ANSI C



C89

C99

C11

Пълен справочник

Първо издание

Димо Петков

ANSI C
Пълен справочник

Първо издание

София, 2014 г.

Всички права запазени. Нито една част от тази книга не може да бъде размножавана или предавана под никаква форма или начин, електронен или механичен, включително фотокопиране, записване или чрез каквито и да е системи за съхранение на информация, без предварително писмено разрешение от автора.

© Димо Петков Петков, автор

ISBN xxxx-xxxx-xxxx-xxxx

Кратко съдържание:

За автора 14
От автора 14
За техническия редактор 14
От техническия редактор 14
Благодарности 15
Предговор 16
За кого е предназначена тази книга 18
Какво да очаквате от тази книга 18
Използвани конвенции 19
Как е организирана книгата 19
Как да четете тази книга 19
Примерите в книгата 20
С стандарти 20

Част I: C89(C90) 22
Лексикални елементи 24
Типове данни 36
Константи 77
Операции и изрази 93
Преобразувания 155
Оператори 195
Изброявания 233
Масиви 243
Указатели 291
Структури 315
Обединения 349
Битови полета 369
Функции 381
Съвместими и композитни типове 413
Декларации 421
Предпроцесор 437

Част II: C99 471
Въведение в C99 473
Описание на нововъведенията в C99 479

Част III: C11 515
Въведение в C11 517
Описание на нововъведенията в C11 521

Част IV: Стандартна Библиотека 537
Въведение в стандартната библиотека 539
Описание на стандартната библиотека 543

Съдържание:

За автора 14
От автора 14
За техническия редактор 14
От техническия редактор 14
Благодарности 15
Предговор 16
За кого е предназначена тази книга 18
Какво да очаквате от тази книга 18
Използвани конвенции 19
Как е организирана книгата 19
Как да четете тази книга 19
Примерите в книгата 20
С стандарти 20

Част I: C89(C90) 22
Лексикални елементи 24
1.1 Общи сведения 26
1.2 Ключови думи 26
1.3 Идентификатор 27
1.4 Константи 29
1.5 Операции 29
1.6 Пунктуатори (разделители) 30
1.7 Имена на хедър-файлове 31
1.8 Триграфи 31
1.9 Предпроцесорни числа 32
1.10 Коментари 33
1.11 Фази на транслиране на С програма 34

Типове данни 36
1.1 Определение за тип 38
1.2 Категории типове 38
1.3 Променливи 41
1.4 Описание на базовите типове данни 42
1.4.1 Видове базови типове данни 42
1.4.2 Модификатори на базовите типове 43
1.4.3 Вътрешно представяне на базовите типове 45
1.5 Квалификатори на типа 45
1.5.1 Квалификатор const 45
1.5.2 Квалификатор volatile 46
1.6 Област на видимост на променливите 47
1.6.1 Област на видимост в прототип на функция 47
1.6.2 Област на видимост във функция 47
1.6.3 Област на видимост в блок с код 49
1.6.4 Област на видимост във файл 52
1.7 Време на живот на променливите 54
1.8 Клас памет на променливите 54
1.8.1 Клас памет auto 55
1.8.2 Клас памет register 55
1.8.3 Клас памет static 56
1.8.4 Клас памет extern 58
1.8.5 Клас памет typedef 60
1.9 Инициализация на променливите 64

1.9.1 Инициализация на променливи от базов тип 64
1.9.2 Инициализация на указатели 65
1.9.3 Инициализация на масиви....66
1.9.4 Инициализация на структрурни-променливи 67
1.9.5 Инициализация на union-променливи 67
1.9.6 Инициализация на enum-променливи....68
1.10 Подравняване на променливите в паметта....68
1.10.1 Подравняване на променливите в 8-битов процесор 68
1.10.2 Подравняване на променливите в 16-битов процесор 70
1.10.3 Подравняване на променливите в 32-битов процесор 71
1.11 Big-endian и Little-endian....74
1.12 Резюме 75

Константи 77
1.1 Видове константи....79
1.2 Литерали и нелитерали 79
1.3 Целочислени константи 80
1.3.1 Десетични константи....81
1.3.2 Шестнадесетични 81
1.3.3 Осмични....82
1.4 Константи с плаваща запетая 82
1.5 Символни константи 83
1.6 Низови константи....85
1.7 Enum-константи 89
1.8 Const-променливи 90
1.9 #define-константи....91

Операции и изрази....93
1.10 L-стойности и R-стойности 95
1.11 Видове операции 95
1.11.1 Аритметични операции (+, -, *, /, %, --, ++) 96
1.11.2 Операции за отношение (<, <=, >, >=)....108
1.11.3 Операции за равенство (==, !=)....113
1.11.4 Логически операции (&&, ||, !)....115
1.11.5 Битови операции (&, |, ^, ~, <<, >>) 121
1.11.6 Операции за присвояване (=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=) .131
1.11.7 Условна операция ?: 133
1.11.8 Операция за последователно изчисляване на изрази , ...135
1.11.9 Операция sizeof за определяна размера на тип 137
1.11.10 Операция за преобразуване на тип (тип) 140
1.11.11 Операции за достъп до членове на структура и обединение (., ->) ...142
1.11.12 Операция за индексиране на масив [ ] 144
1.11.13 Операция за извикване на функция ()....145
1.11.14 Указателни операции *, &....146
1.12 Ред на изчисление на операндите на бинарни и тринарни операции 148
1.13 Изрази 149
1.14 Приоритет и асоциативност на операциите 149
1.15 Странични ефекти и точки на последователност 151
1.15.1 Странични ефекти (side effects) 151
1.15.2 Точки на последователност (sequence points)....152

Преобразувания 155
1.1 Въведение 157
1.2 Целочислено повишение на типовете 157

1.3 Преобразувания между целочислени типове 159
1.3.1 Преобразуване на знакови типове 159
1.3.2 Преобразуване на беззнакови типове 167
1.4 Преобразувания между типове с плаваща запетая 172
1.5 Преобразуване между целочислени типове и типове с плаваща запетая 172
1.5.1 Преобразуване на целочислен тип в тип с плаваща запетая 172
1.5.2 Преобразуване на тип с плаваща запетая в целочислен тип 172
1.6 Преобразувания свързани с указатели 173
1.6.1 Явни (изрични) преобразувания между указатели 174
1.6.2 Неявни(подразбиращи) преобразувания между указатели 177
1.6.3 Други преобразувания свързани с указатели 180
1.7 Преобразувания към тип void 185
1.8 Видове преобразувания 186
1.8.1 Явно преобразуване при използване на операция за преобразуване на типа 186
1.8.2 Преобразувания при присвояване 187
1.8.3 Обичайни унарни преобразувания 188
1.8.4 Обичайни бинарни преобразувания 189
1.8.5 Преобразуване на типа на аргументите при извикване на функция. 191
1.8.6 Преобразуване на типа на стойността на оператора return при връщане на резултат от функция. 192

Оператори 195
1.1 Определение и видове оператори 197
1.2 Операторни изрази 197
1.3 Съставни оператори 198
1.3.1 Декларации и дефиниции в съставен оператор 200
1.4 Условен оператор if-else 202
1.5 Оператор за избор switch 207
1.6 Итеративни оператори (while, do-while, for) 214
1.6.1 Оператор за цикъл while 214
1.6.2 Оператор за цикъл do-while 217
1.6.3 Оператор за цикъл for 218
1.7 Оператори break и continue 222
1.7.1 Оператор break 222
1.7.2 Оператор continue 224
1.8 Оператор return 226
1.9 Оператор goto 227
1.10 Нулев оператор 230

Изброявания 233
1.1 Деклариране на изброяване 235
1.2 Изброяване и typedef 239
1.3 Област на видимост на изброяване 240

Масиви 243
1.1 Едномерни Масиви 245
1.1.1 Определение и дефиниране 245
1.1.2 Инициализиране 245
1.1.3 Индексиране 248
1.1.4 Едномерни масиви и указатели 249
1.1.5 Предаване на масив на функция 252
1.1.6 Едномерен масив от указатели 257
1.1.7 Предаване на масив-от-указатели на функция 261

1.1.8 Масив от указатели-към-функции 262
1.2 Двумерни масиви 263
1.2.1 Определение и дефиниране 263
1.2.2 Инициализиране 265
1.2.3 Индексиране 267
1.2.4 Двумерни масиви и указатели 268
1.2.5 Предаване на двумерен масив на функция 273
1.2.6 Създаване на низова таблица с помощта на двумерен масив 275
1.3 Тримерни масиви 275
1.3.1 Определение и дефиниране 275
1.3.2 Инициализиране 278
1.3.3 Индексиране 280
1.3.4 Тримерни масиви и указатели 282
1.3.5 Предаване на тримерен масив на функция 286
1.4 Многомерни масиви 288
1.4.1 Определение и дефиниране 288
1.4.2 Индексиране 289
1.4.3 N-мерни масиви и указатели 289
1.4.4 Предаване на N-мерен масив на функция 290

Указатели 291
1.1 Определение и свойства 293
1.2 Указатели към обекти 293
1.3 Указатели към функции 296
1.4 Обобщени указатели (void-указатели) 298
1.5 Нулеви указатели 299
1.6 Указатели и квалификатори на типа 300
1.7 Операции с указатели 302
1.7.1 Събиране и изваждане на указател с цяло число 302
1.7.2 Операциите ++ и -- и указатели 303
1.7.3 Изваждане на указатели от един и същи тип 304
1.7.4 Сравнение на указатели 305
1.7.5 Сравнение на указател с цяло число 307
1.8 Указатели като параметри на функция 307
1.9 Указатели връщани от функция 309
1.10 Инициализиране на char-указател с низова константа 310
1.11 Указател към указател 311
1.12 Указател-към-указател и квалификатори на типа 312
1.13 Преобразувания свързани с указатели 313

Структури 315
1.1 Въведение в структурите 317
1.1.1 Деклариране на структура 317
1.1.2 Област на видимост на структура 319
1.1.3 Дефиниране на променливи от структурен тип 319
1.1.4 Инициализация на променливи от структурен тип 322
1.1.5 Структури и typedef 323
1.1.6 Непълна(незавършена) декларация на структура 325
1.1.7 Ограничение на структурите 327
1.2 Операции със структури 329
1.2.1 Достъп до членове на структура 329
1.2.2 Присвояване на структури 331
1.3 Структури и функции 333
1.3.1 Структури като параметри на функции 333

1.3.2 Структури връщани от функции 335
1.4 Масиви от структури 337
1.5 Вложени структури 340
1.6 Разполагане на структурните променливи в паметта 342
1.6.1 Подравняване (alignment) на базовите типове данни 342
1.6.2 Подравняване на членове на структура. Основни правила 342

Обединения 349
1.1 Въведение в обединенията 351
1.1.1 Деклариране на обединение 351
1.1.2 Област на видимост на обединение 352
1.1.3 Дефиниране на union-променливи 353
1.1.4 Инициализация на union-променливи 355
1.1.5 Обединения и typedef 356
1.1.6 Непълна (незавършена) декларация на обединение 357
1.1.7 Ограничение на обединенията 360
1.2 Операции с обединения 361
1.2.1 Достъп до членове на обединение 361
1.2.2 Присвояване на обединения 364
1.3 Разполагане на обединенията в паметта 365
1.4 Използване на обединения 366

Битови полета 369
1.1 Дефиниране на битово поле 371
1.2 Битови полета със специално предназначение 373
1.3 Инициализиране на битови полета 373
1.4 Ограничения на битовите полета 374
1.5 Разполагане на битовите полета в паметта 375

Функции 381
1.1 Въведение 383
1.2 Дефиниране на функция 385
1.3 Функцията main 390
1.4 Деклариране на функция 392
1.4.1 Прототип и област на видимост на прототипа 392
1.4.2 Използване на прототипи 395
1.5 Механизми на подаване на аргументи на функция 401
1.5.1 Подаване на аргументи чрез копие 402
1.5.2 Подаване на аргументи чрез адрес 402
1.6 Глобални (външни) и локални (статични) функции 405
1.6.1 Глобални функции 405
1.6.2 Локални функции 405
1.7 Функции с променлив брой аргументи 406
1.8 Стек 409
1.9 Рекурсивни функции 411

Съвместими и композитни типове 413
1.1 Съвместими типове 415
1.1.1 Съвместимост на аритметични типове 415
1.1.2 Съвместимост на изброявания 415
1.1.3 Съвместимост на структури 416
1.1.4 Съвместимост на обединения 416
1.1.5 Съвместимост на масиви 416
1.1.6 Съвместимост на типове на функции 417

1.1.7 Съвместимост на указатели 418
1.2 Съставни (композитни) типове 418

Декларации 421
1.1 Определение 423
1.2 Обобщен синтаксис на декларации 423
1.3 Декларация и дефиниция 424
1.4 Свързване 426
1.4.1 Въведение 426
1.4.2 Външно свързване 426
1.4.3 Вътрешно свързване 427
1.4.4 Без свързване 428
1.5 Декларатори 429
1.5.1 Прости декларатори 429
1.5.2 Декларатори на указатели 429
1.5.3 Декларатори на масиви 430
1.5.4 Декларатори на функции 431
1.5.5 Дешифриране на сложни декларатори 431

Предпроцесор 437
1.1 Общи сведения 439
1.2 Нулева директива (#) 440
1.3 Директива за включване на файл (#include) 441
1.4 Директива за дефиниране на макроси #define 443
1.4.1 Описание 443
1.4.2 Предефиниране на макроси 450
1.4.3 Повторно сканиране и замяна на макроси 450
1.4.4 Област на видимост на макро-дефинициите 452
1.4.5 Разполагане на макрос на няколко реда 452
1.4.6 Предефинирани макроси 453
1.4.7 Особености на макросите 454
1.5 Директива за отдефиниране на макроси #undef 454
1.6 Директиви за условно компилиране (#if, #ifdef, #ifndef, #else, #elif, #endif) 455
1.6.1 #if 455
1.6.2 #ifdef 460
1.6.3 #ifndef 462
1.7 Директива за номериране на редове (#line) 463
1.8 Директива за грешка (#error) 464
1.9 Имплементационни директиви (#pragma) 465
1.10 Предпроцесорни оператори (defined, # и ##) 465
1.10.1 Предпроцесорен оператор defined 465
1.10.2 Предпроцесорен оператор # (stringizing operator) 467
1.10.3 Предпроцесорен оператор ## (pasting operator) 468

Част II: C99 471
Въведение в C99 473
1.1 Нововъведения в C99 475
1.2 Ключови думи в C99 476
Описание на нововъведенията в C99 479
1.1 Коментар в стил C++ 481
1.2 Смесване на декларации и изпълним код 481
1.3 Нови целочислени типове 482
1.4 Булев тип 483
1.5 Назначени инициализатори 485

1.5.1 Назначен инициализатор на масив 485
1.5.2 Назначен инициализатор на структурна-променлива 486
1.5.3 Назначен инициализатор на обединение 487
1.6 Съставни литерали 488
1.7 VLA-типове 491
1.7.1 Масиви с променлива дължина 491
1.7.2 VLA-модифицирани типове 492
1.7.3 VLA-масиви и typedef 494
1.7.4 VLA-типове като параметри на функции 494
1.8 Вложени функции 495
1.9 Ограничени указатели 497
1.10 Гъвкави масиви 498
1.11 static и квалификатори на типа в декларатор на масив 501
1.12 Макроси с променлив брой аргументи 505
1.13 Нови предефинирани макроси 508
1.14 Стандартни pragma-директиви 509
1.15 _Pragma предпроцесорен оператор 510
1.16 Предефиниран идентификатор __func__ 510
1.17 Други 511

Част III: C11 515
Въведение в C11 517
1.1 Нововъведения в C11 519
1.2 Ключови думи в C11 519
Описание на нововъведенията в C11 521
1.1 Спецификатор _Atomic(тип) 523
1.2 Квалификатор _Atomic 523
1.3 Оператор _Alignof 523
1.4 Спецификатор за подравняване _Alignas 524
1.5 Спецификатор _Noreturn 526
1.6 Анонимни структури и обединения 526
1.7 Многопоточност 527
1.7.1 Управление на многопоточността в C11 527
1.7.2 Примери 529

Част IV: Стандартна Библиотека **537**
Въведение в стандартната библиотека 539
1.1 Общи сведения 541
Описание на стандартната библиотека 543
1.1 Описание на стандартните хедъри 545
1.2 <assert.h> 545
1.2.1 Описание 545
1.2.2 Декларации на типове 545
1.2.3 Декларации на макроси 545
1.2.4 Декларации на функции 545
1.2.5 Описание на функциите (макро-функциите) 545
1.3 <complex.h> 546
1.3.1 Описание 546
1.3.2 Декларации на типове 546
1.3.3 Декларации на макроси 546
1.3.4 Декларации на функции 547
1.3.5 Описание на функциите (макро-функциите) 548
1.4 <ctype.h> 549
1.4.1 Описание 549

1.4.2 Декларации на типове 549
1.4.3 Декларации на макроси 549
1.4.4 Декларации на функции 549
1.4.5 Описание на функциите (макро-функциите) 550
1.5 <errno.h> 558
1.5.1 Описание 558
1.5.2 Декларации на типове 559
1.5.3 Декларации на макроси 559
1.5.4 Декларации на функции 560
1.5.5 Описание на функциите (макро-функциите) 560
1.6 <fenv.h> 560
1.6.1 Описание 560
1.6.2 Декларации на типове 560
1.6.3 Декларации на макроси 560
1.6.4 Декларации на функции 560
1.6.5 Описание на функциите (макро-функциите) 560
1.7 <float.h> 560
1.7.1 Описание 560
1.7.2 Декларации на типове 561
1.7.3 Декларации на макроси 561
1.7.4 Декларации на функции 562
1.7.5 Описание на функциите (макро-функциите) 562
1.8 <inttypes.h> 562
1.8.1 Описание 562
1.8.2 Декларации на типове 562
1.8.3 Декларации на макроси 562
1.8.4 Декларации на функции 562
1.8.5 Описание на функциите (макро-функциите) 562
1.9 <iso646.h> 562
1.9.1 Описание 562
1.9.2 Декларации на типове 562
1.9.3 Декларации на макроси 562
1.9.4 Декларации на функции 563
1.9.5 Оисание на функциите (макро-функциите) 563
1.10 <limits.h> 563
1.10.1 Описание 563
1.10.2 Декларации на типове 563
1.10.3 Декларации на макроси 563
1.10.4 Декларации на функции 564
1.10.5 Описание на функциите (макро-функциите) 564
1.11 <locale.h> 564
1.11.1 Описание 564
1.11.2 Декларации на типове 565
1.11.3 Декларации на макроси 568
1.11.4 Декларации на функции 568
1.11.5 Описание на функциите (макро-функциите) 568
1.12 <math.h> 570
1.12.1 Описание 570
1.12.2 Декларации на типове 571
1.12.3 Деларации на макроси 571
1.12.4 Декларации на функции 571
1.12.5 Описание на функциите (макро-функциите) 572
1.13 <setjmp.h> 582
1.13.1 Описание 582

1.13.2 Декларации на типове 583
1.13.3 Декларации на макроси 583
1.13.4 Декларации на функции 584
1.13.5 Описание на функциите (макро-функциите) 585
1.14 <signal.h> 587
1.14.1 Описание 587
1.14.2 Декларации на типове 589
1.14.3 Декларации на макроси 589
1.14.4 Декларации на функции 589
1.14.5 Описание на функциите (макро-функциите) 589
1.15 <stdalign.h> 592
1.15.1 Описание 592
1.15.2 Декларации на типове 592
1.15.3 Декларации на макроси 592
1.15.4 Декларации на функции 592
1.15.5 Описание на функциите (макро-функциите) 592
1.16 <stdarg.h> 592
1.16.1 Описание 592
1.16.2 Декларации на типове 593
1.16.3 Декларации на макроси (макро-функции) 593
1.16.4 Декларации на функции 594
1.16.5 Описание на функциите (макро-функциите) 594
1.17 <stdatomic.h> 596
1.17.1 Описание 596
1.17.2 Декларации на типове 596
1.17.3 Декларации на макроси 596
1.17.4 Декларации на функции 596
1.17.5 Описание на функциите (макро-функциите) 596
1.18 <stdbool.h> 596
1.18.1 Описание 596
1.18.2 Декларации на типове 596
1.18.3 Декларации на макроси 596
1.18.4 Декларации на функции 596
1.18.5 Описание на функциите (макро-функциите) 596
1.19 <stddef.h> 596
1.19.1 Описание 596
1.19.2 Декларации на типове 597
1.19.3 Декларации на макроси 598
1.19.4 Декларации на функции 599
1.19.5 Оисание на функциите (макро-функциите) 599
1.20 <stdint.h> 599
1.20.1 Описание 599
1.20.2 Декларации на типове 599
1.20.3 Декларации на макроси 599
1.20.4 Декларации на функции 599
1.20.5 Описание на функциите (макро-функциите) 599
1.21 <stdio.h> 600
1.21.1 Описание 600
1.21.2 Декларации на типове 605
1.21.3 Декларации на макроси 605
1.21.4 Декларации на функции 607
1.21.5 Описание на функциите (макро-функциите) 608
1.22 <stdlib.h> 646
1.22.1 Описание 646

1.22.2 Декларации на типове 648
1.22.3 Декларации на макроси 649
1.22.4 Декларации на функции 649
1.22.5 Оисание на функциите (макро-функциите) 649
1.23 <stdnoreturn.h> 676
1.23.1 Описание 676
1.23.2 Декларации на типове 676
1.23.3 Декларации на макроси 677
1.23.4 Декларации на функции 677
1.23.5 Описание на функциите (макро-функциите) 677
1.24 <string.h> 677
1.24.1 Описание 677
1.24.2 Декларации на типове 678
1.24.3 Декларации на макроси 678
1.24.4 Декларации на функции 678
1.24.5 Оисание на функциите (макро-функциите) 679
1.25 <tgmath.h> 695
1.25.1 Описание 695
1.25.2 Декларации на типове 695
1.25.3 Декларации на макроси 695
1.25.4 Декларации на функции 695
1.25.5 Описание на функциите (макро-функциите) 695
1.26 <thread.h> 695
1.26.1 Описание 695
1.26.2 Декларации на типове 695
1.26.3 Декларации на макроси 695
1.26.4 Декларации на функции 695
1.26.5 Описание на функциите (макро-функциите) 695
1.27 <time.h> 695
1.27.1 Описание 695
1.27.2 Декларации на типове 695
1.27.3 Декларации на макроси 696
1.27.4 Декларации на функции 696
1.27.5 Оисание на функциите (макро-функциите) 696
1.28 <uchar.h> 705
1.28.1 Описание 705
1.28.2 Декларации на типове 705
1.28.3 Декларации на макроси 705
1.28.4 Декларации на функции 705
1.28.5 Описание на функциите (макро-функциите) 705
1.29 <wchar.h> 705
1.29.1 Описание 705
1.29.2 Декларации на типове 706
1.29.3 Декларации на макроси 706
1.29.4 Декларации на функции 707
1.29.5 Оисание на функциите (макро-функциите) 707
1.30 <wctype.h> 707
1.30.1 Описание 707
1.30.2 Декларации на типове 707
1.30.3 Декларации на макроси 708
1.30.4 Декларации на функции 708
1.30.5 Оисание на функциите (макро-функциите) 708

За автора



ДИМО ПЕТКОВ завършва Техническия Университет в Габрово през 2001г. Има магистърска степен по специалността **Комуникационна Техника и Технологии**. Започва кариерата си като сервизен инженер на аудио-видео техника. По-късно работи няколко години в белгийската фирма **Мелексис** като приложен инженер. През цялото това време се занимава с изучаване на програмирането на микроконтролери. Понастоящем работи като С-програмист в немската компания **КОСТАЛ**, занимаваща се с разработката на електронни модули за автомобили.

От автора

Има много книги (на английски език), описващи езика С, но малка част от тях претендират, че описват в пълнота езика С. Всяка от тях си има своите предимства и недостатъци. Създадох тази книга с единствената цел да спестя на С-програмистите лутането в търсене на подходяща книга, написана на разбираем език и съдържаща пълно описание на езика С. Надявам се, че съм успял да постигна до голяма степен целта си. Може да изпращате Вашите въпроси и забележки на dimolin@mail.bg.

За техническия редактор



ЕМИЛ КИРИЛОВ завършва Техническия университет в София през 2008г. Има магистърска степен по специалността **Комуникационна Техника и Технологии**. Още като ученик започва кариерата си като автор на технически статии и като техник. През 2005 се преориентира да работи в областта на цифровата обработка на изображения и вградения софтуер. В момента работи в международната фирма **Плейтех** като разработчик на компютърни игри на C++.

От техническия редактор

Благодарности

Благодарности на всички читатели, които ми се довериха и избраха тази книга като справочник на езика C.

Специални благодарности за подкрепата и търпението, което прояви моята скъпа Вики.

Предговор

C е мощен процедурно-ориентиран език, създаден през 1972 година от Денис Ричи. Създаването на C предизвиква революция в програмирането, което го прави най-предпочитания език за програмиране следващите две-три десетилетия. С развитието на технологиите възникват и по сложни за решаване проблеми. Това води до нуждата от нови езици, които да се справят с това. Така се появяват обектно-ориентираните езици C++, Java и C#. Всички те "стъпват" на синтаксиса на C. Въпреки, че тези езици ограничават сферата на използване на C, той си остава популярен език за програмиране и в днешно време. Развитието на микроелектрониката доведе до създаване на "интелигентни" интегрални схеми, наречени микроконтролери. Микроконтролерът е малък компютър, събран в един корпус и с ограничени ресурси в сравнение с един персонален компютър.

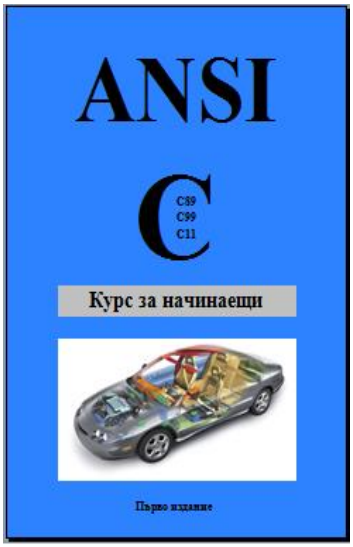


Терминът "интелигентен" е метафоричен, тъй като микроконтролерът се превръща в такъв само когато се програмира да върши определена задача, в противен случай той е просто една безполезна интегрална схема. Микроконтролерите се използват във всички сфери на живота. Почти всички съвременни електронни устройства използват микроконтролери, от малки домакински уреди до печки, хладилници и т.н. Всички съвременни автомобили съдържат редица електронни модули с микроконтролери.



Именно в програмирането на микроконтролери езикът С остава незаменим, а това води и до нужда от С-програмисти.

Тази книга е втората от поредица книги, които ще Ви дадат пълна информация в максимално достъпен вид, която е необходима за да се научите да програмирате микроконтролери на езика С. Първата стъпка е да се запознаете с елементите на езика. За тази цел е предвидена книгата:



Тя е предназначена основно за хора без никакъв опит в програмирането, но ще бъде много полезна и за напреднали С-програмисти. Ако не сте програмирали досега, значи тази книга е точно за Вас. Тя е подготвена така, че да Ви запознае с всички базови понятия и термини, които ще са Ви необходими при изучаването на езика С. Тази книга разглежда почти всички аспекти на езика С, но не влиза в дълбоки детайли, които биха затруднили начинаещия програмист. Настоящата книга разглежда детайлно езика С и може да се използва за справочник дори и от опитни програмисти. След като усвоите езика С, втората стъпка е да се запознаете с устройството и принципа на действие на един микроконтролер. За тази цел е предвидена трета книга:



Книгата е в процес на разработка.

Тя ще Ви даде всички необходими базови знания за градивните елементи и принципа на работа на един микроконтролер от гледна точка на един програмист.

Програмирането се учи най-вече с практика. Това е и третата стъпка. За целта е предвидена четвърта книга, която съдържа практически примери. Тя комбинира наученото от предходните две стъпки и ще Ви даде практически познания за софтуерните и хардуерни средства, необходими

при написването, тестването и дебъгването (откриването и коригирането на грешки) на C код, на базата на конкретен микроконтролер.



Книгата е в процес на разработка.

По мнението на много специалисти C е фундаментален език, който всеки програмист би трябвало да знае. Аз също споделям това мнение. C е "малък" език в сравнение с модерните обектно-ориентирани езици C++, C# и Java и в същото време е актуален и мощен език, който намира широко приложение в индустрията за програмирането на устройства, използващи микроконтролери. Дори и да не започнете да се занимавате с програмиране на микроконтролери или ако просто искате да разширите своите познания, познаването на езика C много ще Ви улесни в случай, че решите да изучавате C++, C# или Java.

За кого е предназначена тази книга

Тази книга е предназначена за хора, които имат вече опит с програмирането на C, но искат да задълбочат своите познания. Книгата е подготвена така, че може да се използва и като справочник от опитни програмисти. Ако сте абсолютно начинаещ, ще трябва първо да се запознаете с основите на езика C. За тази цел е предвидена книгата **ANSI C. Курс за начинаещи**.

Какво да очаквате от тази книга

Тази книга описва всички детайли на езика C, които ще са Ви необходими в ежедневното програмиране.

Използвани конвенции



Текстът в това поле отбелязва някои по-важни моменти, които е добре да запомните.



Текстът в това поле отбелязва съвети, които е препоръчително да следвате.

Забележка: Текстът в това поле съдържа пояснения и коментари.

Как е организирана книгата

Книгата е организирана в четири части.

Част I описва стандарта C89(C90). Това е стандартът, който е добре установен и намира най-широко приложение в програмирането на микроконтролери.

Част II описва разликите и нововъведенията, добавени в следващата версия на езика C: C99. Повечето компилатори в по-малка или по-голяма степен също поддържат този стандарт.

Част III описва разликите и нововъведенията, добавени в текущата версия на езика C: C11. Този стандарт е твърде нов по-време на писането на тази книга и се поддържа едва от няколко компилатора.

Част IV описва стандартната библиотека на езика C.

Как да четете тази книга

Ако сте начинаещ и сте прочели книгата **ANSI C. Курс за начинаещи** и искате да задълбочите познанията си по C, най-добре е да прочетете книгата от началото до края. Ако сте опитен програмист, който се нуждае от конкретна информация свързана с езика, просто потърсете необходимата тема.

Примерите в книгата

Примерите в книгата могат да бъдат изтеглени от:

www.progstarter.com

С стандарти

Езикът С е стандартизиран първо през 1989г от ANSI (American National Standards Institute) комитета. През 1990 този стандарт е приет и от международната организация ISO (International Standards Organization). Затова той се нарича още ANSI/ISO стандарт на езика С. Тези два стандарта са еднакви и често се наричат С89 или С90 и се означават като **ANSI X3.159-1989** и **ISO/IEC 9899:1990** съответно . През следващите няколко години стандартът претърпява няколко корекции и добавки както е показано долу.

ISO/IEC 9899:1990/Cor 1:1994

ISO/IEC 9899:1990/Amd 1:1995 (Наричан понякога формално С95)

ISO/IEC 9899:1990/Cor 2:1996

През 1999г. е приет нов стандарт **ISO/IEC 9899:1999**, който заменя предходния с всички корекции и добавки. Този стандарт е известен още като С99. Той също претърпява изменения през следващите няколко години както е показано долу.

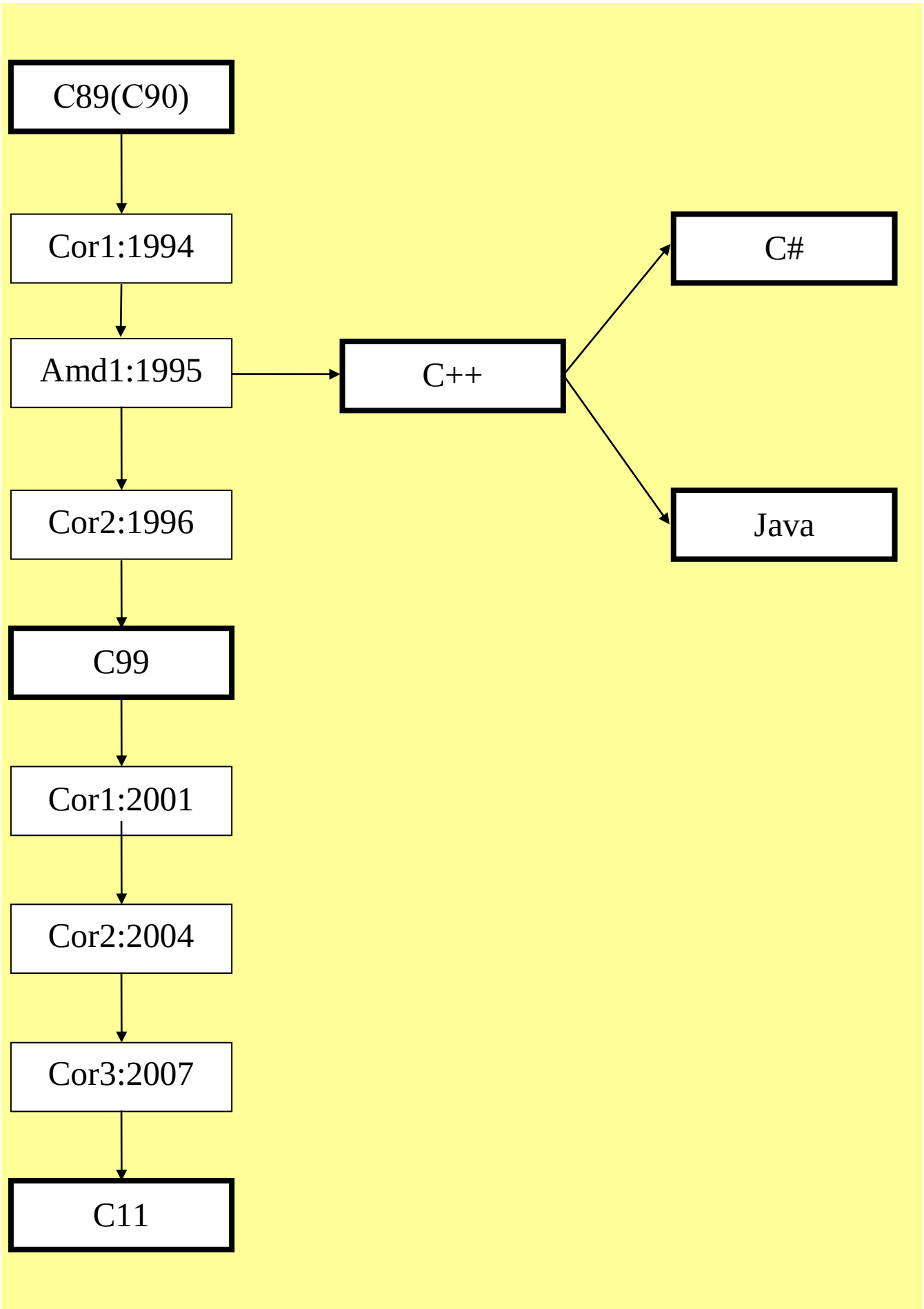
ISO/IEC 9899:1999/Cor 1:2001

ISO/IEC 9899:1999/Cor 2:2004

ISO/IEC 9899:1999/Cor 3:2007

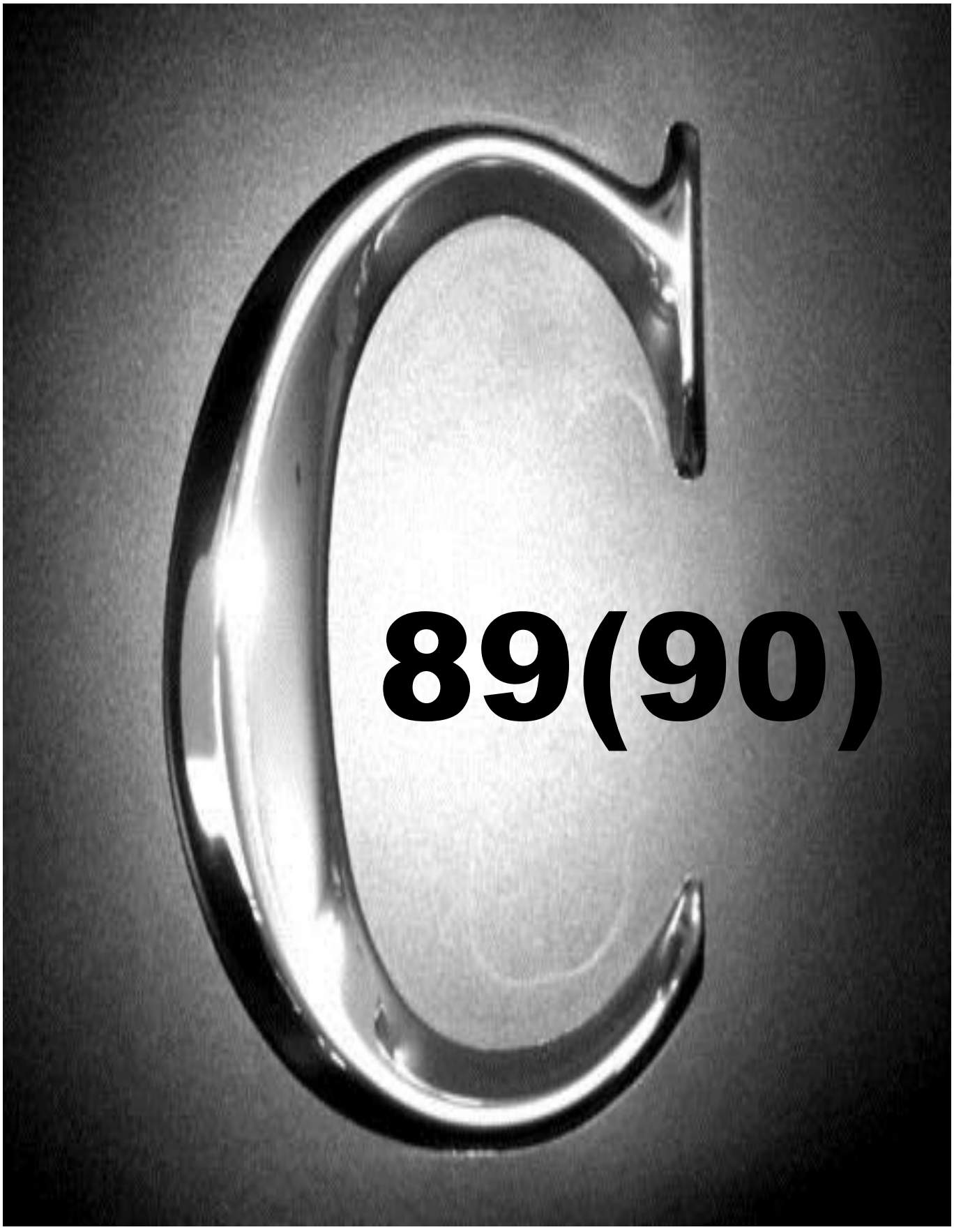
През 2011г. е приет нов стандарт **ISO/IEC 9899:2011**, формално наричан С11.
Стандартът С95 (ISO/IEC 9899:1990/Amd 1:1995) е основа на езика С++, който пък е основа на двата най-популярни езика за обектно-ориентирано програмиране: C# и Java.

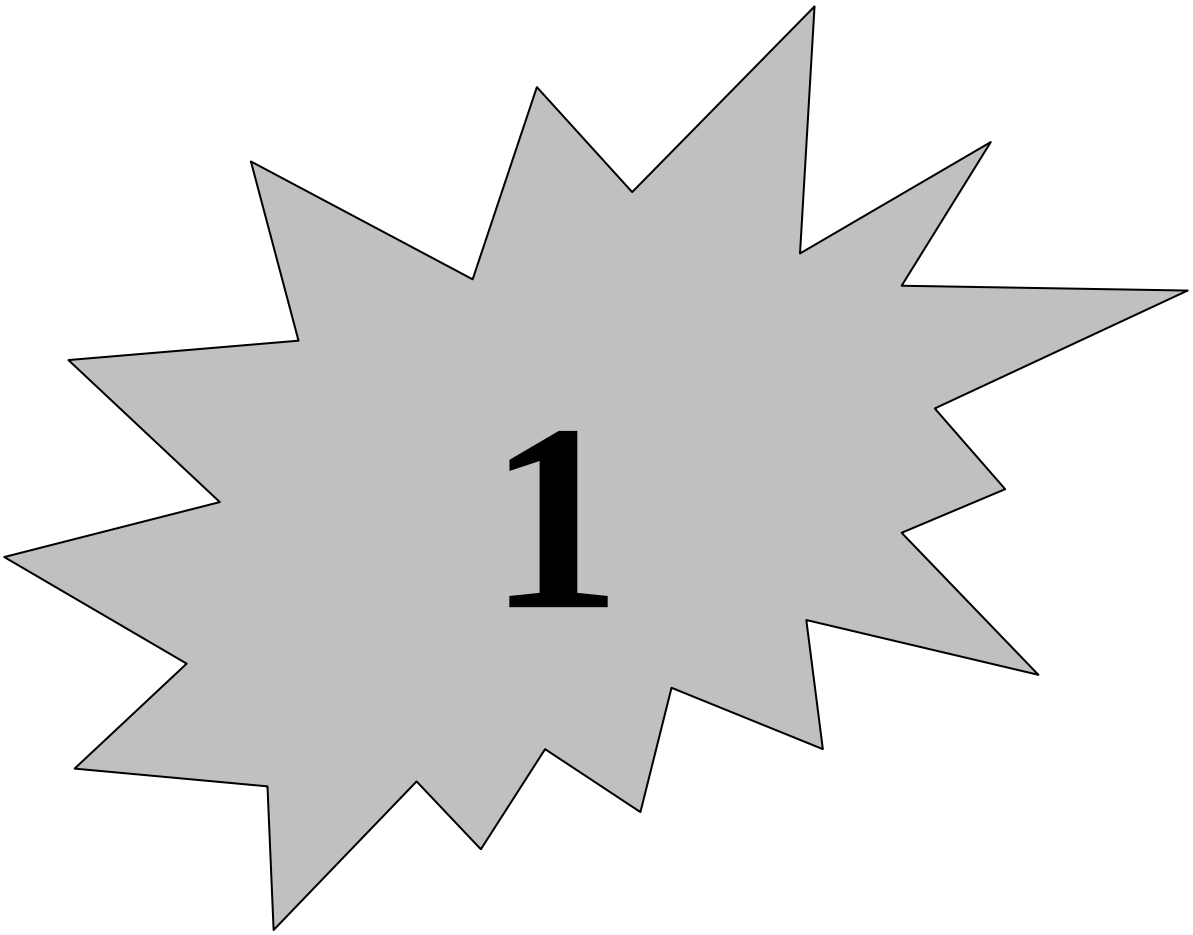
Фиг.1 показва нагледно описаните по-горе периоди на стандартизиране на езика С.



Фиг. 1 Стандартизиране на езика С

Част I: C89(C90)





Лексикални елементи

1.1 Общи сведения

Минималният лексикален елемент, до който една С програма се разбива по време на компилиране се нарича лексема (**token**). Лексемите от които се състои една С програма са:

Лексема:

Ключова дума
Идентификатор
Константа
Операция
Пунктуатор

Предпроцесорна лексема:

Име на хедър-файл
Идентификатор
Предпроцесорно число
Символна константа
Низова константа
Операция
Пунктуатор

Всяка предпроцесорна лексема се преобраува в лексема в процеса на компилация.

1.2 Ключови думи

Табл.1 изброява всичките 32 ключови думи в С89.

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**volatile**
do	**if**	**static**	**while**

Табл. 1 Ключови думи в С89/С90

Изброените ключови думи са резервирани в езика С и не могат да се използват за други цели.

1.3 Идентификатор

Идентификаторът се състои от:

- Латински символи

a b c d e f g h i j k l m
n o p q r s t u v w x y z

A B C D E F G H I J K L M
N O P Q R S T U V X Y Z

- Арабски цифри

0 1 2 3 4 5 6 7 8 9

- Подчертаващо тире

_

Първият символ на идентификатора не може да бъде цифра.

Пример:

sum1 – валиден идентификатор
_sum – валиден идентификатор
1sum – НЕвалиден идентификатор

Езикът С прави разлика между малки и големи букви. Това означава, че sum1, Sum1, SUM1, sUm1 и т.н. се възприемат като различни идентификатори от компилатора.

Един идентификатор може да обозначава една от следните единици в езика С:

- **име на променлива**
- **име на функция**
- **етикет на структура, обединение , изброяване (tag)**
- **етикет на оператор (label)**
- **член на на структура, обединение или изброяване**
- **typedef име**

- **макрос**
- **параметър на макрос**

Пример:

```
int sum; /* sum е идентификатор на променлива */

int sum_xy (int x, int y); /* sum_xy е идентификатор на функция */

struct mystruct /* mystruct  е идентификатор на етикет(име) на структура */
{
      int a; /* a е идентификатор на член на структура */
      int b; /* b е идентификатор на член на структура */
};

union myunion /* myunion  е идентификатор на етикет(име) на обединение */
{
      int i;    /* i е идентификатор на член на обединение */
      double d; /* d е идентификатор на член на обединение */
};

enum myenum /* myenum  е идентификатор на етикет(име) на изброяване */
{
      I,   /* I е идентификатор на член на изброяване   */
      You, /* You е идентификатор на член на изброяване */
      He,  /* He е идентификатор на член на изброяване  */
      She  /* She е идентификатор на член на изброяване */
};

typedef unsigned char uint8; /* uint8 е идентификатор на typedef име */

addition:  /* addition е идентификатор на етикет на оператор  */
sum = sum_xy (10, 20);

#define message    "Hello C!\n" /* message е идентификатор на макрос */

#define print_message(m)  printf("%s\n", m); /* m е идентификатор на параметър на
                                                макрос */
```

Идентификаторите, обозначаващи различни C единици могат да съвпадат стига да се намират в различни категории (именовани пространства).

Именованите пространства в езика C са:

- етикети на оператори
- етикети на структури, обединения и изброявания
- членове на структури и изброявания
- всички останали (имена на променливи, функции, typedef имена и enum-константи)

Пример 1-1:

```c
#include <stdio.h>

struct id /* етикет (име) на структура */
{
	int id; /* член на структура */
};

int main(void)
{
	char id = 'j'; /* име на променлива */
	struct id  s;

	s.id = 10;

id:/* етикет на оператор */
	printf("%c\n", id);
	printf("%d\n", s.id);

	return 0;
}
```

В показания пример идентификаторът id е използван да обозначи няколко различни единици: етикет на структура, член на структура, име на променлива, етикет на оператор. Тъй като всички тези единици принадлежат на различни именовани пространства това не създава проблеми за компилатора.

1.4 Константи

Константите се разделят на следните видове:

- целочислени константи
- константи с плаваща запетая
- enum-константи
- символни константи
- низови константи

Константите са разгледани в Глава 3 Константи.

1.5 Операции

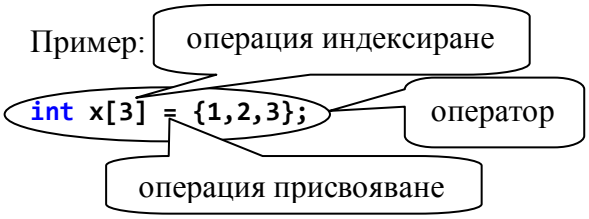
Операцията[1] в C води до изпълнението на някакво действие в резултат, на което се получава стойност например и т.н. Единиците, върху които

операциите действат се наричат операнди. По долу са дадени всички операции в езика C.

[ ]	**()**	**.**	**->**					
++	**++**	**&**	*****	**+**	**-**	**~**	**!**	**sizeof**
/	**%**	**<<**	**>>**	**<**	**>**	**<=**	**>=**	**==**
!=	**^**	**|**	**&&**	**||**	**? :**	**=**	***=**	**/=**
%=	**+=**	**-=**	**<<=**	**>>=**	**&=**	**^=**	**|=**	**,**

Забележка[1]: По принцип в чужденстранната литература термините "оператор" и "операция" са взаимнозаменяеми. В тази книга обаче, всеки един от горните символи се нарича "операция", а терминът "оператор" се използва за обозначаване на цяла програмна инструкция, която включва в себе си една или повече от горните операции, или за обозначаване на управляващите оператори if, if-else, switch, for, while, do-while, break, continue и goto.

Пример:



Допълнително в заглавния-файл <**iso646.h**> са дефинирани макроси, които се разширяват[1] до определени операции както е показано:

```
#define    and       &&
#define    and_eq    &=
#define    bitand    &
#define    bitor     |
#define    compl     ~
#define    not       !
#define    not_eq    !=
#define    or        ||
#define    or_eq     |=
#define    xor       ^
#define    xor_eq    ^=
```

Забележка[1]**:** Процесът на замяна на макрос се нарича разширяване или разгъване на макроса.

1.6 Пунктуатори (разделители)

Пунктуаторът е символ, който има независимо синтактично значение и не извършва действие. В зависимост от контекста на програмата един символ може да бъде операция или пунктуатор. По- долу са дадени всички пунктуатори в C:

[ ] ( ) { } * , : = ; ...

Някои символи за пунктуатори имат алтернативна форма на запис чрез така наречените диаграфи.

пунктуатор	диаграф
{	<%
}	%>
[	<:
]	:>
#	%:
##	%:%:

Табл. 2 Диаграфи

Пример 1-2:

```
int main(void)
<% /* Еквивалентен запис на { */


      return 0;
%> /* Еквивалентен запис на } */
```

1.7 Имена на хедър-файлове

<име-хедър-файл>
"име- хедър-файл"

Този вид предпроцесорна лексема може да се появява само в #include предпроцесорната директива. Имената на хедър-файлове могат да включват всички символи без символ за нов ред и ". Допълнително са в сила следните ограничения:

- Ако символите ', \ , " или /* се появят между разделителите < и >, поведението е недефинирано.

- Ако символите ', \ , или /* се появят между разделителите " и ", поведението е недефинирано.

1.8 Триграфи

Някои символи от таблицата със символи използвани в C не са налични във всички среди. Липсващите символи в тези среди се заменят с

последователности от три символа наречени триграфи. Наличните триграфи са:

символ	триграф
[	??(
\	??/
]	??)
^	??'
{	??<
|	??|
}	??>
~	??-
#	??=

Табл. 3 Триграфи

Пример 1-3:

```
#include <stdio.h>

int main(void)
{
  char arr??(??) = "Hello C??/n"; /* char arr[] =  "Hello C\n"; */

  printf("%s\n" , arr);

  return 0;
}
```

Следните символи имат диаграфно и триграфно представяне:

символ	диаграф	триграф
{	<%	??<
}	%>	??>
[	<:	??(
]	:>	??)
#	%:	??=

Табл. 4 Символи с диаграфно и триграфно представяне

1.9 Предпроцесорни числа

Едно предпроцесорно число започва с цифра, която по избор може да бъде предшествана от точка, и може да бъде последвана от букви, подчертаващи тирета, цифри, точки и e+, e-, E+ или E-. Предпроцесорните числа включват целочислени константи и константи с плаваща запетая плюс

много други форми, които може да не са първоначално валидни числа. Предпроцесорното число няма тип или стойност. То може да ги "получи" след като се преобразува до число с плаваща запетая или целочислена константата след разгъване на макроса, в който участва.

Пример 1-4:

```
#include <stdio.h>

#define str(X)          #X[1]
#define concat(X,Y)     X##Y[1]
#define MAX             100

int main( void )
{
  const char   *s = str(123);       /* const char* s = "123";     */
        int    concat(sum, 1);      /* int    sum1;               */
        float  i = concat (1e,10);  /* float   i = 1e10);         */

        concat (sum, 1) = MAX;      /* sum1 = 100                 */

        printf("%s\n", s);
        printf("%f\n", i);
        printf("%d\n", sum1);

        return 0;
}
```

Забележка[1]: Символите # и ## се използват като предпроцесорни оператори. Те са описани в Глава 16 Предпроцесор.

1.10 Коментари

Коментарите представлява потребителски съобщения затворени между **/*** и ***/.** Всичко поставено между тези символи се игнорира от компилатора и служи единствено за пояснение на програмата. Това е единствения коментар дефиниран в стандарта C89/C90. Вложени коментари не се поддържат.

Пример:

```
/* Това е коментар */

/* Коментар
   разположен
   на няколко
   реда */
```

1.11 Фази на транслиране на C програма

Обобщената компилация на C сорс-файл включва следните фази:

1. Триграфите се заменят с вътрешното представяне на съответните символи.

2. Всички сорс-редове, завършващи с обратно наклонена черта (\) последвана от символ за нов ред, се свързват със следващия ред, формирайки логически ред от няколко физически реда. Освен ако е празен, един сорс-файл трябва да завършва със символ за нов ред, който не е предшестван от обратно наклонена черта (\).

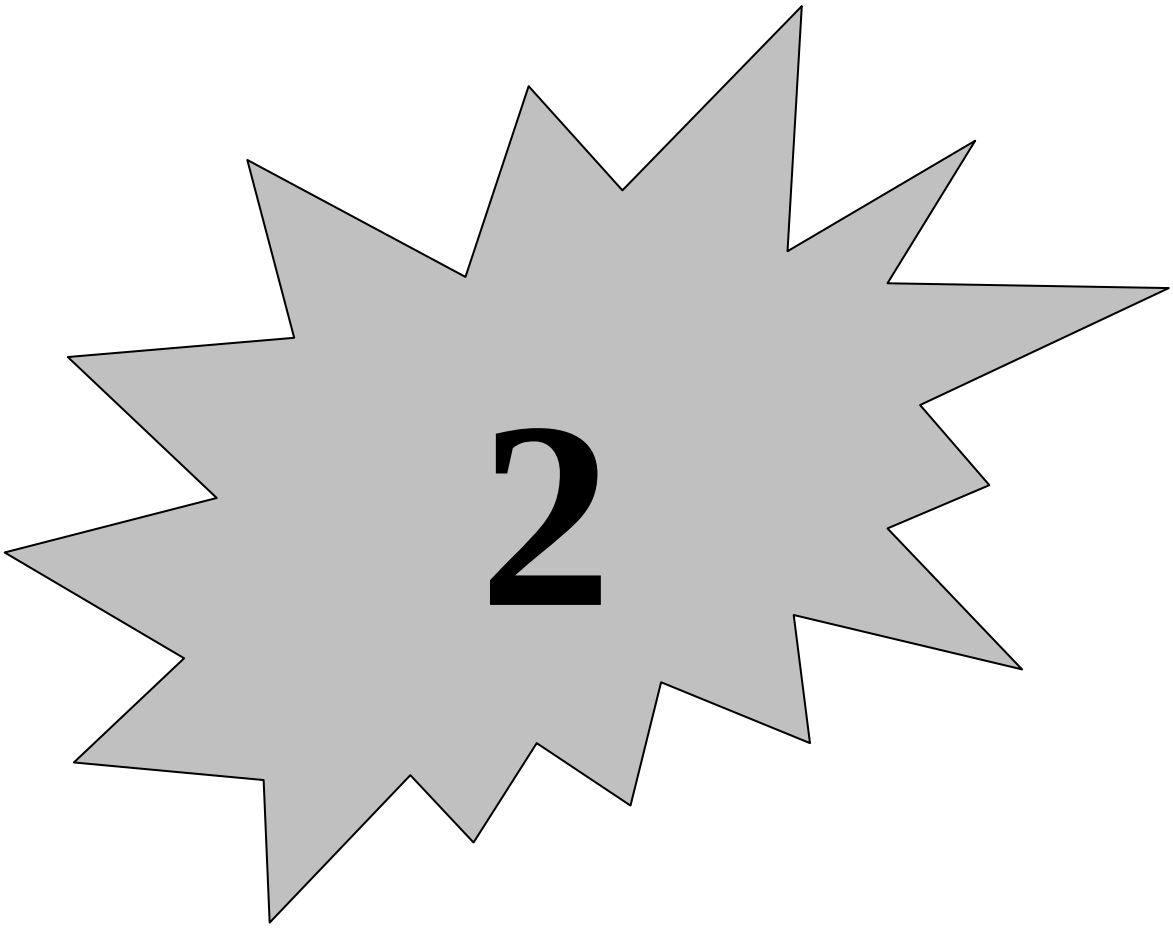
3. Сорс-файлът се разбива на предпроцесорни лексеми и бели-символи (интервал ' ', хор. табулация '\t', вер. табулация '\v', връщане на курсора '\r', нов ред '\n', нова страница '\f'). Всеки коментар се заменя със символ за интервал. Символите за нов ред се запазват.

4. Предпроцесорните директиви се изпълняват и се извършва разширяване на макросите. Фази 1 ÷ 4 се прилагат рекурсивно за всички файлове вмъкнати с предпроцесорната директива #include.

5. Съседните низове се свързват в един низ.

6. Всички лексеми се анализират синтактично и се преобразуват в обектен код.

7. Извиква се линкера.



Типове данни

1.1 Определение за тип

Езикът С включва голямо разнообразие от данни. Всеки отделен вид данни се характеризира с тип. Типът определя диапазона от стойности на конкретния вид данни и операциите, които могат да се прилагат върху тях. Данните могат да се представят по два начина: чрез константи (описани са в Глава 3 Константи) или променливи (вижте точка 3 Променливи).

1.2 Категории типове

В Табл.5 са показани всички категории типове в езика С.

<table>
<tr><th>С типове</th><th colspan="3">категории</th></tr>
<tr><td>char (signed и unsigned)</td><td rowspan="3">Целочислени типове</td><td rowspan="4">Аритметични типове</td><td rowspan="6">Скаларни типове</td></tr>
<tr><td>short, int, long
(signed и unsigned)</td></tr>
<tr><td>enum T{…}</td></tr>
<tr><td>float, double, long double</td><td>Типове с плаваща запетая</td></tr>
<tr><td>T*</td><td colspan="2">Указатели</td></tr>
<tr><td>T […]</td><td colspan="2">Масиви</td><td rowspan="2">Агрегатни типове</td></tr>
<tr><td>struct T{…}</td><td colspan="2">Структури</td></tr>
<tr><td>union T{…}</td><td colspan="3">Обединения</td></tr>
<tr><td>T (…)</td><td colspan="3">Функции</td></tr>
<tr><td>void</td><td colspan="3">тип без стойност</td></tr>
</table>

Табл. 5 Категории типове данни в С

- Целочислени типове

Както подсказва името, тези типове данни могат да представят само цели числа. За по-ефикасно използване на паметта езика С включва няколко целочислени типа, всеки от които може да представи различен диапазон от стойности.

- Типове с плаваща запетая

Тези типове се използват за представяне на реални числа (дробни числа). За по-ефикасно използване на паметта езика С включва няколко типа с плаваща запетая, всеки от които може да представи различен диапазон от

стойности.

Сумарно целочислените типове и типовете с плаваща запетая се наричат аритметични типове. Аритемтичните типове, без enum-типовете, се наричат още базови типове. Това са типовете, на базата на които се получават останалите типове данни в C. Само те ще бъдат разгледани подробно в тази глава. Всички останали типове са разгледани в отделни глави.

- Изброявания

Изброяването е съвкупност от целочислени константи, зададени чрез идентификатори.

Изброяванията са описани подробно в Глава 7 Изброявания.

- Масиви

Масивът описва съвкупност от обекти[1] от един и същ тип и с общо име. Обектите принадлежащи на даден масив се наричат елементи на масива.

Забележка[1]: Под обект в C се разбира променлива от всеки тип, който е различен от функция.

Масивите са описани подробно в Глава 8 Масиви.

- Указатели

Един указателен тип описва обект, чиято стойност представлява адрес на друг обект.

Указателните типове са описани подробно в Глава 9 Указатели.

Сумарно аритметичните и указателните типове се наричат скаларни типове.

- Структури

Структурата представлява съвкупност от обекти, които могат да са от различен тип. Обектите, принадлежащи на една структура се наричат членове на структурата.

Структурите са описани подробно в Глава 10 Структури.

Сумарно структурите и масивите се наричат агрегатни типове.

- Обединения

Обединението представлява съвкупност от припокриващи се (частично или напълно) обекти, които могат да са от различен тип. Припокриващи се обекти означава, че те използват обща памет, която е достатъчно голяма да побере обекта с най-голям размер. Обектите, принадлежащи на едно обединение се наричат членове на обединението.

Обединенията са описани подробно в Глава 11 Обединения.

- Функции

Функцията е група от оператори, които се изпълняват при извикването ѝ в някоя точка от програмата. Функциите могат да получават данни при извикването си и могат да връщат резултат. Типът на една функция се определя от типа на връщания резултат и броя и типа на параметрите на функцията.

Функциите са описани подробно в Глава 13 Функции.

- тип void

Тип void е непълен (незавършен) тип, който представлява тип без стойност. Той се използва за дефиниране на:

1. Обобщени указатели (указатели, които могат да сочат към всякакви типове данни) (вижте Глава 9 Указатели).

2. Функции, които не връщат стойност (вижте Глава 13 Функции).

3. Функции, които не приемат аргументи (вижте Глава 13 Функции).

Най-общо всички гореописани категории типове се разделят на 3 групи:

- **Обектни типове** – това са типове, описващи обекти (променливи)

- **Функционални типове** – това са типове, описващи функции.

- **Непълни типове** – това са типове, които описват обекти, но липсва информация необходима за определяне на техния размер. Непълен е тип void, също могат да се декларират непълни типове за масиви, структури и обединения. Всички непълни типове (без void) могат да станат пълни (завършени). Един пълен тип носи цялата информация, описваща този тип. Повече за непълните типове е описано в Глава 8 Масиви, Глава 9 Указатели, Глава 10 Структури и Глава 11 Обединения.

1.3 Променливи

Преди да преминем към разглеждането на базовите типове данни ще се запознаем с един важен термин в програмирането – променлива. Променлива това е област от паметта на компютъра, която се използва за съхранение на данни. Както и самото понятие подсказва, в процеса на изпълнение на програмата, стойността на тази променлива може да се променя. В C една променлива трябва да бъде дефинирана (обявена) преди да бъде използвана някъде в кода. Общата форма на дефиниране на променлива е:

Дефиниране на променлива

```
тип    име-променлива;

       или

тип    име-променлива1, име-променлива2,... име-променливаN;

където

тип
       Може да бъде от скаларен тип, агрегатен тип или обединение.
```

Втората форма позволява няколко променливи от един и същ тип да бъдат дефинирани наведнъж.

Дефинирането на променлива води до заделяне на памет за нея. Типът на една променлива носи следната информация:

- видът на данните, които тази променлива може да съхранява (цели числа, символи и т.н), а от тук и размера памет, необходим за заделяне за тази променлива.

- видовете операции, които могат да се прилагат върху тази променлива (допустимите операциите върху променлива от един тип може да не са позволени за променлива от друг тип).

- разполагане на променливата в паметта или подравняване в паметта (**memory alignment**). Тази характеристика не е изискване на езика C, а на конкретната хардуерна платформа. Различните хардуерни платформи имат различни изисквания как данните да се разполагат в паметта. Всяка имплементация на C компилатора, трябва да се съобразява с това изискване.

Пример:

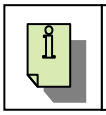
```
/* Дефиниране на променливи от базови типове данни */

signed              si;
```

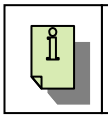
```
unsigned          un;
long              li;
unsigned long     ul;
short             sh;
unsigned short    ush;
float             f1, f2;
double            d1, d2, d3;
```

За задаване на име на променлива се използва идентификатор (вижте Глава 1 Лексикални елементи).

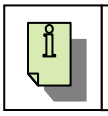
Най-общо казано променливите могат да се дефинират на две места: извън всички функции или в тялото на функция. Променливите дефинирани извън всички функции се наричат глобални. Те са достъпни за всички функции в програмата. Променливите обявени вътре във функция са локални и могат да се използват само от тази функция.



В С дефиницията на променлива е оператор, който води до заделяне на памет за променливата и затова завършва с точка и запетая.



Променливите обявени вътре във функция се наричат локални и могат да се използват само от тази функция.



Променливите дефинирани извън всички функции се наричат глобални и могат да се използват от всички функции.



Винаги задавайте смислени имена на променливите. Това подобрява разбираемостта на програмата и самодокументирането на кода.

1.4 Описание на базовите типове данни

1.4.1 Видове базови типове данни

Езикът С дефинира няколко базови (вградени в самия език) типа данни (Табл.6).

тип	описание	пример
char	Използва се за дефиниране на променливи, които могат да съхраняват символи. Символите в С се затварят в единични кавички. Тип char заема 8 бита, и въпреки че се използва за съхраняване на символи, може да се използва и за съхраняване на цели 8 битови числа.	'z','s','#', 10, 38,127
int	Използва се за дефиниране на променливи, които могат да съхраняват цели числа.	2330, -1325, 1000

float	Използва се за дефиниране на променливи, които могат да съхраняват дробни числа.	2.5, -0.123
double	Същото като float с тази разлика, че може да съхранява по-големи числа и почти двойно по-голям брой значещи цифри.	125464.986537
void	Тип без стойност. Ключовата дума void има по-особено предназначение. Например не може да дефинирате променлива от този тип както с останалите базови типове. Ключовата дума void се използва за : ▪ Дефиниране на функции, които не връщат стойност ▪ Дефиниране на функции без параметри ▪ Дефиниране на обобщени указатели	void foo(void); void *p;

Табл. 6 Базови типове данни

1.4.2 Модификатори на базовите типове

Базовите типове могат да се модифицират с помощта на т.нар модификатори на типа (Табл.7)

модификатор	описание	пример
unsigned	Може да се прилага към базовите типове char и int, за да укаже че тези типове могат да приемат само неотицателни стойности.	unsigned char unsigned int
signed	Може да се прилага към базовите типове char и int за да укаже, че тези типове могат да приемат знакови числа, т.е и положителни и отрицателни.	signed char signed int
long	Може да се прилага към базовите типове double и int, за да увеличи големината на стойностите, които тези типове могат да съхраняват. Този модификатор може да се използва в комбинация с модификаторите unsigned и signed към тип int.	long int unsigned long int signed long int long double
short	Може да се прилага към базовия тип int, за да намали големината на стойностите, които тези типове могат да съхраняват. Този модификатор може да се използва в комбинация с модификаторите unsigned и signed към тип int.	short int unsigned short int signed short int

Табл. 7 Модификатори на базовите типове

Не е задължително да указвате всички тези ключови думи когато дефинирате променливи. Следните конструкции са еквивалентни:

short ≈ **signed short** ≈ **short int** ≈ **signed short int**
unsigned short ≈ **unsigned short int**

signed ≈ **signed int** ≈ **int**
unsigned ≈ **unsigned int**
long ≈ **long int** ≈ **signed long** ≈ **signed long int**
unsigned long ≈ **unsigned long int**

Размерът (обемът заемана памет) и диапазонът от стойности на базовите типове данни е различен за различните процесори и компилатори. Стандартът С дефинира само минималните размери и стойности (Табл.8).

тип	тинимален размер в битове	числов диапазон	коментар
char	8	-127 ÷ +127 или 0 ÷ 255	Стандартът не дефинира дали по подразбиране тип char е знаков или безнаков. Това зависи от имплементацията на компилатора.
signed char	8	-127 ÷ +127	
unsigned char	8	0 ÷ 255	
int[1]	16	-32767 ÷ +32767	По-малък или равен на long int
unsigned int	16	0 ÷ 65535	
signed int	16	-32767 ÷ +32767	
short int	16	-32767 ÷ +32767	По-малък или равен на int
unsigned short int	16	0 ÷ 65535	
signed short int	16	-32767 ÷ +32767	
long int	32	-2 147 483 647 ÷ +2 147 483 647	По-голям или равен на int
unsigned long int	32	0 ÷ +4 294 967 295	
signed long int	32	-2 147 483 647 ÷ +2 147 483 647	
float	32	±1E-37 ÷ ±1E+37	По-малък или равен на double
double	32	±1E-37 ÷ ±1E+37	По-малък или равен на long double
long double	32	±1E-37 ÷ ±1E+37	По-голям или равен на double

Табл. 8 Минимални размери на базовите типове съгласно стандарта С

Забележка[1]: Размерът на тип int типично съвпада с размера на машинната дума на компютъра, т.е за 16-битов компютър тип int е 16 бита, за 32-битов компютър тип int е 32 бита и т.н.

1.4.3 Вътрешно представяне на базовите типове

Стойностите на беззнаковите целочислени типове и неотрицателните стойности на знаковите целочислени типове използват стандартно двоично кодиране. Стандартът С не дефинира метода на кодиране на отрицателните стойности на знаковите целочислени типове. Повечето С компилатори използват код с допълнение до две (**2's complement**).

Стандартът С също така не дефинира и метода на кодиране на числата с плаваща запетая. Повечето С компилатори използват стандарта **IEEE 754-1985** и неговия наследник **IEEE 754-2008**.

1.5 Квалификатори на типа

С89 въвежда така наречените квалификатори на типа – **const** и **volatile**. Тези квалификатори могат да се прилагат върху всички типове с изключение на функциите.

1.5.1 Квалификатор const

Променливите обявени като const не могат да променят своята стойност в програмата. Затова те трябва да се инициализират, т.е. да им се присвои стойност по време на тяхното дефиниране.

Квалификатор const

```
const   тип   име-променлива = инициализатор;
```

където

инициализатор
Формата на инициализаторите зависи от типа на променливата. Правилата за инициализаторите са разгледани в точка 9 Инициализация на променливите.

Пример 2-1:

```
#include <stdio.h>

int main(void)
{
              int  x = 10;
      const   int  y = x;       /* инициализаторът е друга променлива */
      const   int  z = 10;      /* инициализаторът е константа        */

       printf("y = %d\n", y);
       printf("z = %d\n", z);
       /* y = 20; */            /* Грешка, y не може да се променя */
```

```
        /* z = 30; */           /* Грешка, z не може да се променя */

        return 0;
}
```

Всеки опит в програмния код да се промени една const променлива ще доведе до генериране на грешка при компилация.

Компилаторите са свободни да поставят const променливите в ROM (Read Only Memory) паметта на компютъра. Важно е да се запомни, че квалификаторът const гарантира само, че една променлива не може да се промени в кода с помощта на програмни инструкции, като присвояване на нова стойност и т.н.(вижте примера по-горе), но const променливите могат да се променят от хардуера. Например една променлива може да е дефинирана като const и в същото време е назначена на някое периферно устройство, което при приемане на данни автоматично ги съхранява в нея. Тази променлива може само да се чете от програмата, но може да се изменя само от хардуера.

1.5.2 Квалификатор volatile

Квалификаторът volatile информира компилатора, че променливата може да се измени неявно, т.е. без намесата на програмни инструкции, например от хардуера.

Квалификатор volatile

```
volatile    тип    име-променлива;
```

Такава променлива не участва в оптимизацията на кода. Например ако между две четения на една променлива, в кода липсват програмни инструкции, които да я модифицират, компилаторът може да оптимизира кода, така че след като е прочел първия път стойността на променливата да допусне, че тя не е променена и да използва същата стойност. Ако обаче тази променлива се промени междувременно от хардуера, това ще доведе до логическа грешка в програмата. Ако има вероятност една променлива да бъде променена извън програмата, то тази променлива трябва да се дефинира като volatile. Тогава компилаторът ще изключи оптимизацията й.

Квалификаторите const и volatile могат да се използват в комбинация. Редът на квалификаторите е без значение.

Пример: `const volatile int x = 10;` или `volatile const int x = 10;`



Ако има вероятност една променлива да бъде променена извън програмата, направете я volatile. Това ще изключи оптимизацията й.

1.6 Област на видимост на променливите

Областта на видимост на една променлива определя тази част от програмния код, в която променливата е видима и може да се използва. Областта на видимост на променлива се определя от мястото където тя е дефинирана. В С има 4 области на видимост:

- област на видимост в прототип на функция
- област на видимост във функция
- област на видимост в блок с код
- област на видимост във файл

1.6.1 Област на видимост в прототип на функция

Общия вид на прототип на функция е:

Прототип на функция

```
тип име-функция(списък-параметри);
```

Ако списък-параметри съдържа декларации на параметри, то тези параметри са видими само в прототипа.

Пример:
```
void  func(int x, double y);
```

Параметрите x и y са видими само в прототипа.

В действителност компилаторът игнорира идентификаторите използвани за задаване на имена на параметрите в прототипа на функция. Затова те могат да се изпуснат.

Пример:
```
void  func(int , double);
```

1.6.2 Област на видимост във функция

Променливи дефинирани във функция са видими само в нея и не са достъпни за други функции. С89 изисква дефинирането на променливи да става в началото функцията преди всички други изпълними оператори. Такива променливи се наричат още локални променливи. Параметрите на функциите също се явяват локални променливи. Тяхната област на

видимост също е тялото на функцията.

Пример 2-2:

```
#include <stdio.h>

int sum(int x, int y);

int main(void)
{
  int i;

  i = sum(1,2);

  printf("i = %d\n", i);

  /* result = 10; */      /* !!!Грешка, result  не е видима извън функцията sum */

  return 0;
}

int sum(int x, int y)
{
    int result;

    result = x + y;

    return result;
}
```

Област на видимост на i.

Област на видимост на result, x и y.



Променливата result е локална за функцията sum и не може да се използва извън нея. Параметрите x и y също са локални променливи за функцията sum и не са видими извън нея.

Тъй като локалните променливи и параметри са видими само във функциите, в които са дефинирани, то едни и същи имена на променливи или параметри могат да се използват за локални променливи или параметри, дефинирани в различни функции.

Пример 2-3:

```
#include <stdio.h>

int sum(int x, int y);
int sub(int x, int y);

int main(void)
{
        int i;

        i = sum(1,2);
        printf("1 + 2 = %d\n", i);
```

```
        i = sub(1,2);
        printf("1 - 2 = %d\n", i);

        return 0;
}

int sum(int x, int y)
{
    int result;

    result = x + y;

    return result;
}

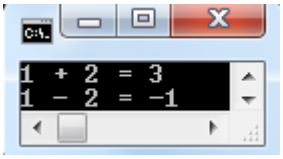
int sub(int x, int y)
{
    int result;

    result = x - y;

    return result;
}
```

Област на видимост на result, x и y от sum.

Област на видимост на result, x и y от sub.



Името на параметър на функция и локална променлива, дефинирана в нея, не могат да съвпадат. Компилаторът няма как да направи разлика между тези две променливи в този случай.

Пример:

```
void foo(int x)
{
      double x; /* !!!Грешка, параметърът също се казва x */

      /*...*/
}
```

1.6.3 Област на видимост в блок с код

Блок с код или още съставен оператор се нарича съвкупността от оператори затворена между отваряща и затваряща фигурна скоба. C89 изисква дефинирането на променливи да става в началото на блок с код преди всички други изпълними оператори.

Блок с код

```
{
    /* Дефиниране на променливи */
    тип име-променлива1;
    тип име-променлива2;
    ........
    тип име-променливаN;

    /* Изпълними оператори */
    оператор1;
    оператор2;
    ........
    операторN;
}
```

Променливите дефинирани в блока са видими само в него. Извън блока те са недостъпни. Всеки опит за достъп до тях извън блока ще генерира грешка при компилиране.

Тялото на функция в частност също е блок с код. Блок със код може да бъде част от управляващите оператори if, for, while и do-while.

Пример 2-4:

```
#include <stdio.h>

int main(void)
{
    int i = 10;

    if(10 == i)
    {
        int x = 20; /* x е локална променлива за if */

        printf("x = %d\n", x);
    }

    /* printf("x = %d\n", x); */ /* Грешка, x не е достъпна извън блока на if */

    return 0;
}
```

Област на видимост на i

Област на видимост на x



Блок с код може да съществува и самостоятелно, но само в тялото на функция.

Пример 2-5:

```
#include <stdio.h>

int main(void)
```

```
{
    int i = 10;

    {/* Блок 1 */
       int x = 20;      /* x е локална за Блок 1 */

       printf("block1:x = %d\n", x);
       printf("main:i   = %d\n", i); /* i е достъпна в Блок1 */
    }/* Край на Блок 1 */

    {/* Блок 2 */
       int x = 30;      /* x е локална за Блок 2 */

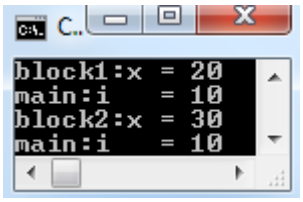
       printf("block2:x = %d\n", x);
       printf("main:  i = %d\n", i); /* i е достъпна в Блок2 */
    }/* Край на Блок 2 */

    /* printf("x = %d\n", x); */ /* Грешка, x не е видима тук */
    return 0;
}
```

Област на видимост на x от Блок 1

Област на видимост на x от Блок 2

Област на видимост на i



Променливата i е дефинирана в началото на main() и е достъпна в цялото тяло на функцията включително и в тялото на Блок 1 и Блок 2. Локалните променливи дефинирани в тези блокове обаче са достъпни само в тях.

Блоковете могат да бъдат вложени един спрямо друг. В горния пример Блок 1 и 2 се явяват вложени по отношение на main().



Въпреки, че локалните променливи могат да се дефинират в началото на всеки блок с код, за предпочитане е дефинициите на всички локални променливи да се разполагат в началото на функцията.

Имената на локални променливи дефинирани във външния блок могат да съвпадат с имената на локалните променливи дефинирани във вложения блок. В този случай променливите на вложения блок "скриват" променливите на външния блок в рамките на вложения блок.

Пример 2-6:

```
#include <stdio.h>

int main(void)
{
    int i = 10;  /* i е локална за main() */

    printf("main:i = %d\n", i);

    {/* Блок 1 */
          int i = 20;  /* i е локална за блок 1 */
          printf("block1:i = %d\n", i);
```

main: i

Блок1: i

```
        {/* Блок 2 */
            int i = 30;  /* i е локална за блок 2 */

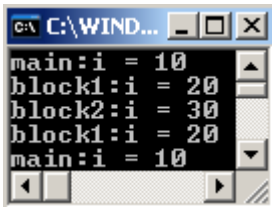
            printf("block2:i = %d\n", i);
        }/* Край на Блок 2 */

        printf("block1:i = %d\n", i);
    }/* Край на Блок 1 */

    printf("main:i = %d\n", i);

    return 0;
}
```

Блок2: i



Променливата i от Блок 1 скрива променливта i от main(). Променливата i от Блок 2 скрива както i от Блок 1 така и i от main().

	**Избягвайте използването на еднакви имена на локални променливи във вложен и обхващащия го блок. Това води до объркване влошава качеството на кода.**

1.6.4 Област на видимост във файл

Променливи дефинирани извън всички функции имат област на видимост от точката на дефиниране до края на файла. Такива променливи се наричат още глобални променливи. Глобалните променливи са достъпни за всички функции, които се намират след дефиницията и. Най-често глобалните променливи се дефинират в началото на файла преди дефинирането на всички функции. Тогава те са достъпни за всички функции във файла.

Пример 2-7:

```
#include <stdio.h>

int i; /* Дефиниране на глобална променлива */

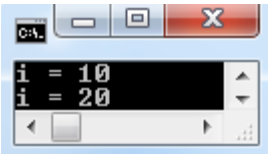
void func(void);

int main(void)
{
      i = 10; /* i може да се използва в main() */

      printf("i = %d\n", i);
      func();
      printf("i = %d\n", i);
      return 0;
```

```
}/* Край на main() */

void func(void)
{
	i = 20; /* i може да се използва в func() */
}
```



Както се вижда от примера променливата i е дефинирана преди всички функции и е достъпна както за main() така и за func().

	Ако не е абсолютно наложително, избягвайте използването на глобални променливи. Тъй като глобалните променливи могат да се променят от всички функции в програмата, това би могло да доведе до трудно откриваеми логически грешки.

Името на глобална променлива може да съвпада с името на локална променлива. Локалната променлива "скрива" глобалната променлива в рамките на функцията или блока, в който е дефинирана.

Пример 2-8:

```
#include <stdio.h>

int i; /* Дефиниране на глбална променлива */

void func(void);

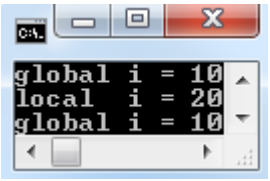
int main(void)
{
	i = 10; /* използва се глобалната променлива i */

	printf("global i = %d\n", i);
	func();
	printf("global i = %d\n", i);

	return 0;
}/* Край на main() */

void func(void)
{
	int i = 20; /* Скрива глобална променлива със същото име */

	printf("local  i = %d\n", i);
}
```



В func() се използва локалната променлива i, а не глобалната.



Избягвайте използването на еднакви имена на локална и глобална променлива. Това води до объркване и влошава качеството на кода.

1.7 Време на живот на променливите

Времето на живот на променливите представлява времето, през което една променлива се съхранява в паметта на компютъра. Глобалните променливи се създават при стартиране на програмата и се съхраняват докато трае изпълнението ѝ. След завършване на програмата паметта, заделена за глобалните променливи, се освобождава. Локалните променливи се съхраняват само докато се изпълнява блока с код, в който са дефинирани. Паметта за локалните променливи се заделя при влизане в блока, в който са дефинирани и се освобождава при излизане от него. Времето на живот на локалните променливи може да се променя с помощта на т.нар. спецификатор за памет static (вижте точка 8.3.1 Локални статични променливи).

1.8 Клас памет на променливите

С дефинира 5 спецификатора за клас памет на променливите:

- auto
- register
- static
- extern
- typedef[1]

Забележка[1]: Спецификаторът typedef не е в действителност спецификатор за клас памет. Той се нарича така само от синтактична гледна точка, тъй като само една от горните ключови думи може да присъства в декларацията на една променлива.

Клас памет на променлива

```
клас-памет   тип   име-променлива;
```

Само един спецификатор за клас-памет може да се използва при дефинирането на променлива.

Класът памет на една променлива определя мястото, където променливата ще бъде съхранена в паметта, областта на видимост и времето и на живот.

1.8.1 Клас памет auto

Клас памет auto може да се прилага само към локални променливи. По подразбиране локалните променливи имат клас на съхранение auto. По тази причина ключовата дума auto е излишна и рядко се използва. Локалните променливи клас auto се наричат още автоматични променливи. Тези променливи се съхраняват в област от паметта, наречена стек. Когато програмата влезе в локална област на действие каквато е тялото на функция и блок с код, компилаторът заделя памет за тях в стека. При излизане от функцията (или блока с код) паметта се освобождава и може да се използва от компилатора за заделяне на памет за други автоматични локални променливи.

Пример 02-9:

```
#include <stdio.h>

int main(void)
{
	auto int x = 10;	/* Същото като int x = 10; */

	printf("x = %d\n", x);

	return 0;
}
```

1.8.2 Клас памет register

Клас памет register може да се прилага само към локални променливи и параметри на функции. Когато локална променлива или параметър на функция се дефинира като register това е заявка за компилатора да задели памет за тази променлива така, че достъпът до нея да бъде максимално бърз, например във вътрешните регистри на процесора вместо в RAM паметта. Ако компилаторът не може да удовлетвори тази заявка, променливата се съхранява като обикновена auto-променлива. За разлика от auto-променливите обаче адресът на register-променливите не може да се получи с операцията за взимане на адрес &.

Пример 2-10:

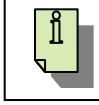
```c
#include <stdio.h>

int main(void)
{
	register int x;

	int *p;

	for(x = 0; x <= 10 ; x ++)
	{
		printf("x = %d\n", x);
	}

	/* p = &x; */ /* Грешка, адреса на register променлива е недостъпен за
			програмата */
	return 0;
}
```



Операцията за извличане на адрес & не може да се прилага към register локални променливи.



Ключовата дума register на практика може да не се използва. Съвременните компилатори са достатъчно интелигентни сами да преценят дали да оптимизират достъпа до дадена променлива или не.

1.8.3 Клас памет static

Клас памет static може да се прилага към глобални и локални променливи (без параметри на функции). Статичните променливи (глобални и локални) се съхраняват извън стека и съществуват през цялото време на изпълнение на програмата. Те се създават при стартиране на програма. Ако дефиницията на променливата е съпроводена и с инициализатор, то стойността на инициализатора се присвоява на променливата също при стартирането на програмата.

1.8.3.1 Локални статични променливи

Когато една локална променлива се дефинира като static се променя времето й на живот. Паметта за съхранение на статичните локални променливи се запазва през изпълнението на цялата програма. Въпреки това областта на видимост си остава блока, в който е дефинирана тази променлива. Това означава, че при излизане от блока променливата не е достъпна, но стойността й се запазва.

Пример 2-11:

```c
#include <stdio.h>
```

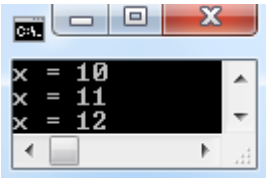
```c
void func(void);

int main(void)
{
	func();
	func();
	func();
	/* x++; */ /* Грешка, x не е достъпна извън func() */

	return 0;
}

void func(void)
{
	static int x = 10;

	printf("x = %d\n", x);
	x++;
}
```



Примерът показва две неща. Първо статичните локални променливи се инициализират само веднъж при стартирането на програмата. Второ след излизане от функцията променливата запазва своята стойност. При следващото влизане във функцията се използва последната съхранена стойност на статичната локална променлива.

1.8.3.2 Глобални статични променливи

Глобалните статични променливи са достъпни само във файла, в който са дефинирани.

Пример 2-12:

```c
/* сорс-файл1 */
#include <stdio.h>

extern void func(void);

static int x;

int main(void)
{
    x = 10;

    func();

    return 0;
}
```

```c
/* сорс-файл2 */
void func(void)
{
    x = 20; /* x не е достъпна тук */
}
```

Примерът показва, че ако една глобална променлива е дефинирана като статична, тя е достъпна само във файла където се намира дефиницията й. Всеки опит за достъп до тази променлива в други файлове води до генериране на грешка при компилиране.

Имената на статични глобални променливи дефинирани в различни файлове могат да съвпадат. В този случай всяка променлива е достъпна само във файла, в който е дефинирана и няма нищо общо с другата променлива.

Пример 2-13:

```c
/* сорс-файл1 */
#include <stdio.h>

static int x;

extern void func(void);

int main(void)
{
      x = 10;

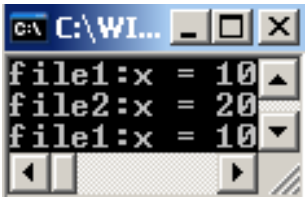
      printf("file1:x = %d\n", x);
      func();
      printf("file1:x = %d\n", x);

      return 0;
}
```

```c
/* сорс-файл2 */
#include <stdio.h>

static int x;

void func(void)
{
   x = 20;
   printf("file2:x = %d\n", x);
}
```



Променливата x от сорс-файл1 няма нищо общо с променливата x в сорс-файл2. Двете променливи са напълно отделни и са достъпни само във файла, в който са дефинирани.

1.8.4 Клас памет extern

Глобалните променливи без никакъв спецификатор за клас-памет са достъпни за всички файлове на програмата съгласно следните правила. Глобалната променлива има област на видимост от точката на дефинирането си до края на файла. Ако тази променлива се използва преди точката на дефинирането си или от друг файл, тя трябва да се декларира. Декларацията изглежда като дефиницията само, че се добавя ключовата дума extern. Декларацията не води до заделяне на памет за променливата, а е просто указание за компилатора, че тази променлива е вече дефинирана на друго място в програмата. Когато компилаторът срещне такива декларации в различни части от програмата (включително в различни файлове) той разбира, че те се отнасят за една и съща променлива (всяка употреба на променливи със същото име в различни файлове се свързва с

една и съща променлива). Ето защо глобалните нестатични променливи се наричат още външни променливи. Външните променливи се съхраняват извън стека и съществуват през цялото време на изпълнение на програмата. Те се създават при стартиране на програма. Ако дефиницията на променливата е съпроводена и с инициализатор, то стойността на инициализатора се присвоява на променливата също при стартирането на програмата.

Пример 2-14:

```
/* сорс-файл1 */
#include <stdio.h>

extern void func(void);

int x; /* Точка на дефиниране на
       глобалната променлива x */

int main(void)
{
       x = 10;
       func();
       printf("x = %d\n", x);

       return 0;
}
```

```
/* сорс-файл2 */
#include <stdio.h>

extern int x; /* Глобална декларация на
                 глобалната променлива x */

void func(void)
{
  printf("x = %d\n", x); /* x е достъпна тук */
  x = 20;
}
```



За да може сорс-файл2 да използва променливата x, дефинирана в сорс-файл1, е необходимо преди това тя да бъде декларирана в този файл. Декларацията може да се направи глобално, в началото на файла, тогава променливата x ще бъде достъпна в целия сорс-файл2 (вижте предния пример), или локално в блока, в който тази променлива се използва, както е показано в следващия пример.

Пример 2-15:

```
/* сорс-файл1 */
#include <stdio.h>

int x; /* Точка на дефиниране на
       глобалната променлива x */
void func(void);

int main(void)
{
    x = 10;
    func();
    printf("x = %d\n", x);

    return 0;
}
```

```
/* сорс-файл2 */
#include <stdio.h>

void func(void);
void func1(void);

void func(void)
{
   extern int x;/* Локална декларация на
                   глобалната променлива x */
   printf("x = %d\n", x);/* x е достъпна тук */
   x = 20;
}

void func1(void)
{
  /*x = 30;*//* Грешка, x не е достъпна тук */
}
```

Променливата x е достъпна в func(), но не е достъпна в func1(). Причината е, че декларацията й не се вижда извън func().

Пример 2-16:

```
#include <stdio.h>

void func(void);

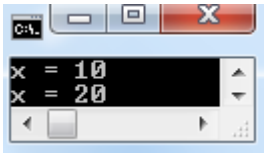
int main(void)
{
	extern int x; /* Декларация на x */

	x = 10;
	func();
	printf("x = %d\n", x);

	return 0;
}

int x; /* Точка на дефиниране на x */

void func(void)
{
	printf("x = %d\n", x);
	x = 20;
}
```



Тъй като x е дефинирана след main(), за да може да се използва от main() е необходимо да се добави декларация на променливата преди мястото, в което променливата се използва (в началото на main или в началото на файла).

	**Въпреки, че глобалните променливи (статични или външни) могат да се дефинират в навсякъде в един файл стига да са извън функция, за предпочитане е дефинициите им да се разполагат в началото на файла преди всички функции.**

	**Ако една глобална променлива се използва само във файла, в който е дефинирана, направете я статична.**

1.8.5 Клас памет typedef

typedef е спецификатор за тип, т.е с негова помощ може да се декларира синоним на вече съществуващ тип. За да разберете как работи този спецификатор нека Ви припомним как изглежда обобщената дефиницията

на променлива:

```
T  име-променлива;
```

където

T може е всеки един обектен тип в C

Ако пред дефиницията поставим спецификатора за клас памет typedef, т.е

```
typedef  T  име-променлива;
```

дефиницията на променливата се превръща в декларация на тип, т.е компилаторът не заделя памет, а е информиран, че **име-променлива** вече е синоним на типа T, който би имала променливата ако го нямаше спецификатора typedef.

Декларация на typedef име

```
typedef  T  ново-име-на-тип;
```

където

ново-име-на-тип
Синоним на типа **T**.

T
Може да бъде всеки обектен тип в езика C:

- скаларен тип (аритметични типове и указатели)
- агрегатен тип (масиви и структури)
- обединение

Пример 2-17:

```
#include <stdio.h>

typedef unsigned char  uint8; /* uint8 става синоним на типа unsigned char */

int main( void )
{
      uint8  x = 10; /* същото като unsigned char x */

      printf("x = %d\n", x);

      return 0;
}
```

Идентификаторът uint8 става синоним на типа unsigned char.

Показаният пример демонстрира една от най-широко използваните употреби на typedef. Тъй като размера на базовите типове данни не е фиксиран от стандарта C, то typedef може да се използва да се дефинират

унифицирани базови типове. Например:

```
typedef unsigned char    uint8;
typedef signed char      sint8;

typedef unsigned short   uint16;
typedef signed short     sint16;

typedef unsigned long    uint32;
typedef signed long      sint32;

typedef float            float32;
typedef double           float64;
```

Тези дефиниции на типове могат да се сложат в хедър файл, например с име stdtypes.h и да се използват унифицираните имена на базовите типовете вместо тези указани от стандарта.



За предпочитане е да използвате унифицирани имена типове вместо тези указани в стандарта. Декларациите на тези унифицирани имена е най-добре да се поставят в хедър-файл, който да се включва във всеки сорс-файл. Това подобрява самодокументирането на кода и повишава преносимостта на програмата.



Създаването на синоним на вече съществуващ тип може да се обобщи по следния начин: Ако пред дефиницията на променлива от кой да е тип се постави typedef, дефиницията се превръща в декларация на typedef име. Идентификаторът използван за име на променливата става синоним на типа (typedef име), който би имала тази променлива ако го нямаше спецификатора typedef.

Другата широка употреба на **typedef** е да свежда сложни дефиниции на променливи до обобщения вид показан по-горе. По-долу са показани обобщени декларации на typedef имена за различни типове.

- `typedef struct име-структура`$_{опц}$`{списък-членове}име-тип;`

Пример:

```
typedef struct {int x; int y}Coord;
Coord coordinates; /* Дефиниране на променлива тип Coord */
```

- `typedef union име-обединеие`$_{опц}$`{списък-членове}име-тип;`

Пример:

```
typedef union {char symbols[100]; int  integers[100/sizeof(int)];} Buffer100Bytes;
Buffer100Bytes  message; /* Дефиниране на променлива тип Buffer100Bytes */
```

- `typedef enum име-изброяване`$_{опц}$`{списък-константи}име-тип;`

Пример:

```
typedef enum {R, G, B} Rgb;

Rgb color; /* Дефиниране на променлива тип Rgb */
```

- `typedef тип  име-тип[размер];`

Пример:

```
typedef uint32 u32Array100elements[100];
u32Array100elements u32array; /* Дефиниране на променлива тип u32Array100elements
*/
```

- `typedef тип-резултат (*име-тип)(списък-параметри);`

Пример:

```
typedef int (*PointerType)(int x, int y);
PointerType  p; /* Дефиниране на променлива тип PointerType */
```

Ключовата дума typedef може да се поставя и пред декларацията на функция. Името на функцията се превръща в синоним на типа на функцията.

Пример 2-18:

```
#include <stdio.h>
#include <math.h> /* Съдържа декларацията на библиотечната функцията sqrt() */

typedef double func_t(double);

int main(void)
{
	func_t* pFunc;
	double y;

	pFunc = sqrt;
	y = pFunc(2.0);
	printf( "The square root of 2 is %f.\n", y );

	return 0;
}
```



Дефиницията **`func_t *pFunc`** е еквивалентна на **`double (*pFunc)(double)`**.

1.9 Инициализация на променливите

Всяка променлива може да се инициализира по време на дефинирането си. Общата форма на инициализиране на променлива е:

Инициализиране на променлива

```
тип    име-променлива = инициализатор;

или

тип    име-променлива1 = инициализатор,...,име-променливаN = инициализатор;

където

инициализатор
    Формата на инициализатора зависи от типа на променливата.
```

Следващите точки разглеждат формата на инициализаторите за различните типове.

1.9.1 Инициализация на променливи от базов тип

Инициализаторът на auto- и register- променливите от базов тип може да бъде всеки израз, чийто тип е съвместим с типа на променливата. Ако променливата е статична или външна, инициализаторът трябва да е константен израз.

Ако липсва инициализатор, външните и статичните променливи се инициализират от компилатора в 0 за целочислени типове и 0.0 за типове с плаваща запетая. Локалните променливи (auto и register) не се инициализират автоматично от компилатора и съдържат произволна стойност.

Пример 2-19:

```
       int g_e = 10; /* Инициализаторът трябва да е константен израз */
static int g_s = 10; /* Инициализаторът трябва да е константен израз */

int foo(void) {return 45;}

int main(void)
{
               int    l_a = 10; /* Инициализаторът може да всеки израз */
      register int    l_r = foo() + 2;/* Инициализаторът може да всеки израз */
      static   int    l_s = 10; /* Инициализаторът трябва да е константен израз */

      /*...*/
      return 0;
}
```

Стандартът C позволява инициализаторът да бъдат затворен във фигурни скоби, но това е излишно.

Пример: `int  x = {10};`

1.9.2 Инициализация на указатели

Инициализаторът на auto-указателите може да бъде всеки израз, чийто тип съвпада с типа на указателя.

За статичните и външните указатели, инициализаторът трябва да е константен израз, формиран по един от следните начини:

1) Целочислен константен израз със стойност 0 (или (void*)0).

Пример:
```
double* dp = 0;
```

2) Името на статична или външна функция от тип "функция връщаща тип T" се преобразува в константа от тип "указател към функция връщаща тип T".

Пример:

```
static int foo(void);
static int (*p_foo)(void) = foo;
```

3) Името на статичен или външен масив от тип "масив от тип T" се преобразува в константа от тип "указател към тип T".

Пример:

```
static int arr[10];
static int* p_int = arr;
```

4) Операцията & приложена към името на статична или външна променлива от тип T дава като резултат константа от тип "указател към тип T".

Пример:

```
static int x;
static int* p_int = &x;
```

5) Операцията & приложена към името на статичен или външен масив от тип "масив от тип T", индексирано с константен израз, дава като резултат константа от тип "указател към тип T".

Пример:

```
static int arr[10];
static int* p_int = &arr[5];
```

6) Целочислена константа преобразувана до указателен тип дава като резултат константа от този указателен тип.

Пример: `static int* p_int = (int*)0xABCD;`

7) Низова константа дава като резултат "указател към тип char", когато се използва като инициализатор на указател към тип char.

Пример: `static char* WellcomeMessage = "Hello C programmers.";`

8) Сумата или разликата на всеки указателен константен израз от точки 3) до 7) с целочислен константен израз, дава като резултат указателен константен израз.

Пример:

```
static int a[5];
static int* p_int = &a[0] + 2;
```

Ако липсва инициализатор, външните и статичните указатели се инициализират от компилатора в 0.

Указателите са разгледани подробно в Глава 9 Указатели.

1.9.3 Инициализация на масиви

Инициализаторът на масив има следната форма:

Инициализатор на масив

```
{инициализатор1, инициализатор2,..., инициализаторN}

където

инициализатор1/.../ инициализаторN
      Трябва да са константни изрази, чийто тип е съвместим с типа на елементите
       на масива.
```

Пример: `int arr[] = {1, 2, 3};`

Обърнете внимание, че ако имате масив от указатели, не може да го инициализирате с адресите на auto-променливи, тъй като те не се считат за

константи в C89.

Пример:

```
int main(void)
{
	int x, y, z;
	int* a[] = {&x, &y, &z}; /* !!!C89 компилатор ще генерира грешка */

	/*...*/

	return 0;
}
```

Масивите са разгледани подробно в Глава 8 Масиви.

1.9.4 Инициализация на структрурни-променливи

Инициализаторът на структурна-променлива има следната форма:

Инициализатор на структурна-променлива

```
{инициализатор1, инициализатор2,..., инициализаторN}

където

инициализатор1/.../ инициализаторN
	Трябва да са константни изрази, чийто тип е съвместим с типа на членовете
	 на структурата.
```

Пример:

```
struct Cbook
{
	int	quantity ;
	double	price;
	char*	name;
} s = {1000, 50, "ANSI C. Complete reference."};
```

Структутрите са разгледани подробно в Глава 10 Структури.

1.9.5 Инициализация на union-променливи

Инициализаторът на union-променлива има следната форма:

Инициализатор на union-променлива

```
{инициализатор}
```

```
където

инициализатор
      Трябва да е константен израз, чийто тип е съвместим с типа на първия член на
      обединението.
```

Пример: `union myunion { int i; char* p; } u = {4};`

Обединенията са разгледани подробно в Глава 11 Обединения.

1.9.6 Инициализация на enum-променливи

Тъй като enum-променливите по същество са променливи от целочислен тип (signed char, unsigned char, short, unsigned short, int или unsigned int), те следват същите правила на инициализация (вижте точка 9.1 инициализация на променливи от базов тип).

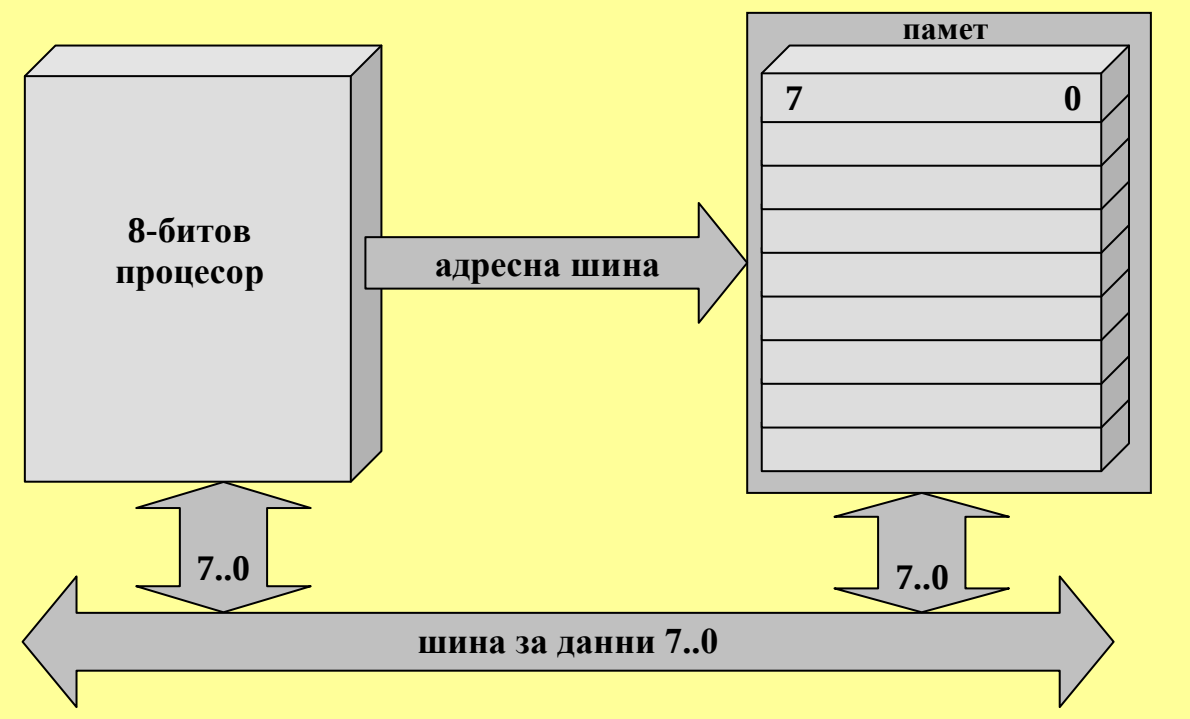
Изброяванията са разгледани подробно в Глава 7 Изброяване.

1.10 Подравняване на променливите в паметта

Подравняването представлява разполагане на променливите на определени адреси в паметта. Кратността на адресите, на които трябва да се поставят променливите се нарича **модул на подравняване**. Променлива с модул на подравняване 1 означава, че тя може да се поставя на всеки адрес. Такива са char-променливите. Общото правило е, че модулът на подравняване на една променлива е равен на нейния размер в байтове, но това не е задължително. Подравняването на променливите в паметта не е изискване на езика C, а на различните компютърни архитектури. Всяка имплементация на компилатора обаче трябва да се съобразява с това изискване.

1.10.1 Подравняване на променливите в 8-битов процесор

Фиг.2 показва в обобщен вид организацията на паметта на 8-битов процесор.



Фиг. 2 Организация на паметта на 8-битов процесор

Процесор с 8-битова шина за данни може да трансферира по 8 бита (1 байт) с една операция. В този случай се казва, че големината на една машинна дума е 8 бита. За трансфер на многобайтови данни като тип short, int, long и т.н. са необходми толкова на брой операции, колкото е размера в байтове на данните. 8-битовите процесори могат да осъществяват достъп до всеки байт, т.е те нямат специални изисквания за подравняване на данните на границата на определени адреси. С други думи казано от гледна точка на хардуера модула на подравняване на еднобайтови и многобайтови данни е 1 (Табл.9).

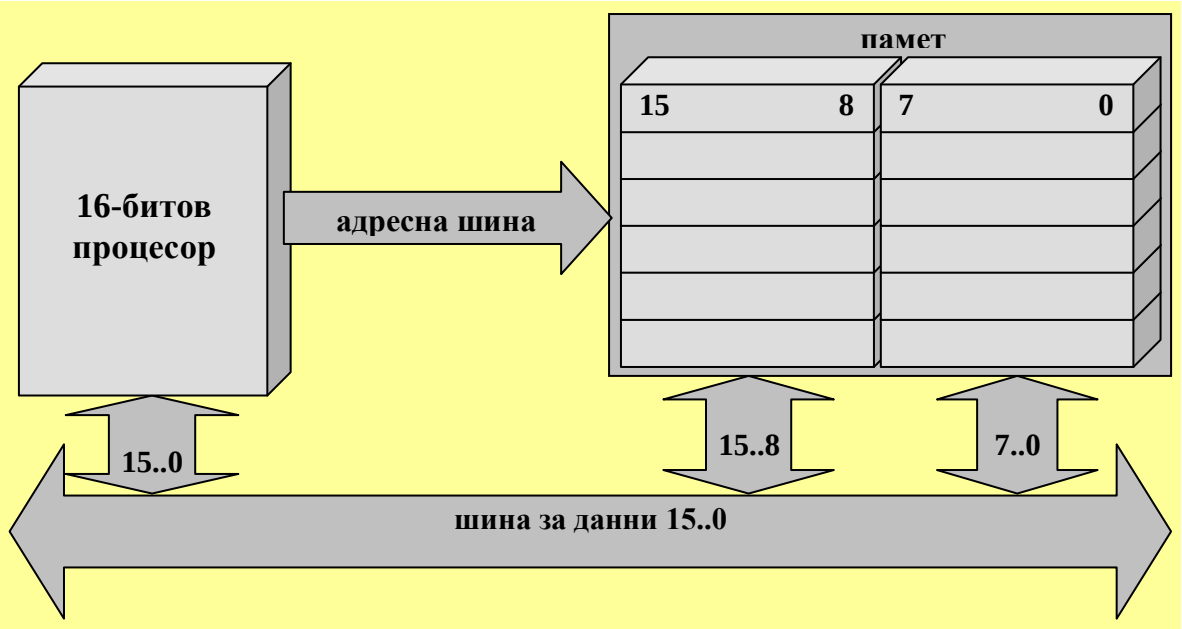
тип	размер в байтове	модул на подравняване
char	1	1
short	2	1
int	2 или 4	1
long	4	1
float	4	1
double	4	1
long double	4 или 8	1
указатели	-	1

Табл. 9 Типични модули на подравняване за 8-битов процесор

Въпреки това компилаторите са свободни да разполагат данните с модул на подравняване различен от 1, въпреки че хардуерът не го изисква.

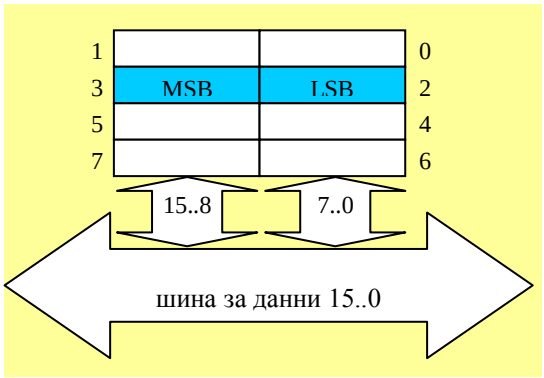
1.10.2 Подравняване на променливите в 16-битов процесор

Процесор с 16-битова шина за данни може да трансферира по 16 бита (2 байта) с една операция, т.е два пъти повече информация при една и съща скорост в сравнение с 8-битов процесор. В този случай се казва, че големината на една машинна дума е 16 бита.

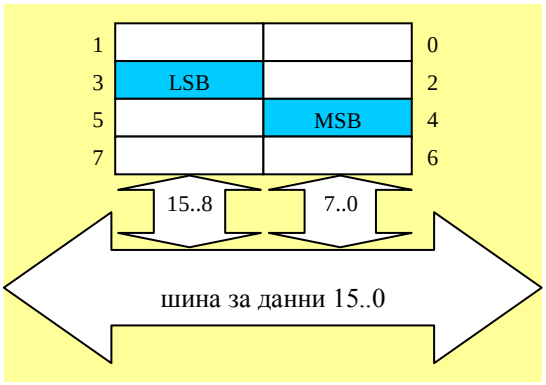


Фиг. 3 Организация на паметта на 16-битов процесор

Фиг.3 показва типична организация на паметта на 16-битовите процесори. Както се вижда паметта е организирана в две банки. Едната банка съдържа само четни адреси и е свързана с младшия байт на шината за данни (D0...D7), а другата само нечетни и е свързана със старшия байт на шината за данни (D8...D15). При такава организация на паметта обаче, трансферът на 16 бита данни с една операция е възможен само ако данните се извличат от адреси кратни на две (Фиг.4).



Фиг. 4 Подравнена 16-битова машинна дума



Фиг. 5 Неподравнена 16-битова машинна дума

Фиг.5 показва неподравнена машинна дума. Младшият байт на думата е разположен на нечетен адрес, а старшия на четен. Това е критична ситуация, която различните процесори обработват по различен начин.

- Процесори, които не са проектирани да осъществяват достъп до неподравнена машинна дума ще генерират изключение при такава ситуация.

- Процесори, които са проектирани да осъществяват достъп до неподравнени думи, извършват това с две операции. Първата операция извлича думата, в която е разположен младшия байт, а с втората операция се извлича думата, в която е разположен старшия байт.

Както се вижда дори и един процесор да може да достъпва неподравнени думи, това става за сметка на производителността. Табл.10 описва типичните модули на подравняване на 16-битови процесори.

тип	размер в байтове	модул на подравняване[1]	модул на подравняване[2]
char	1	1 / 1	1
short	2	1 / 2	2
int	2 или 4	1 / 2	2
long	4	1 / 2	2
float	4	1 / 2	2
double	4 или 8	1 / 2	2
long double	4 или 8	1 / 2	2
указатели	-	1 / 2	2

Табл. 10 Типични модули на подравняване за 16-битови процесори

Забележка[1]: Процесори, които могат да достъпват машинните думи от всякакви адреси. С цел по-добра производителност, компилаторите могат да разполагат променливите на подравнени адреси (колона 2).

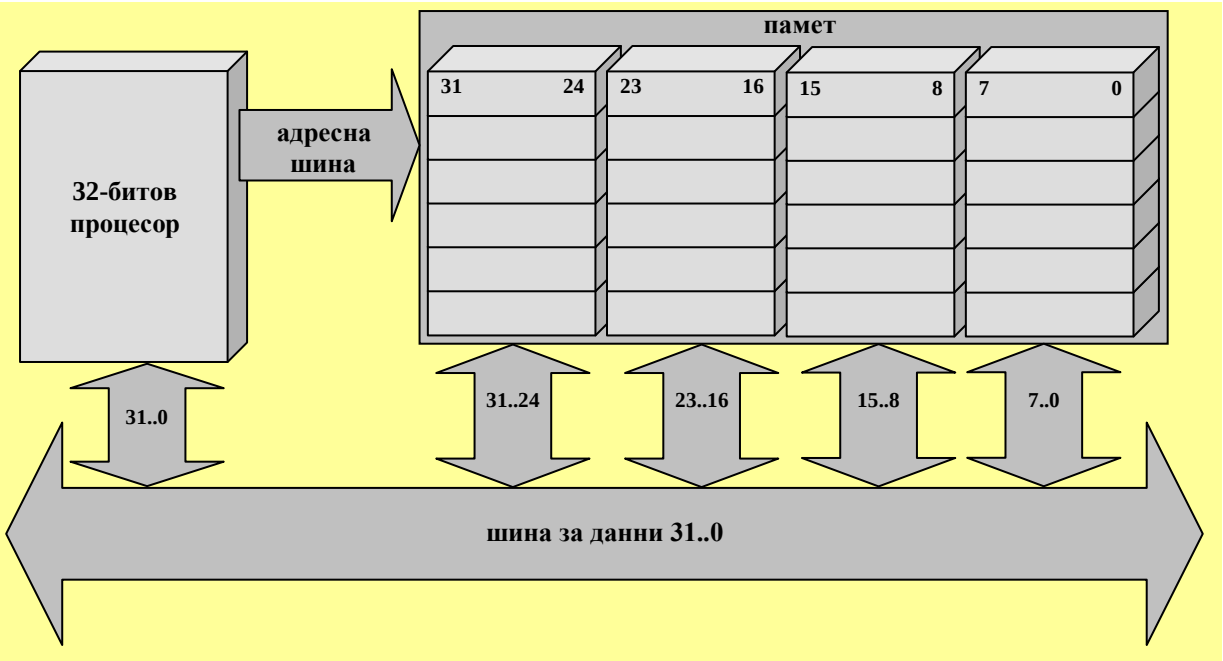
Забележка[2]: Процесори, които могат да достъпват машинните думи само от подравнени адреси.

Независимо от възможностите на процесора, байтовите данни могат да се разполагат на всякакви адреси, т.е за тях модула на подравняване си остава 1.

1.10.3 Подравняване на променливите в 32-битов процесор

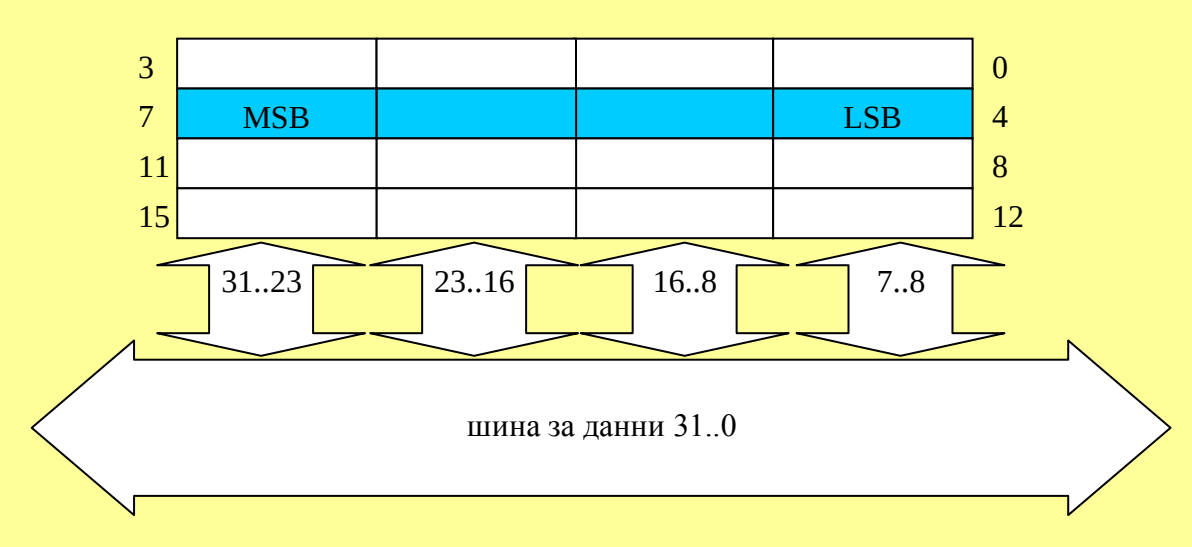
Процесор с 32-битова шина за данни може да трансферира по 32 бита (4 байта) с една операция, т.е два пъти повече информация при една и съща

скорост в сравнение с 16-битов процесор и четири пъти повече информация при една и съща скорост в сравнение с 8-битов процесор . В този случай се казва, че големината на една машинна дума е 32 бита.



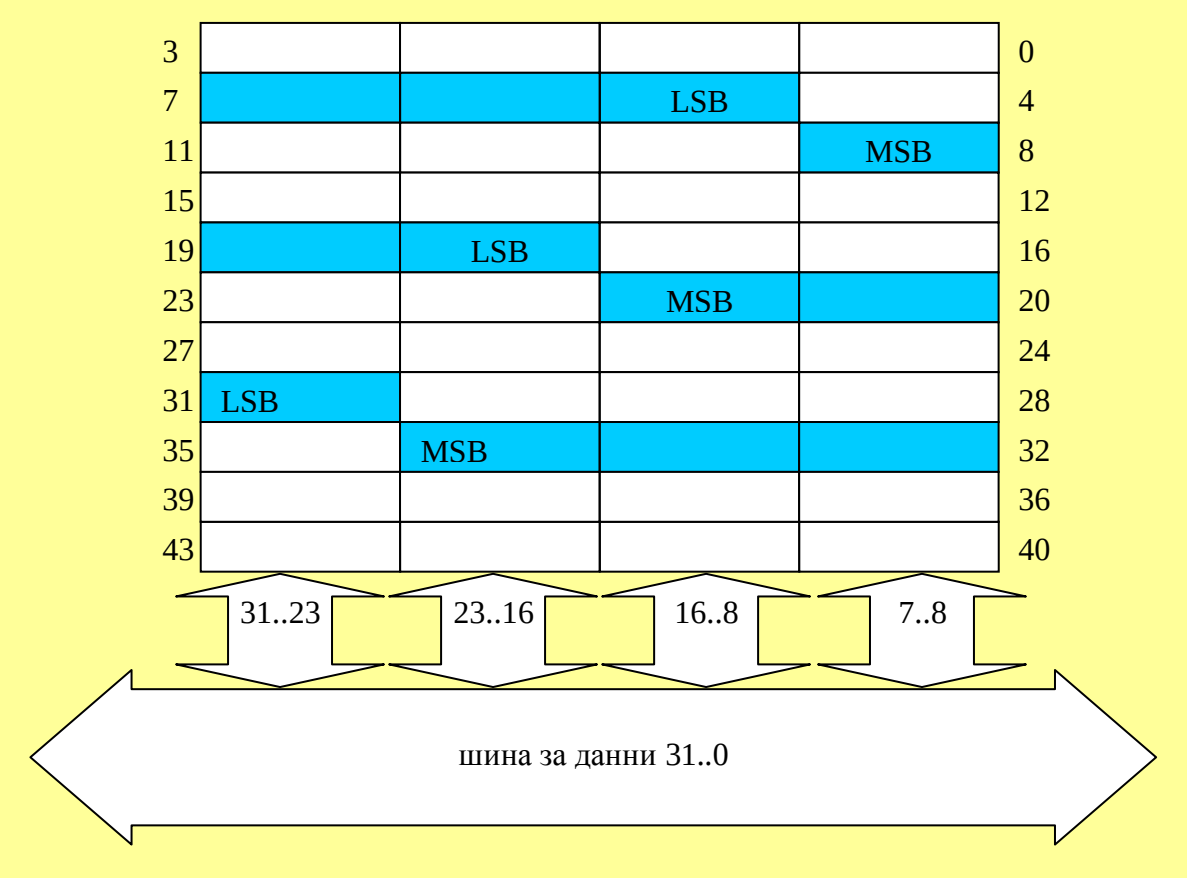
Фиг. 6 Организация на паметта на 32-битов процесор

Фиг.6 показва типична организация на паметта на 32-битовите процесори. Както се вижда паметта е организирана в четири банки. При такава организация на паметта обаче, трансфера на 32 бита данни с една операция е възможен само ако данните се извличат от адреси кратни на 4 (Фиг.7).



Фиг. 7 Подравнена 32-битова машина дума

Фиг.8 показва ситуации на неподравнена машинна дума в 32-битов процесор.



Фиг. 8 Неодравнени 32-битова машини думи

Както и при 16-битовите процесори това е критична ситуация, която различните процесори обработват по различен начин.

- Процесори, които не са проектирани да осъществяват достъп до неподравнена машинна дума ще генерират изключение при такава ситуация.

- Процесори, които са проектирани да осъществяват достъп до неподравнени думи, извършват това с две операции. Първата операция извлича думата, в която са разположени младшите байтове, а с втората операция се извлича думата, в която са разположени старшите байтове.

Както се вижда дори и един процесор да може да достъпва неподравнени думи, това става за сметка на производителността. Табл.11 описва накратко типичните модули на подравняване на 32-битови процесори.

тип	размер в байтове	модул на подравняване[1]	модул на подравняване[2]
char	1	1 / 1	1
short	2	1 / 2	2
int	4	1 / 4	4
long	4	1 / 4	4

float	4	1 / 4	4
double	4 или 8	1 / 4	4
long double	4 или 8	1 / 4	4
указатели	-	1 / 4	4

Табл. 11 Типични модули на подравняване за 32-битови процесори

Забележка[1]: Процесори, които могат да достъпват машинните думи от всякакви адреси. С цел по-добра производителност, компилаторите могат да разполагат променливите на подравнени адреси (колона 2).

Забележка[2]: Процесори, които могат да достъпват машинните думи само от подравнени адреси.

Независимо от възможностите на процесора, байтовите данни могат да се разполагат на всякакви адреси, т.е за тях модула на подравняване си остава 1.

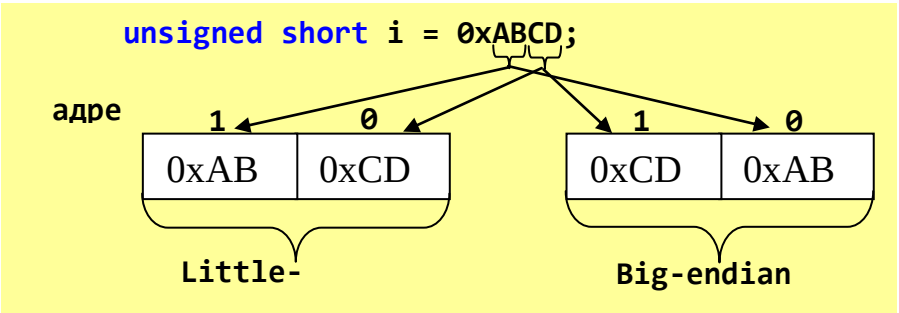
1.11 Big-endian u Little-endian

В зависимост от това как стойността на многобайтовите променливи се разполага в паметта, заделена за тях, различаваме Big-endian и Little-endian компютри.

Little-endian компютрите разполагат стойността на променливата започвайки от най-младшия байт на паметта, заделена за нея.

Big-endian компютрите разполагат стойността на променливата, започвайки от най-старшия байт на паметта, заделена за нея.

Фиг.9 показва нагледно това.



Фиг. 10 Big-endina и Little-endian

Някои компютри позволяват това да се конфигурира.

1.12 Резюме

Пълна обобщена форма на дефиниране на променлива:

```
клас-памет    квалификатор-тип    спецификатор-тип    тип    име-променлива;
```

където

клас-памет може да е:
- `extern`
- `static`
- `auto`
- `register`
- `typedef`

квалификатор-тип може да е:
- `const`
- `volatile`
- `const volatile` **или** `volatile const`

спецификатор-тип се прилага само бързу базови типове и може да е:
- `signed`
- `unsigned`
- `short`
- `long`
- `signed short`
- `signed long`
- `unsigned short`
- `unsigned long`

тип може да е:
- базов тип (**char**, **int**, **float**, **double**)
- указателен тип
- масив
- структура
- обединение

Табл.12 описва накратко категориите променливи.

категория	клас памет
глобални	статични (static)
	външни (extern)
локални	автоматични (auto)
	регистрови (register)
	статични (static)

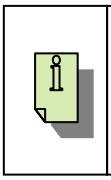
Табл. 12 Свързване на променливите

Табл.13 показава в обощен вид формата и вида на инициализаторите.

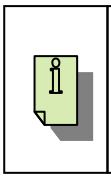
клас памет	тип	инициализатор	подразбиращ се инициализатор
static/ extern	скаларен	константен израз	0 / 0.0

static/ extern	масив	{списък-константни-изрази}	0 / 0.0
static/ extern	структура	{списък-константни-изрази}	0 / 0.0
static/ extern	обединение	константен израз	0 / 0.0
auto/ register	скаларен	всякакъв израз	недефиниран
auto/ register	масив	{списък-константни изрази}	недефиниран
auto/ register	структура	{списък-константни изрази} или struct-променлива от същия тип	недефиниран
auto/ register	обединение	константен израз или union-променлива от същия тип	недефиниран

Табл. 13 Форма на инициализаторите за различните типове



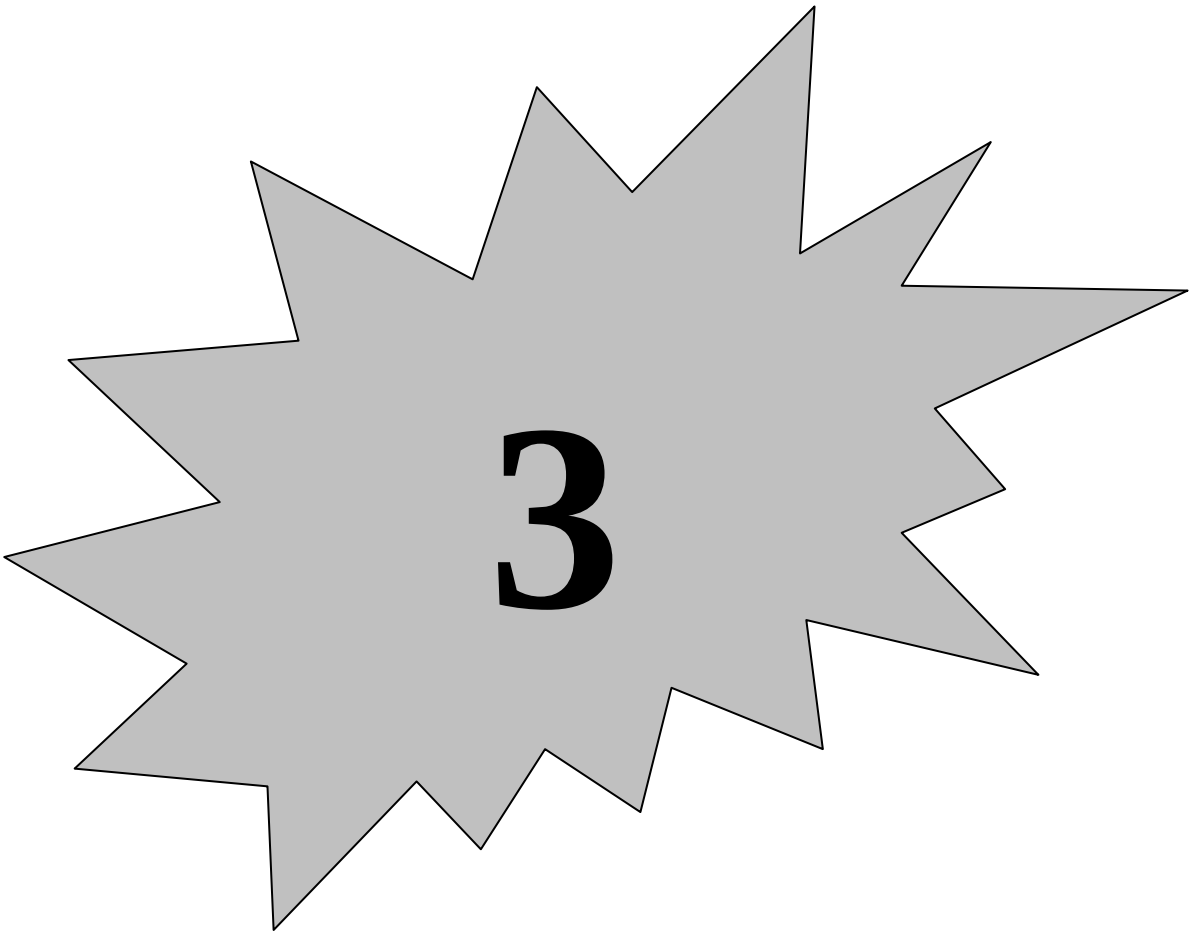
Външните и статичните (глобални и локални) променливи се инициализират при стартирането на програмата. Локалните променливи се инициализират при влизане в блока, в който са дефинирани.



При липса на инициализатор външните и статичните (глобални и локални) променливи се инициализират с 0/0.0 автоматично от компилатора. Локалните променливи НЕ се инициализират автоматично от компилатора и съдържат произволна стойност.



Препоръчително е да не разчитате на компилатора, а винаги явно да инициализирате външните и статичните (глобални и локални) променливи.



Константи

1.1 Видове константи

Константата е фиксирана стойност, която не може да се изменя в програмата. В зависимост от това дали константите заемат памет или не, те могат условно да се разделят на два вида:

- литерални константи (литерали)
- нелитерални константи (нелитерали)

В зависимост от типа константите могат да се разделят на:

- символни константи
- целочислени константи
- константи с плаваща запетая
- низови константи

Те се използват за инициализиране на променливи, в изрази, като аргументи на функции и т.н.

1.2 Литерали и нелитерали

- литерални константи (литерали)

Пример:

```
int     i = 10;
double  d = 12.3;
char    c = 'a';
char*   p = "ANSI C";
```

Константите 10, 12.3, 'a' , "ANSI C" и са пример за литерали. Литералите не заемат памет[1].

Забележка[1]: Низовите константи са изключение от това правило. Компилаторът разполага символите на низа последователно в паметта (вижте точка 6 Низови константи).

- нелитерални константи (нелитерали)

Пример:

```
const  int    i = 10;
const  double d = 12.3;
const  char   c = 'a';
const  char*  p = "ANSI C";
```

Променливите i, d, c и p се инициализират с литерали. Тези променливи обаче се дефинирани с ключовата дума const и не могат да се променят в програмата, т.е те също са константи. За разлика от литералите обаче, тези константи заемат памет и можем да ги наречем константни "променливи". Променливи, защото заемат памет, като всяка променлива, и константни, защото не могат да променят стойността си.

Това разделение на литерали и нелитерали на константите е необходимо тъй като в някой конструкции на езика C могат да се използват само литерални константи, докато в други и двата вида. Например, статичните и външните променливи могат да се инициализират само с литерални константи, размерът на масив се задава само с помощта на литерални целочислени константи и др.. Ако се използват нелитерални константи или променливи в тези случаи, компилаторът ще генерира грешка при компилиране.

Пример 3-1:

```
#include <stdio.h>

const int size = 20;

int i = 10;

/* int y = size; */ /* !!!Грешка, инициализиращият израз трябва да е литерал */

int main(void)
{
      int j = size; /* Позволена инициализация */

      /* char arr[size] = {0}; */ /* !!!Грешка, размерът на масив трябва да е
                                     целочислен литерал */
      /*...*/

      return 0;
}
```

1.3 Целочислени константи

Размерите на целочислените константи за всеки компилатор са описани в хедър-файла <limits.h>.

1.3.1 Десетични константи

Десетичните константи се състоят от последователност от арабските цифри 0 ÷ 9, като не могат да започват с 0.

Пример: 10, 123, 64674 и т.н.

Типът на десетичната константа е първия целочислен тип, започвайки от int, който може да представи стойността на тази константа (Табл.14).

вид константа	тип
Десетична целочислена константа	int
	long
	unsigned long

Табл. 14 Подразбиращ се тип на десетична целочислена константа

Минималният тип на десетичните константи може изрично да бъде указан с помощта на следните суфикси.

U или u – unsigned
L или l – long
UL или ul – unsigned long

Например нека тип int е 16 бита и long е 32 бита:

1000 – на числото 1000 се назначава тип int.
1000U – на числото 1000 се назначава тип unsigned int.
100000U – това число е твърде голямо да се побере в unsigned int, затова компилаторът ще му назначи следващия по големина целочислен тип, който може да побере тази стойност, а именно long.

Ако стойността на константата е твърде голяма, за да се побере и в най-големия целочислен тип, резултатът е недефиниран.

1.3.2 Шестнадесетични

Шестнадесетичните константи се състоят от префикса 0x (или 0X) последван от арабските цифри 0 ÷ 9 и латинските букви a ÷ f (A ÷ F).

Пример: 0x12, 0xa1, 0xFF и т.н.

Типът на шестнадесетичнта константа е първия целочислен тип, започвайки от int, който може да представи стойността на тази константа

(Табл.15).

вид константа	тип
Шестнадесетична константа	int
	unsigned
	long
	unsigned long

Табл. 15 Подразбиращ се тип на шестнадесетична константа

Ако стойността на константата е твърде голяма, за да се побере и в най-големия целочислен тип, резултатът е недефиниран.

1.3.3 Осмични

Осмичните константи се състоят от арабските цифри 0 ÷ 7 и започват с нула.

Пример: 012, 023, 077 и т.н

Типът на осмичната константа е първия целочислен тип, започвайки от int, който може да представи стойността на тази константа.

вид константа	тип
Осмична константа	int
	unsigned
	long
	unsigned long

Табл. 16 Подразбиращ се тип на осмична константа

Ако стойността на константата е твърде голяма, за да се побере и в най-големия целочислен тип, резултатът е недефиниран.

1.4 Константи с плаваща запетая

Константите с плаваща запетая имат цяла и дробна част. Те могат да се записват по два начина:

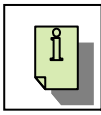
цяла-част. дробна-част

Пример: 123.093213, 0.7124 и т.н.

цяла-част. дробна-част E(или e) ± степен

Пример: 123.093E+2, 0.123E-2 и т.н.

Като разделител между цялата и дробната част се използва точка, а не запетая.



В С дробните числа използват десетична точка, а не запетая, за разделяне на цялата и дробната част.

В случай, че цялата или дробната част е 0, тя може да се изпусне, но десетичната точка задължително трябва да присъства. Именно, тя е указание за компилатора, че дадената константа трябва да се представи вътрешно като константа с плаваща запетая.

Когато степента е положителна, знакът + пред нея може да се изпусне.

Пример:

0.123 ≈ .123
123.0 ≈ 123.
0.123E-2 ≈ .123E-2
123.0E1 ≈ 123.0E+1 ≈ 123.E1

По подразбиране типът на константите с плаваща запетая е double. Минималният тип може изрично да бъде зададен чрез следните суфикси:

F или f – float
L или l – long double

Пример:

23.3 – тип double
23.3f – тип float

Размерите на типовете с плаваща запетая за всеки компилатор са указани в заглавния файл <float.h>.

1.5 Символни константи

Символната константа представлява символ затворен в апострофи.

'символ'

Пример: 'a', '!', '-' и т.н.

Стандартът C дефинира също и многобайтови (състоящи се от един или повече байтове) и широки (типично 16 бита) символи за представяне на всякакви символи. Широките символи имат тип **wchar_t**, който не е част от базовите типове, а е синоним на някой от беззнаковите целочислени типове (например unsigned int). Типът wchar_t е деклариран във файла <stddef.h>. Широките символни константи се въвеждат с главно L пред символната константа затворена в апострофи.

L'символ'

Пример: L'a', L'&' и т.н.

В действителност символните константи се заменят от компилатора със съответния им числов код от използваната от имплементацията кодова символна таблица (най-често ASCII). Символните константи имат тип int. Например ако имплементацията на даден компилатор използва ASCII кодовата таблица, на символа 'a' отговаря числовия код 97, а на 'A' 65 и т.н.

Пример 3-2:

```
#include <stdio.h>

int main(void)
{
	int i;

	printf("Value\tCodeTable\n");
	for(i = 0; i <= 255 ; i++)
	{
		printf("%d\t%c\n", i, i);
	}

	return 0;
}
```

Програмата отпечатва числата от 0 до 255 и съответните им символи от използваната кодова таблица на символите.

Стандартът C дефинира също и т.нар. escape-последователности, които се използват да представят символи без печатен образ (табулации и др.) и някои символи, които освен като обикновени символи имат и специално предназначение в синтаксиса на езика C (апостроф, двойни кавички, обратно наклонена черта). Escape-последователностите имат следния синтаксис:

\escape-символ

Въпреки, че се състоят от два символа, те се възприемат като един символ.

escape-последователност	описание
\a	Издава звуков сигнал
\b	Връща курсора една позиция назад
\f	Премества курсора на нова страница
\n	Премества курсора на нов ред
\r	Връща курсора в началото на реда
\t	Хоризонтална табулация
\v	Вертикална табулация
\\	Обратно наклонена черта
\?	Въпросителен знак
\'	Апостроф
\"	Кавички
\xHH	Символ , зададен чрез шестандесетично число

Табл. 17 escape-последователности

Пример 3-3:

```
#include <stdio.h>

int main(void)
{
	char c1 = '\'';  /* символ  '   */
	char c2 = '\"';  /* символ  "   */
	char c3 = '"';   /* символ  "   */
	char c4 = '\\';  /* символ  \   */

	printf("%c %c %c %c\n ", c1,c2,c3,c4);

	return 0;
}
```



1.6 Низови константи

Низовата константа е последователност от символи, затворена в двойни кавички.

Нормална низова константа:

"последователност-символи"

Широка низова константа:

L"последователност-символи"

Символите на низовата константа се съхраняват последователно в паметта на компютъра и компилаторът автоматично добавя нулев байт като последен символ. Нулевия байт (нулев символ) е байт, чийто битове са нули и се означава с \0.

По време на компилация две съседни низови константи се свързват и образуват една низова константа.

Пример 3-4:

```
#include <stdio.h>

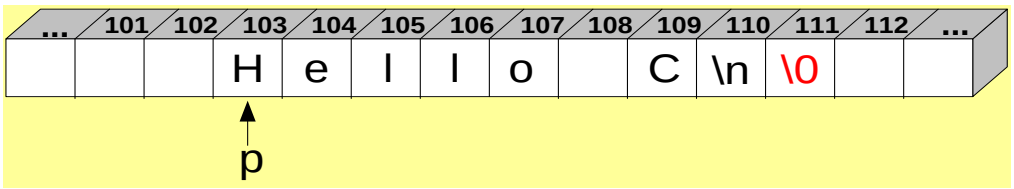
int main(void)
{
	char *p = "Hello " "C\n";

	printf("%s" , p);

	return 0;
}
```



Двата съседни низа "Hello " и "C\n" се свързват и образуват един низ "Hello C\n". Символите на така образуваната низова константа се разполагат някъде в паметта на компютъра и се добавя нулев байт '\0' (Фиг.10).



Фиг. 10 Разполагане на низова константа в паметта

Ако низова константа се появи някъде, с изключение като аргумент на адресната операция &, sizeof или като инициализатор на символен масив, резултатът е адреса на първият символ на масива.

Пример 3-5:

```
#include <stdio.h>

int main(void )
{
	char name[] = "Dimo Petkov"; /* Инициализиране на масива с низ */
	char     *p = "Dimo Petkov"; /* Адресът на първия символ на низа се
                                    присвоява ма указателя p */
	printf("%s\n",name);
	printf("%s\n",p);
```

```
	printf("%d\n",sizeof("Dimo Petkov"));

	return 0;
}
```



Компилаторът е свободен да поставя низовите константи в ROM-памет. Също така C89 позволява използването на една и съща памет за съхранение на еднакви низови константи. По тези причини опита да се модифицира някой от символите на низова константа може да доведе грешка по време на изпълнение на програмата.

Пример 3-6:

```
#include <stdio.h>

int main(void)
{
	char *p1 = "Dimo Petkov";
	char *p2 = "Dimo Petkov";

	p1[0] = 'P'; /* !!!Грешка */
	printf("%s\n", p1);
	printf("%s\n", p2);

	return 0;
}
```

В примера са използвани две еднакви низови константи. Първо компилаторът може да използва една и съща памет да съхрани двата низа, т.е указателите p1 и p2 ще сочат към един и същ символ. Опитът да се промени първият символ на низа, сочен от p1, ще доведе до логическа грешка в този случай. Второ тази памет може да е тип ROM.

Дали компилаторът използва обща памет за еднакви низови константи лесно може да се провери както е показано в следния пример:

Пример 3-7:

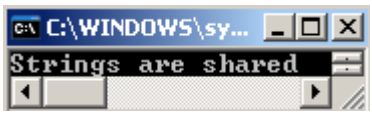
```
#include <stdio.h>

int main( void )
{
	char *p1 = "Dimo Petkov";
	char *p2 = "Dimo Petkov";

	if(p1 == p2)
	{
		printf("Strings are shared\n");
	}
```

```
	else
	{
		printf("Strings are not shared\n");
	}

	return 0;
}
```



Низовете обикновено се пишат на един ред. Но ако са твърде дълги е по удобно да се напишат на няколко реда. Това може да стане по два начина. Първият е чрез използване на обратно наклонена черта \.

"последователност-символи
последователност-символи
последователност-символи"

Обратно наклонената черта и символът за нов ред се игнорират в този случай и низа се възприема все едно е написан на един ред.

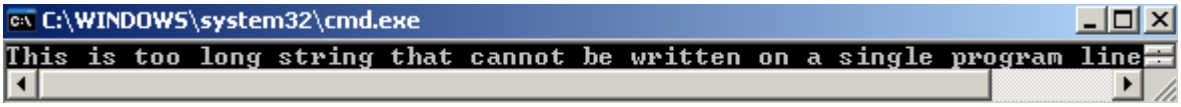
Пример 3-8:

```
#include <stdio.h>

int main(void)
{
	char arr[] = "This is too long string \
that cannot be written on a single program line\n";

	printf("%s", arr);

	return 0;
}
```



Вторият начин е чрез разбиване на низа съседни поднизове. Компилаторът автоматично свързва два съседни низа в един низ. Низовете могат да бъдат разделени от един или повече интервали, хоризонтална табулация и нов ред.

"последователност-символи" "последователност-символи"

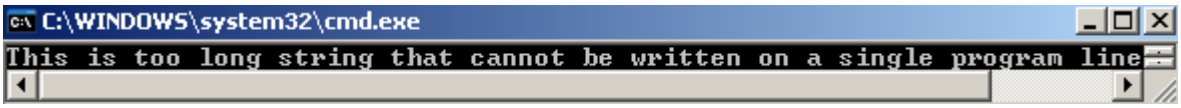
Пример 3-9:

```
include <stdio.h>

int main(void)
{
```

```
        char arr[] = "This is too long string "
                     "that cannot be written on a single program line\n";
        printf("%s", arr);

        return 0;
}
```



1.7 Enum-константи

Enum-константите са едно от трите средства за задаване на имена на целочислени константи. Задаването на имена на константите е много полезна практика тъй като използването на име вместо литералната стойност подсказва предназначението на константата и подобрява самодокументирането на кода .

enum-константи

```
enum име-изброяване_опц { enum-константа1, enum-константа2, ..., enum-константаN };
```

enum е ключова дума следвана от името на изброяването. То не е задължително и може да се пропусне ако не се дефинират enum-променливи (вижте Глава 7 Изброявания). Следва списък с именовани константи, затворени във фигурни скоби. За задаване на имена на константите се използват идентификатори. Enum-константите имат тип int. По подразбиране компилаторът задава стойност 0 на първия идентификатор, и на всеки следващ с единица по-голяма стойност от предходната. Enum-константите могат да се използват навсякъде където могат да се използват целочислени константи.

Пример 3-10:

```
#include <stdio.h>

enum digits {ZERO,ONE,TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT,NINE,TEN};
/* или enum {ZERO,ONE,TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT,NINE,TEN}; */

int main( void )
{
        printf("%d ", ZERO);
        printf("%d ", ONE);
        printf("%d ", TWO);
        printf("%s","... ");
        printf("%d ", TEN);

        return 0;
}
```



На идентификаторите могат да се задават всякакви целочислени стойности, които могат да се представят от тип int. Enum-константите дори могат да имат еднакви стойности.

```
enum име-изброяване {enum-константа1 = стойност, ...,enum-константаN = стойност};
```

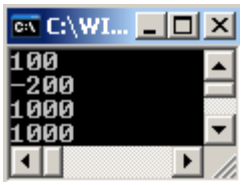
Пример 3-11:

```
#include <stdio.h>

enum vehicle{CAR = 100, BUS = -200, TRUCK = 1000, TRACTOR = 1000};

int main( void )
{
	printf("%d\n", CAR);
	printf("%d\n", BUS);
	printf("%d\n", TRUCK);
	printf("%d\n", TRACTOR);

	return 0;
}
```



Enum-константите са разгледани подробно в Глава 7 Изброявания.

1.8 Const-променливи

Ключовата дума const е едно от трите средства за задаване на имена на константи (целочислени, с плаваща запетая, символни и т.н.).

const-променливи

```
const  тип   име-константа = инициализатор;

където

тип
	Може да бъде от скаларен тип, агрегатен тип или обединение.

инициализатор
	Формата на инициализатора зависи от тип.
```

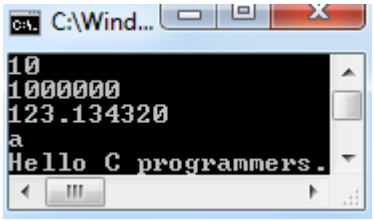
Пример 03-12:

```
#include <stdio.h>

int main( void )
{
	const int    ci  = 10;
	const long   cl  = 1000000;
	const double cd  = 123.13432;
	const char   cc  = 'a';
	const char*   p     = "Hello C programmer.";

	printf("%d\n",  ci);
	printf("%ld\n", cl);
	printf("%f\n",  cd);
	printf("%c\n",  cc);

	return 0;
}
```



Квалификаторът const е разгледан подробно в Глава 2 Типове Данни.

1.9 #define-константи

Предпроцесорната директива #define е едно от трите средства за задаване на имена на константи (целочислени, с плаваща запетая, символни и т.н.).

#define-константи

```
#define  име-константа  стойност
```

С помощта на тази директива могат да се задават имена на всякакви константи.

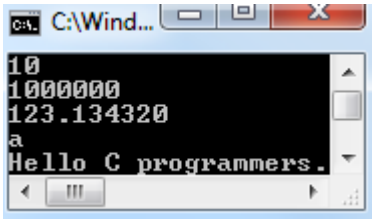
Пример 3-13:

```
#include <stdio.h>

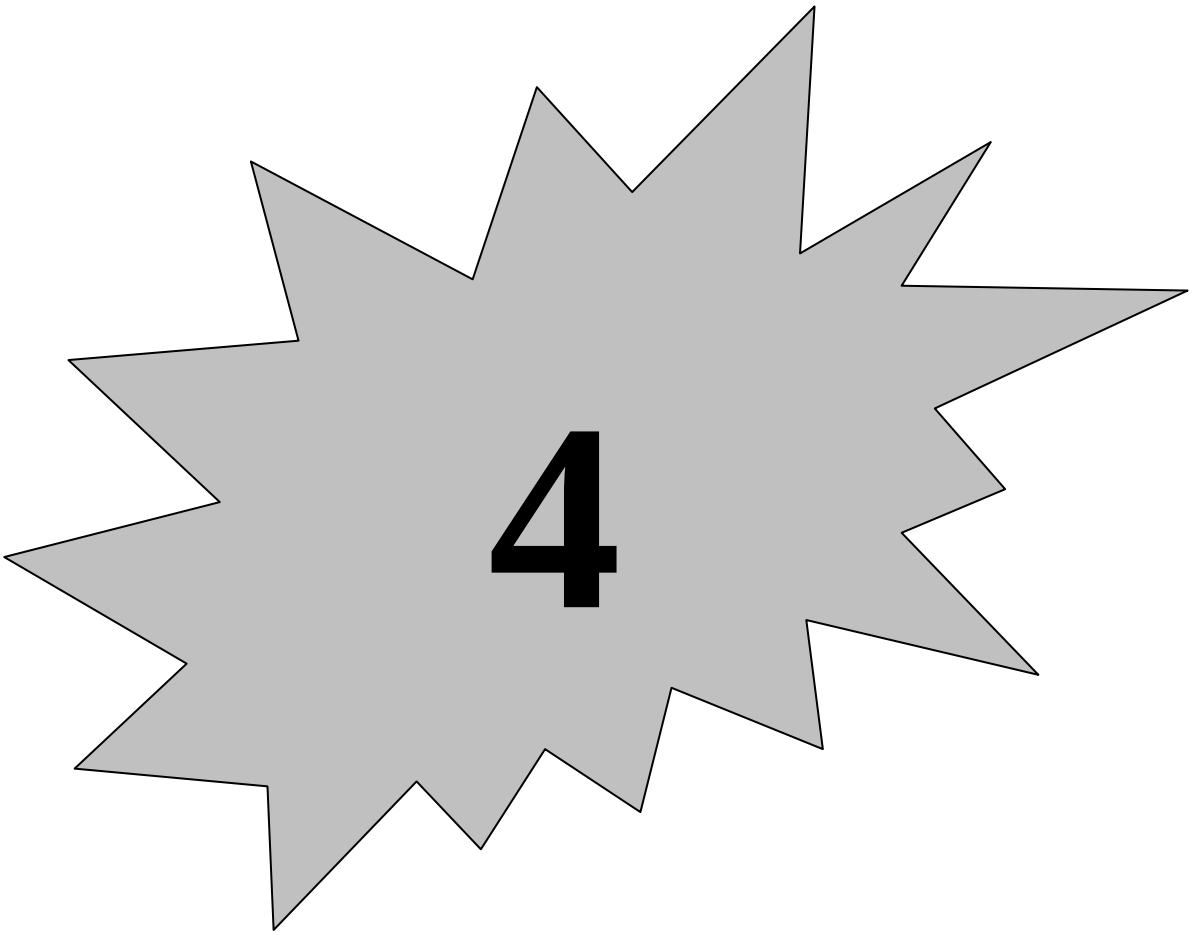
#define ci 10
#define cl 1000000
#define cd 123.13432
#define cc 'a'
#define WELCOME_MESSAGE   "Hello C programmers."
```

```
int main( void )
{
	printf("%d\n", ci);
	printf("%ld\n", cl);
	printf("%f\n", cd);
	printf("%c\n", cc);
	printf("%s\n", WELCOME_MESSAGE);

	return 0;
}
```



Директива #define е разгледана подробно в Глава 16 Предпроцесор.



Операции и изрази

1.1 L-стойности и R-стойности

Преди да разгледаме операциите и изразите в C ще Ви запознаем с три термина: L-стойност (**L-value**), модифицируема L-стойност (**modifiable L-value**) и R-стойност (**R-value**).

- L-стойност се нарича всяка стойност, която има адрес в паметта на компютъра. Такива са всички променливи. Литералните константи обаче нямат адрес и не са L-стойности. Термина L-стойност се е появил в началото от израз за присвояване E1 = E2, където операнда E1 вляво на знака за присвояване трябва да бъде променлива (на константи не могат да се присвояват нови стойности).

- модифицируема L-стойност се нарича всяка L-стойност, която може да се променя от програмата. Както видяхте в предходните глави, с въвеждането на ключовата дума const, не всички променливи могат да се променят от програмата. Обобщено можем да кажем, че всички модифицируеми L-стойности са и L-стойности, но обратното не винаги е вярно.

- R-стойност е всяка стойност, която може да се намира вдясно на символа за присвояване (операнд E2). Това могат да бъдат константи, L-стойности, модифицируеми L-стойности.

1.2 Видове операции

По отношение на функционалността си операциите могат да се разделят на следните групи:

- Аритметични операции (+, -, *, /, %, --, ++)
- Операции за сравнение (<, <=, >, >=, ==, !=)
- Логически операции (&&, ||, !)
- Битови операции (&, |, ^, ~, <<, >>)
- Операции за присвояване (=, +=, -=, *=, /=,%=, &=, |=, ^=, <<=, >>=)
- Условна операция (?:)
- Операция за последователно изчисляване на изрази (,)
- Операция за определяне на размер на тип (sizeof)

- Операция за преобразуване на тип ((тип))
- Операции за достъп до членове на структури и обединения (., ->)
- Операция за индексиране на масив ([])
- Операция за извикване на функция (())
- Указателни операции (&, *)

В зависимост от броя на операндите операциите се разделят на:

- унарни – имат един операнд
- бинарни – имат два операнда
- тринарни – имат три операнда

1.2.1 Аритметични операции (+, -, *, /, %, --, ++)

В Табл.18 са дадени всички аритметични операции заедно с кратко описание.

операция	описание
-	Изваждане или унарен минус
+	Събиране или унарен плюс
*	Умножение
/	Деление
%	Деление по модул
--	Намаление с 1
++	Увеличение с 1

Табл. 18 Аритметични операции

1.2.1.1 Операция изваждане -

Операция изваждане

```
операнд1 – операнд2
```

Резултатът от операцията изваждане е разликата, получена от изваждането на десния операнд от левия. Резултатът не е L-стойност. Операндите на

операцията изваждане трябва да спазват следните правила:

- Двата операнда трябва да бъдат от аритметични типове. Ако операндите са от различни аритметични типове, те се преобразуват до един общ тип съгласно обичайните бинарни преобразувания, който става и тип на резултата.

- Двата операнда трябва да бъдат указатели към обекти от съвместими типове (квалификаторите на типа се игнорират). Резултатът е от знаков целочислен тип **ptrdiff_t**, който е дефиниран в <stddef.h>. Тази операция има смисъл само ако обектите, към които сочат указателите са взаимно свързани, например указателите сочат към елементи на един и същ масив. Ако указателите сочат към обекти, нямащи връзка помежду си, резултатът е недефиниран.

- Левият операнд (операнд1) е указател, а десният (операнд2) е обект от целочислен тип. Резултатът е указател. Тази операция има смисъл само ако оригиналният и резултатният указател сочат към взаимно свързани обекти, например към елементи на един и същ масив. Ако указателите сочат към обекти, нямащи връзка помежду си, резултатът от дерефенцирането на резултатния указател е недефиниран.

Пример 4-1:

```
#include <stdio.h>

int main(void)
{
	char    c1, c2;
	int     i1, i2;
	double  d1, d2;
	int    *p1, *p2;
	int     *p, arr[5];

	c1 = 1;
	c2 = 2;
	printf("c1 - c2 = %d\n", c1-c2);

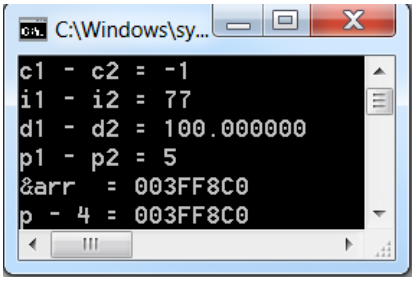
	i1 = 100;
	i2 = 23;
	printf("i1 - i2 = %d\n", i1-i2);

	d1 = 100.23;
	d2 = 0.23;
	printf("d1 - d2 = %f\n", d1-d2);

	p1 = arr;
	p2 = &arr[5];
	printf("p1 - p2 = %d\n", p2-p1);

	p = &arr[4];
	printf("&arr  = %p\n", arr);
	printf("p - 4 = %p\n", p-4);
```

```
	return 0;
}
```



Операцията изваждане може да доведе до непредсказуеми разултати ако настъпи препълване и операндите (след преобразуване на типовете) са знакови числа или числа с плаваща запетая или единия от операндите е указател. При изваждане на беззнакови цели числа препълване не може да се получи и резултатът е винаги ясно определен.

1.2.1.2 Операция унарен минус -

Унарен минус

```
-операнд
```

Резултатът от унарния минус е отрицание на неговия операнд. Операндът на унарния минус трябва да бъде от аритметичен тип. Преди да се извърши операцията, операндът се подлага на обичайните унарни преобразувания.

Пример 4-2:

```
#include <stdio.h>

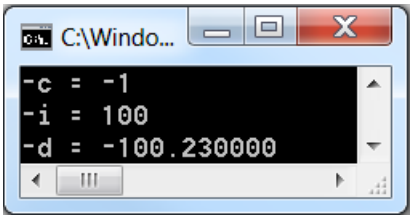
int main(void)
{
	signed char   c;
	int           i;
	double        d;

	c = 1;
	printf("-c = %d\n", -c);

	i = -100;
	printf("-i = %d\n", -i);

	d = 100.23;
	printf("-d = %f\n", -d);

	return 0;
}
```



1.2.1.3 Операция събиране +

операнд1 + операнд2

Резултатът от операцията събиране е сумата на двата операнда. Резултатът не е L-стойност. Операндите на операцията събиране трябва да спазват следните правила:

- Двата операнда трябва да са обекти от аритметичен тип. Ако операндите са от различни аритметични типове, те се преобразуват до един общ тип съгласно обичайните бинарни преобразувания, който става и тип на резултата.

- Единият от двата операнда трябва да е указател към обект, а другият трябва да е обект от целочислен тип. Резултатът е указател. Тази операция има смисъл само ако оригиналният и резултатният указател сочат към взаимно свързани обекти, например към елементи на един и същ масив. Ако указателите сочат към обекти, нямащи връзка помежду си, резултатът от дерефенцирането на резултатния указател е недефиниран.

Пример 4-3:

```
#include <stdio.h>

int main(void)
{
	char   c1, c2;
	int    i1, i2;
	double d1, d2;
	int   *p, arr[5];

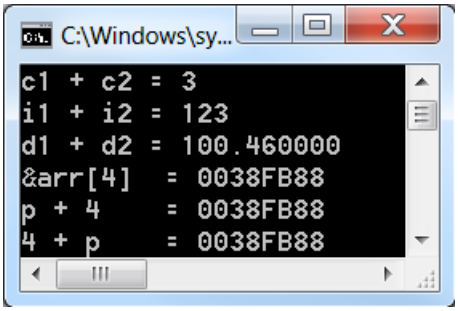
	c1 = 1;
	c2 = 2;
	printf("c1 + c2 = %d\n", c1+c2);

	i1 = 100;
	i2 = 23;
	printf("i1 + i2 = %d\n", i1+i2);

	d1 = 100.23;
	d2 = 0.23;
	printf("d1 + d2 = %f\n", d1+d2);
```

```
        p = &arr[0];
        printf("&arr[4]  = %p\n", &arr[4]);
        printf("p + 4    = %p\n", p+4);
        printf("4 + p    = %p\n", 4+p);

        return 0;
}
```



1.2.1.4 Операция унарен плюс +

Унарен плюс

+операнд

Резултатът от унарния плюс е стойността на неговия операнд. Операндът на унарния плюс трябва да бъде от аритметичен тип. Унарният плюс е излишна операция в C и не се използва на практика.

1.2.1.5 Операция умножение *

Операция умножение

операнд1 * операнд2

Резултатът от операцията умножение е произведението на двата операнда. Резултатът не е L-стойност. Операндите могат да са от всеки аритметичен тип. Ако типовете на операндите се различават, те се преобразуват до един общ тип съгласно обичайните бинарни преобразувания, който става и тип на резултата.

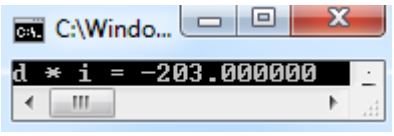
Пример 4-4:

```
#include <stdio.h>

int main(void)
{
    int     i;
    double  d;
```

```
    i = -10;
    d = 20.3;
    printf("d * i = %f\n", d * i);

    return 0;
}
```



В дадения пример операндите на операцията * се различават. Съгласно обичайните бинарни преобразувания типът на променливата **i** се преобразува в double преди да се извърши умножението. Резултатът също е тип double.

Операцията умножение може да доведе до непредсказуем резултат, ако настъпи препълване и операндите (след преобразуването) са знакови числа или числа с плаваща запетая.

Пример 4-5:

```
#include <stdio.h>
#include <float.h> /* Дефинира макроса DBL_MAX */

int main(void)
{
    int     i;
    double  d;

    i = 5;
    d = DBL_MAX; /* Максималното число тип double за конкретната система */
    printf("d * i = %f\n", d * i);

    return 0;
}
```



На променливата d се присвоява максималната стойност, която може да се представи от тип double за конкретната ситема. Изразът d*i в случая ще доведе до препълване и резултатът е недефиниран.

1.2.1.6 Операция деление /

Операция деление

```
операнд1  /  операнд2
```

Резултатът от операцията деление е частното на двата операнда. Резултатът не е L-стойност. Операндите могат да са от всеки аритметичен тип. Ако типовете на операндите се различават, те се преобразуват до един общ тип съгласно обичайните бинарни преобразувания, който става и тип на резултата.

Ако операндите са от целочислен тип и имат положителна стойност, то дробната част на резултата (ако има такава) се изхвърля.

Пример 4-6:

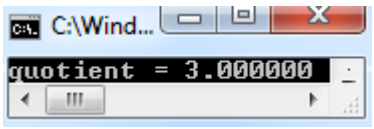
```
#include <stdio.h>

int main(void)
{
    int    numerator;
    int    denominator;
    double quotient;

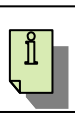
    numerator   = 10;
    denominator = 3;

    quotient = numerator / denominator;
    printf("quotient = %f\n", quotient);

    return 0;
}
```



Както се вижда от резултата дробната част е изхвърлена.



При деление на цели числа, дробната част на резултата (ако има такава) се изхвърля.

Ако операндите са от целочислен тип и единия от тях има отрицателна стойност, резултатът зависи от имплементацията. Възможни са два сценария.

- Дробната част се изхвърля (както и при положителни операнди). Тази операция се нарича още закръгляване към нулата.

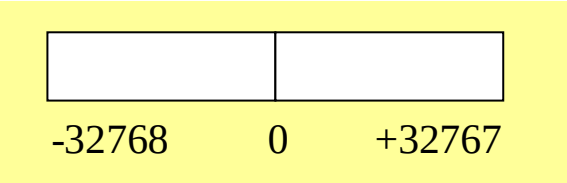
Пример: (-10) / 3 = -3 /* математически резултат -3.3(3) */

- Дробната част се изхвърля и резултата се закръглява към най-близкото цяло число, което е по-малко от резултата. Тази операция се нарича още закръгляване нагоре от нулата.

Пример: (-10) / 3 = -4 /* математически резултат -3.3(3) */

Библиотечните функции div() и ldiv() могат да се използват, тъй като тяхното поведение е ясно дефинирано когато единия операнд е отрицателен.

Операцията деление може да доведе до непредсказуеми разултати ако настъпи препълване и операндите (след преобразуването) са знакови цели числа или числа с плаваща запетая. Препълване може да настъпи за знакови цели числа, представени в код **2's complement** ако най-малкото отрицателно цяло число, което може да се представи от левия операнд се раздели на -1. Тогава резултатът е положителен, който не може да се представи от резултатния тип. Нека обясним това нагледно:

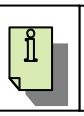


Фиг. 11 Числов обхват на тип signed short int

Нека да вземем най-малкото цяло число -32768, което може да се представи от тип signed short int и го разделим на -1. Резултатът 32768 излиза извън обхвата на този тип (Фиг.11).

Ако операндите са цели положителни числа препълване не може да настъпи.

Ако десният операнд е 0 резултата е недефиниран.



Ако десният операнд на операцията / е 0, резултатът е недефиниран.

1.2.1.7 Операция деление по модул %

Операция деление по модул

```
операнд1  %  операнд2
```

Резултатът от тази операция е остатъка от делението на двата операнда. Резултатът не е L-стойност. Операндите могат да са само целочислени аритметични типове. Ако типовете на операндите се различават, те се преобразуват до един общ тип съгласно обичайните бинарни преобразувания, който става и тип на резултата.

Пример 4-7:

```
#include <stdio.h>

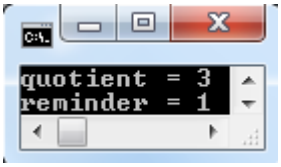
int main(void)
{
	signed char	numerator;
	signed char	denominator;
	signed char	quotient;
	signed char	reminder;

	numerator   = 10;
	denominator = 3;

	quotient = numerator / denominator;
	reminder = numerator % denominator;

	printf("quotient = %d\n", quotient);
	printf("reminder = %d\n", reminder);

	return 0;
}
```



	Операндите на операцията % могат да бъдат само от целочислен тип (char, short, int и т.н).

Следното равенство е винаги в сила:

`(a/b)*b + a%b = a`

От тук следва, че **a%b = a – (a/b)*b**, т.е. операцията % е пряко свързана с операцията /. В стандарта C89, ако единия от операндите на операцията / е цяло отрицателно число, то резултатът зависи от имплементацията. Следователно и резултата от операцията % ще зависи от имплементацията ако единия операнд е отрицателен.

Пример:

a = -5
b = 2

Закръгляване към нулата

a % b = -5 - (-5/2)*2 = -5 - (-2)*2 = -5 + 4 = -1

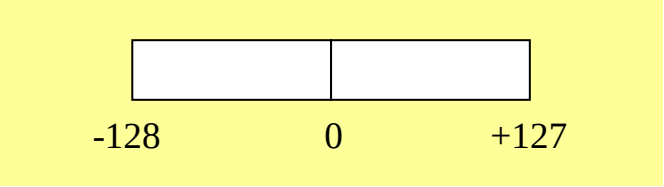
Закръгляване нагоре от нулата

a % b = -5 – (-5/2)*2 = -5 – (-3)*2 = -5 + 6 = +1

Библиотечните функции div(), ldiv() и fmod() могат да се използват за изчисление на остатъка от делението на цели числа и числа с плаваща запетая. Поведението на тези функции, ако единия операнд е отрицателен, е ясно дефинирано.

Операцията деление по модул може да доведе до непредсказуеми резултати ако настъпи препълване. Препълване може да настъпи за знакови цели числа, представени в код **2's complement**, ако най-малкото отрицателно цяло число, което може да се представи от левия операнд се раздели на -1. Тогава резултатът е положителен, който не може да се представи от резултатния тип и следователно резултатите са непредсказуеми, въпреки че остатъкът в този случай (0) може да се представи от резулатния тип.

Нека обясним това нагледно използвайки тип signed char (Фиг.12).



Фиг. 12 числов обхват на тип signed char

-128 / (-1) = 128 /* този резултат излиза извън обхвата на signed char */
-128 % (-1) = -128 - (-128/-1)*(-1) = -128 + 128 = 0

Ако десният операнд на операцията % е 0 резултатът е недефиниран.



Ако десният операнд на операцията % е 0, резултатът е недефиниран.

1.2.1.8 Операция увеличаване/намаляване с единица ++/--

Тези операции имат две форми: постфиксна и префиксна.

Постфиксна форма

```
операнд++
операнд--
```

Префиксна форма

```
++операнд
--операнд
```

Операндът трябва да е модифицируема L-стойност и може да е от всеки аритметичен или указателен тип. Резултатът от тези операции е увеличаване/намаляване на операнда с 1 и не е L-стойност. Обичайните бинарни преобразувания се извършват върху операнда и 1, и обичайните преобразувания при присвояване, когато резултатът се съхранява обратно в операнда. Типът на резултата е този на операнда преди преобразуването.

Двете форми са еквивалентни когато се използват самостоятелно.

Пример 4-8:

```
#include <stdio.h>

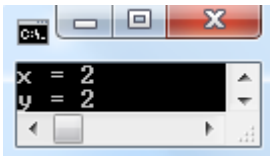
int main(void)
{
    int  x, y;

    x = 1;
    y = 1;

    x++;
    ++y;

    printf("x = %d\n", x);
    printf("y = %d\n", y);

    return 0;
}
```



Разликите между двете форми се проявяват когато операциите --/++ са част от по-голям израз.

За постфиксната форма е в сила следното правило: първо се извлича стойността на операнда и тази стойност ще се използва, ако операцията е част от друг израз. След това операндът се увеличава/намалява с 1.

Пример 4-9:

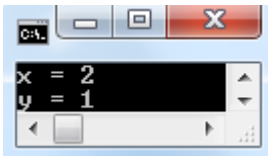
```
#include <stdio.h>

int main(void)
{
      int  x, y;
```

```
        x = 1;
        y = x++;

        printf("x = %d\n", x);
        printf("y = %d\n", y);

        return 0;
}
```



Нека разгледаме конструкцията **y = x++**. Съгласно описаното правило действията, които се извършват са еквивалентни на следното:

```
y = x;       /* Стойността на x се присвоява на y */
x = x +1;    /* x се увеличава с 1                 */
```

За префиксната форма е в сила следното правило: първо стойността на операнда се увеличава/намалява с 1 и тази стойност ще се използва, ако операцията е част от друг израз.

Пример 4-10:

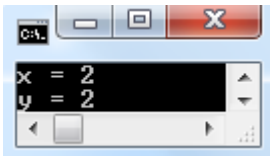
```
#include <stdio.h>

int main(void)
{
    int  x, y;

    x = 1;
    y = ++x;

    printf("x = %d\n", x);
    printf("y = %d\n", y);

    return 0;
}
```



Нека разгледаме конструкцията **y = ++x**. Съгласно описаното правило действията, които се извършват са еквивалентни на следното:

```
x = x +1;    /* x се увеличава с 1                 */
y = x;       /* Стойността на x се присвоява на y */
```

Тези операции могат да доведат до непредсказуеми разултати ако настъпи

препълване и операндът е знаково цяло число или число с плаваща запетая.

1.2.2 Операции за отношение (<, <=, >, >=)

В Табл.19 са дадени операциите за отношение заедно с кратко описание.

операция	описание
<	По-малко
<=	По-малко или равно
>	По-голямо
>=	По-голямо или равно

Табл. 19 Операции за отношение

Операции за отношение

```
операнд1 операция операнд2

операция: <, <=, >, >=
```

Операндите трябва да спазват следните правила:

- Двата операнда трябва да са от аритметичен тип. Ако операндите са от различни аритметични типове, те се преобразуват до един общ тип съгласно обичайните бинарни преобразувания.

- Двата операнда трябва да са указатели (квалифицирани или неквалифицирани) към съвместими обектни типове.

- Двата операнда трябва да са указатели (квалифицирани или неквалифицирани) към непълни типове.

Резултатът от операциите за отношение може да бъде само 1 или 0. Ако условието е изпълено, резултатът е 1 , в противен случай е 0.Независимо от типа на операндите, типът на резултата е винаги int и не е L-стойност.

Пример 4-11:

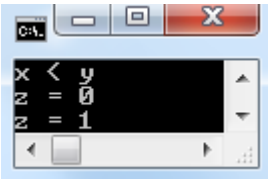
```
#include <stdio.h>
int main(void)
{
    int  x;
    int  y;
    int  z;

    x = 1;
    y = 2;
```

```
    if (x > y)
    {
       printf("x > y\n");
    }
    else
    {
       printf("x < y\n");
    }

    z = x > y;
    printf("z = %d\n", z);
    z = x < y;
    printf("z = %d\n", z);

    return 0;
}
```





Резултатът от операциите за отношение е 0 или 1.

Имплементацията на числа с плаваща запетая в C може да включва стойности като NaNs, които не са числа. Използвайки такива стойности в изрази за отношение може да предизвика генериране на изключение и резултатът от операцията ще бъде 0.

Когато двата операнда са указатели, резултатът зависи от относителното разположение на сочените обекти в адресното пространство. Сравняването на указатели има смисъл само ако те сочат към един и същ агрегатен тип (масив или структурна-променлива).

Ако указателите сочат към един и същ масив, указателя, който сочи към елемент с по-голям индекс се счита за "по-голям" от този, който сочи към елемент с по-малък индекс. Обяснението е просто – елементите на масив се разполагат последователно в паметта започвайки от по-малък адрес към по-голям.

Пример 4-12:

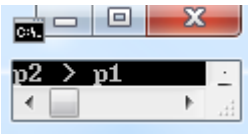
```
#include <stdio.h>

int main(void)
{
       int a[5] = {1,2,3,4,5};
       int *p1, *p2;

       p1 = &a[0];
       p2 = &a[4];
```

```
	if (p2 > p1)
	{
		printf("p2 > p1\n");
	}

	return 0;
}
```



Ако указателите сочат към структурна-променлива, указателя, който сочи към член, който е дефиниран по-късно в структурата, се счита за "по-голям" от този, който сочи към член, дефинират по-рано в структурата. Обяснението е просто – членовете на структурната-променлива се разполагат последователно в паметта започвайки от по-малък адрес към по-голям.

Пример 4-13:

```
#include <stdio.h>

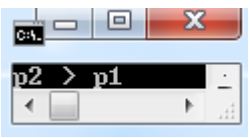
struct MyStruct
{
	int x1;
	int x2;
	int x3;
};

int main(void)
{
	struct MyStruct s = {1,2,3};
	int *p1, *p2;

	p1 = &s.x1;
	p2 = &s.x3;

	if (p2 > p1)
	{
		printf("p2 > p1\n");
	}

	return 0;
}
```



Указатели към членовете на едно и също обединение в действителност сочат към един и същ адрес и се считат за равни.

Пример 4-14:

```c
#include <stdio.h>

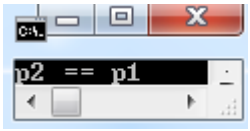
union MyUnion
{
	int x1;
	int x2;
	int x3;
};

int main(void)
{
	union MyUnion s = {1};
	int   *p1, *p2;

	p1 = &s.x1;
	p2 = &s.x3;

	if (p2 > p1)
	{
		printf("p2 > p1\n");
	}
	else if (p2 < p1)
	{
		printf("p2 < p1\n");
	}
	else
	{
		printf("p2 == p1\n");
	}

	return 0;
}
```



Ако обектите, сочени от сравняваните указатели, не са членове на един и същ агрегатен обект или обединение, резултатът е недефиниран с едно изключение: ако указател P сочи към елемент на масив и указател Q сочи към последния елемент на масива, то израза Q+1 ще бъде "по-голям" от P, въпреки, че Q+1 не сочи към елемент на масива.

Пример 4-15:

```c
#include <stdio.h>

int main(void)
{
	int a[5] = {1,2,3,4,5};
	int *p1, *p2;

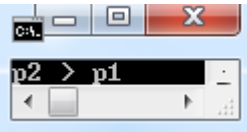
	p1 = &a[0];
	p2 = &a[4];

	p2++; /* Сега p  сочи след последния елемент на масива */

	if (p2 > p1)
	{
```

```
            printf("p2 > p1\n");
      }

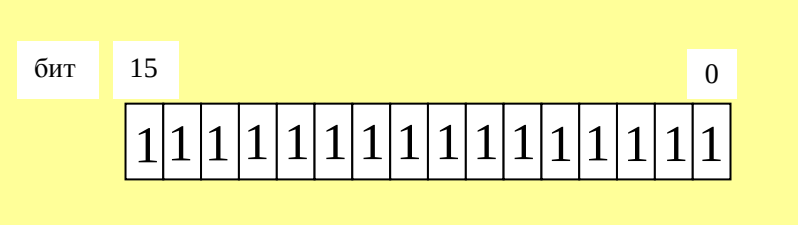
      return 0;
}
```



Следната особеност трябва да се има предвид когато се сравняват беззнакови и знакови целочислени типове. Съгласно обичайните бинарни преобразувания "по-малък" знаков тип ще се преобразува в "по-голям" безнаков тип. Това може да доведе до напълно различно поведение от очакваното при сравнение на такива операнди. Например:

-1 < (unsigned) 0

От математическа гледна точка 0 е по-голямо от -1 и този израз трябва да върне като резултат 1 в контекста на C. Но поради правилата за преобразуване операндът -1, който има тип int ще се преобразува до unsigned int. Това на практика означава, че компилаторът ще интерпретира вътрешното представяне на -1 по съвсем друг начин. Нека приемем, че тип int е 16 бита и отрицателните числа се представят във формат **2's complement**. Фиг.13 показва вътрешното представяне на -1 при тези условия.



Фиг. 13 Вътрешно представяне на -1 в 16 битов 2's complemet формат

Това число интерпретирано като unsigned int ще даде като резултат 65535 и изразът -1 < (unsigned) 0 ще бъде интерпретиран като 65535 < 0 и съответно ще върне 0 като резултат, вместо 1 както се очаква на пръв поглед.

Пример 4-16:

```
#include <stdio.h>

int main(void)
{
      if (-1 < (unsigned)0)
      {
            printf("This will not be executed\n");
      }
```

```
        return 0;
}
```

1.2.3 Операции за равенство (==, !=)

В Табл.20 са дадени операциите за равенство заедно с кратко описание.

операция	описание
==	равно
!=	различно

Табл. 20 Операции за равенство

Операции за равенство

операнд1 операция **операнд2**

операция: ==, !=

Операндите трябва да спазват следните правила:

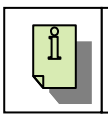
- Двата операнда трябва да са от аритметичен тип. Ако операндите са от различни аритметични типове, те се преобразуват до един общ тип съгласно обичайните бинарни преобразувания.

- Двата операнда трябва да са указатели (квалифицирани или неквалифицирани) към съвместими типове. Двата указателя могат да са тип void*.

- Единият операнд може да бъде указател към обект или непълен тип, а другия може да бъде квалифициран или неквалифициран void* тип. Първият операнд ще се преобразува към квалифициран или неквалифициран void* тип.

- Единият операнд може да бъде указател, а другия нулев константен указател ((void*)0).

Резултатът от операциите за равенство може да бъде само 1 или 0. Ако условието е изпълено, резултатът е 1, в противен случай е 0. Независимо от типа на операндите типът на резултата е винаги int и не е L-стойност.



Резултатът от операциите за равенство е 0 или 1.

Ако операндите са указатели, те ще бъдат равни само ако са изпълнени

следните условия:

1) Двата указателя сочат към един и същ обект или функция.

2) Двата указателя са нулеви указатели

3) Двата указателя сочат към един и същ елемент, който се намира след последния елемент на масив.

Пример 4-17:

```
#include <stdio.h>

int main(void)
{
	int x, y;
	int a[5] = {1,2,3,4,5};
	int *p1, *p2;

	x = 1;
	y = 1;
	p1 = &a[0];
	p2 = &a[4];

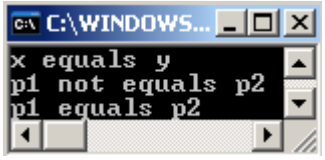
	if(x == y)
	{
		printf("x equals y\n");
	}

	if(p1 != p2)
	{
		printf("p1 not equals p2\n");
	}

	p1 = 0;
	p2 = 0;

	if(p1 == p2)
	{
		printf("p1 equals p2\n");
	}

	return 0;
}
```





Внимавайте да не използвате операцията = вместо операцията ==. Ако сравнявате променлива и константа, препоръчително е константата да се използва като ляв операнд. Така ако по невнимание използвате операцията =, компилаторът ще генерира грешка.

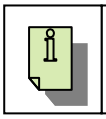
1.2.4 Логически операции (&&, ||, !)

В Табл.21 са дадени логическите операции заедно с кратко описание.

операция	описание
&&	Логическо И (AND)
||	Логически ИЛИ (OR)
!	Логическо НЕ (NOT)

Табл. 21 Логически операции

Логическите операции се използват най-често за да комбинират няколко израза за отношение и/или равенство.



Резултатът от всички логически операции е 0 или 1.

1.2.4.1 Логическо И &&

Логическо И

```
операнд1  &&  операнд2
```

Операндите могат да бъдат от всеки скаларен тип. Няма връзка между типовете на двата операнда. Всеки от тях се подлага на обичайните унарни преобразувания. Резултатът от операцията винаги е от тип int и има стойност 0 или 1. В C всяка стойност различна от нула се възприема за TRUE (ИСТИНА), а нулата за FALSE (ЛЪЖА). Операцията && се извършва съгласно следната таблица на истинност:

операнд1	операнд2	резултат
FALSE	FALSE	0
FALSE	TRUE	0
TRUE	FALSE	0
TRUE	TRUE	1

Табл. 22 Таблица на истинност на операцията &&

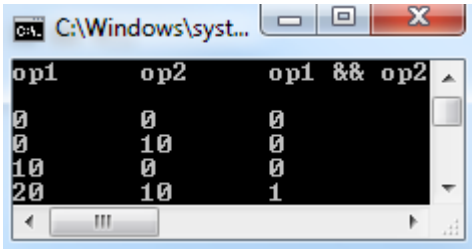
Както се вижда от Табл.22 резултатът ще бъде 1 само когато и двата операнда имат стойности различни от 0 (за аритметични типове) или NULL (за указателни типове).

Пример 4-18:

```
#include <stdio.h>
```

```
int main(void)
{
	printf("op1\top2\top1 && op2\n\n");
	printf("0\t0\t%d\n", 0 && 0);
	printf("0\t10\t%d\n", 0 && 10);
	printf("10\t0\t%d\n", 10 && 0);
	printf("20\t10\t%d\n", 20 && 10);

	return 0;
}
```



Операцията && има следната важна особеност. Ако левият операнд е FALSE, резултатът е 0 независимо от стойността на десния операнд. Езикът C използва тази особеност и извършва следните действия когато изчислява резултата от операцията:

1) Първо се изчислява стойността на левия операнд

2) Ако стойността на левия операнд е 0, то резултатът от цялата операция е 0 и десният операнд не се изчислява.

3) Ако стойността на левия операнд е различна от 0, се изчислява и стойността на десния операнд и резултатът се изчислява съгласно таблицата на истинност.

Тази особеност трябва да се има предвид когато десният операнд е по-сложен израз, който съдържа в себе си странични ефекти.

Пример 4-19:

```
#include <stdio.h>

int main(void)
{
	int x = 0, y = 10, z = 0;

	(x != 0) && (z = y/x);
	printf("z = %d\n", z);

	return 0;
}
```



Нека разгледаме този пример подробно. Левият операнд на операцията && е израза (x != 0), а десният операнд е израза (z = y/x). Както се вижда от резултата, десният операнд, който съдържа в себе си страничен ефект, изразяващ се в присвояване на резултата от делението на y на x на променливата z, не е изчислен. В резултат променливата z остава непроменена. Причината е ясна, изразът (x != 0) в дадения пример има стойност 0 и компилаторът не изчислява десния операнд. Ако зададем на x стойност различна от 0, например 2, ще видим че стойността на z сега е 5, което означава, че десния операнд също е изчислен. В показания пример, крайният резултат от операцията && не се използва и той се изхвърля от компилатора. В практиката операцията && се използва в условните изрази на операторите if, switch, for, while и do-while за да комбинира няколко израза, които са най-често изрази за отношение и сравнение.

Пример 4-20:

```
#include <stdio.h>

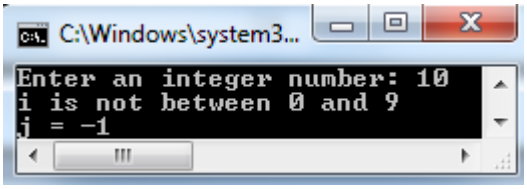
int main(void)
{
	int i;
	int j = -1;

	printf("Enter an integer number: ");
	scanf("%d", &i);

	if ( (i >=0) && (i <= 9) )
	{
		j = i;
	}
	else
	{
		printf("i is not between 0 and 9\n");
	}

	printf("j = %d\n", j);

	return 0;
}
```



Както показва резултатът от изпълнението на програмата, на променливата j ще се присвои стойността на променливата i само когато и двете условия са изпълнени, i >= 0 и i <= 9. В противен случай стойността на j не се променя.

1.2.4.2 Логическо ИЛИ ||

Логическо ИЛИ

операнд1 || операнд2

Операндите могат да бъдат от всеки скаларен тип. Няма връзка между типовете на двата операнда. Всеки от тях се подлага на обичайните унарни преобразувания. Резултатът от операцията винаги е от тип int и има стойност 0 или 1. В C всяка стойност различна от нула се възприема за TRUE (ИСТИНА), а нулата за FALSE (ЛЪЖА). Операцията || се извършва съгласно следната таблица на истинност:

операнд1	операнд2	резултат
FALSE	FALSE	0
FALSE	TRUE	1
TRUE	FALSE	1
TRUE	TRUE	1

Табл. 23 Таблица на истинност на операцията ||

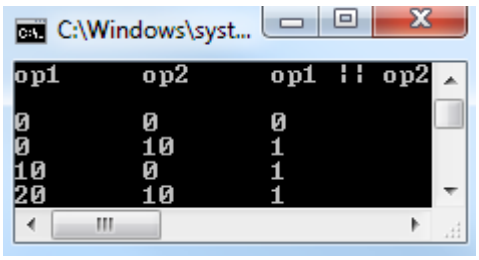
Както се вижда от Табл.23 резултатът ще бъде 0 само когато и двата операнда имат стойност 0 (за аритметични типове) или NULL (за указателни типове).

Пример 4-21:

```
#include <stdio.h>

int main(void)
{
	printf("op1\top2\top1 || op2\n\n");
	printf("0\t0\t%d\n", 0 || 0);
	printf("0\t10\t%d\n", 0 || 10);
	printf("10\t0\t%d\n", 10 || 0);
	printf("20\t10\t%d\n", 20 || 10);

	return 0;
}
```



Операцията || има следната важна особеност. Ако левият операнд е TRUE, резултатът е 1 независимо от стойността на десния операнд. Езикът C

използва тази особеност и извършва следните действия когато изчислява резултата от операцията:

1) Първо се изчислява стойността на левия операнд.

2) Ако стойността на левия операнд е различна от 0, то резултатът от цялата операция е 1 и десният операнд не се изчислява.

3) Ако стойността на левия операнд е 0, се изчислява и стойността на десния операнд и резултатът се изчислява съгласно таблицата на истинност.

Тази особеност трябва да се има предвид когато десният операнд е по-сложен израз, който съдържа в себе си странични ефекти.

Пример 4-22:

```
#include <stdio.h>

int main(void)
{
	int x = 0, y = 10, z = 1;

	(x == 0) || (z = y/x);
	printf("z = %d\n", z);

	return 0;
}
```



Нека разгледаме този пример подробно. Левият операнд на операцията || е израза (x == 0), а десният операнд е израза (z = y/x). Както се вижда от резултата, десния операнд, който съдържа в себе си страничен ефект, изразяващ се в присвояване на резултата от делението на y на x на променливата z, не е изчислен. В резултат променливата z остава непроменена. Причината е ясна, изразът (x == 0) в дадения пример има стойност 1 и компилаторът не изчислява десния операнд. Ако зададем на x стойност различна от 0, например 2, ще видим че стойността на z сега е 5, което означава, че десният операнд също е изчислен. В показан ия пример, крайният резултат от операцията || не се използва и той се изхвърля от компилатора. В практиката операцията || се използва в условните изрази на операторите if, switch, for, while и do-while за да комбинира няколко израза, които са най-често изрази за отношение и сравнение.

Пример 4-23:

```
#include <stdio.h>
```

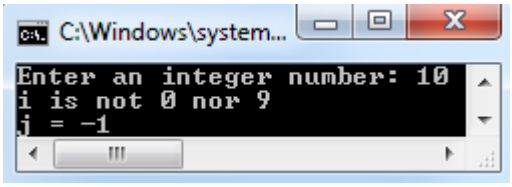
```
int main(void)
{
	int i;
	int j = -1;

	printf("Enter an integer number: ");
	scanf("%d", &i);

	if ( (i == 0) || (i == 9) )
	{
		j = i;
	}
	else
	{
		printf("i is not 0 nor 9\n");
	}

	printf("j = %d\n", j);

	return 0;
}
```



Променливата i ще се присвои на j само ако i е 0 или 9. В противен случай стойността на j не се променя.

1.2.4.3 Логическо НЕ !

Логическо НЕ

!операнд

Операндът може да бъде от всеки скаларен тип. Върху операнда се прилагат обичайните унарни преобразувания. Резултатът от операцията е от тип int и приема стойност 0 или 1. В C всяка стойност различна от нула се възприема за TRUE (ИСТИНА), а нулата за FALSE (ЛЪЖА). Операцията ! се извършва съгласно следната таблица на истинност:

операнд	резултат
FALSE	1
TRUE	0

Табл. 24 Таблица на истинност на операцията !

Ако стойността на операнда е 0, резултатът от логическото НЕ е 1. Ако стойността на операнда е различна от нула, резултатът е 0. Например !0

или !0.0, или !NULL ще даде като резултат 1, а !100 или !2.3, или !ptr, където ptr е ненулев указател към обект или функция, ще даде като резултат 0.

Пример 4-24:

```
#include <stdio.h>

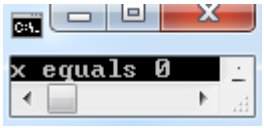
int main(void)
{
	int x = 0;

	if(!x)
	{
		printf("x equals 0\n");
	}

	x = 100;

	if(!x)
	{
		printf("This will not be executed\n");
	}

	return 0;
}
```



Следните два израза са еквивалентни: !(x) и (x) == 0. Това лесно може да се провери. Ако x е 0 и двата израза връщат 1 като резултат. Ако x е различно от 0, например 10, и двата израза връщат 0 като резултат.



Следните изрази са еквивалентни: !(x) и (x) == 0.

1.2.5 Битови операции (&, |, ^, ~, <<, >>)

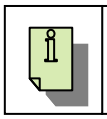
В Табл.25 са дадени битовите операции заедно с кратко описание.

операция	описание
&	Битово И (Bitwise AND)
|	Битово ИЛИ (Bitwise OR)
^	Битово Изключващо ИЛИ (Bitwise Exclusive OR)
~	Битово инвертиране
<<	Битово преместване на ляво

>>	Битово преместване на дясно

Табл. 25 Битови операции

Както показва самото име, тези операции въздействат върху битовете на своите операнди. Битовите операции могат да се прилагат само върху целочислени (знакови и беззнакови) типове (char, short, int, long и enum-типове).



Битовите операции могат да се прилагат само върху целочислени (знакови и беззнакови) типове (char, short, int, long и enum-типове).

1.2.5.1 Битово И &

Битово И

```
операнд1 & операнд2
```

Операндите могат да бъдат само от целочислен тип. Ако двата операнда са от различни целочислени типове те се преобразуват до един общ целочислен тип съгласно обичайните бинарни преобразувания, който става и тип на резултата. Резултатът не е L-стойност. Операцията битово И се извършва съгласно следната таблица на истинност:

бит n - операнд1	бит n -операнд2	резултат
0	0	0
0	1	0
1	0	0
1	1	1

Табл. 26 Таблица на истиност на операцията &

Както се вижда от Табл.26 резултатният бит е 1 само когато двата бита на операндите също са 1.

Нека разгледаме следния пример: 10_{10} & 9_{10}

Битовото представяне на тези операнди изглежда така: 00001010_2 (10_{10}) и 00001001_2 (9_{10}).

00001010 операнд1 (10_{10})
&
00001001 операнд2 (9_{10})

00001000 резултат (8_{10})

Битовото И се използва за нулиране на един или повече битове на даден операнд. За целта се използва втори операнд (наричан още маска), който съдържа нули в желаните позиции за нулиране и единици във всички останали позиции.

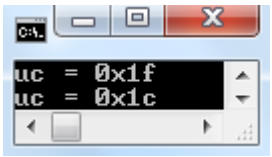
Пример 4-25:

```
#include <stdio.h>

int main(void)
{
	unsigned char uc = 0x1F;

	printf("uc = %#x\n", uc);
	uc = uc & 0xFC; /* Нулиране на бит 0 и 1 */
	printf("uc = %#x\n", uc);

	 return 0;
}
```



В показания пример операндът uc има стойност 0x1F (00011111). За да нулираме например битове 0 и 1 (без да променяме останалите битове) е необходимо да използваме маска, която да съдържа нули в тези позиции и единици във всички останали – 0xFC (11111100). Както се вижда от резултата, битовете 0 и 1 са нулирани – 0x1C (00011100).

Битовото И се използва и за проверка дали даден бит на един операнд е 0 или 1. За целта се използва маска, която съдържа единица в желаната позиция за проверка и нули във всички останали. Ако резулатът от операцията е 0, значи проверяваният бит е нула, а ако резулатът е различен от нула, проверяваният бит е 1.

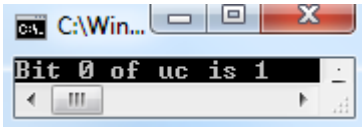
Пример 4-26:

```
#include <stdio.h>

int main(void)
{
    unsigned char uc = 0x1F;

    /* Проверка на бит 0 */
    if (uc & 0x01)
    {
       printf("Bit 0 of uc is 1\n");
    }

    return 0;
}
```



1.2.5.2 Битово ИЛИ |

Битово ИЛИ

операнд1 | операнд2

Операндите могат да бъдат само от целочислен тип. Ако двата операнда са от различни целочислени типове те се преобразуват до един общ целочислен тип съгласно обичайните бинарни преобразувания, който става и тип на резултата. Резултатът не е L-стойност. Операцията побитово ИЛИ се извършва съгласно следната таблица за истинност:

бит n - операнд1	бит n -операнд2	резултат
0	0	0
0	1	1
1	0	1
1	1	1

Табл. 27 Таблица на истиност на операция Логическо ИЛИ

Както се вижда от таблицата резултатният бит е 0 само когато и двата бита на операндите са 0.

Нека разгледаме следния пример: 10_{10} | 9_{10}

Битовото представяне на тези операнди изглежда така: 00001010_2 (10_{10}) и 00001001_2 (9_{10}).

00001010 операнд1 (10_{10})
|
00001001 операнд2 (9_{10})

00001011 резултат (11_{10})

Битовото ИЛИ се използва за установяване в 1 на един или повече битове на даден операнд. За целта се използва втори операнд (наричан още маска), който съдържа единици в желаните позиции за установяване и нули във всички останали позиции.

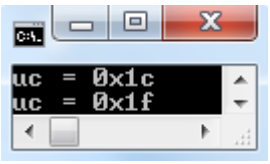
Пример 4-27:

```c
#include <stdio.h>

int main(void)
{
	unsigned char uc = 0x1C;

	printf("uc = %#x\n", uc);
	uc = uc | 0x03; /* Установяване на бит 0 и 1 */
	printf("uc = %#x\n", uc);

	return 0;
}
```



В показания пример операндът uc има стойност 0x1C (00011100). За да установим например битове 0 и 1 (без да променяме останалите битове) е необходимо да използваме маска, която да съдържа единици в тези позиции и нули във всички останали – 0x03 (00000011). Както се вижда от резултата, битовете 0 и 1 са установени – 0x1F (00011111)

1.2.5.3 Битово Изключващо ИЛИ ^

Битово Изключващо ИЛИ

операнд1 ^ операнд2

Операндите могат да бъдат само от целочислен тип. Ако двата операнда са от различни целочислени типове те се преобразуват до един общ целочислен тип съгласно обичайните бинарни преобразувания, който става и тип на резултата. Резултатът не е L-стойност. Операцията побитово Изключващо ИЛИ се извършва съгласно следната таблица за истинност:

бит n - операнд1	бит n -операнд2	резултат
0	0	0
0	1	1
1	0	1
1	1	0

Табл. 28 Таблица на истинност на операцията битово Изключващо ИЛИ

Както се вижда от таблицата резултатният бит е 1 само когато единия от двата бита на операндите е 1.

Нека разгледаме следния пример: 10_{10} ^ 9_{10}

Битовото представяне на тези операнди изглежда така: 00001010_2 (10_{10}) и 00001001_2 (9_{10}).

```
00001010  операнд1 (10₁₀)
    ^
00001001  операнд2 (9₁₀)
--------
00000011  резултат (3₁₀)
```

Типична употреба на тази операция е да инвертира определени битове на даден операнд.

Пример 4-28:

```
#include <stdio.h>

int main( void )
{
	unsigned char uc;
	unsigned char res;

	uc = 0x55;
	res = uc ^ 0x03;
	printf("res = %#x\n", res);

	return 0;
}
```



В показания пример операндът uc има стойност 0x55 (01010101). За да инвертираме например битове 0 и 1 (без да променяме останалите битове) е необходимо да използваме маска, която да съдържа единици в тези позиции и нули във всички останали – 0x03 (00000011). Както се вижда от резултата, битовете 0 и 1 са инвертирани – 0x56 (01010110)

Интересно свойство на операцията битово Изключващо ИЛИ, че ако се приложи между резултата и единия операнд, резултатът ще бъде другия операнд. Ако вземем по-горния пример, то

```
00000011   резултат (3)          00000011   резултат (3)
    ^                                 ^
00001001   операнд2 (9)           00001010   операнд1 (10)
--------                          --------
00001010   операнд1 (10)          00001001   операнд2 (9)
```

1.2.5.4 Битово инвертиране (отрицание) ~

Битово Инвертиране

```
~операнд
```

Операндът може да бъде от всеки целочислен тип. Преди да се извърши операцията върху операнда се прилагат обичайните унарни преобразувания. Резултатът не е L-стойност. Операцията ~ инвертира битовете на своя операнд, т.е всички битове, които са 1 стават 0, а всички битове, които са 0 стават 1.

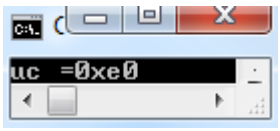
Пример 4-29:

```
#include <stdio.h>

int main(void)
{
    unsigned char uc;

    uc = ~0x1F;
    printf("uc =%#x\n", uc);

    return 0;
}
```



В дадения пример битовете на операнда 0x1F (00011111) се инвертират с операцията ~. Резултатът е 0xE0 (11100000).

Изразът **~операнд**, където операнд е от беззнаков тип, е еквивалнтен на **(ULONG_MAX - операнд)**, ако операндът е повишен до тип unsigned long, или на (**UINT_MAX - операнд)** ако операндът е повишен до тип unsigned int. Макросите **ULONG_MAX** и **UINT_MAX** са дефинирани в хедър файла <limits.h>.

1.2.5.5 Битово преместване наляво <<

Битово преместване наляво

```
операнд1 << операнд2
```

Двата операнда трябва да са от целочислен тип. Няма връзка между двата операнда, всеки от тях се подлага поотделно на обичайните унарни преобразувания. Типът на левия операнд след преобразуването е и типа на

резултата. Резултатът не е L-стойно
ост. Операцията << премества левия операнд наляво с толкова бита, колкото е указано с десния операнд. При преместване наляво на n-битово цяло число с m на бройт бита, където $0 \leq m \leq (n - 1)$, най-левите m на брой бита се изгубват, а освободените m на брой най-десни бита се попълват с нули.

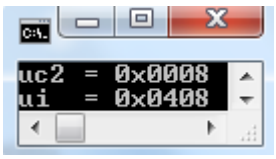
Пример 4-30:

```
#include <stdio.h>

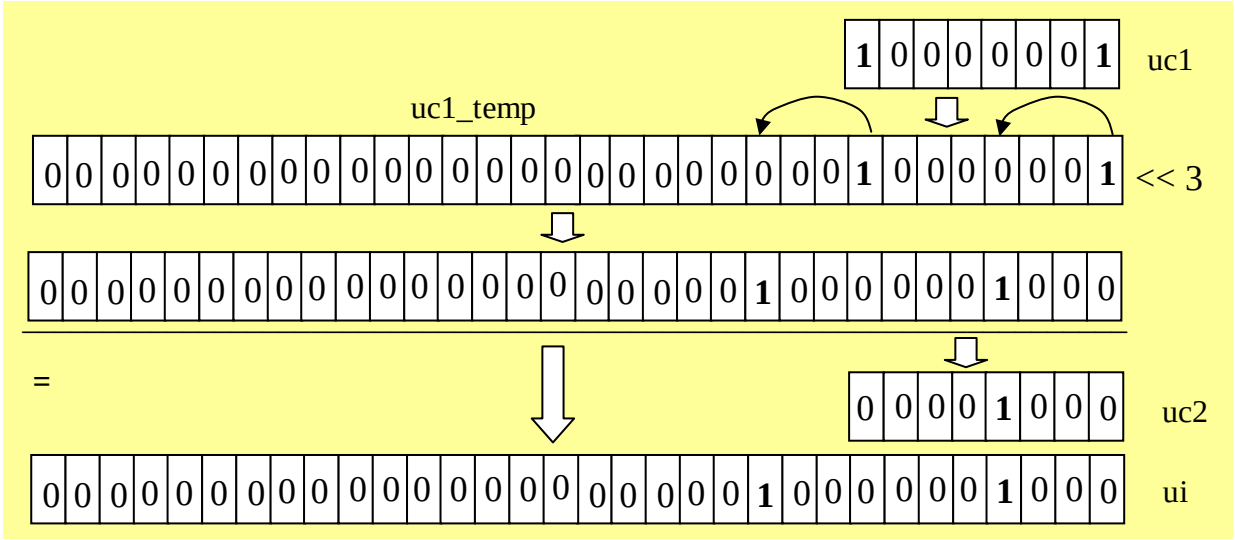
int main(void)
{
    unsigned char uc1, uc2;
    unsigned int  ui;

    uc1 = 0x81;
    uc2 = uc1 << 3;
    ui  = uc1 << 3;
    printf("uc2 = %#.4x\n", uc2);
    printf("ui  = %#.4x\n", ui);

    return 0;
}
```



Нека разгледаме този пример отблизо. Левият операнд на операцията << е uc1, който е от тип char, а десният е литералната константа 3, която е тип int. Съгласно описаните правила левият операнд се повишава до тип int поради обичайните унарни преобразувания, след което се премества с три бита наляво. Полученият резултат е от тип int и се присвоява на uc2, тип char, и на ui, тип int. Фиг. 14 илюстрира описаните по-горе процеси (приема се, че тип int е 32бита).



Фиг. 14 Илюстрация към Пример 4-29

Ако стойността на левия операнд е от беззнаков тип, преместването на ляво с m на брой бита, където $0 \leq m \leq (n - 1)$ е еквивалентно на умножение на левия операнд с 2^m, т.е $n << m \approx n*2^m$.



Ако трябва да умножите един беззнаков целочислен операнд с число 2^n за предпочитане е да използвате операцията преместване наляво << пред операцията умножение *. Това ще подобри производителността на програмата.

Ако стойността на десния операнд е отрицателна, равна или по-голяма от броят на битовете на левия операнд след преобразуването, поведението е недефинирано. Ако стойността на десния операнд е 0, никакви действия не се извършват.

Следващият пример демонстрира два макроса за нулиране и установяване на определен бит на целочислена променлива с помощта на операцията <<.

Пример 4-31:

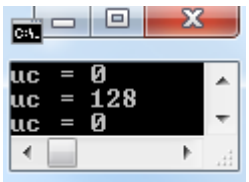
```
#include <stdio.h>

#define setbit(operand, bit)    ((operand) |= (1 << (bit)))
#define clearbit(operand, bit)  ((operand) &= ~(1 << (bit)))

int main(void)
{
    unsigned char uc = 0;

    printf("uc = %u\n",uc); /* Преди установяване на бит 7 */
    setbit(uc,7);
    printf("uc = %u\n",uc); /* След установяване на бит 7  */
    clearbit(uc,7);
    printf("uc = %u\n",uc); /* След нулиране на бит 7      */

    return 0;
}
```



Може да добавите проверка за валидност на параметъра bit, за да подобрите макросите.

Пример:

```
#define setbit(operand, bit)\
if( (bit >= 0) && ((sizeof(operand)*8) > bit) )\
{\
```

```
        ((operand) |= (1 << (bit)));\
}\
else\
{\
        printf("Error: the right operand is too big. Undefined behaviour!\n");\
}

#define clearbit(operand, bit)\
if( (bit >= 0) && ((sizeof(operand)*8) > bit) )\
{\
        ((operand) &= ~(1 << (bit)));\
}\
else\
{\
        printf("Error: the right operand is too big. Undefined behaviour!\n");\
}
```

Тъй като формата на представяне на отрицателните цели числа не е дефиниран от C стандарта, то резултата от преместването наляво на такива числа няма да бъде преносим.

1.2.5.6 Битово преместване надясно >>

Битово преместване надясно

```
операнд1 >> операнд2
```

Двата операнда трябва да са от целочислен тип. Няма връзка между двата операнда, всеки от тях се подлага поотделно на обичайните унарни преобразувания. Типът на левия операнд след преобразуването е и типа на резултата. Резултатът не е L-стойност. Операцията >> премества левия операнд надясно с толкова бита, колкото е указано с десния операнд. При преместване надясно на n-битово цяло число с m на бройт бита, където $0 \leq m \leq (n - 1)$, най-десните m на брой бита се изгубват. Освободените m на брой най-леви бита се попълват съгласно следните правила:

- Ако левият операнд е от беззнаков тип, освободените m на брой най-леви бита се запълват с нули.

- Ако левият операнд е от знаков тип и стойността му е неотрицателна, освободените m на брой най-леви бита се запълват с нули.

- Ако левият операнд е от знаков тип и стойността му е отрицателна, резултатът зависи от имплементацията. Някои компилатори ще попълнят освободените m на брой най-леви бита с нули (логическо преместване на дясно), а други ще ги запълнят със стойността на знаковия бит (знаково преместване надясно).

Ако стойността на левия операнд е от беззнаков тип или е от знаков тип, но има неотрицателна стойност, преместването на дясно с m на брой бита, където $0 \leq m \leq (n - 1)$, е еквиваленто на деление на левия операнд с 2^m, т.е $n >> m \approx n/2^m$

	**Ако трябва да разделите един целочислен беззнаков операнд или целочислен знаков операнд, но с положителна стойност, с число 2^n, за предпочитане е да използвате операцията преместване надясно >> пред операцията деление /. Това ще подобри производителността на програмата и ще избегне проблема, който би възникнал ако десният операнд е 0.**

Ако стойността на десния операнд е отрицателна, равна или по-голяма от броят на битовете на левия операнд след преобразуването, поведението е недефинирано. Ако стойността на десния операнд е 0, никакви действия не се извършват.

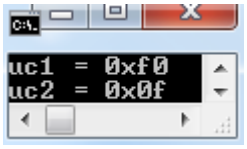
Пример 4-32:

```
#include <stdio.h>

int main(void)
{
    unsigned char uc1, uc2;

    uc1 = 0xF0;
    uc2 = uc1 >> 4;
    printf("uc1 = %#.2x\n", uc1);
    printf("uc2 = %#.2x\n", uc2);

    return 0;
}
```



1.2.6 Операции за присвояване (=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=)

1.2.6.1 Операция за просто присвояване =

Просто присвояване

```
операнд1 = операнд2
```

Стойността на десния операнд се присвоява на левия операнд. Операнд1 трябва да е модифицируема L-стойност. Резултатът от присвояването не е

L-стойност. Преди да се извърши присвояването, типът на десния операнд се преобразува в типа на левия. Ако преобразуването е невъзможно компилаторът ще генерира грешка.

Табл.29 описва позволените типове на операндите на операцията присвояване.

тип операнд1	тип операнд2
аритметичен	аритметичен
struct-/union-	съвместим struct-/union-
указател към T1	указател към T2, където T1 и T2 са съвместими
void*	указател към T, където T е обектен или непълен тип
указател към T, където T е обектен или непълен тип	void*
всеки указател	NULL, където NULL е (void*)0

Табл. 29 Позволени типове на опрандите при присвояване

Допълнителни ограничения върху операндите на операцията =:

- Левият операнд не може да има квалификатор const
- Ако операндите са от аритметични типове, те могат да бъдат квалифицирани или неквалифицирани.
- Ако и двата операнда са указатели към обекти или указатели към функции, те трябва да бъдат квалифицирани или неквалифицирани версии на указатели към съвместими типове, и типът сочен от левия операнд трябва да има всички квалификатори на типа, сочен от десния операнд. Това предпазва указатели тип const int* да бъдат присвоявани на указатели тип int*, което би позволило модифициране на обекта, сочен от const int* указателя.
- Ако единия операнд е квалифицирана или неквалифицирана версия на void* указател, другият операнд трябва да бъде указател към обект или непълен тип. Типът, сочен от левия операнд трябва да има всички квалификатори на типа, сочен от десния операнд. Причината е същата като в предходния случай.

Типът на резултата от присвояването е равен на (непреобразувания и неквалифициран) типа на левия операнд. Резултатът е стойността съхранена в левия операнд.

1.2.6.2 Операции за съставни присвояваня +=, -=, *=, /=,%=, &=, |=, ^=, <<=, >>=

Съставно присвояване

```
операнд1 операция= операнд2

операция: +, -, *, /, %, &, |, ^, <<, >>
```

Тази конструкция е еквивалентна на следното:

операнд1 = операнд1 операция op операнд2

Разликата е, че при използване на съставен оператор за присвояване, операнд1 се изчислява само веднъж от компилатора. Табл.30 описва позволените типове на операндите на операциите за съставно присвояване.

съставен оператор за присвояване	тип операнд1	тип операнд2
*=, /=	аритметичен	аритметичен
%=	целочислен	целочислен
+=, -=	аритметичен	аритметичен
+=, -=	указател	целочислен
<<=, >>=	целочислен	целочислен
&=, |=, ^=	целочислен	целочислен

Табл. 30 Позволени типове на операндите на op=

Левият операнд трябва да бъде модифицируема L-стойност. Двата операнда се подлагат на съответните преобразувания в зависимост от операцията.

Типът на резултата от присвояването е равен на (непреобразувания) типа на левия операнд. Резултатът е стойността съхранена в левия операнд и не е L-стойност.

Всички ограничения при простото присвояване са валидни и за съставните операции за присвояване.

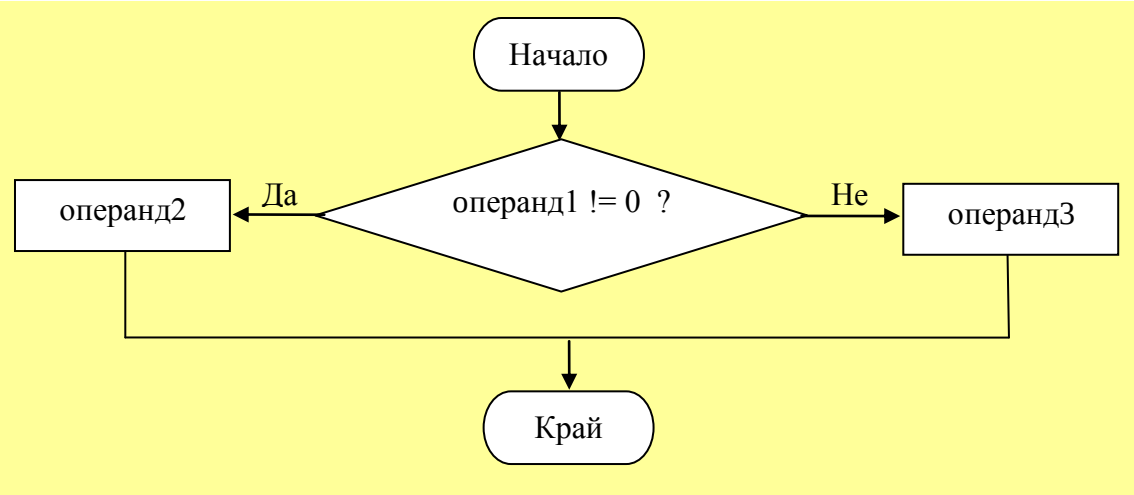
1.2.7 Условна операция ?:

Условна операция

```
операнд1 ? операнд2 : операнд3
```

Това е единствената тринарна (с три операнда) операция в C. Ако

стойността на операнд1 е различна от нула, се изчислява операнд2 и неговата стойност става резултат от цялата операция, в противен случай се изчислява операнд3 и неговата стойност става резултат от цялата операция (Фиг.15).



Фиг. 15 Условна операция ?:

Резултатът не е L-стойност и неговия тип се определя по описаните по-долу правила.

Операнд1 трябва да бъде от скаларен тип. Вторият и третият операнд могат да са от различни типове и се подлагат на обичайните унарни преобразувания. Типът на резултата зависи от типа на втория и третия операнд (Табл.31).

тип операнд2	тип операнд3	тип на резултата
аритметичен	аритметичен	Типа на операндите след обичайните бинарни преобразувания
структура	съвместима структура	структура
обединение	съвместимо обединение	обединение
void	void	void
Указател към квалифициран или неквалифициран тип T1	Указател към квалифициран или неквалифициран тип T2, ако T1 и T2 са съвместими	Указател към тип, имащ всичките квалификатори на типовете T1 и T2
Указател към обект или непълен тип	Квалифициран или неквалифициран void*	void*, където сочения тип има всичките квалификатори на типовете, сочени от двата указателя
Указател към T, където T може да е от всеки тип	Нулев константен указател	Указател към T

Табл. 31 Типове на операнд2 и операнд3

Пример:

операнд2 (операнд3)	операнд3 (операнд2)	тип на резултата
const void*	const int*	const void*
volatile int*	0	volatile int*
const int*	volatile int*	const volatile int*
void*	const char*	const void*
int*	const int*	const int*
void*	int*	void*

1.2.8 Операция за последователно изчисляване на изрази ,

В С символът запетя играе роля на пунктуатор в един контекст (например в списъка с аргументи на функция), и на операция за последователно изчисление на изрази в друг.

Операция за последователно изчисляване на изрази

```
операнд1, операнд2,.....,операндN
```

Операндите не е задължително да имат някаква връзка помежду си. Всеки операнд може да бъде произволен израз, от обикновена променлива до сложен израз, също и израз, който не връща стойност (void израз). Изразите се изчисляват от ляво на дясно и резултатът от изчислението на последния (най-десния) израз става резултат на целия израз. Ако операндите вляво от последния израз връщат някаква стойност, то тя просто се губи. Например изразът

```
r = (a,b,c);
```

е еквивалентен на

```
a;
b;
r = c;
```

Скобите в показания пример са много важни тъй като операцията ',' има най-нисък приоритет от всички операции.

Много е важно да се запомни, че само в контекст където символът ',' играе роля на операция, изразите се изчисляват последователно от ляво на дясно. В контекст където символът ',' играе ролята на пунктуация, редът на изчисление на изразите, разделени със запетая не е дефиниран от

стандарта. Например в извикването на функцията **f(a,b,c);** запетаята играе ролята на пунктуатор (разделител) и редът на изчисление на аргументите a,b и c е недефиниран. Всеки един от аргументите a, b и c обаче може да бъде последователност от изрази, разделени със запетая. Например:

f((a = 2, 2*a), b,c);

Първият аргумент на функцията f() е израза **(a = 2, 2*a)**. Обърнете внимание на скобите. Те информират компилатора, че това е един аргумент, състоящ се от изрази, разделени със запетая. Съгласно описаните правила, стойността на първия аргумент се изчислява до 4. Ако скобите се пропуснат получения израз f(a=2,2*a,b,c); ще се интерпретира от компилатора като извикване на функция с 4 аргумента вместо с 3, което ще доведе до генериране на грешка при компилиране (при условие, че функцията е покрита с прототип).

Пример 4-33:

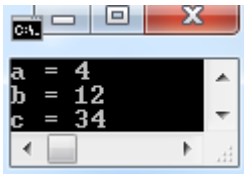
```
#include <stdio.h>

void f(int a, int b, int c);

int main(void)
{
	int i;

	f((i = 2, 2*i), 12, 34);

	return 0;
}
void f(int a, int b, int c)
{
	printf("a = %d\n",a);
	printf("b = %d\n",b);
	printf("c = %d\n",c);
}
```



Операцията ',' може да си използва за формирането на странни програмни конструкции като в следващия пример.

Пример 4-34:

```
#include <stdio.h>

void f(void);
void f1(void);

int main(void)
```

```
{
    f(), f1(); /* същото като */
               /* f();        */
               /* f1();       */

    return 0;
}

void f(void)
{
    printf("Hello ");
}

void f1(void)
{
    printf("C\n");
}
```



1.2.9 Операция sizeof за определяна размера на тип

Операция sizeof

```
sizeof (тип)
sizeof израз
```

Операндът може да бъде име на тип или израз. Ако операндът е тип той трябва да бъде сложен в скоби, докато ако е израз, скобите не са задължителни. Резултатът от операцията е размера на операнда в байтове. И в двата случая, размерът се определя от типа на операнда. Резултатът не е L-стойност и е от тип **size_t**, дефиниран в хедър файла <stddef.h>.

Операцията sizeof има следните особености:

- Ако операндът е тип char, unsigned char, signed char (или техните квалифицирани версии), резултатът е 1.
- Операндът не може да бъде непълен тип, име на функция, битово поле или типът void.
- Резултатът от sizeof(израз) е същия като sizeof(тип), където тип е типа на израза. Операцията sizeof не води до някакви изчисления на израза, когато определя неговия тип. Но ако изразът съдържа операции, които изискват обичайни преобразувания на типа на операндите си, това се отчита при определяне на типа на израза.

- sizeof(име-масив) връща броя на байтовете използвани от масива, т.е. тази операция е еквивалентна на израза **N*sizeof(тип)**, където **N** е броя на елементите на масива, а **тип** е типа на елементите.

- sizeof(име-структура) връща броя на байтовете, заемани от членовете на структурата, включително и уплътняващите байтове.

- Резултатът на операцията sizeof се изчислява по време на компилиране, а не по време на изпълнение на програмата. Ако операндът е израз, той не се изчислява, а само се определя неговия тип.

Пример 4-35:

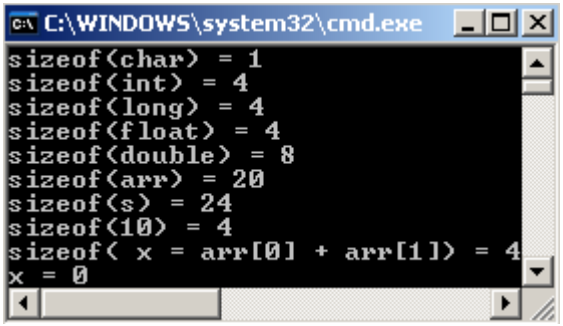
```
#include <stdio.h>

struct mystruct
{
	int    i;
	double d;
	char c1;
	char c2;
};

int main(void)
{
	int                arr[5] = {1,2,3,4,5};
	struct mystruct    s      = {100, 3.14};
	int                x      = 0;

	printf("sizeof(char) = %d\n", sizeof(char));
	printf("sizeof(int) = %d\n", sizeof(int));
	printf("sizeof(long) = %d\n", sizeof(long));
	printf("sizeof(float) = %d\n", sizeof(float));
	printf("sizeof(double) = %d\n", sizeof(double));
	printf("sizeof(arr) = %d\n", sizeof(arr));
	printf("sizeof(s) = %d\n", sizeof(s));
	printf("sizeof(10) = %d\n", sizeof(10));
	printf("sizeof( x = arr[0] + arr[1]) = %d\n", sizeof( x = arr[0] + arr[1]));
	printf("x = %d\n", x);

    return 0;
}
```



Както се вижда от изхода на програмата, операцията sizeof, приложена

върху базовите типове, връща броя байтове заемани от тях. С изключение на тип char, размерът на всички останали базови типове зависи от компилатора и архитектурата на компютъра. Резултатът от sizeof(arr) е 20. Това е така, защото масива arr има 5 елемента тип int, а за дадения компилатор тип int заема 4 байта, т.е. N*sizeof(int) = 5*4 = 20. Резултатът от sizeof(s) е 16. Структурната променлива s има два члена, i тип int и d тип double. Общата памет заемана от тези променливи е 12 байта за дадения компилатор, но резултата е 16. Това се дължи на уплътняващите байтове, добавени при разполагане на членовете на стурктурата в паметта. Резултатът от sizeof(10) е 4. Целочислената константа 10 е от тип int, а за дадения компилатор, тип int заема 4 байта. Резултата от sizeof (x = arr[0] + arr[1]) е 4. Типа на израза x = arr[0] + arr[1] е int (тъй като x е тип int), а за дадения компилатор, тип int заема 4 байта. На пръв поглед, може да останете с впечатление, че на променливата x се присвоява стойността на израза arr[0] + arr[1], т.е. 3. Но както се вижда от изхода на програмата x е 0. Както казахме по-рано операцията sizeof не извършва никакви изчисления когато операнда е израз, а само определя типа на резултата на този израз и връща броя байтове заемани от този тип.

Следващата програма няма да се компилира, тъй като използваните операнди са непозволени.

Пример 4-36:

```
#include <stdio.h>
#include <stddef.h>

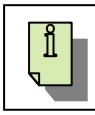
extern int arr[];

struct mystruct
{
    int b0 : 3;
    int b1 : 3;
    int b2 : 2;
};

void func(void)
{
      /* Потребителски код */
}

int main(void)
{
    size_t               i;
    struct mystruct bit_fields;

    printf("sizeof(arr) = %d\n", sizeof(arr)); /* Грешка, arr е от непълен тип */
    i = sizeof(void); /* Грешка, тип void няма размер */
    i = sizeof(func); /* Грешка, име на функция не може да се използва като
                         операнд */
    i = sizeof(bit_fields.b0); /* Грешка,  битово поле не може да се използва като
                                  операнд  */
    return 0;
}
```



Резултатът от операцията sizeof се изчислява по време на компилиране, а не по време наизпълнение на програмата.

1.2.10 Операция за преобразуване на тип (тип)

Преобразуване на тип

```
(тип)операнд
```

Операндът може да бъде само от скаларен тип. Операндът се преобразува към зададения в скоби тип.

Пример 4-37:

```
#include <stdio.h>

int main(void)
{
	int x;

	x = (int)12.3;
	printf("x = %d\n", x);

	return 0;
}
```



Следните особености трябва да се имат предвид когато се извършват преобразувания от и към указателни типове:

- Един указател, може да бъде преобразуван в целочислен тип. Резултатът зависи от имплементацията. Ако размерът на целочисления тип не е достатъчно голям да побере стойността на указателя, резултатът е недефиниран.

Пример 4-38:

```
#include <stdio.h>

int main(void)
{
	int x;
	unsigned long ul;
	int* p = &x;

	ul = (unsigned long)p;
```

```
	printf("&x = %p\n", p);
	printf("ul = %#x\n", ul);

	return 0;
}
```



- Произволно цяло число може да бъде преобразувано в указател. Резултатът зависи от имплементацията.

Пример:

```
int* p;
p = (int*)1000;
```

- Указател към обектен или непълен тип може да бъде преобразуван в указател към друг обектен или непълен тип. Резултатният указател може да бъде невалиден ако подравняването на типа, сочен от резултатния указател и подравняването на типа, сочен от преобразувания указател е несъвместимо.

Пример 4-39:

```
#include <stdio.h>

int main(void)
{
	int* pi;
	char c, *pc = &c;

	pi = (int*)pc;
	*pi = 100; /* Възможно е да се генерира изключение */
	printf("c = %d\n", c);

	return 0;
}
```

В зависимост от хардуерната платформа изпълнението на този пример може да доведе до генериране на изключение при изпълнение на конструкцията *p = 100;. Причината, е че char-променливи нямат изискване за подравняване, те могат да се разполагат на всякакви адреси, докато някои хардуерни платформи имат изискването многобайтовите променливи да се разполагат само на адреси кратни на техния размер. Ако това условие не е изпълнено, всеки опит за достъп до неправилно разположена (неподравнена) в паметта променлива ще доведе до генериране на изключение.

1.2.11 Операции за достъп до членове на структура и обединение (., ->)

1.2.11.1 Операция за директен достъп .

Директен достъп до член на структура/обединение

```
struct-променлива . име-член
union-променлива  . име-член
```

Резултатът от тази операция е член на структурата/обединението и е L-стойност.

Пример 4-40:

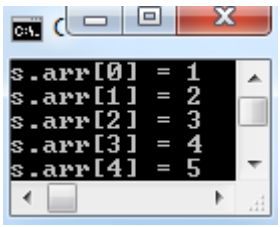
```
#include <stdio.h>

struct mystruct
{
	int i;
	int arr[5];
};

int main(void)
{
	/* Дефиниране на struct-променлива s */
	struct mystruct s = {
				0,		/* Инициализатор на члена i   */
				{1,2,3,4,5} /* Инициализатор на члена arr */
			};

	for( ; s.i < 5 ; s.i++)
	{
		printf("s.arr[%d] = %d\n", s.i , s.arr[s.i]);
	}

	return 0;
}
```



Пример 4-41:

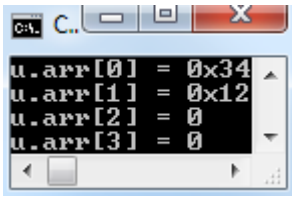
```
#include <stdio.h>

union myunion
{
	int i;
	unsigned char arr[sizeof(int)];
};
```

```
int main(void)
{
	/* Дефиниране на struct-променлива s */
	union myunion u = { 0x1234 /* Инициализатор на члена i */ };
	int counter;

	for (counter = 0 ; counter < sizeof(int) ; counter++)
	{
		printf("u.arr[%d] = %#x\n", counter, u.arr[counter]);
	}

	return 0;
}
```



Структурите са разгледани подробно в Глава 10 Структури. Обединенията са разгледани подробно в Глава 11 Обединения.

1.2.11.2 Операция за косвен достъп ->

Косвен достъп до член на структура/обединение

```
указател-към-struct-променлива -> име-член
указател-към-union-променлива  -> име-член
```

Резултатът от тази операция е член на структурата/обединение и е L-стойност.

Пример 4-42:

```
#include <stdio.h>

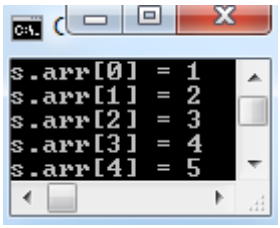
struct mystruct
{
	int i;
	int arr[5];
};

int main(void)
{
	/* Дефиниране на struct-променлива s */
	struct mystruct s = {
				0,		/* Инициализатор на члена i   */
				{1,2,3,4,5}	/* Инициализатор на члена arr */
			};
	struct mystruct *ps = &s;

	for( ; ps->i < 5 ; ps->i++)
```

```
	{
		printf("s.arr[%d] = %d\n", ps->i , s.arr[ps->i]);
	}

	return 0;
}
```



Пример 4-43:

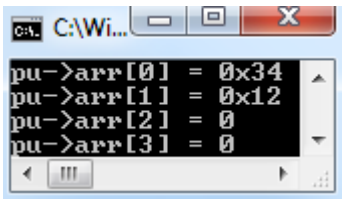
```
#include <stdio.h>

union myunion
{
	int i;
	unsigned char arr[sizeof(int)];
};

int main(void)
{
	/* Дефиниране на union-променлива u */
	union myunion u = {0x1234}; /* Инициализатор на члена i */
	/* Дефиниране на указател към union-променлива */
	union myunion *pu = &u;
	int counter;

	for (counter = 0 ; counter < sizeof(int) ; counter++)
	{
		printf("pu->arr[%d] = %#x\n", counter, pu->arr[counter]);
	}

	return 0;
}
```



Структурите са разгледани подробно в Глава 10 Структури. Обединенията са разгледани подробно в Глава 11 Обединения.

1.2.12 Операция за индексиране на масив [ ]

Индексиране на масив

```
име-масив[индекс]
```

Резултатът от тази операция е елемента на масива, указан с индекса и е L-стойност. Индексът може да бъде всеки целочислен израз. Отговорност на програмистта е да следи индексът да не излиза извън рамките на масива.

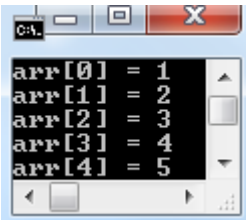
Пример 4-44:

```
#include <stdio.h>

int main(void)
{
	int index;
	int arr[5] = {1,2,3,4,5};

	for(index = 0; index < 5 ; index++)
	{
		printf("arr[%d] = %d\n", index , arr[index]);
	}

	return 0;
}
```



Конструкцията **име-масив[индекс]** се преобразува от компилатора в ***(име-масив + индекс)**, т.е **име-масив[индекс] ≈ *(име-масив + индекс).** Обичайните бинарни преобразувания се прилагат върху двата операнда.

Въпреки, че не е много очеивдно, следващата конструкция е напълно валидна: **индекс[име-масив].**

Масивите са разгледани подробно в Глава 8 Масиви.

1.2.13 Операция за извикване на функция ()

Извикване на функция

```
име-функция(списък-аргументи)

или

(*указател-към-функция)( списък-аргументи)
```

Резултатът от извикването на функцията има някакъв тип T и никога не е L-стойност. Ако T е void, функцията не връща стойност и не може да се използва в изрази, изискващи стойност. Функциите са разгледани

подробно в Глава 13 Функции.

1.2.14 Указателни операции *, &

1.2.14.1 Операция за извличане на адрес &

Извличане на адрес

&операнд

Операндът може да бъде или име на функция или L-стойност, обозначаваща обект. Резултатът от тази операция е адреса на операнда и не е L-стойност. Тъй като име на функция автоматично се преобразува от компилатора в указател към функцията, то тази операция е излишна когато операндът е име на функция.

Пример 4-45:

```
#include <stdio.h>

void func(void)
{
	printf("Hello C\n");
}

int main(void)
{
	int   x;
	int  *px;
	void (*pfunc)(void);

	px = &x; /* Адресът на x се присвоява на указателя px */
	*px = 10; /* На x се присвоява 10 косвено чрез указателя */
	/* Адресът на функцията func се присвоява на указателя pfunc  */
	pfunc = &func; /* Същото като pfunc = func */
	/* Извикване на func косвено чрез указателя */
	(*pfunc)(); /* Същото като pfunc() и func() */

	return 0;
}
```



Адресите на register-обекти и битови полета на могат да се извличат.

Пример 4-46:

```
#include <stdio.h>
```

```
struct mystruct
{
	int	b0:3;
	int	b1:5;
};

int main(void)
{
	register int x;
	int *px;
	struct mystruct	s;
	int *p_bitField;

	px = &x;/* !!!Грешка, адресът на register обект е недостъпен */
	p_bitField = &s.b0; /* !!!Грешка, адресът на битово поле е недостъпен */

	return 0;
}
```

Операцията за извличане на адрес е разгледана подробно в Глава 9 Указатели.

1.2.14.2 Операция за косвен достъп *

Тази операция се нарича още дереференциране на указател.

Дереференциране на указател

```
*име-указател
```

Операндът може да бъде указател към обект или функция. В първия случай резултатът от операцията е обекта и е L-стойност. Когато операндът е указател към функция, резултатът е името на функцията.

Пример: Вижте примерите по горе.

Ако операндът е нулев указател резултатът е недефиниран.

Пример 4-47:

```
#include <stdio.h>

void (*pfunc)(void) = NULL;
int *px = NULL;

int main(void)
{
	*px = 10; /* !!!Грешка, указателят px не сочи към валиден обект */

	(*pfunc)(); /* !!!Грешка, указателят pfunc не сочи към валидна функция */

	return 0;
}
```

Операцията за косвен достъп е разгледана подробно в Глава 9 Указатели и Глава 13 Функции.

1.3 Ред на изчисление на операндите на бинарни и тринарни операции

Единствените операции, за които стандартът С ясно дефинира реда на изчисление на операндите са '**&&**', '||', '**?:**' и '**,**'. При тези операции изчислението винаги започва от левия операнд. За всички останали бинарни операции редът на изчисление на операндите не е дефиниран. Например за операцията събиране '+', умножение '*' и т.н. първи може да се изчисли както левия така и десния операнд. Код, който разчита на определен ред на изчисление на операндите на дадена операция е непреносим.

Пример 4-48:

```
#include <stdio.h>

static int x = 5;

int f1(void)
{
	return x;
}

int f2(void)
{
	x++;
	return x;
}

int main(void)
{
	int res;

	res = f1() + f2();
	printf("res = %d\n", res);

	return 0;
}
```

```
Резултат: 11 или 12
```

Различните компилатори могат да дадат различен резултат.Той може да бъде или 11 (първо се извиква f1(), след това f2()) или 12 (първо се извиква f2(), след това f1()).



Не създавайте изрази, които разчитат на определен ред на изчисление на операндите. Такъв код ще намали преносимостта на програмата от една компютърна система на друга.

1.4 Изрази

Един израз е всяка валидна комбинация от константи, променливи, функции и операции. Всеки израз има тип. Типът на израз е типа на стойността, получена след изчисляване на израза. Ако един израз не връща стойност, неговия тип е void. Един израз може да даде стойност, да породи **странични ефекти(side effects)** (вижте точка 14 Странични ефекти и точки на последователност) или и двете. Табл. 32 съдържа примери за изрази.

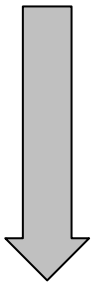
израз	тип	коментар
5	int	Целочислена константа
i + 1	int	Целочислен израз
d + 1.0	double	Израз с плаваща запетая
b < 55.5	int	Израз за отношение
"A string literal"	char*	Резултатът е указател към първия символ на низа
abort()	void	Не връща стойност
sqrt(2.0)	double	Връща стойност тип double

Табл. 32 Пример за изрази

Една константа, низов литерал, име на променлива, извикване на функция представляват сами по себе си изрази. Обобщено такива изрази се наричат прости изрази. Простите изрази в комбинация с операции формират по-сложни изрази. Когато израз завършва с точка и запетая, той се превръща в операторен израз (вижте Глава 6 Оператори).

1.5 Приоритет и асоциативност на операциите

Всяка операция в C има приоритет и асоциативност. Приоритетът указва, коя операция се извършва първа измежду операции с различен приоритет, а асоциативността указва коя операция се извършва първа измежду операции с еднакъв приоритет. Табл.33 показва приоритета и асоциативността на всички C операции.

операция	описание	асоциативност	приоритет
a[i] **f(…)** **.** **->** **x++, x--**	индексиране на масив извикване на функция директен достъп до struct-/union- членове косвен достъп до struct-/union- членове постинкремент, постдекремент	отляво-надясно	най-висок 
++x, --x **sizeof** **~** **!** **-** **+**	преинкремент, предекремент размер на обект битово отрицание логическо отрицание унарен минус унарен плюс	**отдясно-наляво**	

& ***** **(тип)**	извличане на адрес косвен достъп до обект преобразуване на тип	
*** / %**	умножение, деление, деление по модул	отляво-надясно
+ -	събиране, изваждане	отляво-надясно
<< >>	битово преместване на ляво и дясно	отляво-надясно
< <= > >=	оператори за сравнение	отляво-надясно
== !=	оператори за равенство	отляво-надясно
&	битово И	отляво-надясно
^	битово Изкл. ИЛИ	отляво-надясно
|	битово ИЛИ	отляво-надясно
&&	логическо И	отляво-надясно
||	логическо ИЛИ	отляво-надясно
?:	условен оператор	**отдясно-наляво**
= **+= -=** ***= /= %=** **<<= >>=** **&= |= ^=**	просто присвояване, съставни присвоявания	**отдясно-наляво**
,	последователно изчисление	отляво-надясно

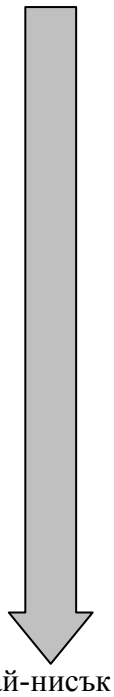


Табл. 33 Приоритети на операциите

Когато в един израз има повече от една операция, изчислението започва винаги от операцията с най-висок приоритет. Например в израза

```
5 * 2 + 3
 \  /    |
  10  +  3
   \     /
     13
```

операцията умножение е с по-висок приоритет от операцията събиране и се извършва първа.

Ако изразът включва няколко операции с еднакъв приоритет или няколко еднакви операции, редът на изчислението им се определя от асоциативността на операторите. Асоциативност отляво-надясно означава, че изчислението започва от най-левия оператор. Например в израза

```
5 + 3 + 4
 \  /    |
  8  +  4
   \     /
     12
```

първо се изпълнява левия + и след това десния.

Асоциативност отдясно-наляво означава, че изчислението започва от най-

десния оператор. Например в израза

```
a = b = c
|     \   /
a       b
 \     /
  a = b
```

първо се изпълнява операцията b = c, а след това a = b.

Редът на изчисление на операциите може да се променя с помощта на скоби. Израз в скоби се изчислява първи.

Пример:

```
5 * (2 + 3)          5 + (3 + 4)
|     \   /          |     \   /
 5   *  5            5   +  7
   \   /              \     /
    25                  12
```

Както се вижда от горните примери в стойността на един израз може да зависи от реда на изчисление на операциите в него.

Ако израз в скоби съдържа друг израз в скоби, то вътрешния израз се изчислява първи.

Пример:

```
(5 * (4 + 3)) - 2
 |     \   /     |
 5    *   7   -   2
  \       /        |
     35     -      2
       \          /
            33
```

1.6 Странични ефекти и точки на последователност

1.6.1 Странични ефекти (side effects)

Един израз освен, че може да даде някаква стойност като резултат, може и да породи някакви изменения, наречени **странични ефекти**. Пример за такива ефекти са промяна на стойността на променлива и т.н. Например в

следващия фрагмент

```
int x = 5;
int y;

y = x++;
```

изразът x++ ще върне като резултат 5 и ще увеличи x с 1 като страничен ефект.

1.6.2 Точки на последователност (sequence points)

Точка на последователност е специална точка от програмата, в която всички странични ефекти, които могат да се породят при последователното изчислението на един израз, са приключили, а страничните ефекти, които могат да се породят от следващия израз още не са се изпълнили. Ако един обект се променя повече от веднъж между две точки на последователност, резултатът от изчислението на израза, в който обекта участва, е недефиниран: Например:

```
int i = 1;
i = i++;   /* !!!Неправилно */
```

Резултатът от втория израз е неопределен. Възможни са два сценария:

- Извлича се стойността на на израза i++, в случая 1, тази стойност се присвоява на i, и след това i се увеличава с 1. Крайния резултат е i = 2.

- Извлича се стойността на на израза i++, в случая 1, след това i се увеличава с 1 и става 2, и най-накрая на i се присвоява 1. Крайният резултат е i = 1.

Стандартът C дефинира следните точки на последователност:

- След изчисление на всички аргументи при извикване на функция

Всички странични ефекти, които могат да се породят при изчислението на аргументите при извикване на функция ще бъдат изпълнени преди управлението на програмата да се предаде на функцията.

Пример 4-49:

```
#include <stdio.h>

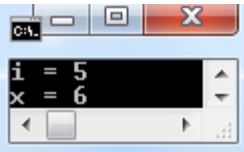
static int x = 5;
```

```
void foo(int i)
{
	printf("i = %d\n", i);
	printf("x = %d\n", x);
}

int main(void)
{
    foo(x++);

    return 0;
}
```

Точка на последователност



Преди да се влезе в тялото на функцията, всички странични ефекти при изчислението на аргумента са приключили. В случая стойността на x се увеличава с 1 преди управлението на програмата да се предаде на функцията. Всяка употреба на x във функцията ще използва стойност 6 както показва и резулата от извикване на foo().

- Края на първия операнд на операциите '&&', '||', '?:' и ','

Пример 4-50:

```
#include <stdio.h>

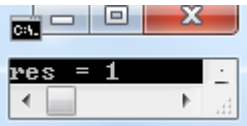
static int x = 5;

int main(void)
{
	int res;

	res = (0 != x++) && (6 == x);
	printf("res = %d\n", res);

	return 0;
}
```

Точка на последователност



Преди да се изчисли десния операнд на операцията &&, всички странични ефекти при изчисление на левия са приключили, т.е x е увеличена с 1.

- В края на всеки пълен израз. Това включва:
  - край на инициализатор (на всякакви обекти)
  - край на операторен израз

- край на управляващ израз на if, switch, while, do-while и всички изрази на for

- край на return-израз

Пример 4-51:

```
#include <stdio.h>

static int x = 5;

int foo(void)
{
	return x++;
}

int main(void)
{
	int a[] = {1,2,3};
	int y;

	if (a[0]++)
	{
		printf("a[0] = %d\n", a[0]);
	}

	y = a[1] + a[2]++;
	printf("a[2] = %d\n", a[2]);

	x = foo();
	printf("x = %d\n", x);

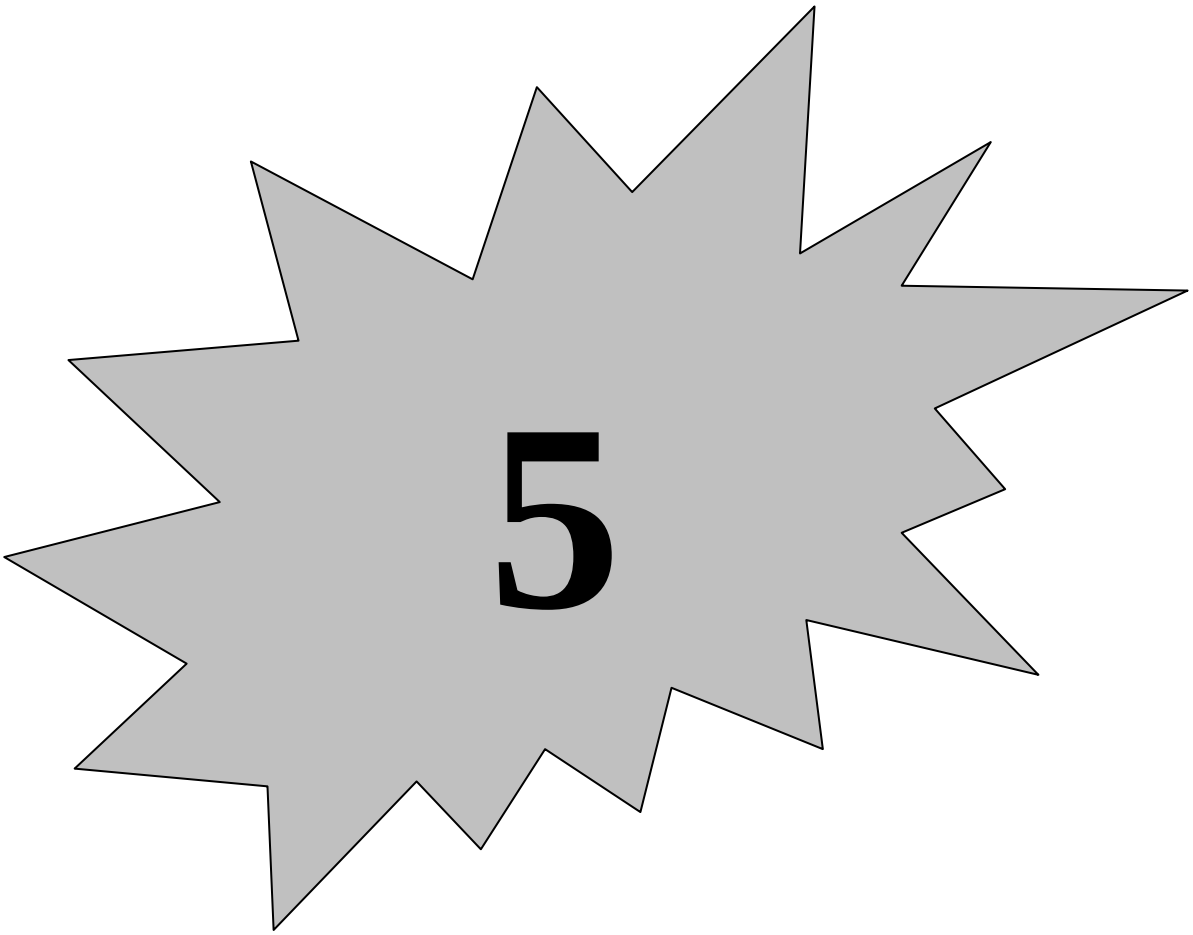
	return 0;
}
```

Край на return-израз

Край на инициализатор

Край на управляващ израз на if

Край на операторен израз



Преобразувания

1.1 Въведение

Една част от преобразуванията са ясно дефинирани в езика C, друга част зависи от имплементацията. В тази глава се разглеждат подробно всички тези правила за преобразуване на типове.

1.2 Целочислено повишение на типовете

Стойности от следните типове

- char (signed или unsigned)
- short int (signed или unsigned)
- битово поле тип int (signed или unsigned)
- enum-типове

могат да се използват навсякъде където могат да се използват стойности от тип int или unsigned int. Ако стойността на типа може да се представи от тип int, то тя се преобразува автоматично в тип int. В противен случай се преобразува в тип unsigned int. Това преобразуване се нарича целочислено повишаване. Табл.34 показва, кои от гореизброените типове се преобразуват в int и кои в unsigned int.

тип	int	unsigned int
signed char	●	
unsigned char	●	
signed short	●	
unsigned short[1]	●	●
битово поле int[2]	●	●
битово поле signed int	●	
битово поле unsigned int		●
enum-тип[3]	●	

Табл. 34 Правила за целочислено повишаване

Забележка[1]: Тип unsigned short се преобразува в тип int когато размерът на тип int е 32-бита и в unsinged int когато размерът на тип int е 16-бита.

Забележка[2]: Битово поле дефинирано само като int може да бъде както signed така и unsigned. Това зависи от имплеметацията.

Забележка[3]: Типът на enum-променливите е имплементационно зависим. В зависимост

от стойностите на enum-константите компилаторът може да избере най-подходящия от целочислените типове char (signed, unsigned), short (signed, unsigned) или int. Ето защо всички стойности на един enum-тип могат да се представят винаги от тип int.

Целочисленото повишаване на типа е част от т.нар. обичайни унарни преобразувания (вижте 8.3 Обичайни унарни преобразувания).

Пример 5-1:

```
#include <stdio.h>

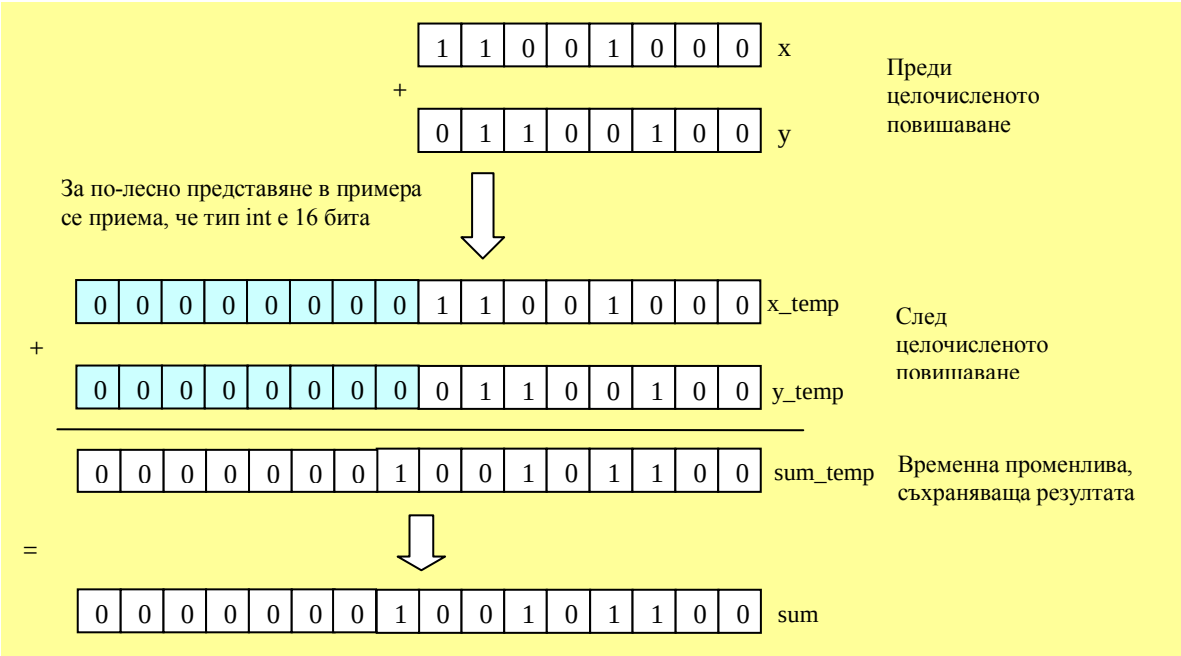
int main( void )
{
	unsigned char	x = 200, y = 100;
	int sum;

	sum = x + y;
	printf("sum = %d\n", sum);

	return 0;
}
```



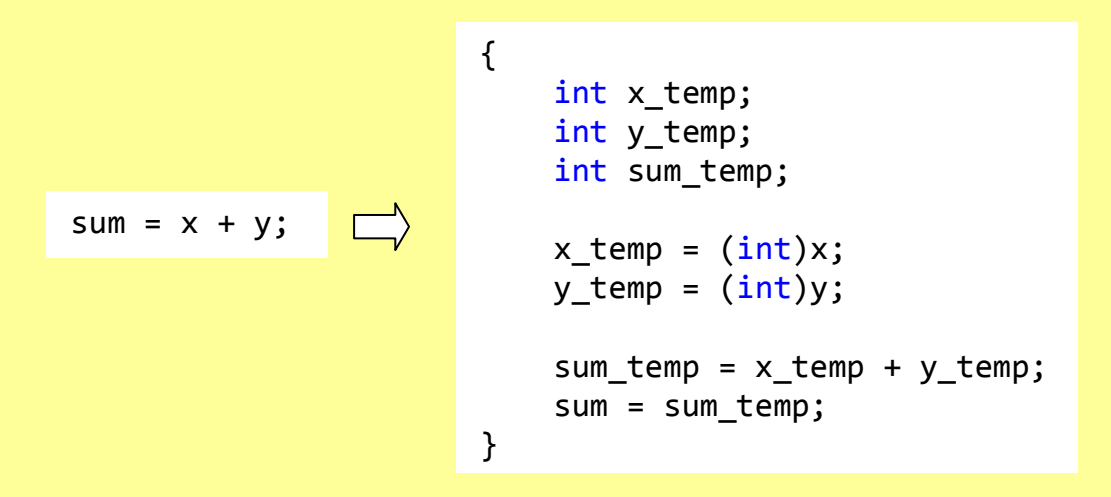
Фиг.16 показва вътрешното представяне на израза **sum = x + y**.



Фиг. 16 Вътрешно представяне на израза sum = x + y

Обърнете внимание, че когато променлива от един тип се преобразува в променлива от друг тип, оригиналната променлива не се променя, а компилаторът създава временна променлива, представляваща резултата от

преобразуването. Именно тази временна променлива се използва в израза, от който оригиналната променлива е част. Фиг.17 представя еквивалентен код, описващ израза sum = x + y.

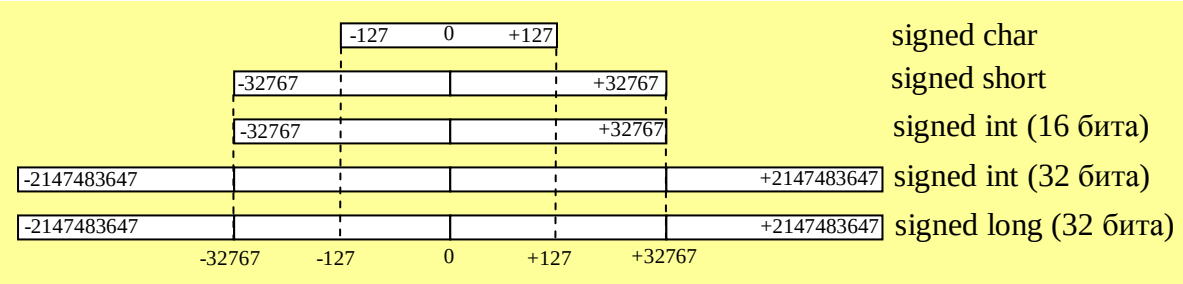


Фиг. 17 Еквивалентен код на израза sum = x + y

1.3 Преобразувания между целочислени типове

1.3.1 Преобразуване на знакови типове

Фиг.18 показва минималните стойностите, които могат да се представят от различните знакови целочислени типове.



Фиг. 18 Диапазон от стойности на знаковите целочислени типове

1.3.1.1 Преобразуване на знаков тип в друг знаков тип със същия или по-голям размер

Възможните комбинации на преобразуване са показани в Табл. 35.

преобразуван тип	резултатен тип
signed char	signed short signed int signed long

signed short	signed int (16 или 32 бита) signed long
signed int (16 бита)	signed short signed long
signed int (32 бита)	signed long

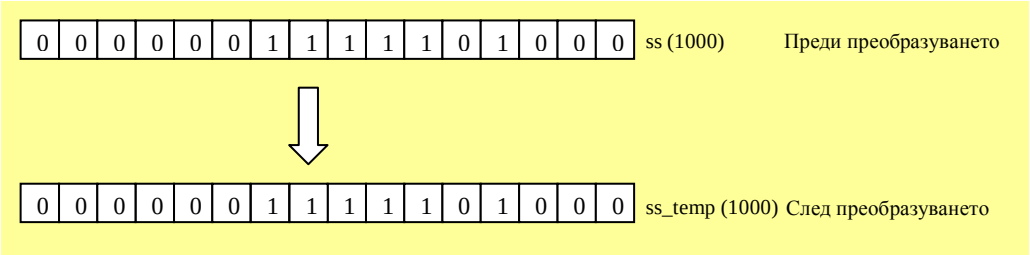
Табл. 35 Комбинации на преобразуване на знакови типове в други знакови типове с еднакъв или по-голям размер

При това преобразуване резултатният тип винаги може да представи всички стойности на преобразувания тип.

Ако стойността на преобразувания тип е неотрицателна и резултатният тип е със същия размер, преобразуването не води до вътрешно изменение на стойността.

Пример: `signed short -> signed int (16 бита)`

```
signed short  ss = 1000;
(signed int)ss
```

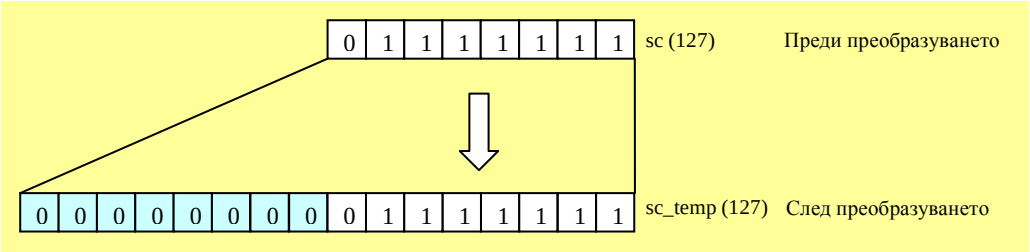


Фиг. 19 Преобразуване на неотрицателна стойност signed short в signed int (16 бита)

Ако резултатният тип е по-голям, старшите битове просто се запълват с нули.

Пример: `signed char -> signed short`

```
signed char   sc = 127;
(signed short)sc
```



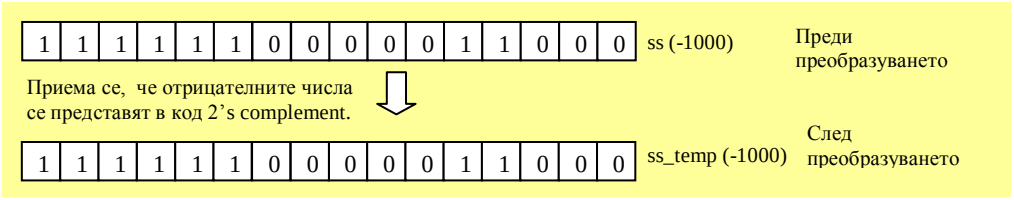
Фиг. 20 Преобразуване на неотрицателна стойност signed char в signed short

Ако стойността на преобразувания тип е отрицателна и резултатният тип е със същия размер, преобразуването не води до изменение на вътрешното

представяне на стойността.

Пример: `signed short -> signed int (16 бита)`

```
signed short  ss = -1000;
(signed int)ss
```



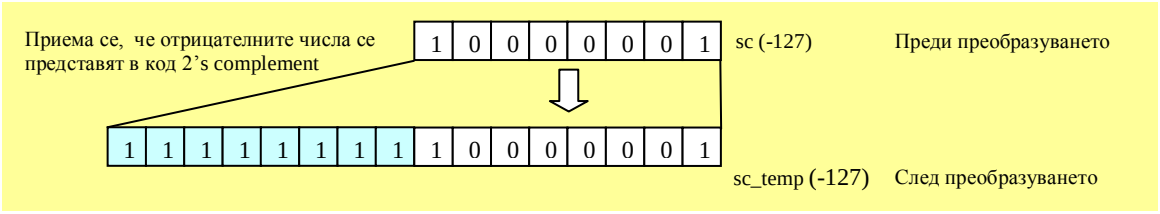
Фиг. 21 Преобразуване на отрицателна стойност signed short в signed int (16 бита)

Ако резултатният тип е с по-голям размер, стойността на преобразувания тип просто се преобразува в съответния еквивалент на резултатния тип[1].

Забележка[1]: Ако отрицателните числа се представят в код 2's complement преобразуването се състои просто в запълване на старшите битове на резултатния тип със знаковия бит на преобразувания тип.

Пример: `signed char -> signed short`

```
signed char   sc = -127;
(signed short)sc
```



Фиг. 22 Преобразуване на отрицателна стойност signed char в signed short

1.3.1.2 Преобразуване на знаков тип в друг знаков тип с по-малък размер

Възможните комбинации на преобразуване са показани в Табл.36.

преобразуван тип	резултатен тип
signed short	signed char
signed int (32 бита)	signed char signed short
signed long	signed char

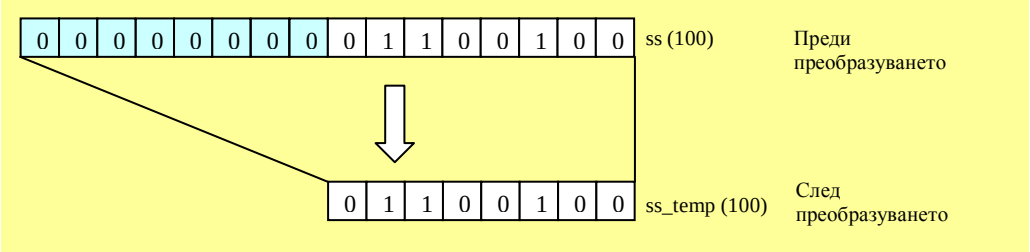
	signed short signed int (16 бита)

Табл. 36 Комбинации на преобразуване на знакови целочислени типове в други знакови целочислени типове с по-малък размер

Ако стойността на преобразувания тип е неотрицателна и резултатният тип може да представи стойността на преобразувания тип, преобразуването се извършва просто чрез изхвърляне на старшите байтове на преобразувания тип.

Пример: `signed short -> signed char`

```
signed short  ss = 100;
(signed char)ss
```



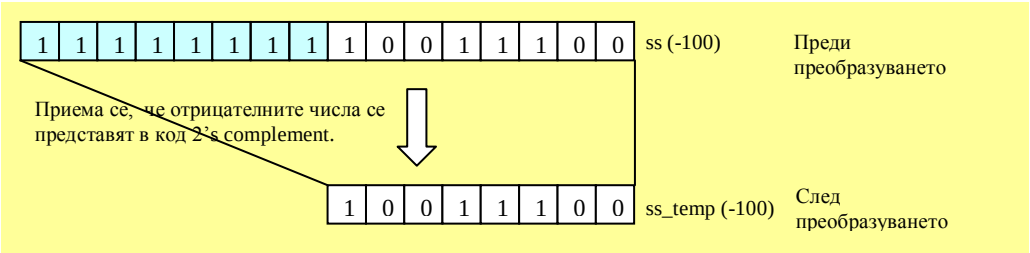
Фиг. 23 Преобразуване на неотрицателна стойност signed short в signed char

Ако стойността на преобразувания тип е отрицателна и резултатният тип може да представи стойността на преобразувания тип, стойността на преобразувания тип просто се преобразува в съответния еквивалент на резултатния тип[1].

Забележка[1]: Ако отрицателните числа се представят в код 2's complement преобразуването се състои в изхвърляне на старшите байтове на преобразувания тип.

Пример: `signed short -> signed char`

```
signed short  ss = -100;
(signed char)ss
```



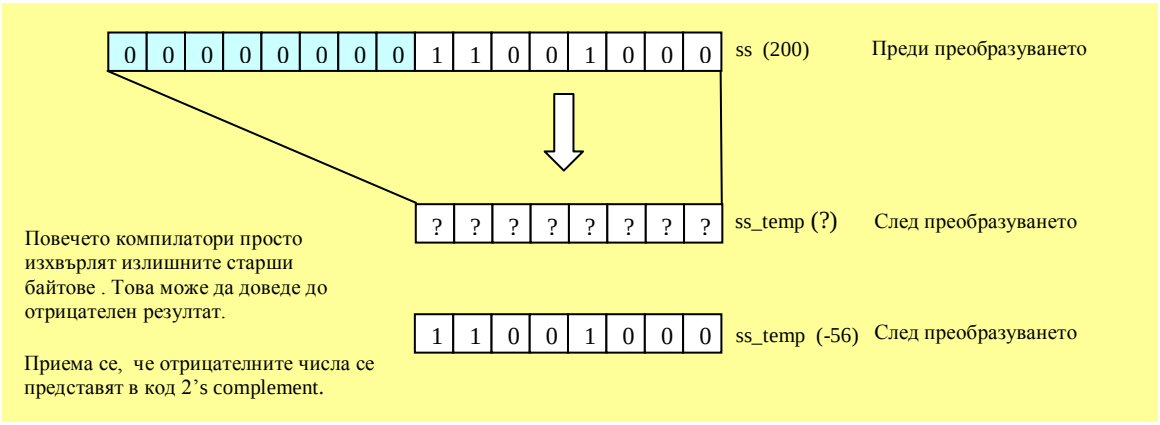
Фиг. 24 Преобразуване на отрицателна стойност signed short в signed char

Ако резултатният тип не може да представи стойността на преобразувания

тип резултатът зависи от имплементацията. Тази ситуация е известна като целочислено препълване.

Пример: `signed short -> signed char`

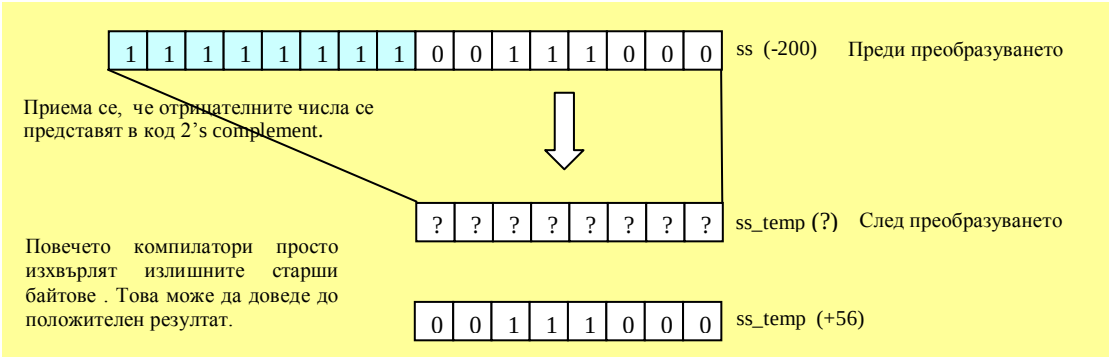
```
signed short  ss = 200;
(signed char)ss
```



Фиг. 25 Преобразуване на неотрицателна стойност signed short в signed char

Пример: `signed short -> signed char`

```
signed short  ss = -200;
(signed char)ss
```



Фиг. 26 Преобразуване на отрицателна стойност signed short в signed char

1.3.1.3 Преобразуване на знаков тип в безнаков тип със същия или по-голям размер

Възможните комбинации на преобразуване са показани в Табл.37.

преобразуван тип	резултатен тип
signed char	unsigned char unsigned short unsigned int

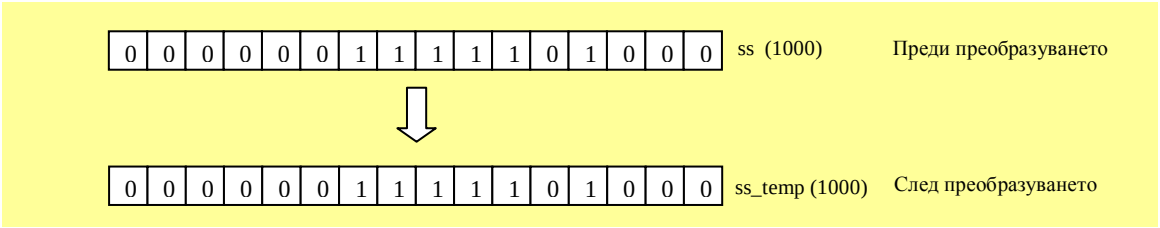
	unsigned long
signed short	unsigned short unsigned int unsigned long
signed int (16 бита)	unsigned short unsigned int unsigned long
signed int (32 бита)	unsigned int unsigned long
signed long	unsigned int (32 бита) unsigned long

Табл. 37 Комбинации на преобразуване на знакови типове в беззнакови типове със същия или по-голям размер

Ако стойността на преобразувания тип е неотрицателна и преобразуваният и резултатният тип са с еднакви размери, преобразуването се извършва без никакво изменение в стойността и знака на преобразувания тип (Фиг.27).

Пример: `signed short -> unsigned short`

```
signed short ss = 1000;
(unsigned short)ss
```

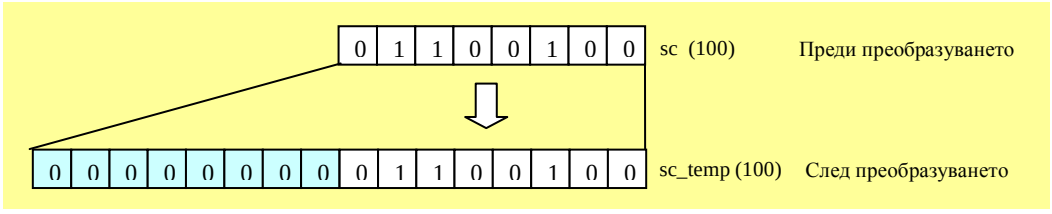


Фиг. 27 Преобразуване на знаков тип с неотрицателна стойност в беззнаков тип със същия размер

Ако стойността на преобразувания тип е неотрицателна и преобразуваният тип е по-малък от резултатния, старшите битове на резултатния тип се запълват с нули.

Пример: `signed char -> unsigned short`

```
signed char sc = 100;
(unsigned short)sc
```



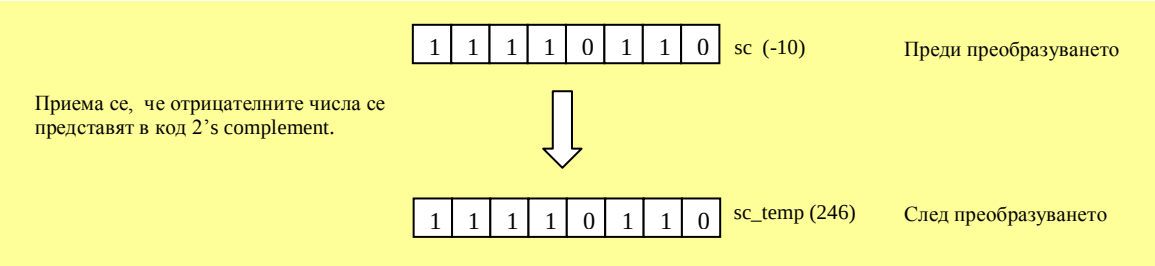
Фиг. 28 Преобразуване на знаков тип с неотрицателна стойност в беззнаков тип с по-голям размер

Ако стойността на преобразувания тип е отрицателна и резултатният тип има същия размер, то преобразуването се извършва като се добави число с 1 по-голямо от най-голямото число, което може да се представи от резултатния тип[1].

Забележка[1]: Ако отрицателните числа се представят в код 2's complement, това преобразуване не води до изменение на вътрешното представяне на стойността на преобразувания тип, а само до изменение интерпретацията. Ако обаче се използва друг код за представяне на отрицателните числа, преобразуването води и до изменение на вътрешното представяне на стойността на преобразувания тип.

Пример: `signed char -> unsigned char`

```
signed char   sc = -10;
(unsigned char)sc /* sc + (UCHAR_MAX + 1) = -10 + (255 + 1) = 246 */
```

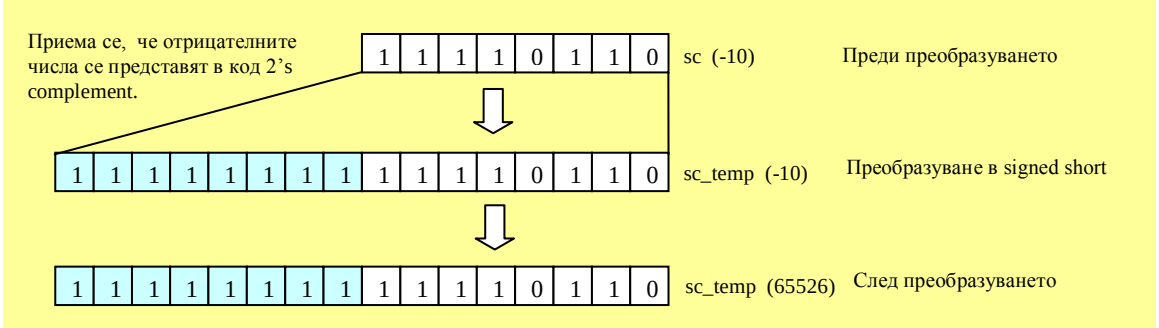


Фиг. 29 Преобразуване на знаков тип с отрицателна стойност в беззнаков тип със същия размер

Ако стойността на преобразувания тип е отрицателна и резултатният тип е с по-голям размер, то преобразуването се извършва като първо се извърши преобразуване до знаковата версия на резултатния тип и след това се добави число с 1 по-голямо от най-голямото число, което може да се представи от резултатния тип.

Пример: `signed char -> unsigned short`

```
signed char   sc = -10;
(unsigned short)sc /* (signed short)sc + (USHRT_MAX + 1) = -10 + (65535 + 1) =
65526 */
```



Фиг. 30 Преобразуване на знаков тип с отрицателна стойност в беззнаков тип с по-голям размер

1.3.1.4 Преобразуване на знаков тип в беззнаков тип с по-малък размер

Възможните комбинации на преобразуване са показани в Табл.38.

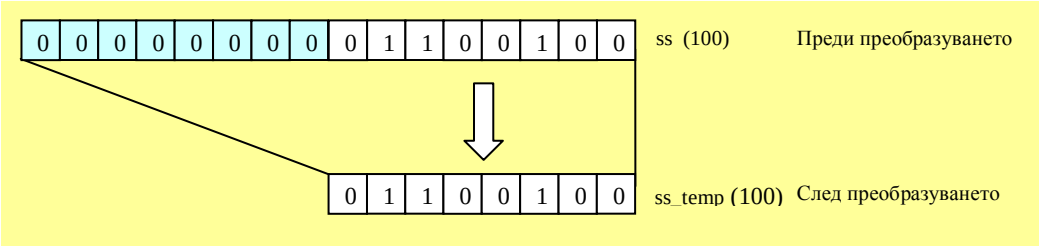
преобразуван тип	резултатен тип
signed short	unsigned char
signed int (16 бита)	unsigned char
signed int (32 бита)	unsigned char unsigned short
signed long	unsigned char unsigned short unsigned int (16 бита)

Табл. 38 Комбинации на преобразуване на знакови типове в беззнакови типове с по-малък размер

Ако стойността на преобразувания тип е неотрицателна и може да се представи от резултатния тип, преобразуването се извършва просто чрез изхвърляне на старшите байтове на преобразувания тип.

Пример: `signed short -> unsigned char`

```
signed short  ss = 100;
(unsigned char)ss
```

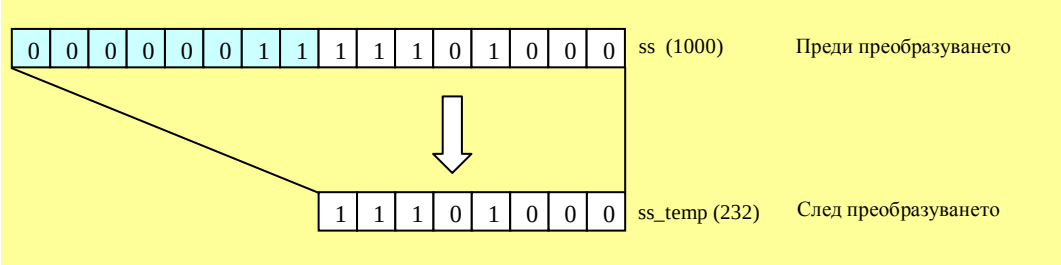


Фиг. 31 Преобразуване на знаков тип с неотрицателна стойност в беззнаков тип с по-малък размер

Ако стойността на преобразувания тип е неотрицателна и не може да се представи от резултатния тип, резултатът от преобразуването се получава като от преобразуваната стойност се изважда число с 1 по-голямо от максималното число, което може да се представи от резултатния тип, толкова пъти докато получената стойност попадне в обхвата на резултатния тип. Същия резултат от преобразуването се получава, ако се вземе остатъка от делението на преобразуваната стойност на число с 1 по-голямо от максималното число, което може да се представи от резултатния тип.

Пример: `signed short -> unsigned char`

```
signed short  ss = 1000;
(unsigned char)ss /* ss - (UCHAR_MAX + 1)*3 = 1000 - (255 + 1)*3   = 232 */
                  /* ss % (UCHAR_MAX + 1)   = 1000 % (255 + 1)     = 232 */
```

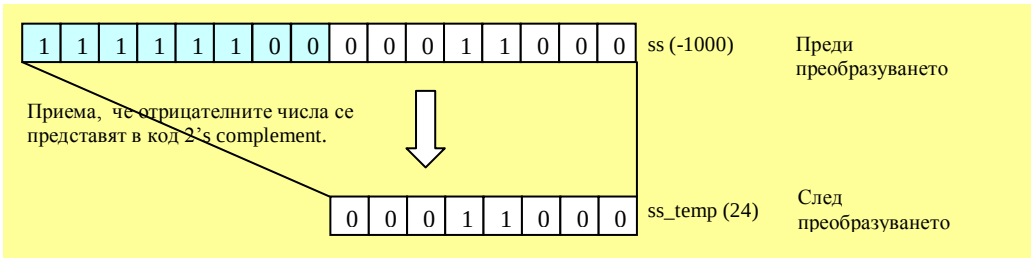


Фиг. 32 Преобразуване на знаков тип с неотрицателна стойност в беззнаков тип с по-малък размер

Ако стойността на преобразувания тип е отрицателна и не може да се представи от резултатния тип, резултатът от преобразуването се получава като към преобразуваната стойност се добавя число с 1 по-голямо от максималното число, което може да се представи от резултатния тип, толкова пъти докато получената стойност попадне в обхвата на резултатния тип.

Пример: `signed short -> unsigned char`

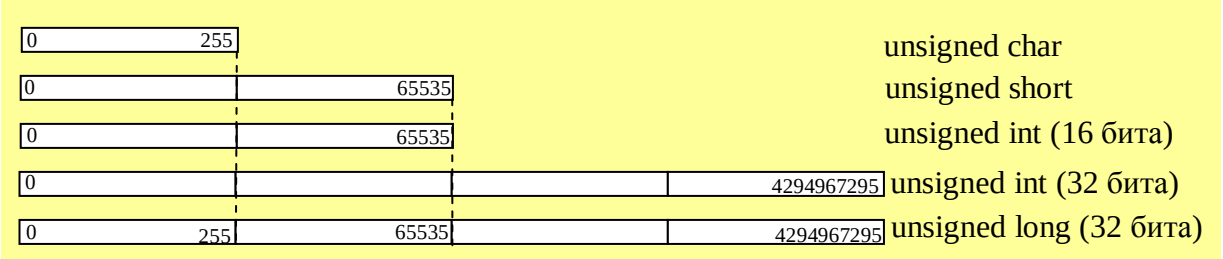
```
signed short  ss = -1000;
(unsigned char)ss /* ss + (UCHAR_MAX + 1)*4 = -1000 + (255 + 1)*4 = 24 */
```



Фиг. 33 Преобразуване на неотрицателна стойност signed short в unsigned char

1.3.2 Преобразуване на беззнакови типове

Фиг.34 показва стойностите, които могат да се представят от различните беззнакови целочислени типове.



Фиг. 34 Диапазон от стойности на беззнаковите целочислени типове

1.3.2.1 Преобразуване на беззнаков тип в друг безнаков тип със същия или по-голям размер

Възможните комбинации на преобразуване са показани в Табл.39.

преобразуван тип	резултатен тип
unsigned char	unsigned short unsigned int unsigned long
unsigned short	unsigned int (16 или 32 бита) unsigned long
unsigned int (16 бита)	unsigned short unsigned long
unsigned int (32 бита)	unsigned long

Табл. 39 Комбинации на преобразуване на беззнаков целочислен тип в друг беззнаков целочислен тип със същия или по-голям размер

При това преобразуване резултатният тип може да представи всички стойности на преобразувания тип. Ако резултатният тип е с по-голям размер, старшите байтове се запълват с нули.

1.3.2.2 Преобразуване на беззнаков тип в друг беззнаков тип с по-малък размер

Възможните комбинации на преобразуване са показани в Табл.40.

преобразуван тип	резултатен тип
unsigned short	unsigned char
unsigned int (16 бита)	unsigned char
unsigned int (32 бита)	unsigned char unsigned short
unsigned long	unsigned char unsigned short unsigned int (16 бита)

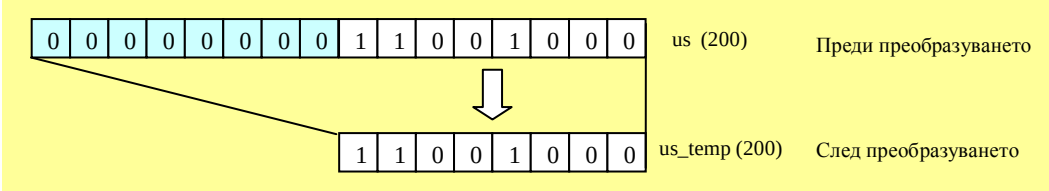
Табл. 40 Комбинации на преобразуване на беззнаков целочислен тип в друг беззнаков целочислен тип с по-малък размер

Общото правили при това преобразуване гласи, че старшите байтове на преобразувания тип се изхвърлят.

Ако стойността на преобразуваният тип може да се представи от резултатния тип, преобразуването не води до загуба на информация.

Пример: `unsigned short -> unsigned char`

```
unsigned short  us = 200;
(usigned char)us
```

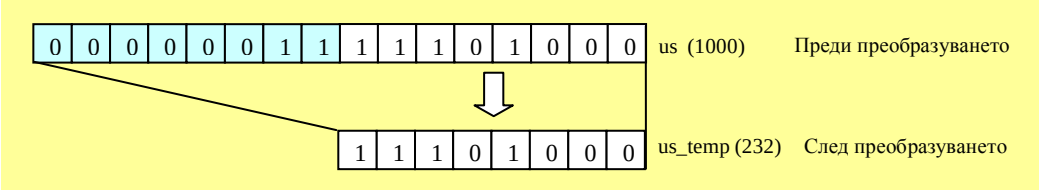


Фиг. 35 Преобразуване на unsigned short в unsigned char без загуба на информация

Ако стойността на преобразуваният тип не може да се представи от резултатния тип, преобразуването води до загуба на информация.

Пример: `unsigned short -> unsigned char`

```
unsigned short  us = 1000;
(usigned char)us
```



Фиг. 36 Преобразуване на unsigned short в unsigned char със загуба на информация

1.3.2.3 Преобразуване на беззнаков тип в знаков тип със същия размер

Възможните комбинации на преобразуване са показани в Табл.41.

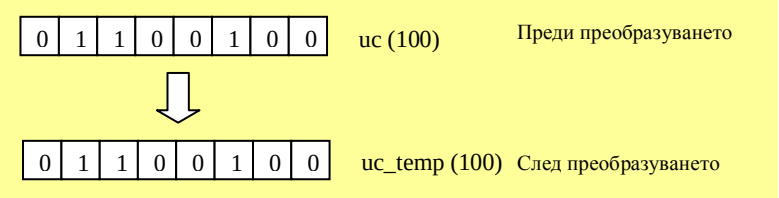
преобразуван тип	резултатен тип
unsigned char	signed char
unsigned short	signed short signed int (16 бита)
unsigned int (16 бита)	signed short signed int
unsigned int (32 бита)	signed int
unsigned long	signed long signed int (32 бита)

Табл. 41 Комбинации при преобразуване на беззнаков целочислен тип в знаков целочислен тип със същия размер

Ако стойността на преобразувания тип може да се представи от резултатния тип, преобразуването не води до никакво изменение.

Пример: `unsigned char -> signed char`

```
unsigned char   uc = 100;
(signed char)uc
```

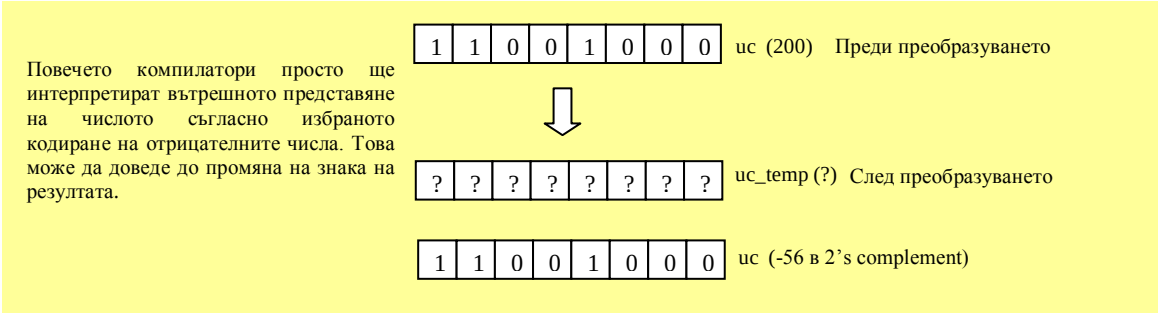


Фиг. 37 Преобразуване на unsigned char в signed char

Ако стойността на преобразувания тип не може да се представи от резултатния тип, резултатът от преобразуването зависи от имплементацията. Тази ситуация е известна като целочислено препълване.

Пример: `unsigned char -> signed char`

```
unsigned char   uc = 200;
(signed char)uc
```



Фиг. 38 Преобразуване на unsigned char в signed char

1.3.2.4 Преобразуване на беззнаков тип в знаков тип с по-голям размер

Възможните комбинации на преобразуване са показани в Табл.42.

преобразуван тип	резултатен тип
unsigned char	signed short signed int signed long
unsigned short	signed int (32 бита) signed long
unsigned int (16 бита)	signed long

Табл. 42 Комбинации на преобразуване на беззнаков целочислен тип в знаков целочислен тип с по-голям размер

При това преобразуване резултатният тип може да представи всички стойности на преобразувания тип. Старшите байтове на преобразувания тип се запълват с нули.

1.3.2.5 Преобразуване на беззнаков тип в знаков тип с по-малък размер

Възможните комбинации на преобразуване са показани в Табл.43.

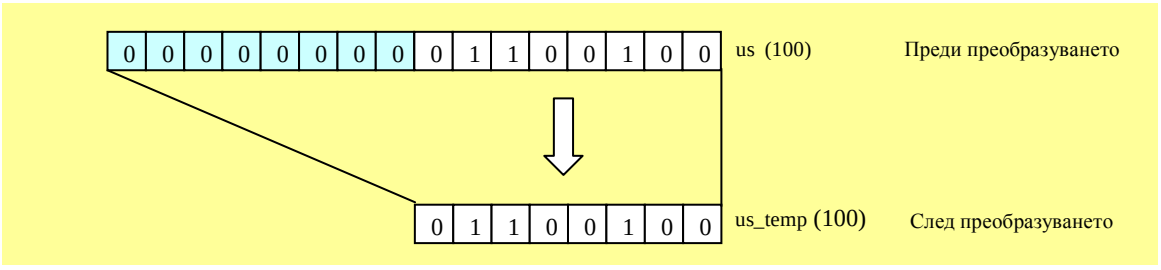
преобразуван тип	резултатен тип
unsigned short	signed char
unsigned int (16 бита)	signed char
unsigned int (32 бита)	signed char signed short
unsigned long	signed char signed short signed int (16 бита)

Табл. 43 Комбинации на преобразуване на беззнаков целочислен тип в знаков целочислен тип с по-малък размер

Ако стойността на преобразувания тип може да се представи от резултатния тип, преобразуването се извършва просто чрез изхвърляне на старшите байтове на преобразувания тип.

Пример: `unsigned short -> signed char`

```
unsigned short   us = 100;
(signed char)us
```

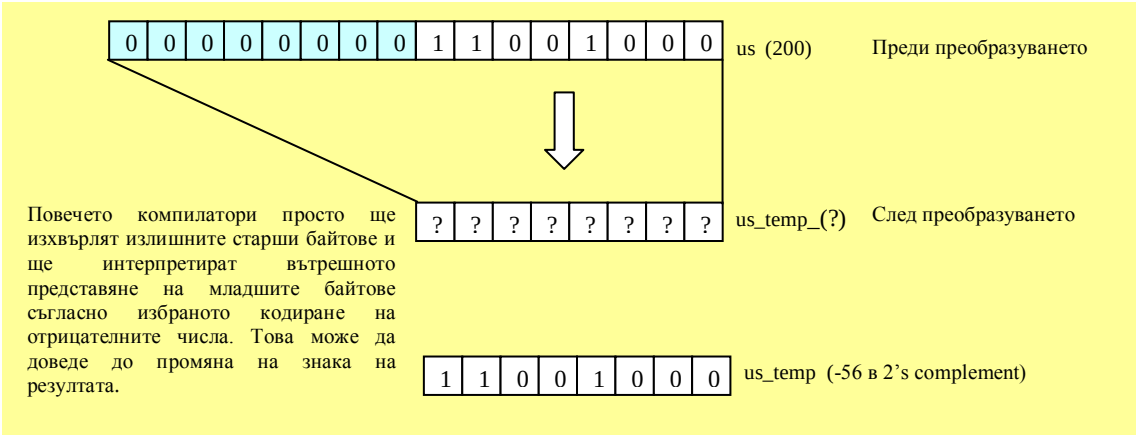


Фиг. 39 Преобразуване на unsigned short в signed char

Ако стойността на преобразувания тип не може да се представи от резултатния тип, резултатът от преобразуването зависи от имплементацията. Тази ситуация е известна като целочислено препълване.

Пример: `unsigned short -> signed char`

```
unsigned short   us = 200;
(signed char)us
```



Фиг. 40 Преобразуване на unsigned short в signed char

1.4 Преобразувания между типове с плаваща запетая

Когато тип **float** се преобразува в **double** или в **long double**, или тип **double** се преобразува в тип **long double**, стойността на преобразувания тип се запазва.

Когато се преобразува тип **long double** в **double** или в **float** или **double** в **float**, ако преобразуваната стойност не може да се представи от резултатния тип, поведението е недефинирано. Ако преобразуваната стойност е в обхвата от стойности, които могат да бъдат представени, но не могат да бъдат представени точно, резултатът е или най-близката по-голяма или най-близката по-малка стойност в зависимост от имплементацията.

1.5 Преобразуване между целочислени типове и типове с плаваща запетая

1.5.1 Преобразуване на целочислен тип в тип с плаваща запетая

Ако преобразуваната стойност може да се представи от резултатния тип, но не може да се представи точно, резултатът е или най-близката по-голяма или най-близката по-малка стойност в зависимост от имплементацията.

1.5.2 Преобразуване на тип с плаваща запетая в целочислен тип

При това преобразуване дробната част на преобразувания тип се изхвърля.

Ако цялата част на преобразувания тип може да се представи от резултатния тип, тя става резултат от преобразуването.

Ако цялата част на преобразувания тип не може да се представи от резултатния тип, резултатът е недефиниран. Това е вид препълване.

Пример 5-2:

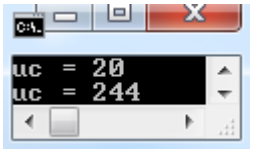
```
#include <stdio.h>

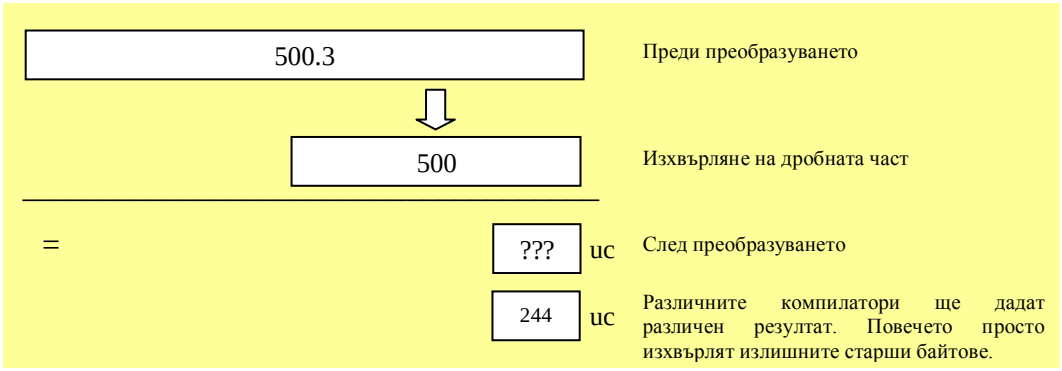
int main(void)
{
	unsigned char	uc;

	uc = 20.3; /* 20 -> uc */
	printf("uc = %d\n", uc);

	uc = 500.3; /* ??? -> uc */
	printf("uc = %d\n", uc);

	return 0;
}
```





Фиг. 41 Преобразуване на double в unsigned char

1.6 Преобразувания свързани с указатели

В С указателите се разделят на две основни групи:

- Указатели към обекти (**object pointers**)
- Указатели към функции (**function pointers**)

Преобразуването между указатели към обекти и указатели към функции не е дефинирано от стандарта С.

1.6.1 Явни (изрични) преобразувания между указатели

1.6.1.1 Преобразувания между указатели към обекти

Указател към един обект може да се преобразува изрично в указател към друг обект. Въпреки това не всички преобразувания между указатели към обекти са безопасни. Всички базови типове данни имат атрибут **m**, наречен модул на подравняване в паметта (**memory alignment**). Различните хардуерни архитектури имат различни изисквания как данните да се разполагат в паметта. Например някои хардуерни архитектури изискват променливите от скаларен тип да се разполагат в паметта на адреси, кратни на модула на подравняване на съответния тип, т.е следното условие трябва да е изпълнено:

```
(адрес-променлива  %  m) == 0
```

Ако това условие не е спазено, достъпът до неподравнена променлива може да породи едно от следните две неща в зависимост от хардуерната архитектура:

1) Увеличение на времето за достъп до променливата. Това се случва за хардуерни платформи, които поддържат достъп до неподравнени данни.

2) Генериране на изключение (аварийно прекъсване). Това се случва за хардуерни платформи, които не поддържат достъп до неподравнени данни.

Някои компилатори винаги подравняват променливите в паметта, дори хардуерната платформа да не го изисква, с цел подобряване производителността на програмата.

Ако имаме типове S и D, съответно с модули на подравняване m_S и m_D, следните правила с валидни:

- Ако $m_S \geq m_D$, то преобразуването на **указател-към-S** в **указател-към-D** е "безопасно". Под "безопасно" се има предвид, че резултатният указател ще работи както се очаква, ако се използва за достъп до обект от тип D, и преобразуването му обратно в първоначалния указателен тип ще възстанови първоначалния указател.

- Ако $m_S < m_D$, то преобразуването на **указател-към-S** в **указател-към-D** е "опасно". "Опасно" означава, че са възможни едно от следните две неочаквани поведения:

1) Опит да се достъпи обект от тип D може да причини грешка, спирайки изпълнението на програмата.

2) Хардуерът или имплементацията може да "настрои" резултатния указател да бъде валиден, обикновено преобразувайки го към най-близкия предходен валиден адрес. Последващо преобразуване към първоначалния указателен тип може да не възстанови първоначалния указател.

Пример:

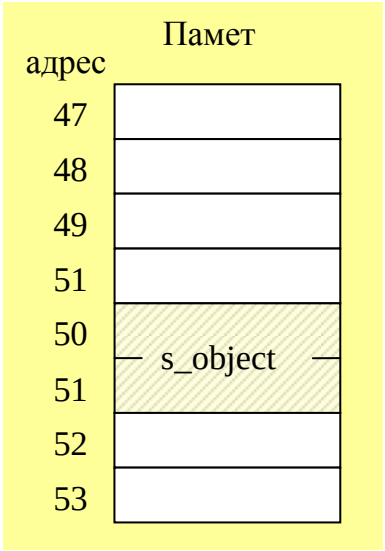
Нека допуснем следните условия:

Типовете S и D имат модули на подравняване

m_S = 2 (модул на подравняване на тип S)
m_D = 4 (модул на подравняване на тип D)

Дефинирани са следните обекти и указатели от тип S (2 байта) и D (4 байта):

```
S  *ps, s_object;
D  *pd;
```



Компилаторът разполага променливата s_object примерно на адрес 50, т.е. променливата е подравнена, тъй като условието е (50 % 2) == 0 е изпълнено.

Нека адреса на s_object се присвои на укзателя pd

```
pd = (D* ) &s_object;
```

и указателя се дереференцира

```
*pd = стойност; .
```

Резултатът от дерефенцирането на указателя ще зависи от хардуерната архитектура, на която този код се изпълнява.

Хардуерна архитектура, която не поддържа достъп до неподравнени данни

Резултатът ще бъде генериране на изключение (и вероятно до спиране на програмата), тъй като указателят pd е предназначен да сочи към обекти от тип D, за които трябва

да е изпълнено условието

(&d_object % m_D) == 0.

В случая обаче указателят pd съдържа адреса на s_object, който не е кратен на 4, т.е. (50 % 4) е различно от 0.

Както споменахме в началото, някои имплементации на компилатора или дори самата хардуерна архитектура, могат да "настроят" указателя, така че да сочи към валиден адрес. Например на указателя pd може да се присвои адрес 48 вместо 50, който е кратен на 4. При това положение първоначалния указател не може да се възстанови от резултатния указател.

Хардуерна архитектура, която поддържа достъп до неподравнени данни

Операцията ще се изпълни, но вероятно с цената на повече машинни цикли. Някои компилатори е възможно да поставят данните винаги на подравнени адреси с цел по-добра производителност, дори и това да не се изисква от архитектурата на процесора.

Посоченият пример демонстрира един от недостатъците на езика C, а именно че позволява създаването на код, който не е безопасен, като отговорността това да не се случва лежи изцяло на програмистта.

Допълнително информация за подравняването на данните в паметта, може да се намери в Глава 2 Типове данни.

1.6.1.2 Преобразувания между указатели към функции

Типът на една функция включва типа на връщания от нея резултат и списъка с параметри. Указател към функция от един тип може да се преобразува в указател към функция от друг тип. Стандартът C обаче не дефинира поведението ако указател към функция се използва за извикване на функция, чийто тип не съвпада точно с типа на указателя.

Пример 5-3:

```
#include <stdio.h>

void foo1(int x)
{
   printf("%d\n", x);
}

void foo2(char *message)
{
   printf("%s\n", message);
}

typedef void func_t(int); /* Дефиниране на тип func_t */

int main(void)
{
```

```
    func_t* p; /* Дефиниране на указател към тип func_t */

    p = foo1; /* Адреса на функцията foo1() се присвоява на указателя p */
    p(3); /* Коректно извикване на foo1() чрез указателя p */
    p = (func_t*)foo2; /* Адреса на функцията foo2() се присвоява на указателя p */
    p(3); /* !!!Внимание, поведението на програмата е недефинирано
             в тази ситуация */
    return 0;
}
```

1.6.2 Неявни(подразбиращи) преобразувания между указатели

1.6.2.1 Преобразувания към и от void-указатели

Void-указателите са универсални указатели, тъй като те могат да сочат към всякакви обекти независимо от типа им. Преобразуването на указател към обект в void-указател и обратно става автоматично. Изрично преобразуване в този случай не се изисква. Достъпът до обект, сочен от void указател, обаче изисква явно преобраузване на void-указателя в указател към типа на обекта, в противен случай компилаторът ще генерира грешка.

Пример 5-4:

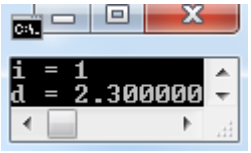
```
#include <stdio.h>

int main(void)
{
	void*  p_void;
	int	i;
	double	d;

	p_void = &i; /* неявно преобразуване на int* в void* */
	/* *p_void = 1; */ /* !!!Грешка, необходимо е явно преобразуване */
	*(int*)p_void = 1;
	printf("i = %d\n", *(int*)p_void);

	p_void = &d; /* неявно преобразуване на double* в void* */
	*(double*)p_void = 2.3;
	printf("d = %lf\n", *(double*)p_void);

	return 0;
}
```



1.6.2.2 Указатели към квалифицирани обекти

В C всеки обект може да бъде квалифициран с квалификаторите **const** и

volatile. Компилаторът може да преобразува неявно указател към неквалифициран-обект в указател към квалифициран-обект където е необходимо. Например указател към int се преобразува неявно от компилатора в указател към const int.

Пример 5-5:

```
#include <stdio.h>

int main(void)
{
	const int  const_object    = 3;
	      int  nonconst_object = 5;

	      int* p_to_nonconst;
	const int* p_to_const;

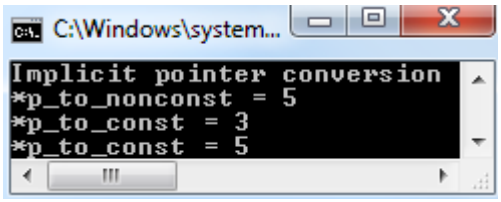
	printf("Implicit pointer conversion\n");

	p_to_nonconst = &nonconst_object;
	printf("*p_to_nonconst = %d\n", *p_to_nonconst);

	p_to_const    = &const_object;
	printf("*p_to_const = %d\n", *p_to_const);

	/* Неявно преобразуване на int* в const int* */
	p_to_const = &nonconst_object;
	printf("*p_to_const = %d\n", *p_to_const);

	return 0;
}
```



Преобразуването на указател към квалифициран-обект в указател към неквалифициран-обект изисква явно преобразуване.

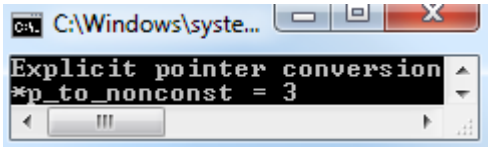
Пример 5-6:

```
#include <stdio.h>

int main(void)
{
	const int  const_object    = 3;
	      int* p_to_nonconst;

	printf("Explicit pointer conversion\n");
	/* Явно преобразуване на const int* в int* */
	p_to_nonconst = (int*)&const_object;
	printf("*p_to_nonconst = %d\n", *p_to_nonconst);

	return 0;
}
```



Const-обектите могат да се променят от хардуера, но не могат да се променят от софтуера. Преобразуването на указател към константен обект в указател към неконстантен обект нарушава това правило. Такъв указател трябва да се използва само за четене на сочената от него променлива. Поведението е недефинирано, ако такъв указател се използва за модифициране на сочения от него обект.

Пример 5-7:

```
#include <stdio.h>

int main(void)
{
	const int  const_object    = 3;
		int* p_to_nonconst;

	printf("Undefined behavior\n");

	p_to_nonconst = (int*)&const_object;
	printf("*p_to_nonconst = %d\n", *p_to_nonconst);

	*p_to_nonconst = 10; /* !!!Внимание, недефинирано поведение */
	printf("*p_to_nonconst = %d\n", *p_to_nonconst);

	return 0;
}
```

1.6.2.3 Нулеви указателни константи

Целочислената константа 0 или целочислена константа 0 преобразувана в указател към void се нарича нулев константен указател. Нулевият константен указател може да се използва за две неща:

- Да се присвоява на всеки тип указател.
- Да се сравнява за равенство с всеки тип указател.

И в двата случая нулевият константен указател се преобразува в типа на другия указател. Указател, на който е присвоен нулев константен указател се нарича **нулев указател** (**null pointer**). Приема се, че нулевият указател е невалиден, т.е не сочи към нищо и се различава от всеки друг валиден указател. Вътрешното представяне на нулевите указатели за различните указателни типове може да се различава, т.е. не е задължително всички битове на нулевите указатели да бъдат нули. Нулата е единственото цяло число, което може директно да се присвоява на указатели или да се

сравнява с тях. Макросът NULL представлява нулев константен указател и е дефиниран в хедъра <stddef.h> като

```
#define  NULL  ((void *)0)
```

Пример 5-8:

```
#include <stdio.h> /* този хедър включва в себе си и stddef.h */
int main(void)
{
	int* p;
	int  x;

	p = 0; /* Сега p е нулев указател */

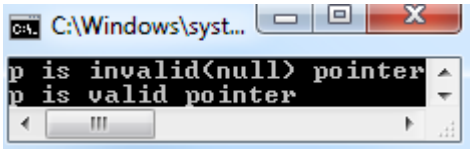
	p = NULL; /* Същото като p = 0 */

	if (NULL == p) /* Проверка за валидност на указателя */
	{
		printf("p is invalid(null) pointer\n");
	}

	p = &x; /* Сега p сочи към валиден обект */

	if (NULL != p) /* Проверка за валидност на указателя */
	{
		printf("p is valid pointer\n");
	}

	return 0;
}
```



1.6.3 Други преобразувания свързани с указатели

1.6.3.1 Преобразуване на цели числа в указатели и обратно

Всеки указател може изрично да се преобразува в цяло число и обратно, но резултатът зависи от имплементацията и не е преносим. Когато указател се преобразува в целочислен тип, който не може да побере стойността на указателя, резултатът е недефиниран. Размерът в байтове на указателите не е твърдо фиксиран, а зависи от размера на паметта и архитектурата на компютъра. За по ефикасно използване на паметта, компилаторите добавят нестандартни ключови думи като **near**, **far**, **huge** и подобни, с които може да се променя размера на указателя, а от там и размера на паметта, която може да се адресира с този указател. Един 16-битов указател може да адресира до 64 килобайта памет (0x0000 ÷ 0xFFFF), а един 32-битов указател до 4 гигабайта (0x00000000 ÷ 0xFFFFFFFF). От това следва, че не

всяко цяло число преобразувано в указател, ще се побере в него, съответно и резултатът не винаги ще бъде валиден указател.

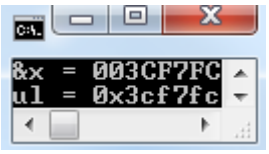
Пример 5-9:

```
#include <stdio.h>

int main(void)
{
	int  x;
	unsigned long ul;


	printf ("&x = %p\n", &x);
	ul = (unsigned long)&x;
	printf ("ul = %#x\n", ul);

	return 0;
}
```



1.6.3.2 Преобразуване на име на масив в указател

Името на масив се преобразува автоматично в указател към първия елемент на масива с изключение на следните ситуации:

- Когато името на масив е операнд на операцията за определяне на размера **sizeof**.

  ```
  sizeof(име-масив)
  ```

  В този случай резултатът е броя на байтовете, заемани от елементите на масива.

- Когато името на масив е операнд на адресната операция **&**

  &име-масив

  В този случай резултатът е указател към целия масив, т.е

  ```
  тип (*)[размер-на-масива]
  ```

- Когато низов литерал се използва за инициализация на **char** или **w_char** масив

  ```
  char име-масив[] = "Some string";
  ```

В този случай елементите на масива се инициализират със символите на низа.

Пример 5-10:

```
#include <stdio.h>
int main(void)
{
	int  array[] = {1,2,3,4,5};
	char str[] = "Hello C";
	int  *p;
	int  i;
	int  (*p_to_array)[5]; /* Указател към масив от 5 int елемента */

	/* Отпечатване на елементите на масива чрез името на масива */
	for (i = 0; i < sizeof(array)/sizeof(int); i++)
	{
		printf ("array[%d] = %d\n", i, array[i]);
	}

	printf("\n\n");
	p = array;

	/* Отпечатване на елементите на масива чрез указател към масива */
	for (i = 0; i < sizeof(array)/sizeof(int); i++)
	{
		printf ("array[%d] = %d\n", i, *(p + i));
	}

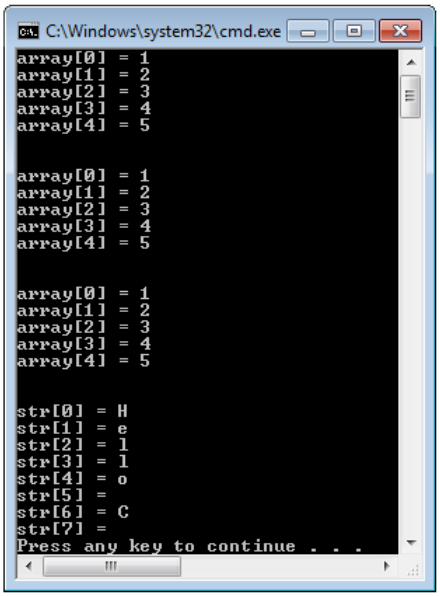
	printf("\n\n");
	p_to_array = &array;

	/* Отпечатване на елементите на масива чрез указател към целия масив */
	for (i = 0; i < sizeof(array)/sizeof(int); i++)
	{
		printf ( "array[%d] = %d\n", i, *(((int*)p_to_array) + i) );
	}

	printf("\n\n");

	/* Отпечатване на символите на char масив */
	for (i = 0; i < sizeof(str)/sizeof(char); i++)
	{
		printf ("str[%d] = %c\n", i, str[i]);
	}

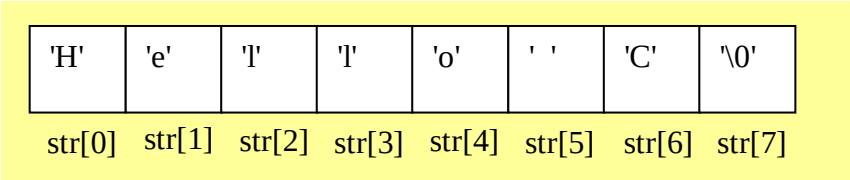
	return 0;
}
```



Нека да разгледаме някои части от този пример по-подробно като започнем от конструкцията

```
char str[] = "Hello C";
```

Тази конструкция дефинира масив с име str, чийто елементи са от тип **char**, а не указател str, който сочи към низа "Hello C". Елементите на масива се инициализират със символите на низа "Hello C" (Фиг.42).



Фиг. 42 Масив str инициализиран с низа "Hello C"

Конструкцията

```
sizeof(array)
```

връща броя на байтовете заемани от елементите на масива array (20 ако int е 32 бита и 10 ако int е 16 бита), т.е в този контекст array не се преобразува в указател, в който случай резултата би бил 2 или 4 в зависимост от размера на указателя.

Конструкцията

```
array[i]
```

връща i-тия елемент на масива array. В този контекст името на масив автоматично се преобразува в указател към първия елемент на масива, т.е конструкцията array[i] е еквивалента на **(&array[0])[i]** и на ***(array + i)**.

Получените конструкции се дължат на връзката между масиви и указатели в C (вижте Глава 8 Масиви).

В конструкцията

```
p = array;
```

array се преобразува в &array[0], т.е в указател към първия елемент на масива.

В конструкцията

```
p_to_array = &array;
```

p_to_array е указател към обект от тип **масив с 5 елемента тип int**, а не просто указател към тип **int**. Конструкцията &array в този случай връща адреса на целия масив. Въпреки, че технически този адрес съвпада с адреса на първия елемент на масива, конструкциите &array и &array[0] връщат като резултат указатели към различни типове. В първия случай това е указател към **5 елементен масив тип int**, а във втория указател към тип **int**. Ето защо указателя p_to_array се преобразува в указател към int чрез конструкцията **(int*)p_to_array** преди да се използва за достъп до елементите на масива.

Преобразуването на име на масив в указател към първия елемент на масива, е част от т. нар. обичайни унарни преобразувания (вижте точка 8.3 Обичайни унарни преобразувания).

1.6.3.3 Преобразуване на име на функция в указател

Името на функция автоматично се преобразува в указател към типа на функцията с изключение на следните ситуации:

- Когато името на функция е операнд на **sizeof**.

  ```
  sizeof(име-функция)
  ```

  В този случай компилаторът ще генерира грешка, тъй като това е невалидна конструкция в C.

- Когато името на функция е операнд на адресния оператор **&**.

  ```
  &име-функция
  ```

  Тази конструкция изрично преобразува името на функцията в указател към нея, т.е.

```
тип (*)(списък-параметри)
```

Пример 5-11:

```
#include <stdio.h>

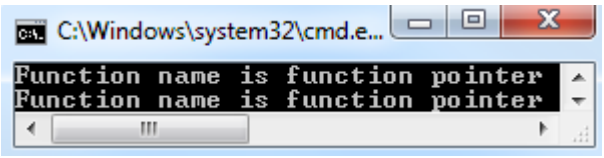
void myfunc(void)
{
	printf("Function name is function pointer\n");
}

int main(void)
{
	void (*p)(void);

	p = myfunc; /* Подразбиращо се преобразуване */
	p();

	p = &myfunc; /* Изрично преобразуване */
	p();

	return 0;
}
```



Преобразуването на име на функция в указател към нея е част от т. нар. обичайни унарни преобразувания (вижте точка 8.3 Обичайни унарни преобразувания).

1.7 Преобразувания към тип void

Всяка стойност може да бъде преобразувана в тип **void**. Такова преобразуване има смисъл само когато искаме изрично да подчертаем, че не се интересуваме от стойността, върната от дадена функция.

Пример 5-12:

```
#include <stdio.h>

int main(void)
{
	(void)printf("Cast to void\n");

	return 0;
}
```

В C ако резултатът, връщан от функция, не се използва той просто се губи. Преобразувайки го към **void** е ясно указание, че не се интересуваме от него в дадена ситуация както е показано в примера по-горе.



Ако резултата от извикване на функция не се използва, за препочитане е използването на преобразуване към тип void. Това подчертава, че изхвърлянето на резултата е умишлено, а не по невнимание.

1.8 Видове преобразувания

В С стойността на един тип може да се преобразува в друг тип при следните обстоятелства:

- Явно(изрично) преобразуване при използване на операция за преобразуване на типа.
- Преобразуване на типа на десния операнд в типа на левия операнд при присвояване.
- Преобразуване на типовете в аритметични и логически изрази.
- Преобразуване на типа на аргументите при извикване на функция.
- Преобразуване на типа на стойността на оператора **return** при връщане на резултат от функция.

1.8.1 Явно преобразуване при използване на операция за преобразуване на типа

Табл.44 описва позволените[1] преобразувания между различните типове данни чрез явно преобразуване с оператора за преобразуване.

резултатен тип	преобразуван тип
Всеки аритметичен тип	Всеки аритметичен тип
Всеки целочислен тип	Всеки указателен тип
Указател към обект T или void-указател	▪ Всеки целочислен тип ▪ void указател ▪ Указател към обект Q
Указател към функция T	▪ Всеки целочислен тип ▪ void-указател ▪ Указател към функция Q
struct- или union-тип	-
масив от T	-
Функция връщаща T	-
void	Всеки тип

Табл. 44 Позволени явни преобразувания

Забележка[1]: Под "позволени" преобразувания се има предвид, че компилаторът ще позволи дадено преобразуване без да генерира грешка. Отговорност на програмистта е да определи дали резултатът от преобразуването е ясно дефиниран от стандарта или зависи от имплементацията.

1.8.2 Преобразувания при присвояване

При просто присвояване нормално типът на десния операнд е същия като този на левия операнд. Ако това не е така компилаторът ще се опита автоматично да преобразува типа на десния операнд в типа на левия. Ако преобразуването е невъзможно, компилаторът ще генерира грешка.

Табл.45 описва позволените типове на левия и десния операнд, при които преобразуването на левия в десния операнд става автоматично от компилатора.

ляв операнд	десен операнд
Всеки аритметичен тип	Всеки аритметичен тип
struct- или union-тип	struct- или union- съвместим тип
void-указател[1]	▪ Константа 0 ▪ Указател към обект ▪ void-указател
Указател към обект T_1[1,2]	▪ Константа 0 ▪ Указател към обект T_2, където T_1 и T_2 са съвместими и имат еднакви квалификатори ▪ void-указател
Указател към функция F_1[1]	▪ Константа 0 ▪ Указател към функция F_2, където F_1 и F_2 са съвместими

Табл. 45 Позволени преобразувания при присвояване

Забележка [1]: Соченият тип в лявата страна трябва да има всички квалификатори на сочения тип в дясната страна.

Забележка [2]: T_1 може да бъде от непълен тип ако другият указател има тип void*.

Опит за всякакви други присвоявания, различни от описаните в Табл.45 без изрично преобразуване, ще доведе до генериране на грешка при компилиране.

1.8.3 Обичайни унарни преобразувания

Обичайните унарни преобразувания се прилагат върху:

- операндите на унарните операции !, -, +, ~, *
- по отделно върху операндите на битовите операции << и >>
- по отделно върху операндите на логическите операции **&&** и **||**
- име на масив
- име на функция.

Табл. 46 описва всички обичайни унарни преобразувания.

тип на операнда	резултат от обичайните унарни преобразувания
float	Не се преобразува
Масив от тип T[1]	Указател към тип T
Функция връщаща тип T[2]	Указател към функция, връщаща тип T
char, unsigned char, signed char	int
signed short	int
unsigned short[3]	▪ int (ако int може да представи всички стойности на signed short) ▪ unsigned int (ако int не може да представи всички стойности на signed short)
Битово поле int[4]	▪ int (ако тип int е еквивалентен на signed int) ▪ unsigned int (ако тип int е еквивалентен на unsigned int)
Битово поле signed int	int
Битово поле unsigned int	unsigned int
enum-тип[5]	int

Табл. 46 Обичайни унарни преобразувания

Забележка[1]: Име на масив не се подлага на преобразуване в следните ситуации:

- Когато името на масив е операнд на оператора sizeof: sizeof(име-масив)

  В този случай резултатът е броя на байтове, заемани от елементите на масива.

- Когато името на масив е операнд на адресния оператор &: &име-масив

В този случай резултатът е указател към целия масив, т.е **тип (*)[размер-на-масива].**

- Когато низов литерал се използва за инициализация на char или w_char масив:
  char име-масив[] = "Some string";

  В този случай елементите на масива се инициализират със символите на низа.

Забележка[2]: Името на функция не се подлага на преобразуване в следните ситуации:

- Когато името на функция е операнд на оператора sizeof: sizeof(име-функция).

  В този случай компилаторът ще генерира грешка, тъй като това е невалидна конструкция в C.

- Когато името на функция е операнд на адресния оператор &: &име-функция

  Тази конструкция изрично преобразува името на функцията в указател към нея, т.е. **тип (*)(списък-параметри)**

Забележка[3]: Тип unsigned short се преобразува в тип int когат размерът на тип int е 32-бита и в unsigned int когато размерът на тип int е 16-бита.

Забележка[4]: Битово поле дефинирано само като int може да бъде както signed така и unsigned. Това зависи от имплеметацията.

Забележка[5]: Типът на enum-променливите е имплементационно зависим. В зависимост от стойностите на enum-константите компилаторът може да избере най-подходящия от целочислените типове char (signed, unsigned), short (signed, unsigned) или int. Ето защо всички стойности на един enum-тип могат да се представят винаги от тип int.

1.8.4 Обичайни бинарни преобразувания

Обичайните бинарни преобразувания се прилагат върху операндите на бинарните оператори[1] и върху втория и третия операнд на условния оператор **?:**.

Забележка[1]: Бинарните двойки операторите << и >> и && и || са изключение от това правило. Върху операндите на тези оператори се прилагат само обичайните унарни преобразувания.

Преди да се приложат обичайните бинарни преобразувания, операндите на бинарните оператори се подлагат на обичайните унарни преобразувания ако е необходимо. Табл.47 описва правилата, по които се извършват обичайните бинарни преобразувания.

ако единият операнд е	а другия е	то компилаторът преобразува операндите в
long double	double, float, unsigned long, signed long, unsigned int, signed int	long double
double	float unsigned long signed long unsigned int signed int	double
float	unsigned long signed long unsigned int signed int	float
unsigned long	signed long unsigned int signed int	unsigned long
signed long	unsigned int	signed long ако signed long може да представи всички стойности на unsigned int, иначе двата операнда се преобразуват в unsigned long[1]
signed long	signed int	signed long
unsigned int	signed int	unsigned int
char signed char unsigned char signed short unsigned short enum-тип	char signed char unsigned char signed short unsigned short enum-тип	signed int или unsigned int съгласно правилата за целочислените повишения (виж Табл.46)

Табл. 47 Обичайни бинарни преобразувания

Забележка[1] : Ако тип long и int са с еднакъв размер (32 бита), signed long не може да представи всички стойности на unsigned int.

1.8.5 Преобразуване на типа на аргументите при извикване на функция.

1.8.5.1 Преобразувания при липса на прототип

Когато една функция не е покрита с прототип се извършва подразбиращо се преобразуване на аргументите съгласно следните правила:

- Аргументите от тип char (signed или unsigned), short (signed или unsigned), enum-типове и битови полета се подлагат на целочислено повишаване на типа преди да бъдат подадени на функцията (вижте точка 2 Целочислено повишение на типовете).

- Аргументите от тип float се преобразуват в тип double.

Ако типовете на аргументите след преобразуването не са съвместитми с типовете на параметрите, поведението е недефинирано.

Пример 5-13:

```
/* сорс-файл1 */
#include <stdio.h>

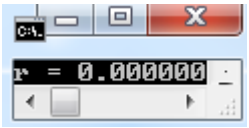
double Sqr();

int main(void)
{
    printf("r = %f\n", Sqr(3));

    return 0;
}
```

```
/* сорс-файл2 */

double Sqr(double d)
{
    return d*d;
}
```



Както се вижда резултатът от изпълнението на прогамата е грешен. Причината е, че функцията Sqr() се извиква от сорс-файл1 без да е осигурен прототип (а само предварителна декларация). Аргументът, който се подава е тип int и е различен от типа на параметъра, в случая double. Както беше казано, поведението в тази ситуация е недефинирано, тъй като компилаторът няма информация за типа на параметъра на функцията и не знае дали трябва да предприеме някакви преобразувания. Следователно генерирания код, който извършва копиране на стойността на аргумента в параметъра ще бъде грешен, а от там и резултата от изпълнението на функцията.

1.8.5.2 Преобразувания при наличие на прототип

Когато една функция е покрита с прототип типовете на аргументите се преобразуват към типовете на параметрите по същите правила както при

присвояване (вижте точка 8.2 Преобразувания при присвояване). Подразбиращи се преобразувания на типа на аргументите не се извършват.

Пример 5-14:

```
/* сорс-файл1 */
#include <stdio.h>

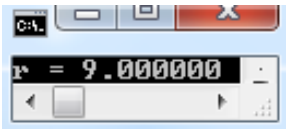
double Sqr(double);

int main(void)
{
    printf("r = %f\n", Sqr(3));

    return 0;
}
```

```
/* сорс-файл2 */

double Sqr(double d)
{
    return d*d;
}
```



Това е същият пример, както в предходната подточка с тази разлика, че в сорс-файл1 е добавен протоптипа на функцията Sqr(). Сега компилаторът знае, че преди да копира стойността на аргумента в параметъра, той трябва да се преобразува в тип double, т.е компилаторът ще генерира допълнителен код, който извършва това пробразуване и след това преобразуваната стойност ще бъде копирана в параметъра.

1.8.6 Преобразуване на типа на стойността на оператора return при връщане на резултат от функция.

Ако типа на израза, връщан от оператора return се различава от връщания тип, указан в дефиницията на функцията, компилаторът автоматично ще се опита да го преобразува преди да върне стойността. Например нека разгледаме следната обобщена дефиниция на функция:

```
тип-резултат    име-функция(списък-параметри)
{
        ...
        return    връщан-израз;
}
```

Ако типът на **връщан-израз** е различен от **тип-резултат,** то действието, което ще извърши компилатора е еквивалентно на следното:

```
тип-резултат    име-функция(списък-параметри)
{
        ...
        return   (тип-резултат)връщан-израз;
}
```

Пример 5-15:

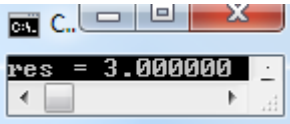
```
#include <stdio.h>
```

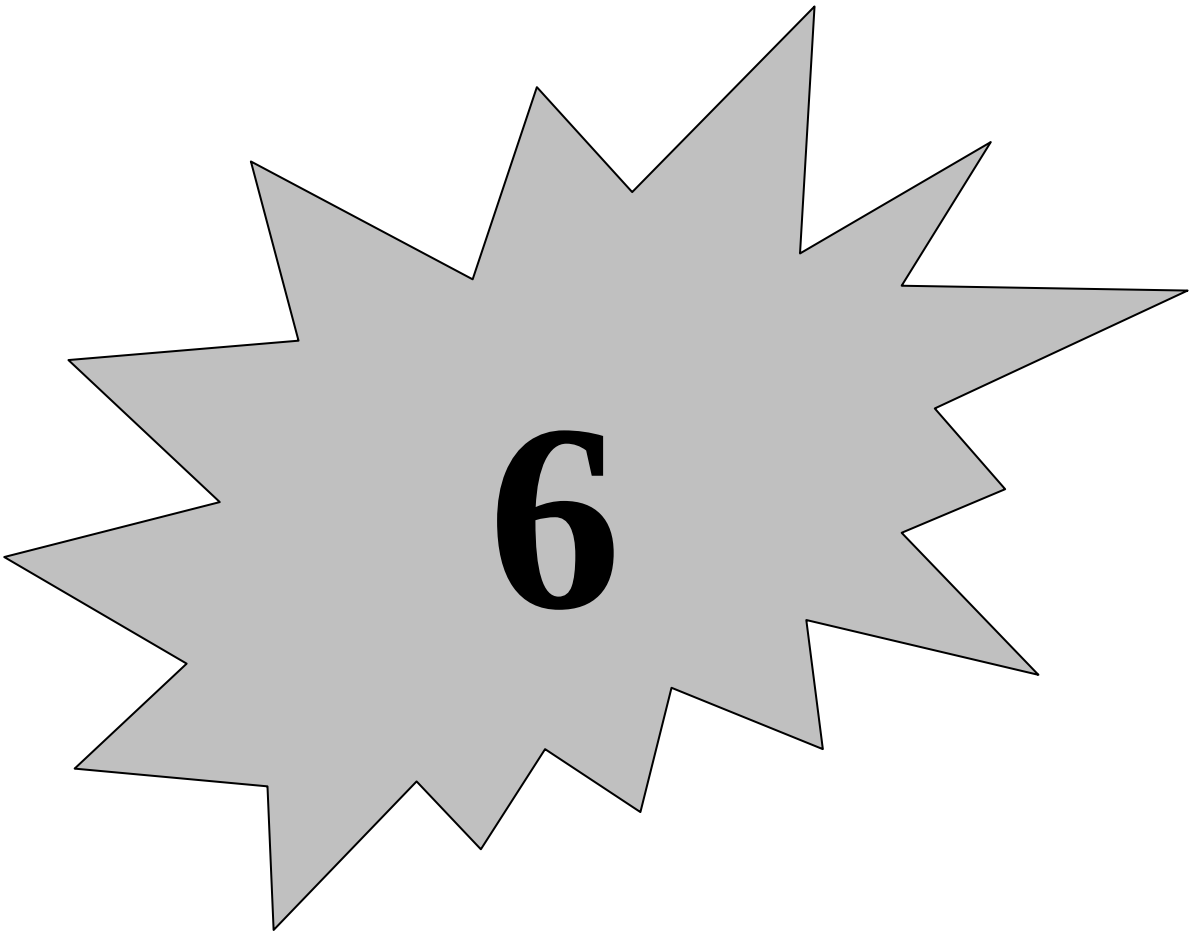
```c
int RetInt(void)
{
	return 3.3; /* Еквивалентно е на return (int)3.3 */
}

int main(void)
{
	double res;

	res = RetInt();
	printf("res = %f\n", res);

	return 0;
}
```





Оператори

1.1 Определение и видове оператори

Операторът е езикова конструкция, която извършва някакво действие. Това действие може да бъде присвояване на стойност на променлива, извикване на функция и други.

Езикът С дефинира следните видове оператори:

- Операторни изрази
- Съставни оператори
- Условни оператори if-else
- Оператор за избор switch
- Итеративни оператори (оператори за цикли: for, while, do-while)
- Оператори break и continue
- Оператор return
- Оператор goto
- Нулев оператор

С изключение на съставните оператори, операторите завършват с точка и запетая.

1.2 Операторни изрази

Всеки израз, завършващ с точка и запетая, се третира като оператор. Компилаторът изчислява израза и изхвърля резултата ако има такъв.

Операторен израз

израз;

където

израз

- Константа
- Променлива
- Аритметичен израз
- Логически израз
- Израз за присвояване

```
        ▪ Извикване на функция
        ▪ Други
```

Операторният израз има смисъл само ако изчислението му включва странични ефекти (вижте Глава 4 Операции и изрази). Компилаторът не е задължен да изчислява израз, който не включва странични ефекти и чийто резултат се изхвърля.

Пример 6-1:

```
void foo(void);

int main(void)
{
  int x;

  4;     /* Константа като операторен израз  */
  x;     /* Променлива като операторен израз */
  x + 4; /* Аритметичен израз без странични ефекти */
  foo(); /* Израз за извикване на функция */
  x = 1; /* Израз за присвояване */
  x++;   /* x = x + 1 */

  return 0;
}

void foo(void)
{
  /*...*/
}
```

1.3 Съставни оператори

Съставният оператор се нарича още блок с код и се състои от декларации и оператори, затворени във фигурни скоби { }.

Съставен оператор (блок с код)

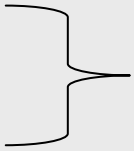
```
{
   /* Декларации на променливи и функции */
   extern тип име-променива1;
   ...
   extern тип име-променливаN;
   extern тип име-функция1(списък-параметри);
   ...
   extern тип име-функцияN(списък-параметри);

   /* Декларации на типове с typedef, struct, union, enum */
   typedef декларация-на-тип    ново-име-на-тип;
   ...

   /* Дефиниране на променливи */
   тип име-променлива1;
   ...
   тип име променливаN;
```

Декларации

```
   /* Оператори */
   оператор1;
   ...
   операторN;
}
```



Съставният оператор може да се използва навсякъде където може да се използва единичен оператор. Той въвежда нова област на видимост. Всеки идентификатор, дефиниран вътре, е видим само от точката на дефиниране до края на блока, освен когато е скрит от дефиниция на идентичен идентификатор във вътрешен блок. Всички декларации в блока също са видими само в него.

Пример:

```
/* Съставен оператор - блок с код */
{
   /* Декларации */
   extern int  limit;

   /* Дефиниции */
   double      result = 0.0;
   static long status = 0;

   /* Оператори */
   if(status == 0)
   {/*  Вътрешен блок */
      int i = 0;

      while(status == 0 && i < limit )
      {/* По-вътрешен блок */

            /* ... */

      }
   }
   else
   {/*  Вътрешен блок */

        /* ... */

   }
}/* Край на съставния оператор */
```

Декларацията на променливата limit е видима само в тялото на съставния оператор, което означава, че limit също е видима само в тялото на съставния оператор. Променливите result и status са видими също само в тялото на съставния блок. Променливата i е видима само в тялото на оператора if.

Един съставен оператор може да се дефинира самостоятелно, както е показано в горния пример, но само в рамките на функция, или като тяло на операторите if-else, switch, while, do-while и for.

1.3.1 Декларации и дефиниции в съставен оператор

Ако идентификатор на функция се декларира без спецификатор за клас памет в блок се допуска, че класа памет е extern. Когато една функция се декларира в блок единствения спецификатор за клас памет, който може да се укаже е extern. Ако идентификатор на променлива се декларира без спецификатор за клас памет в блок се допуска, че класът памет е auto.

Пример:

```
{
	void foo(int x); /* Декларация на fooo(), клас памет extern подразбиране  */
	int i;		/* Дефиниция на променлива, клас памет auto по подразбиране */

	/*...*/

	foo(i);

	/*...*/
}
```

Ако една променлива се декларира с изричен спецификатор за клас памет extern в блок, памет за нея не се заделя и инициализатор за променливата не е позволен. Тази декларация указва, че променливата е дефинирана някъде другаде, или в същия или в друг сорс файл.

Пример:

```
{
	extern int i; /* Декларация на външна променлива */
	extern int j = 10; /* !!!Грешка, инициализатор не е позволен тук */

	/*...*/
}
```

Ако една променлива със спецификатор за клас памет auto или register се декларира в блок, компилаторът заделя памет за нея всеки път при влизане в блока и я освобождава при излизане от него. Ако липсва инициализатор, стойността на променливата е недефинирана. Времето на живот на auto- и register- променливите се простира в целия блок, т.е съвпада с областта им на видимост.

Пример 6-2:

```
#include <stdio.h>

int main(void)
{
	register int x; /* Съдържа  недефинирана стойност */

	for(x = 0; x < 5 ; x ++)
	{
		int i = 0; /* Инициализира се в 0 всеки път при влизане в блока */
```

```
            i++;

            printf("i = %d\n", i);
      }

      /*i = 10;*/ /* !!!Грешка, променливата i не се вижда извън блока, в който е
                     дефинирана */

      return 0;
}
```



Регистровата променлива x има област на видимост и време на живот в тялото на функцията main, докато променлвата i има област на видимост и време на живот само в тялото на цикъла for.

Ако една променлива се декларира с изричен спецификатор за клас памет static в блок, компилаторът заделя памет за нея още преди програмата да започне изпълнението си. Ако липсва инициализатор, компилаторът инициализира променливата в 0 (за целочислени типове), 0.0 (за типове с плаваща запетая), NULL (за указателни типове). Ако се използва явен инициализатор той трябва да бъде литерален константен израз. Инициализацията се извършва само веднъж преди програмата да започне изпълнението си. При излизане от блока статичната променлива запазва стойността си. Областта на видимост на такава променлива си остава блока, но времето на живот се запазва докато трае изпълнението на програмата.

Пример 6-3:

```
#include <stdio.h>

int main(void)
{
      static int x; /* Инициализира се в 0 автоматично само веднъж (при стартиране
                       на програмата) */
      for( ; x < 5 ; x ++)
      {
            static int i = 0; /* Инициализира се в 0 автоматично само веднъж (при
                                 стартиране на програмата) */
            i++;
            printf("i = %d\n", i);
      }

      /*i = 10;*/ /* !!!Грешка, променливата i не се вижда извън блока в който е
                     дефинирана */

      return 0;
}
```



1.4 Условен оператор if-else

Условният оператор if-else служи за отклонение на програмата по един от два възможни пътя в зависимост от дадено условие. По-долу са показани различните синтактични разновидности на този оператор.

Условни оператор if-else

1
```
if (условие)
  if-оператор;
```

2
```
if (условие)
{
   if-оператор1;
    ...
   if-операторN;
}
```

3
```
if (условие)
  if-оператор;
else
  else-оператор;
```

4
```
if (условие)
{
   if-оператор1;
   ...
   if-операторN;
}
else
{
   else-оператор1;
   ...
   else-операторN;
}
```

където

условие
Може да бъде всеки израз, чийто резултат е от скаларен тип (аритметичен или указателен тип).

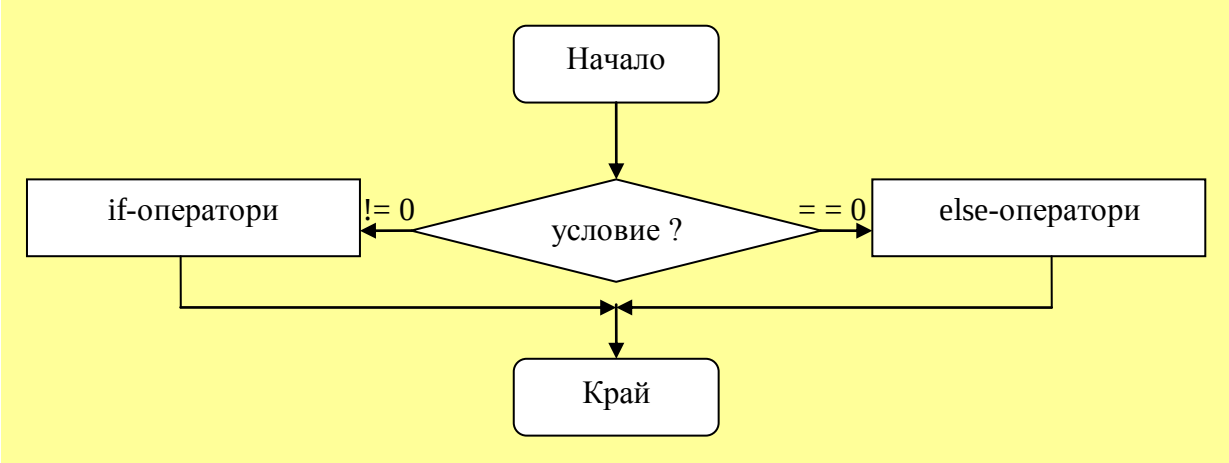
if-оператор/if-operator1,..., if-operatorN
else-оператор/else-оператор1,..., else-операторN
Може да бъде всеки един от С операторите, включително и вложен if-else оператор.

Както се вижда else-частта е незадължителна. Операторът if-else работи по следния начин:

1. Изчислява се условието в скобите на if-частта.

2. Ако резултатът е различен от нула, се изпълнява оператора if-оператор (или операторите if-оператор1,...,if-операторN).
3. Ако резултатът е нула и има else-част, се изпълнява оператора else-оператор (или операторите else-оператор1,...,else-операторN).

Фиг.42 представя if-else оператора в графичен вид.



Фиг. 43 if-else оператор

Ако е необходимо да се изпълнят няколко оператора наведнъж те трябва да бъдат затворени в съставен оператор (вижте синтаксис 2. и 4.).

Когато условието е различно от нула е прието да се казва, че то е ИСТИНА (TRUE), а ако е нула, че е ЛЪЖА (FALSE).

Пример 6-4:

```
#include <stdio.h>

int main(void)
{
	int x = 4;

	if( (x % 2) == 0 )
		printf("x is even\n");
	else
		printf("x is odd\n");

	return 0;
}
```



В показания пример условието на оператора if е израза **(x % 2) == 0**. При x равно на 4 той дава като резултат 1, т.е ИСТИНА.

Не е задължително резултата от изчислението на условието да се проверява изрично дали е равен или различен от нула. Например предходният пример може да се пренапише по следния начин.

Пример 6-5:

```
#include <stdio.h>

int main(void)
{
```

```
    int x = 4;

    if(x % 2)
        printf("x is odd\n");
    else
        printf("x is even\n");

    return 0;
}
```

или

```
if( !(x % 2) )
    printf("x is even\n");
else
    printf("x is odd\n");
```



Въпреки че операторът if-else позволява изпълнението на програмата да продължи по един от два възможни пътя, е възможно този избор да бъде увеличен. За целта като оператор на else-частта трябва да се използва друг if-else оператор. Подобна конструкция се нарича **if-else-if стълбица**.

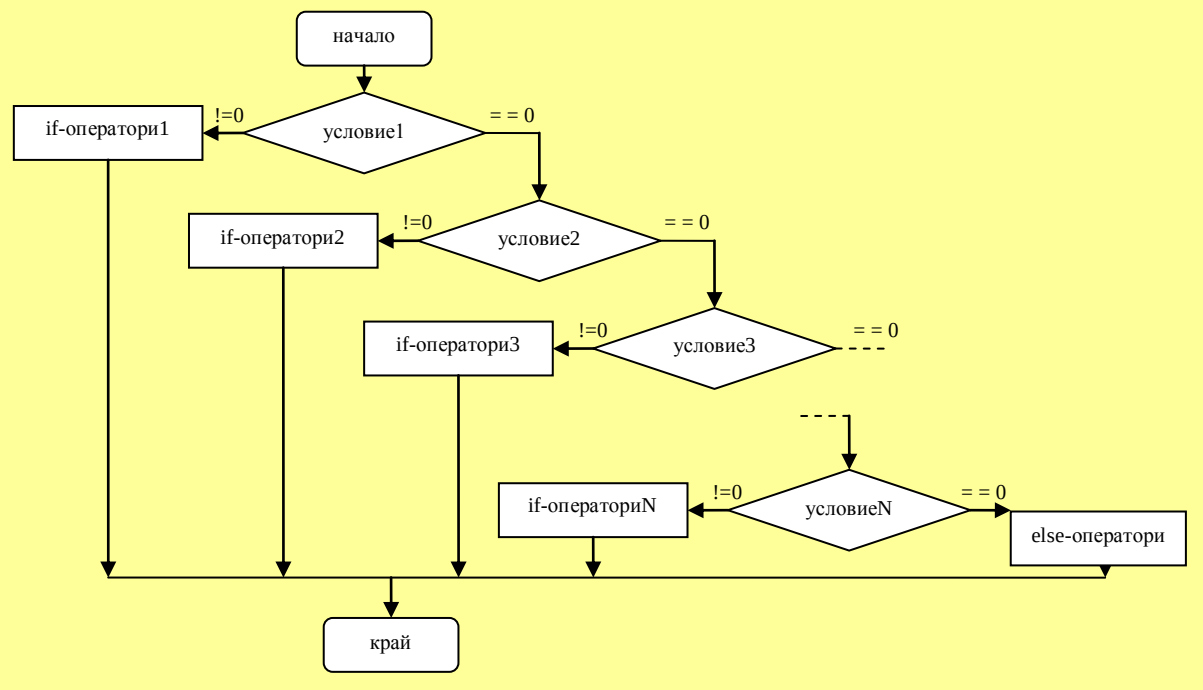
if-else-if стълбица

```
if(условие1)
  {if-оператори1}
else if(условие2)
  {if-оператори2}
else if(условие3)
  {if-оператори3}
...
else
  {else-оператори}
```

Тази конструция е компактен запис на следната конструкция.

```
if(условие1)
  {if-оператори1}
else
    if(условие2)
      {if-оператори2}
    else
        if(условие3)
          {if-оператори3}
        else
            ...
                else
                   {else-оператори}
```

Фиг.43 представя if-else-if стълбицата в графичен вид.



Фиг. 44 if-else-if стълбица

При if-else-if стълбицата се изпълняват само if-операторите на първото условие което е ИСТИНА. Ако нито едно от условията не е ИСТИНА, се изпълняват else-операторите на завършващата else-част. Ако липсва завършваща else-част, нито един от операторите на if-else-if стълбицата не се изпълнява.

Обърнете внимание на следният често срещан проблем, който може да възникне при използване на оператора if-else.

```
if(k > 0 )
      if( m > 0)
            printf ("k and m > 0\n");
else
      printf ("k < 0 \n");
```

На пръв поглед поради начина на подравняване на кода изглежда, че else-частта принадлежи на първия if. Това е логичекса грешка, тъй като в действителност else-частта принадлежи на втория if както е показано по-долу.

```
if(k > 0 )
      if( m > 0)
            printf ("k and m > 0\n");
      else
            printf ("k < 0 \n");
```

За да коригирате това е необходимо да използвате съставен оператор както е показано в следващия пример.

```
if(k > 0 )
{
      if( m > 0)
```

```
            printf ("k and m > 0\n");
}
else
      printf ("k < 0 \n");
```

За да се избягва този проблем се препоръчва тялото на if и else-частта да бъде съставен оператор, състоящ се дори и от един оператор, т.е:

```
if(k > 0 )
{
      printf ("k > 0\n");
}
else
{
      printf ("k < 0 \n");
}
```

Използването на съставен оператор води до избягване и на следния проблем:

```
if(k > 0 )
      printf ("k > 0\n");
else
      printf ("error \n");
      printf ("k < 0 \n");
```

Тук отново поради подравняването на кода изглежда, че последните два оператора принадлежат на else-частта. Това е логичекса грешка, тъй като в действителност само първия от тях принадлежи на else-частта както е показано по-долу.

```
if(k > 0 )
      printf ("k > 0\n");
else
      printf ("error \n");
printf ("k < 0 \n");
```

За да коригирате това е необходимо да използвате съставен оператор както е показано в следващия пример.

```
if(k > 0 )
      printf ("k > 0\n");
else
{
      printf ("error \n");
      printf ("k < 0 \n");
}
```



Винаги използвайте съставен оператор за тяло на if и else операторите както е показано в синтактична конструкция 4, дори то се състои само от един оператор Това ще Ви предпази от логическа грешка, ако впоследствие решите да добавите други оператори.

1.5 Оператор за избор switch

Операторът switch служи за отклонение на програмата по един от няколко възможни пътя в зависимост от стойността на целочислен израз.

Оператор switch

```
switch (целочислен-израз)
{
case константен-израз1:

     case1-оператор1;
     ...
     case1-операторN;

     break;

case константен-израз2:

     case2-оператор2;
     ...
     case2-операторN;

     break;

...

case константен-изразN:

     caseN-оператор1;
     ...
     caseN-операторN;

     break;

default:

     deafault-оператор1;
     ...
     deafault-операторN;

     break;
}
```

Тяло на оператора switch

където

целочислен-израз
Може да бъде всеки израз, който връща стойност от целочислен тип.

константен-израз1...константен-изразN
Може да бъде всеки литерален константен целочислен израз.

case1-оператор1...caseN-операторN / deafault-оператор1...deafault-операторN
Може да бъде всеки един от С операторите, включително и вложен switch оператор.

Тялото на оператора switch съдържа два типа етикети:

- case-етикети

- default-етикет

Общо двата типа етикети се наричат switch-етикети. Всеки от тези етикети е свързан с целочислен константен израз и оператори.

Операторът switch работи по следния начин:

1. Изчислява се стойността на целочисления израз.

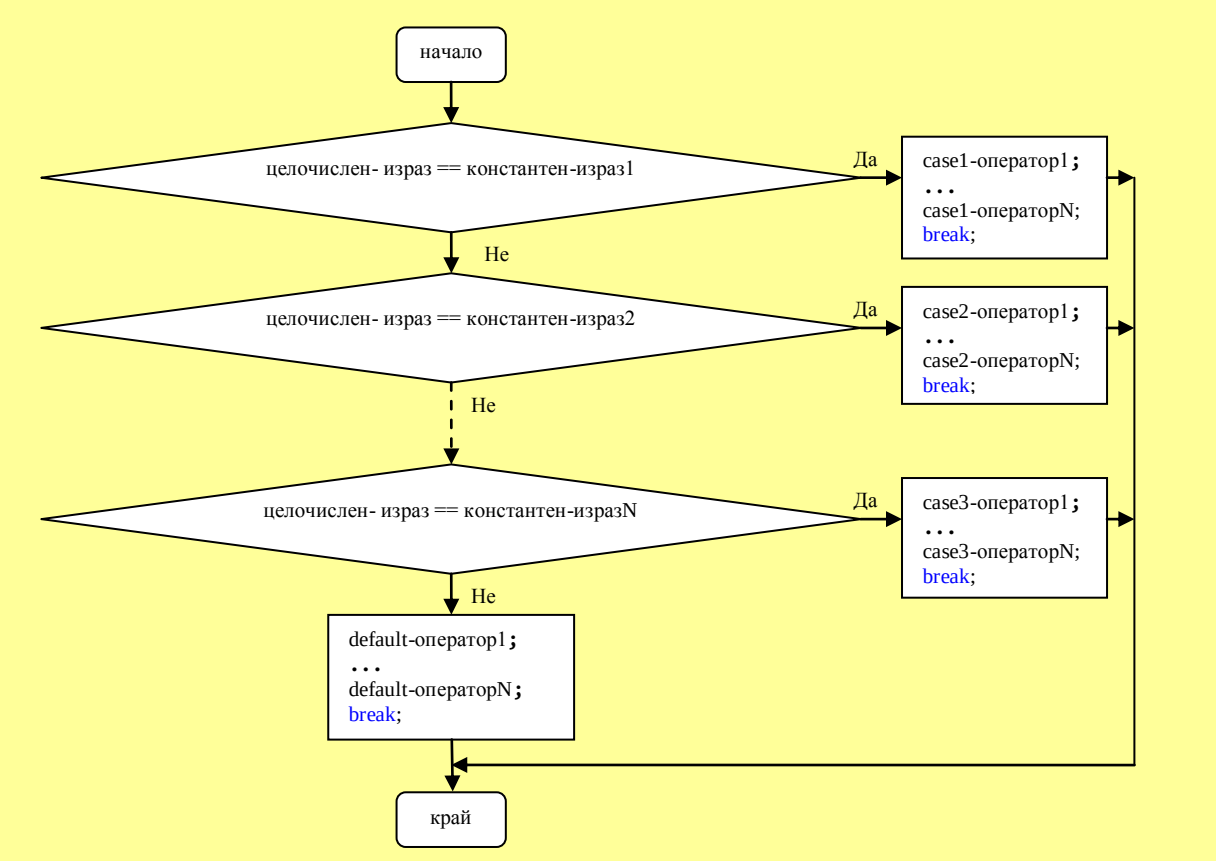
2. Тази стойност се сравнява със стойностите на константните целочислени изрази, принадлежащи на case-етикетите.

3. Ако се намери съвпадение, се изпълняват операторите принадлежащи на съответния case-етикет. Изпълнението продължава докато се срещне оператор break или затварящата фигурна скоба на оператора switch.

4. Ако няма съвпадение, се изпълняват операторите принадлежащи на default-етикета.

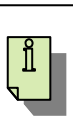
   Default-етикета не е задължителен. Ако няма нито едно съвпадение между **`целочислен-израз`** и **`константен-израз1...константен-изразN`** и липсва default-етикет, операторът switch се прескача и програмата продължава с изпълнението на следващия оператор след него.

Фиг. 45 Представя оператора switch в графичен вид.

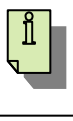


Фиг. 45 Оператор switch

По време на изчислението **`целочислен-израз`** се подлага на обичайните унарни преобразувания (вижте Глава 5 Преобразувания). По време на сравнението съответният литерален целочислен константен израз **`константен-израз1...константен-изразN`** се повишава до преобразувания тип на **`целочислен-израз`**. Стандартът не дефинира редът на сравнение.



`целочислен-израз` се подлага на обичайните унарни преобразувания преди сравнението.



По време на сравнението съответният литерален целочислен константен израз `константен-израз1...константен-изразN` се повишава до преобразувания тип на `целочислен-израз`.



C89 стандартът не дефинира редът на сравнение на `целочислен-израз` и `константен-израз1...константен-изразN`.



`константен-израз1...константен-изразN` трябва да имат различни стойности.

Пример 6-6:

```
#include <stdio.h>

int main(void)
{
	char ch;

	printf("Enter your choice between 1-3 or 4-9: ");
	scanf("%c", &ch);

	switch (ch)
	{
	case '1':
		printf ("Your choice is 1.\n");
		break;

	case '2':
		printf ("Your choice is 2.\n");
		break;

	case '3':
		printf ("Your choice is 3.\n");
		break;

	default:
		printf ("Your choice is between 4-9.\n");
		break;

	}

	return 0;
}
```





Не е задължително default-етикета да бъде на последно място. Той може да се разположи в началото или между case-етикетите.

Пример 6-7:

```
#include <stdio.h>

int main(void)
{
	char ch;

	printf("Enter your choice between 1-3 or 4-9: ");
	scanf("%c", &ch);
```

```
	switch (ch)
	{
	case '1':
		printf ("Your choice is 1.\n");
		break;

	case '2':
		printf ("Your choice is 2.\n");
		break;

	default:

		printf ("Your choice is between 4-9.\n");
		break;

	case '3':
		printf ("Your choice is 3.\n");
		break;

	}

	return 0;
}
```

Ако в някой от switch-етикетите е пропуснато да се добави оператор break след поредицата оператори, принадлежащи на етикета, програмата продължава с изпълнението на операторите на следващия етикет. Това продължава докато се срещне оператор break или се стигне до затварящата фигурна скоба на оператора switch. Сравнете резултата от следващия пример и този от **Пример 6-6**.

Пример 6-8:

```
#include <stdio.h>

int main(void)
{
	char ch;

	printf("Enter your choice between 1-3 or 4-9: ");
	scanf("%c", &ch);

	switch (ch)
	{
	case '1':
		printf ("Your choice is 1.\n");

		/* Липсва оператор break */

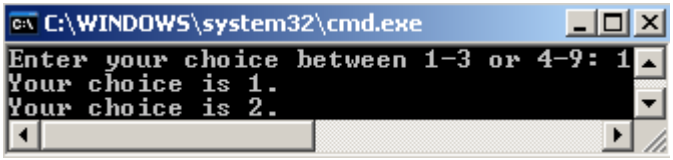
	case '2':
		printf ("Your choice is 2.\n");
		break;

	case '3':
		printf ("Your choice is 3.\n");
		break;

	default:
		printf ("Your choice is between 4-9.\n");
		break;
```

```
	}

	return 0;
}
```



Няколко етикета могат да се добавят един след друг. Това означава, че един и същ код може да се изпълни при различни стойности на **целочислен-израз**.

Пример 6-9:

```
#include <stdio.h>

int main(void)
{
	char ch;

	printf("Enter your choice between 1-9: ");
	scanf("%c", &ch);

	switch (ch)
	{
	case '1':
		printf ("Your choice is 1.\n");

	case '2':
		printf ("Your choice is 2.\n");
		break;

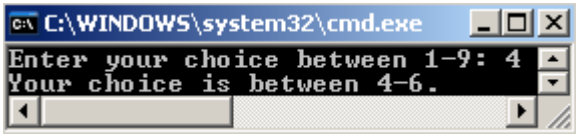
	case '3':
		printf ("Your choice is 3.\n");
		break;

	case '4':
	case '5':
	case '6':
		printf ("Your choice is between 4-6.\n");
		break;

	default:
		printf ("Your choice is between 7-9.\n");
		break;

	}

	return 0;
}
```



Както показва резултатът за стойности на ch '4', '5' или '6' се изпълнява един и същ код.

Обърнете внимание на следната особеност на оператора switch. Операторът switch извършва "скокообразно" предаване на управлението на програмата към съответния switch-етикет (подобно на оператора goto, вижте точка 9 Оператор goto). Това означава, че ако добавите код веднага след отварящата фигурна скоба на switch-оператора преди всички switch-етикети, той няма да се изпълни. Ако този код съдържа дефиниции на auto или register променливи, компилаторът ще задели памет за тях в стека при влизане в тялото на switch оператора, но няма да ги инициализира дори и да имат явен инициализатор.

Пример 6-10:

```
#include <stdio.h>

void foo(void)
{
	printf ("This function will not be called\n");
}

int main(void)
{
	char ch;

	printf("Enter value between 1-3: ");
	scanf("%c", &ch);

	switch (ch)
	{
		int x = 10;/* !!!ВНИМАНИЕ: Променливата x няма да се инициализира */
		foo();	   /* !!!ВНИМАНИЕ: Функцията foo() няма да се извика      */

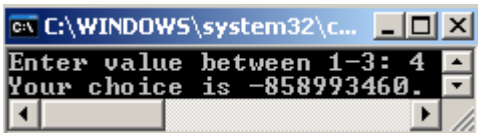
	case '1':
		x = 1;
		printf ("Your choice is %d.\n", x);
		break;

	case '2':
		x = 2;
		printf ("Your choice is %d.\n", x);
		break;

	case '3':
		x = 3;
		printf ("Your choice is %d.\n", x);
		break;

	default:
		printf ("Your choice is %d.\n", x); /* !!!ВНИМАНИЕ: Променливата x е
		                                       неинициализирана */
		break;
	}

	return 0;
}
```



1.6 Итеративни оператори (while, do-while, for)

Итеративните оператори се наричат още оператори за цикъл. Те служат за многократно изпълнение на определен програмен код в зависимост от някакво условие. Езикът C включва три оператора за цикъл: **while**, **do-while** и **for**. Във всеки от тези оператори броят на повторенията на програмния код се определя от условие, което се нарича още управляващ израз. Управляващият израз може да бъде всеки израз, чийто резултат е от скаларен тип (аритметичен или указателен тип).

1.6.1 Оператор за цикъл while

Оператор за цикъл while

1

```
while(условие)
      оператор;
```

2

```
while(условие)
{
      оператор1;
      ...
      операторN;
}
```

където

условие
Може да бъде всеки израз, чийто резултат е от скаларен тип (аритметичен или указателен тип).

оператор/оператор1...операторN
Може да бъде всеки един от C операторите, включително и вложен while оператор.

Операторът while работи по следния начин:

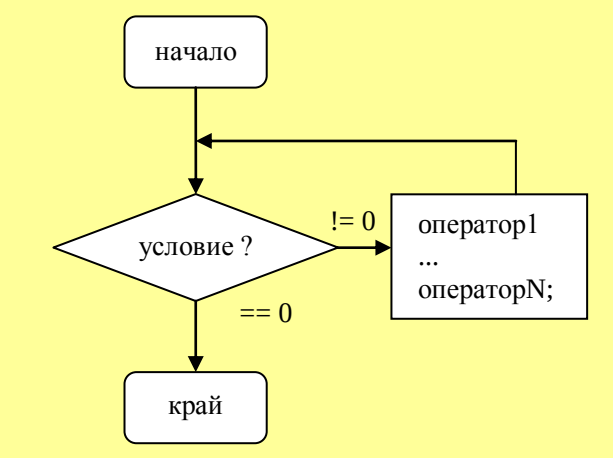
1. Изчислява се условието в скобите

2. Ако резултатът е различен от нула (е ИСТИНА), тялото на цикъла while се изпълнява и условието се изчислява и проверява отново. Описаният цикъл се повтаря докато стойността на условието е различна от нула.

3. Ако резултатът е нула (е ЛЪЖА) тялото на цикъла се прескача и програмата продължава с изпълнението на оператора след цикъла

while.

Фиг.46 представя while оператора в графичен вид.



Фиг. 46 Оператор за цикъл while

Ако тялото на оператора е изградено от повече от един оператор, те трябва да се затворят във фигурни скоби, т.е да се използва съставен оператор (вижте синтаксис 2.).

Тялото на цикъла се изпълнява дотогава докато управляващият израз е ИСТИНА. Ако още при първата итерация (т.е. при първото изчисление на управляващия израз) управляващият израз е 0, тялото на цикъла не се изпълнява въобще.

Ясно е, че стойността на управляващия израз трябва да се променя по някакъв начин, в противен случай цикъла ще бъде безкраен.

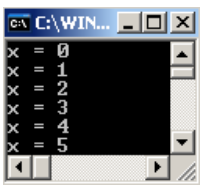
Пример 6-11:

```
#include <stdio.h>

int main(void)
{
	int x = 0;

	while(x < 6)
	{
		printf ("x = %d\n", x);
		x++;
	}

	return 0;
}
```



Махнете конструкцията x++ в горния пример и го стартирайте отново. Ще забележите, че резултата от програмата е безкрайно отпечатване на x = 0 на екрана. Това е напълно очаквано, управляващия израз x < 6 винаги ще дава стойност различна от 0, тъй като никъде в тялото на while-оператора променливата x не се променя.

Излизането от безкарен цикъл може да стане само с операторите break (вижт точка 7.1 Оператор break), return (вижте точка 8 Оператор return) и goto (вижте точка 9 Оператор goto).

Обърнете внимание на следната често срещана грешка:

Пример 6-12:

```
#include <stdio.h>

int main(void)
{
	int x = 0;

	while(x < 6)
		printf ("x = %d\n", x); /* 1) */
		x++;                    /* 2) */

	return 0;
}
```

Начина, по който е подравнен кода може да остави впечатление, че тялото на цъкъла while се състои от операторите 1) и 2). В действителност само първият оператор принадлежи на цикъла while както е показано по-долу.

```
#include <stdio.h>

int main(void)
{
	int x = 0;

	while(x < 6)
		printf ("x = %d\n", x); /* 1) */

	x++;                        /* 2) */

	return 0;
}
```

	**Винаги използвайте съставен оператор за тяло на цикъла while както е показано в синтактична конструкция 2, дори то се състои само от един оператор Това ще Ви предпази от логическа грешка, ако впоследствие решите да добавите други оператори.**

1.6.2 Оператор за цикъл do-while

Оператор за цикъл do-while

1

```
do
   оператор;
while(условие);
```

2

```
do
{
   оператор1;
   ...
   операторN;
} while(условие);
```

където

условие
Може да бъде всеки израз, чийто резултат е от скаларен тип (аритметичен или указателен тип).

оператор/оператор1...операторN
Може да бъде всеки един от С операторите, включително и вложен do-while оператор.

Операторът do-while работи по следния начин:

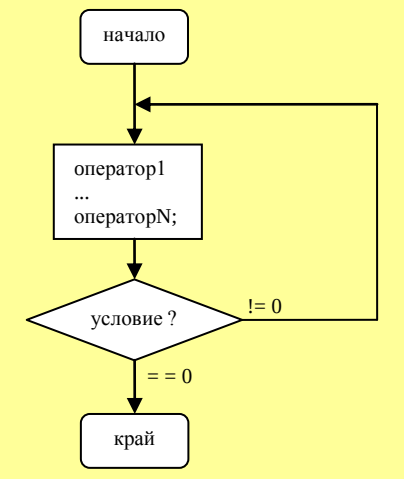
1. Изпълнява се оператора/операторите в тялото на цикъла

2. Изчислява се условието в скобите след while.

3. Ако резултатът е различен от нула (е ИСТИНА), тялото на цикъла do-while се изпълнява отново. Описаният цикъл се повтаря докато стойността на управляващия израз е различна от нула.

4. Ако резултатът е нула (е ЛЪЖА), тялото на цикъла се прескача и програмата продължава с изпълнението на оператора след цикъла do-while.

Фиг.47 представя do-while оператора в графичен вид.



Фиг. 47 Цикъл do-while

Както се вижда цикълът do-while се различава от while само по това, че тялото на do-while се изпълнява поне един път.

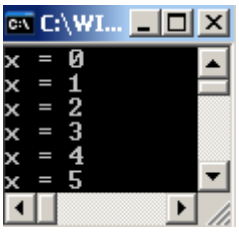
Пример 6-13:

```
#include <stdio.h>

int main(void)
{
	int x = 0;

	do
	{
		printf ("x = %d\n", x);
		x++;
	}while(x < 6);

	return 0;
}
```



	Винаги използвайте съставен оператор за тяло на цикъла do-while както е показано в синтактична конструкция 2, дори то се състои само от един оператор Това ще Ви предпази от грешка при компилиране, ако впоследствие решите да добавите други оператори.

1.6.3 Оператор за цикъл for

Оператор за цикъл for

1

```
for (инициализация; условие; изменение)
    оператор;
```

2

```
for (инициализация; условие; изменение)
{
    оператор1;
    ...
    операторN;
}
```

където

инициализация/изменение
Може да бъде всеки валиден C ираз или изрази разделени със запетая.

условие
Може да бъде всеки израз, чийто резултат е от скаларен тип (аритметичен или указателен тип).

```
оператор/оператор1...операторN
      Може да бъде всеки един от С операторите, включително и вложен for оператор.
```

От всички оператори за цикъл операторът for е най-гъвкав. Той работи по следния начин:

1. Изчислява се изразът/изразите от секцията **инициализация.**

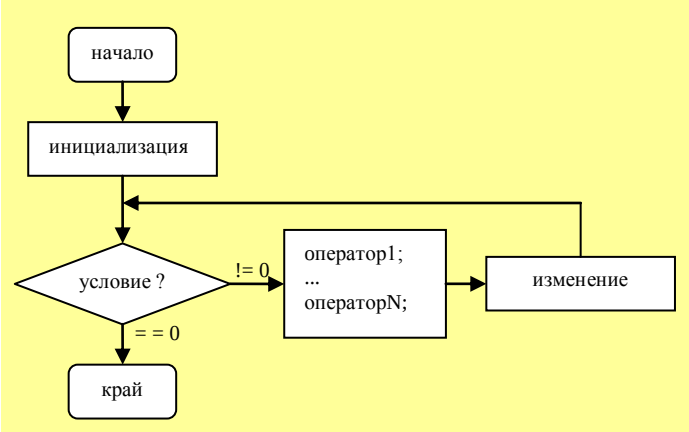
2. Изчислява се управляващия израз от секцията **условие.**

3. Ако резултатът е различен от нула (е ИСТИНА), тялото на цикъла for се изпълнява.

4. Изчислява се изразът/изразите от секцията **изменение**, след което управляващия израз от секцията **условие** се изчислява и проверява отново. Описаният цикъл се повтаря докато стойността на управляващия израз е различна от нула.

5. Ако резултатът от изчислението на управляващия израз е нула (е ЛЪЖА) тялото на цикъла се прескача и програмата продължава с изпълнението на оператора след цикъла for.

Фиг.47 представя for оператора в графичен вид.



Фиг. 48 Оператор за цикъл for

Както и при while оператора ако още при първото изчисление стойността на управляващия израз е нула, тялото на for цикъла не се изпълнява въобще. С изключение на поведението на оператора continue (вижте точка 7.2 Оператор continue), следната конструкция с while е еквивалента на цъкъл for.

```
инициализация; /* Може да включва няколко израза, които се изпълняват
                  последователно един след друг */
while (условие)
{
      оператор1;
      ...
```

```
        операторN;

        изменение; /* Може да включва няколко израза, които се изпълняват
                      последователно един след друг */
}
```

Типично секцията **инициализация** представлява инициализиране на променлива/променливи, която/които управлява/управляват цикъла. В секцията **условие**, тази/тези променлива/променливи се проверява/проверяват дали дадено условие е изпълнено, а в секция **изменение** променливата/променливите се изменя/изменят по някакъв начин.

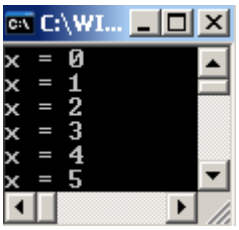
Пример 6-14:

```
#include <stdio.h>

int main(void)
{
        int x;

        for (x = 0 ; x < 6 ; x++)
        {
                printf ("x = %d\n", x);
        }

        return 0;
}
```



Описаната процедура на използване на оператора for е типична, но не и задължителна. Стандартът С допуска и трите секции на цикъла да бъдат празни, точките и запетаите, разделящи секциите, обаче трябва да присъстват. Например секциите **инциализация** и **изменение** в следващия пример са оставени празни, тъй като променливата x се инициализира преди цикъла и се изменя вътре в тялото му.

Пример 6-15:

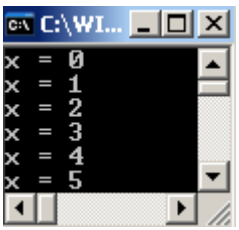
```
#include <stdio.h>

int main(void)
{
        int x = 0;

        for ( /* Празна */ ; x < 6 ; /* Празна */ )
        {
                printf ("x = %d\n", x);
                x++;
```

```
	}

	return 0;
}
```



Също така изразите в тези секции не е задължително да са свързани логически. Например в следващия пример променливата y, която няма отношение към for цикъла, се инициализира в секцията **инициализация**, заедно с променливата x.

Пример 6-16:

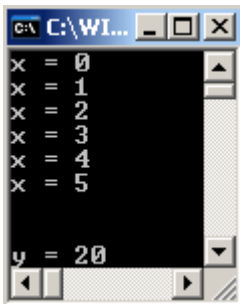
```
#include <stdio.h>

int main(void)
{
	int x, y;

	for (y = 20, x = 0; x < 6 ; x++)
	{
		printf ("x = %d\n", x);
	}

	printf ("\n\ny = %d\n", y);

	return 0;
}
```

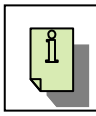


Подобно използване на цикъла for обаче е объркващо и е лоша програмна практика.

Ако секцията **условие** се остави празна, това се интерпретира от компилатора като ИСТИНА и цикълът става безкраен.

```
for ( ; ; )
{
	printf ("This loop never terminates\n", x);
}
```

Излизането от безкарен цикъл може да стане само с операторите break (вижте точка 7.1 Оператор break), return (вижте точка 8. Оператор return) и goto (вижте точка 9 Оператор goto).



Ако управляващ-израз на for е празен, това се интерпретира от компилатора като ИСТИНА и цикъла for става безкраен.

1.7 Оператори break и continue

1.7.1 Оператор break

Операторът break може да се появавя само в тялото на оператор switch (вижте точка 5 Оператор за избор switch) или в тялото на операторите за цикли while, do-while и for (вижте точка 6 Итеративни оператори). Използването на break извън този контекст ще доведе до грешка при компилиране.

Оператор break

```
оператор-switch-или-цикъл
{
      ...
      break;
      ...
      break;
      ...
}

където

оператор-switch-или-цикъл
      Може да е ключовата дума swtict, while, do-while или for
```

Резултатът от изпълнението на оператора break е прекратяване на изпълнението на съответния оператор към когото принадлежи и преход към следващия оператор непосредствено намиращ се след затварящата фигурна скоба на тялото на **оператор-switch-или-цикъл**.

```
switch(управляващ-израз)
{
      ...
      break;
      ...
}
следващ-оператор
```

```
while(управляващ-израз)
{
        ...
        break;
        ...
}
следващ-оператор
```

```
do
{
        ...
        break;
        ...
} while(управляващ-израз);
следващ-оператор
```

```
for(израз1 ; управляващ-израз ; израз3)
{
        ...
        break;
        ...
}
следващ-оператор
```

В следващия пример операторът break се използва за излизане от безкраен цикъл while при достигане на определено условие.

Пример 6-17:

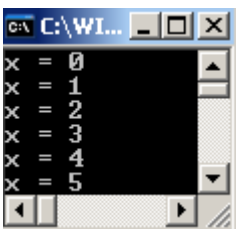
```
#include <stdio.h>

int main(void)
{
        int x = 0;

        while(1)
        {
                printf ("x = %d\n", x);
                x++;

                if (x == 6)
                {
                        break;
                }
        }/* Затваряща фигурна скоба на while */

        return 0;
}
```



Обърнете внимание, че ако операторът към когото принадлежи break е вложен в друг такъв оператор, то break води до излизане само от вложения оператор, т.е.

```
switch(управляващ-израз)
{
        ...
        while (условие)
        {
                ...
                break;
                ...
        }
        следващ-оператор
        ...
        break;
        ...
}
следващ-оператор
```

оператор while, вложен в оператор switch.

```
for(израз1 ; управляващ-израз ; израз3)
{
        ...
        for(израз1 ; управляващ-израз ; израз3)
        {
                ...
                break;
                ...
        }
        следващ-оператор
        ...
        break;
        ...
}
следващ-оператор
```

оператор for, вложен в друг оператор for.

1.7.2 Оператор continue

Операторът continue може да се появявя само в тялото на операторите за цикли while, do-while и for (виж точка 6 Итеративни оператори). Използването на continue извън този контекст ще доведе до грешка при компилиране.

Оператор continue

```
оператор-за-цикъл
{
        ...
        continue;
        ...
        continue;
```

```
        ...
}

където

оператор-за-цикъл
        while, do-while или for
```

Изпълнението на оператора continue води до прескачане на всички оператори, намиращи се след него, и стартиране на нова итерация (завъртане) на цикъла. За циклите while и do-while това означава преход към изчисление на управляващия израз, а за цикъл for преход към секцията **изменение**.

```
while(условие)
{
        ...
        continue;
        ...
}
```

```
do
{
        ...
        continue;
        ...
} while(условие);
```

```
for(инициализация; условие ; изменение)
{
        ...
        continue;
        ...
}
```

Следващият пример отпечатва на екрана само четните числа от 1 до 10 с помощта на оператора continue.

Пример 6-18:

```
#include <stdio.h>

int main(void)
{
        int x;

        for (x = 1; x <= 10 ; x++)
        {
                if (0 != (x % 2))
                {
                        continue; /* Прескача всички оператори след него */
                }

                printf("x = %d\n", x);
        }

        return 0;
```

```
}
```



Обърнете внимание, че ако оператора към когото принадлежи continue е вложен в друг оператор за цикъл, то continue води до стартиране на нова итерация само на вложения цикъл, т.е

```
for(израз1; управляващ-израз ; израз3)
{
      ...
      while(управляващ-израз)
      {
            ...
            continue;
            ...
      }
      ...
      continue;
      ...
}
```

оператор while, вложен в оператор for.

1.8 Оператор return

Операторът return води до прекратяване на изпълнението на функцията в която се намира. Управлението на програмата се предавава непосредствено след точката, в която функцията се извиква.

Оператор return

1

```
return;
```

2

```
return израз;
```

където

израз

Може да бъде всеки израз, различен от void и чийто тип съвпада или може да се преобразува към типа указан в декларацията на функцията.

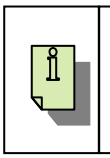
Първата форма се използва за функции с тип void, а втората за функции с тип, различен от void.

```
void  име-функция(списък-параметри)
{
      ...
      return;
      ...
```

```
        return;
}

тип-резултат  име-функция(списък-параметри)
{
        ...
        return израз1;
        ...
        return изразN;
}
```

Ако типа на **израз1...изразN** се различава от **тип-резултат**, компилаторът ще се опита да го преобразува, използвайки същите правила за преобразуване както при присвояване.



C89 стандартът позволява функции с тип, различен от void да използват оператор return без израз, но поведението в този случай е недефинирано.

Повече за return вижте в Глава 13 Функции.

1.9 Оператор goto

Операторът goto е оператор за безусловен преход към друг оператор, указан с етикет (**label**). Областта на действие на goto е само тялото на функцията, в която операторът се използва. Преход между функции с помощта на goto е невъзможен.

Оператор goto

```
тип-резултат име-функция(списък-параметри)
{
        ...
        ...
        goto име-етикет-на-оператор;
        ...
        ...
име-етикет-на-оператор:
        оператор;
        ...
        ...
        return израз;
}

където

име-етикет-на-оператор
        Задава се с помощта на идентификатор.
```

На всеки оператор може да се назначи етикет. Етикетът завършва с двуеточие. Операторът може да се намира непосредствено след

двуеточието, но за по-добра нагледност обикновено се поставя на следващия ред.

```
име-етикет:оператор;

или

име-етикет:
          оператор;
```

Обозначеният с етикет оператор може да се намира преди или след goto конструкцията, която прави преход към него.

Следващият пример симулира цикъл while с помощта на операторите goto и if.

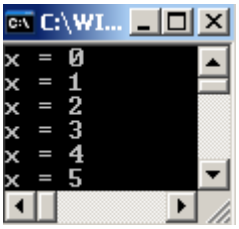
Пример 6-19:

```
#include <stdio.h>

int main(void)
{
      int x = 0;

repeat:
      if (x < 6)
      {
            printf("x = %d\n", x);
            x++;
            goto repeat;
      }

      return 0;
}
```



Следващият пример използва goto за излизане от цикъл при първото срещане на нечетно число в масив.

Пример 6-20:

```
#include <stdio.h>

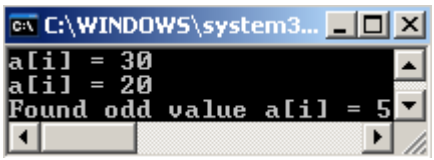
int main(void)
{
      int i;
      int a[4] = {30,20,5,60};

      for (i = 0 ; i < 10 ; i++)
```

```
	{
		if(0 == (a[i] % 2))
		{
			printf("a[i] = %d\n", a[i]);
		}
		else
		{
			goto PrintMessage;
		}
	}

PrintMessage:
    printf("Found odd value a[i] = %d\n", a[i]);

	return 0;
}
```



Използването на goto може да доведе до т.нар. "спагети код". Това е код, при който по време на изпълнението му програмата прави преходи напред-назад, водещи до трудно проследяване и разрушаване на принципите на структурното програмиране. Освен това goto позволява преход в средата на съставен оператор, заобикаляйки инициализирането на променливите дефинирани в началото на блока.

Пример 6-21:

```
#include <stdio.h>

int main(void)
{
	int x = 10;

	goto print_i;

	if( x < 6)
	{
		int i = 22;

print_i:
		printf (" i = %d\n", i);
	}

	return 0;
}
```



Поради тези причини използването на goto не се препоръчва. В действителност този оператор е излишен. Всяка една програма може да се напише без използването на goto. Може би единственото смислено

използване на goto е за излизане от дълбока вложеност.



Операторът goto може да направи програмата доста заплетена. Повечето специалисти препоръчват този оператор да не се използва изобщо.

1.10 Нулев оператор

Нулевият оператор се състои само от точка и запетая.

Нулев оператор ;

Нулевият оператор не извършва никакви действия. Той е полезен главно в две ситуации.

- За обозначаване на празно тяло на операторите за цикли

Пример 6-22:

```
#include <stdio.h>

int main( void )
{
	unsigned long i;

	printf("Wait...\n");
	for (i = 1000000000; i > 0 ; i--); /* Цикъл for с празно тяло */
	printf("Done! You may continue\n");

	return 0;
}
```

В показания пример операторът for се използва само да осигури определено закъснение преди програмата да продължи изпълнението си и не е необходимо да изпълнява някакви действия в тялото си. По тази причина след затварящата скоба на цикъла се поставя нулев оператор. Ако се изпусне точката и запетая след цикъла for, следващият оператор става част от него.

Пример 6-23:

```
#include <stdio.h>

int main( void )
{
	unsigned long i;

	printf("Wait...\n");
	for (i = 1000000000; i > 0 ; i--) /* !!! ВНИМАНИЕ !!! */
	printf("Done! You may continue\n"); /* Сега това е част от цикъла for */
```

```
	return 0;
}
```

Следните две конструкции са еквивалентни:

```
for (i = 1000000000; i > 0 ; i--);
```

и

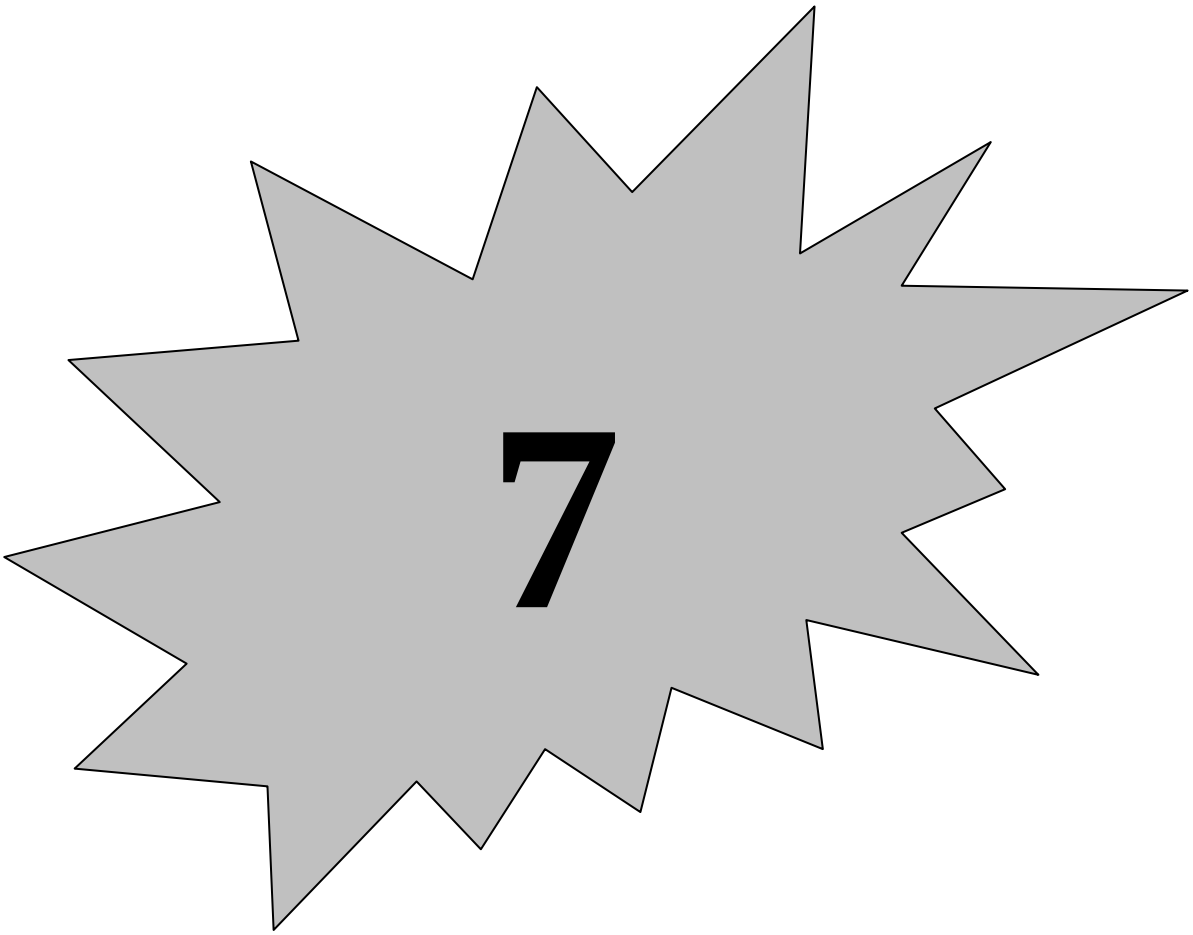
```
for (i = 1000000000; i > 0 ; i--) { /* Тяло без оператори */ }
```

- За поставяне на етикет преди затварящата фигурна скоба на съставен оператор.

Пример:

```
if (expr)
{
	...
	goto Label;
	...
Label:
	; /* Празен оператор */
}
```

Операцията goto може да прави преход само към оператор. Затова ако искаме да прескочим всички оператори в един съставен блок с помощта на goto, е необходимо пред затварящата скоба да добавим нулев оператор с етикет.



Изброявания

1.1 Деклариране на изброяване

Изброяването е потребителски тип, който представалява набор от целочислени константи, зададени чрез идентификатори. Изброяване се декларира по следния начин:

Деклариране на изброяване

```
enum име-изброяване_опц                    } Тип на изброяването
{                                          }
      име-enum-константа1,                 }
      име-enum-константа2,                 } Тяло на изброяването
      ...                                  }
      име-enum-константаN                  }
};
```

Декларацията на изброяване се въвежда с ключовата дума enum, последвана от името и тялото на изброяването. Забележете, че декларацията на изброяването завършва с точка и запетая. Името на изброяването се нарича още етикет (**tag**). Идентификаторите **име-enum-константа1,..., име-enum-константаN**, декларирани в тялото на изброяването се наричат enum-константи или членове на изброяването. По подразбиране първият идентификатор приема стойност 0, следващия 1 и т.н., т.е. всяка enum-константа е с единица по-голяма от предходната.

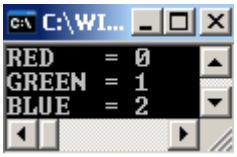
Пример 7-1:

```
#include <stdio.h>

/* Деклариране на изброяване */
enum rgb {RED, GREEN, BLUE};

int main(void)
{
      printf("RED   = %d\n", RED);
      printf("GREEN = %d\n", GREEN);
      printf("BLUE  = %d\n", BLUE);

      return 0;
}
```



Пример 7-1 дефинира изброяване с име rgb, което съдържа три константи, представени с идентификаторите RED, GREEN и BLUE. Стойностите, които тези идентификатори представляват са съответно 0, 1 и 2. Това се вижда и от резултата от изпълнението на програмата.

Освен стойности по подразбиране, enum-константите могат да се

инициалицират явно с инициализатори.

Деклариране на изброяване и явно задаване на стойности

```
enum име-изброяване_опц                                  } Тип на изброяването
{                                                        }
      име-enum-константа1 = инициализатор,               }
      име-enum-константа2 = инициализатор,               } Тяло на изброяването
      ...                                                }
      име-enum-константаN = инициализатор                }
};

където

инициализатор
      Целочислен литерален константен израз от тип int.
```

Тази форма позволява задаването на едни и същи стойности на различни идентификатори. Ако enum-константа не е изрично инициализирана, нейната стойност е с 1 по-голяма от предходната.

Пример 7-2:

```
#include <stdio.h>

/* Деклариране на изброяване */
enum rgby {RED = 10, GREEN, BLUE = 3, YELLOW = 3};

int main(void)
{
      printf("RED     = %d\n", RED);
      printf("GREEN   = %d\n", GREEN);
      printf("BLUE    = %d\n", BLUE);
      printf("YELLOW  = %d\n", YELLOW);

      return 0;
}
```



Името на изброяването може да се пропусне.

Пример 7-3:

```
#include <stdio.h>

/* Деклариране на изброяване */
enum {RED = 10, GREEN, BLUE = 3, YELLOW = 3};

int main(void)
{
      printf("RED     = %d\n", RED);
      printf("GREEN   = %d\n", GREEN);
```

```
        printf("BLUE    = %d\n", BLUE);
        printf("YELLOW  = %d\n", YELLOW);

        return 0;
}
```

Enum-константите са от тип int и могат да се използват навсякъде където могат да се използват константи от този тип.

Също така може да дефинирате и променливи от enum-тип. Това може да стане по време на декларирането на изброяването или по късно в кода. Комбинацията от ключовата дума **enum** и **име-изброяване** представлява типа на изброяването. Именно този тип се използва за дефиниране на enum-променливи по-късно в кода.

Деклариране на изброяване и enum-променливи едновременно

```
enum име-изброяване[1]опц                          } Тип на изброяването
{                                                  }
        име-enum-константа1 = инициализатор,       }
        име-enum-константа2 = инициализатор,       } Тяло на изброяването
        ...                                        }
        име-enum-константаN = инициализатор        }
}списък-enum-променливи;
```

където

инициализатор
Целочислен литерален константен израз от тип int.

списък-enum-променливи
име-enum-променлива1, име-enum-променлива2,..., име-enum-променливаN

Забележка[1]: Ако няма да дефинирате други enum-променливи, освен тези указани в **списък-enum-променливи**, **име-изброяване** може да се изпусне. Ако обаче възнамерявате да дефинирате enum-променливи по-късно в кода, ще трябва да добавите и **име-изброяване**.

Дефиниране на enum-променливи отделно от декларацията на изброяване

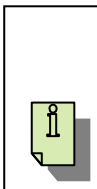
```
enum име-изброяване       списък-enum-променливи;
```

където

списък-enum-променливи
име-enum-променлива1, име-enum-променлива2,..., име-enum-променливаN

След като enum-променливите се дефинират на тях могат да им се присвояват стойности от списъка с enum-константи (или може да ги инициализирате по време на дефинирането им като всяка променлива). Типът на enum-променливите е имплементационно зависим. В зависимост от стойностите на enum-константите компилаторът може да избере най-

подходящия от целочислените типове char (signed, unsigned), short (signed, unsigned) или int.



Типът на enum-променливите е имплементационно зависим. В зависимост от стойностите на enum-константите компилаторът може да избере най-подходящия от целочислените типове char (signed, unsigned), short (signed, unsigned) или int.

Пример 7-3:

```
#include <stdio.h>

/* Деклариране на изброяване rgby и enum-променлива едновременно */
enum rgby{RED = 10, GREEN, BLUE = 3, YELLOW = 3} color;

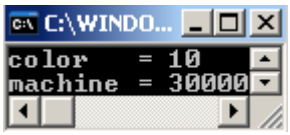
/* Деклариране на изброяване vehicles */
enum vehicles {CAR = 4, BUS = 50, TRAIN = 30000};

int main(void)
{
    enum vehicles  machine; /* Дефиниране на enum-променлива по-късно в кода */

    color = RED;
    printf("color   = %d\n", color);

    machine = TRAIN;
    printf("machine = %d\n", machine);

    return 0;
}
```



Тъй като enum-променливите по същество са целочислени типове, то на тях може да им се присвояват и стойности извън списъка с enum-константи без да е необходимо явно преобразуване. Например на enum-променливата machine от Пример 7-3 може да й присвоим стойност 1000. Подобно смесване на стойности може да е объркващо и не се препоръчва. Освен това, тъй като типа на enum-променливите зависи от имплементацията, подобни практики може да доведат до препълване. Например компилаторът може да избере за enum-променливата color тип signed char. Тогава конструкцията

```
color = 200;
```

ще се компилира, но ще доведе до препълване с недефиниран резултат.



Използвайте само enum-константи за присвояване на enum-променливи, в противен случай рискувате да създадете логическа грешка, ако типа на изброяването не може да представи стойността на целочислената константа.

1.2 Изброяване и typedef

Както обяснихме в началото, типът на изброяването се състои от ключовата дума enum и името на изброяването, т.е. **enum име-изброяване**. Използвайки ключовата дума typedef, можете да създадете синоним на enum-тип, който се състои просто от един идентификатор.

Деклариране на синоним на enum тип

```
/* Деклариране на изброяване */
enum име-изброяване_опц
{
      име-enum-константа1,
      име-enum-константа2,
      ...
      име-enum-константаN
};

/* Деклариране на синоним на изброяването */
typedef enum име-изброяване   синоним-на-изброяването;
```

Пример 7-4:

```
/* Деклариране на изброяване _rgb*/
enum _rgb{RED, GREEN, BLUE};

/* Деклариране на синоним rgb на типа enum _rgb */
typedef enum _rgb  rgb;

int main(void)
{
      rgb color1;       /* Дефиниране на променлива от тип rgb      */

      enum _rgb color2; /* Дефиниране на променлива от тип enum _rgb */

      /*...*/

      return 0;
}
```

Както показва горния пример променливи от enum тип могат да се дефинират както чрез използване на синонима на изброяването, така и чрез ключовата дума enum и името на изброяването. Променливите color1 и color2 са от един и същ тип.

Ключовата дума typedef може да се приложи и директно върху

декларацията на изброяването.

Деклариране на синоним на enum-тип

```
typedef enum име-изброяване[1]опц
{
      име-enum-константа1,
      име-enum-константа2,
      ...
      име-enum-константаN

} синоним-на-изброяването;
```

Заележка[1]: име-изброяване може да се изпусне. За дефиниране на enum-променливи в този случай обаче може да се използва само синонима, зададен с **синоним-на-изброяването.**

Пример 7-5:

```
/* Деклариране на изброяване и синоним едновременно */
typedef enum _rgb {RED, GREEN, BLUE} rgb;

int main(void)
{
      rgb color1;       /* Дефиниране на променлива от тип rgb      */
      enum _rgb color2; /* Дефиниране на променлива от тип enum _rgb */

      /*...*/

      return 0;
}
```

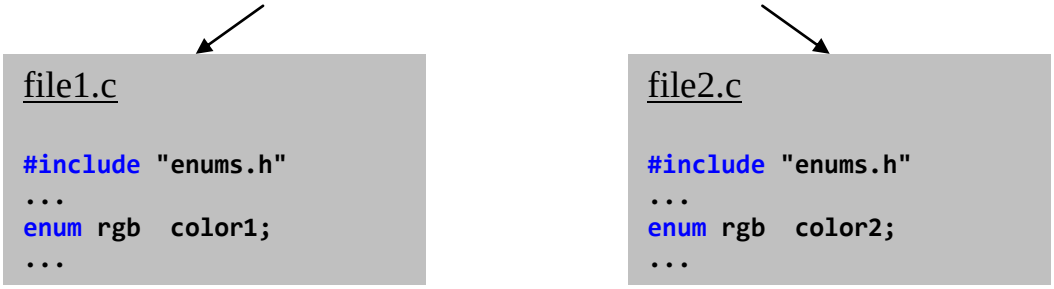
1.3 Област на видимост на изброяване

Декларацията на едно изброяване е видима от точката на деклариране до края на файла или блока, в който се намира декларацията. Всеки файл, който използва изброяването трябва да включва и неговата декларация. В такива случаи е най-добре изброяването да се декларира в хедър-файл и този файл да се включва във всеки сорс-файл, който го използва както е показано по-долу.

enums.h

```
#ifndef ENUMS_H
#define ENUMS_H
...
enum rgb { RED, GREEN, BLUE};
...
#endif
```





Ако изброяването се използва само в един сорс-файл, обикновено декларацията се поставя в началото на файла преди дефинициите на функциите.

Обърнете внимание, че за разлика от структурите и обединенията, тялото на изброяването не дефинира отделна област на видимост. Това означава, че идентификаторите на enum-константите трябва да се различават от други идентификатори в същата област на видимост. Въпреки това име на изброяване може да съвпада с име на enum-константа.

Пример 7-6:

```
/* Деклариране на изброяване rgb*/
enum rgb {RED, GREEN, BLUE, rgb};

/* int rgb; */ /* !!!Грешка, има същата област на видимост като enum-константата
                  rgb */

int main(void)
{
      int  rgb; /* Допустимо, но скрива enum-константата rgb */

      /*...*/

      return 0;
}
```

Име на изброяване може да съвпада с друго име в същата област на видимост. От контекста компилаторът определя кой идентификатор се използва.

Пример 7-7:

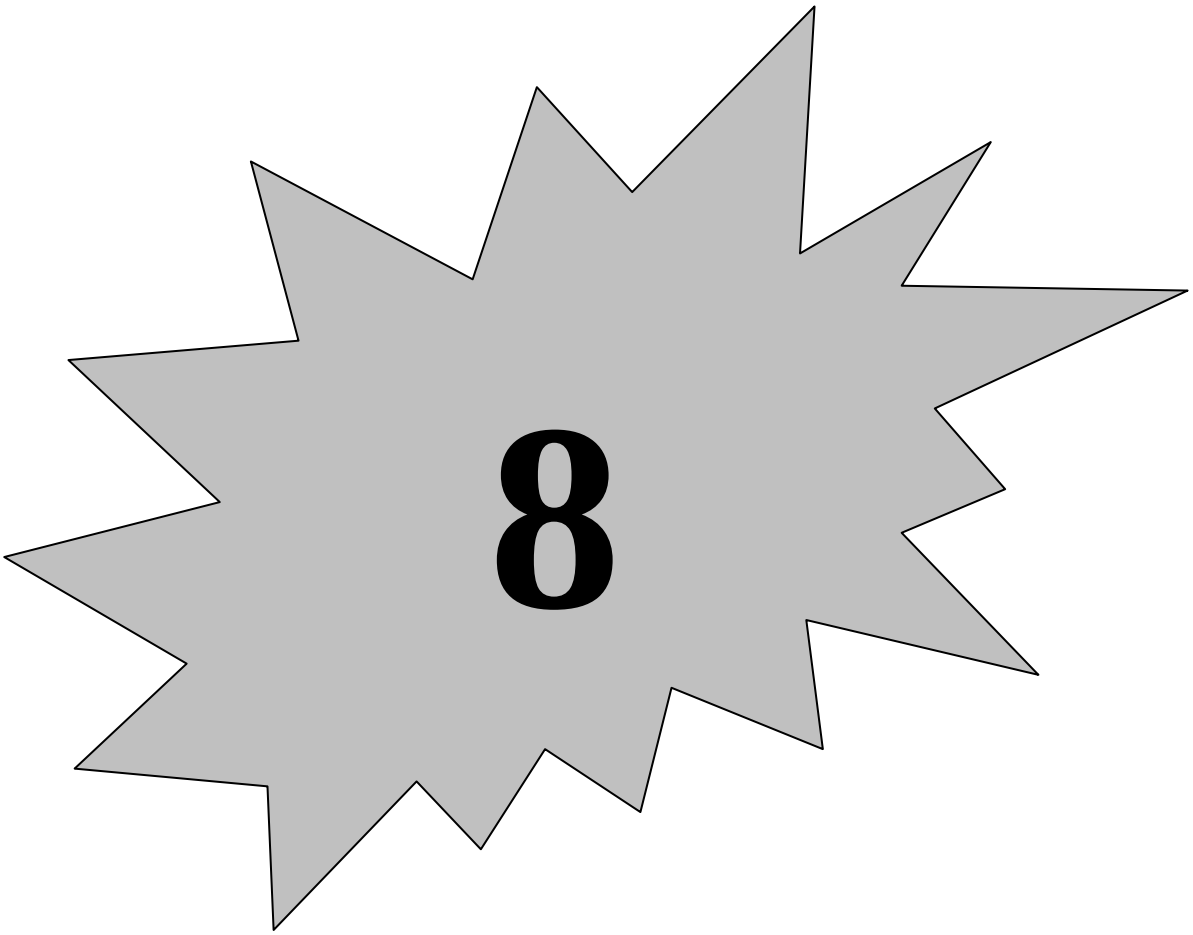
```
/* Деклариране на изброяване rgb*/
enum  rgb {RED, GREEN, BLUE};

enum  rgb    color; /* Дефиниране на глобална променлива color от тип enum rgb */
int          rgb;   /* Дефиниране на глобална променлива rgb от тип int        */

int main(void)
{
      color = RED;
      rgb   = 100;

      /*...*/

      return 0;
}
```



Масиви

1.1 Едномерни Масиви

1.1.1 Определение и дефиниране

Масивът е съвкупност от променливи от един и същ тип и с общо име. Променливите на масива се наричат елементи и се разполагат последователно в паметта като първият елемент на масива се разполага на най-младшия адрес, вторият елемент на следващия адрес и т.н. Елементите на масива могат да бъдат от всеки тип с изключение на битови полета и функции, т.е масив от битови полета и масив от функции са недопустими в езика С.

Дефиниране на едномерен масив:

```
тип    име-масив[размер];

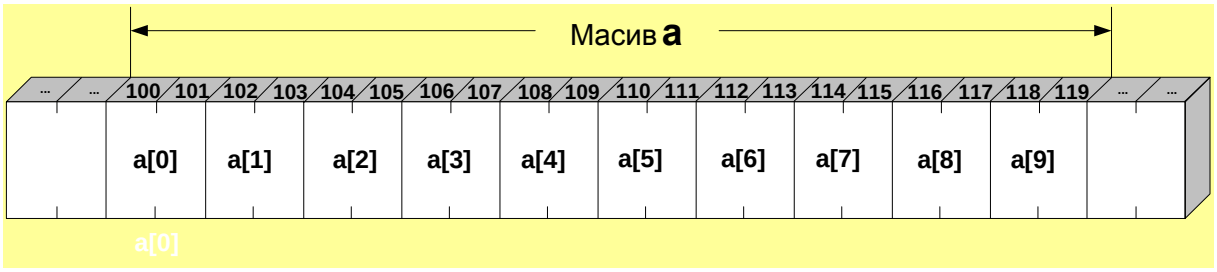
където

тип
       Дефинира типа на елементите на масива. Елементите на масив могат да бъдат от
       всеки тип с изключение на битови полета и функции.

размер
       Определя броя на елементите на масива. Трябва да бъде литерален константен
       целочислен израз.
```

Пример: `int a[10];/* дефиниране на масив с име a, с 10 елемента от тип int */`

Когато компилаторът срещне тази конструкция той заделя памет за 10 int елемента в паметта на компютъра както е показано на Фиг.49 (приема се, че тип int заема 2 байта).



Фиг. 49 Разполагане на едномерен масив в паметта

1.1.2 Инициализиране

Елементите на масива могат да се инициализират по време на дефинирането му.

Инициализиране на едномерен масив:

```
тип име-масив[размер] = {списък-инициализатори[1]};
```

или

тип име-масив[][2] **= {списък-инициализатори};**

където

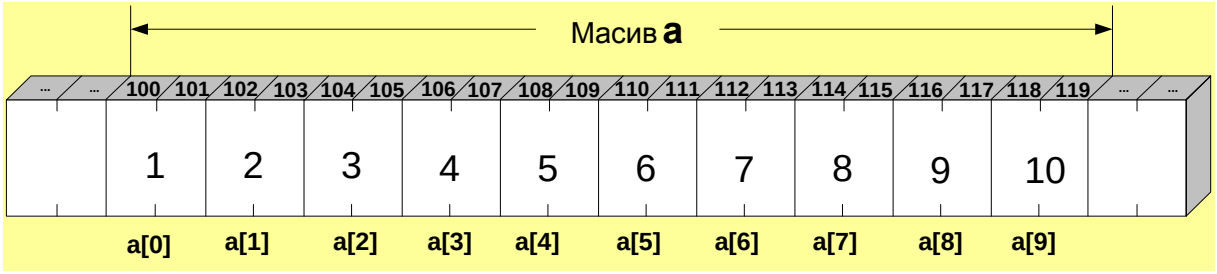
списък-инициализатори
Литерални константни изрази, разделени със запетая.

Забележка[1]**:** Броят на инициализаторите трябва да е по-малък или равен на броя на елементите на масива, указан с **размер**. Ако броят на инициализаторите е по-малък от броя на елементите, елементите, за които не достигат инициализатори се инициализират с 0 (за целочислени типове), 0.0 (за типове с плаваща запетая) или NULL (указателни типове).

Забележка[2]**:** Когато масив се инициализира по време на дефинирането си, **размер** може да се пропусне. Броят на елементите се определя от броя на инициализаторите.

Пример: `int a[10] = {1,2,3,4,5,6,7,8,9,10};`

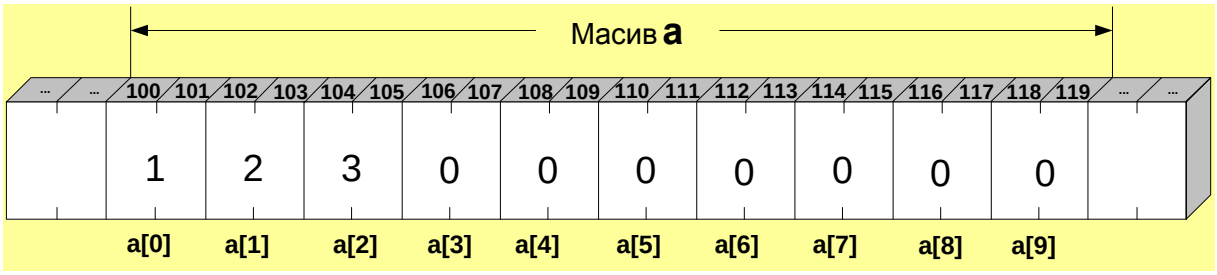
Когато компилаторът срещне тази конструкция той заделя памет за 10 int елемента в паметта на компютъра и ги инициализира както е показано на Фиг.50 (Приема се, че тип int заема 2 байта).



Фиг. 50 Инициализиране на едномерен масив

Ако броя на инициализиращите стойности е по-малък от размера на масива, елементите на масива, за които не достигат инициализиращи стойности, се нулират (Фиг.51).

Пример: `int a[10] = {1,2,3};`



Фиг. 51 Инициализиране на едномерен масив с по-малък на брой инициализатори

Ако масив се инициализира по време на неговата дефиниция, то размера

може да се пропусне. В този случай компилаторът автоматично изчислява размера на масива въз основа на броя на инициализиращите стойности.

Пример: `int a[] = {1,2,3,4,5,6,7,8,9,10};`

Спомнете си, че низовите константи представляват съвкупност от символи, разположени последователно в паметта и завършват с нулев символ '\0', т.е низовите константи са едномерни масиви от символи, чийто размер е с единица по-голям от броя на символите в низа. Низовите константи могат да се използват за инициализиране на символни масиви (масиви тип char, signed char и unsigned char).

Инициализиране на символен масив с низова константа:

```
char име-масив[размер[1]] = "низова-константа";

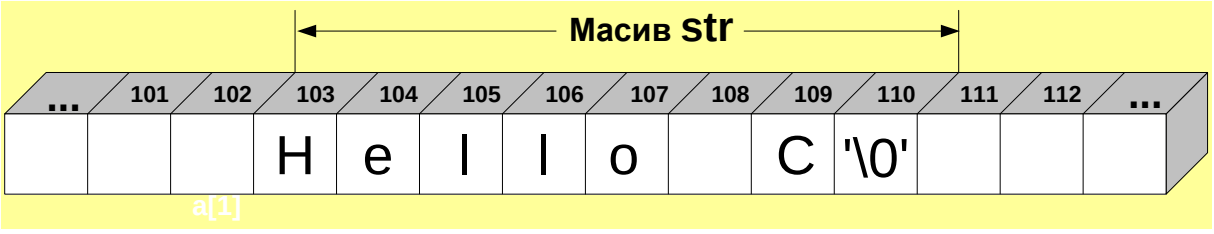
char име-масив[[2]] = "низова-константа";
```

Забележка[1]: Размерът на символен масив трябва да е с единица по-голям от броя на символите на низовата константа.

Забележка[2]: Размерът на символен масив се определя автоматично от броя на символите в низа. В това число влиза и нулевия символ.

Пример: `char str[] = "Hello C";`

Когато компилаторът срещне подобна конструкция той автоматично заделя памет за всички символи на низа плюс един байт за нулевия символ '\0' (Фиг.52).



Фиг. 52 Инициализиране на символен масив с низова константа

Масивът може да се инициализира с низ и символ по символ.

Пример: `char str[] = { 'H', 'e', 'l', 'l', 'o', ' ', 'C', '\0'};`

Ако нулевият символ се пропусне char-масива представлява просто обикновен масив от символи, а не низ.

1.1.3 Индексиране

Елементите на масива се достигат чрез името му и номера (индекса) на елемента в квадратни скоби, т.е **име-масив[индекс]**. Тази операция се нарича индексиране на масива. Първият елемент на масива има номер 0, а последният има номер с единица по-малък от размера на масива. Елементите на масива по нищо не се отличават от обичайни променливи от същия тип и могат да се използват навсякъде където и обичайните променливи могат да се използват.

Индексиране на едномерен масив

име-масив[индекс[1]]

където

индекс
Може да бъде всеки целочислен израз, чиято стойност изпълнява условието

0 ≤ индекс < размер-на-масива, т.е. индекс Є [0...(размер-1)]

Забележка[1]: Отговорност на програмистта е да следи индекса да не излиза извън рамките на масива, в противен случай резултатът е недефиниран.

Пример 8-1:

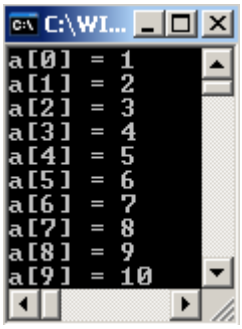
```
#include <stdio.h>

int main(void)
{
	int a[10];
	int i;

	/* Запис на стойности в елементите на масива */
	for( i = 0; i < 10 ; i++ )
	{
		a[i] = i + 1 ;
	}

	/* Отпечатване на стойностите на елементите на масива на екрана */
	for( i = 0; i < 10 ; i++ )
	{
		printf("a[%d] = %d\n", i, a[i]);
	}

	return 0;
}
```

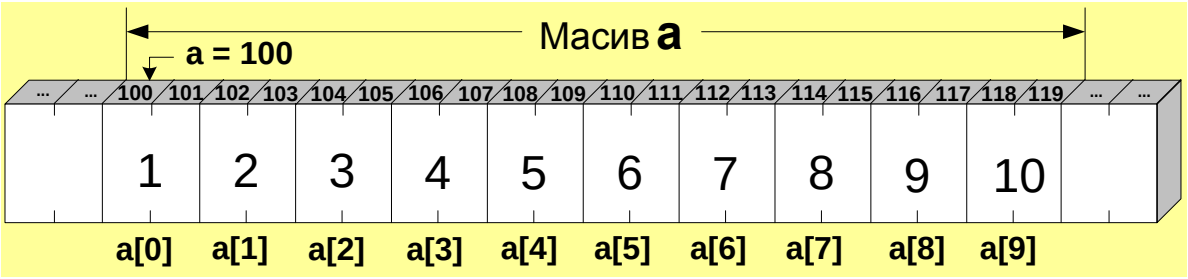


	Езикът C не прави проверка дали индексът излиза извън границите на масива. Отговорност на програмистта е да следи това да не се случва.

	Винаги проверявайте дали стойността на индекса е в границите на масива преди да го индексирате.

1.1.4 Едномерни масиви и указатели

Масивите и указателите в C са тясно свързани. Името на масив в действителност представлява адреса на първия елемент на масива, т.е. името на масив е указател към първият елемент на масива (Фиг.53). Ето защо елементите на масива могат да се достигат и чрез указателна аритметика.



Фиг. 53 Едномерен масив и указатели

- достъп до елементите на масив чрез името на масива като указател

Пример 8-2:

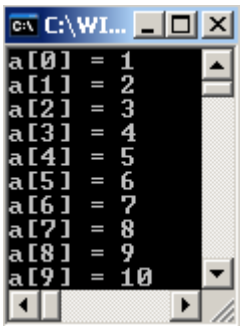
```
#include <stdio.h>

int main(void)
{
	int a[10] = {1,2,3,4,5,6,7,8,9,10};
	int i;

	for(i = 0; i < 10 ; i++ )
```

```
	{
		printf("a[%d] = %d\n", i, *(a + i));
	}

	return 0;
}
```





В С всяка конструкция от вида *x[i]* се преобразува от компилатора в конструкция от вида **(x + i)*, т.е *x[i] == *(x + i)*, където x е име на масив или указател, а i е индекс.

- достъп до елементите на масив чрез указател към началото на масива

Пример 8-3:

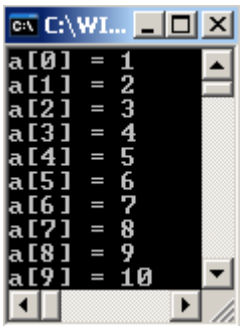
```
#include <stdio.h>

int main(void)
{
	int a[10] = {1,2,3,4,5,6,7,8,9,10};
	int *p;
	int i;

	p = a; /* еквивалентно е на p = &a[0]; */

	for( i = 0; i < 10 ; i++ )
	{
		printf("a[%d] = %d\n", i, *(p + i));
	}

	return 0;
}
```



Указателите могат да се индексират когато се използват за достъп до елементите на масив. Следователно горният пример може да се пренапише

както е показано по-долу.

Пример 8-4:

```
#include <stdio.h>

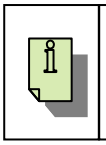
int main(void)
{
	int a[10] = {1,2,3,4,5,6,7,8,9,10};
	int *p;
	int i;

	p = a; /* еквивалентно е на p = &a[0]; */

	for( i = 0; i < 10 ; i++ )
	{
		printf("a[%d] = %d\n", i, p[i]);
	}

	return 0;
}
```

И все пак има една важна разлика между името на масив като указател и обикновен указател. Името на масива е константен указател и за него не се заделя отделно място в паметта на компютъра, т.е. не може да се променя да сочи към други обекти. Просто компилаторът свързва името на масива с адреса на първият му елемент.



Името на масив е константен указател към първия елемент на масива, т.е ако *a* е име на масив, то следните две конструкции са еквивалентни: *a = = &a[0].*

За да се определи броят на елементите на масив по време на изпълнение на програмата може да се използва следната конструкция:

```
sizeof (име-масив) / sizeof (тип-масив)
```

Първият оператор sizeof връща размера на масива в байтове, а вторият sizeof връща размерът в байтове, който всеки елемент на масива заема. Частното на тези две стойности дава броя на елементите на масива.

Предходната програма може да бъде пренаписана по следния начин:

Пример 8-5:

```
#include <stdio.h>

int main(void)
{
	int a[10] = {1,2,3,4,5,6,7,8,9,10};
	int *p;
	int i;
```

```
        p = a; /* еквивалентно е на p = &a[0]; */

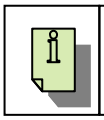
        for( i = 0; i < (sizeof (a) / sizeof (int)) ; i++ )
        {
                printf("a[%d] = %d\n", i, p[i]);
        }

        return 0;
}
```

Предимството на този подход е, че ако трябва да променим размера на масива, е необходимо да променим само размера в дефиницията на масива.

1.1.5 Предаване на масив на функция

Масивите също могат да се подават като аргументи на функция. За целта името на масива служи като аргумент. Тъй като името на масив е указател, то логично е и параметъра, който го приема, да е указател към същия тип данни, т.е към данни от типа на елементите на масива. Това води до следния важен извод: когато масив се предава на функция, това което в действителност се предава е адреса към първия елемент на масива.



Когато масив се предава на функция, това което в действителност се предава е адреса към първия елемент на масива.

Езикът C допуска три алтернативни форми на декларация на функция, която приема масив за аргумент:

1) **тип име-функция(тип име-масив[размер]);**

2) **тип име-функция(тип име-масив[]);**

3) **тип име-функция(тип *име-указател);**

В първата декларация, въпреки че е посочен и размера на масива, той се игнорира от компилатора. Ето защо размерът може да се изпусне както е показано във втората декларация. В крайна сметка тези две декларации се трансформират до третата декларация. Тя най-точно отразява механизма на предаване на масив на функция, но има един недостатък. За външен наблюдател тя означава едно от следните две неща:

- Декларация на функция, която приема единични променливи от декларирания тип.

- Декларация на функция, която приема масив от декларирания тип и с неизвестен брой елементи.

Разгледайте внимателно следния пример.

Пример 8-6:

```
#include <stdio.h>

void print_single(int* p);
void print_array(int* p);

int main(void)
{
	int a[10] = {1,2,3,4,5,6,7,8,9,10};
	int x = 99;

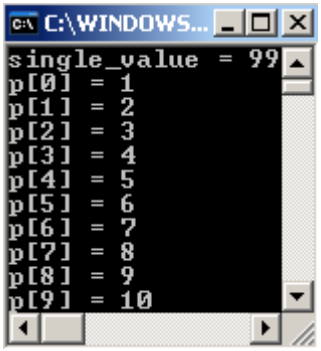
	print_single(&x);
	print_array(a);

	return 0;
}

void print_single(int* p)
{
	printf("single_value = %d\n", *p);
}

void print_array(int* p)
{
	int i;

	for( i = 0; i < 10 ; i++ )
	{
		printf("p[%d] = %d\n", i, p[i]);
	}
}
```



Обърнете внимание, че параметрите на двете функции изглеждат абсолютно по един и същ начин, но са предназначени да обработват различна информация: print_single() отпечатва стойността само на единична променлива от тип int, докато print_array() отпечатва съдържанието на 10 елементен масив от тип int.

	**За по-добро самодокументиране на кода е препоръчително първите две декларации да се използват ако една функция приема масив за аргумент, а третата да се използва ако се приема адрес на единична променлива.**

Подаването на масив на функция има следната важна особеност. Тъй като параметърът на функцията съдържа адреса на първия елемент на масива, то всяка промяна на елементите на масива във функцията се отразяват и на масива-аргумент.

Пример 8-7:

```
#include <stdio.h>

void init_array(int p[10]);

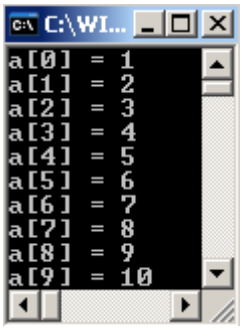
int main(void)
{
	int a[10];
	int i;

	init_array(a); /* Инициализиране на масива a */
	for( i = 0; i < 10 ; i++ ) /* Отпечатване на масива a */
	{
		printf("a[%d] = %d\n", i, a[i]);
	}

	return 0;
}

void init_array(int p[10])
{
	int i;

	for( i = 0; i < 10 ; i++ )
	{
		p[i] = i + 1;
	}
}
```



От механизма на предаване на масив на функция става ясно, че приемащия параметър на функцията е указател, на който се предава адреса на първия елемент на масива. По този начин функцията няма информация за големината на масива. При необходимост може да се декларира втори формален параметър, на който да се предава броя на елементите на масива.

Пример 8-8:

```
#include <stdio.h>

void init_array(int p[10], int count);
```

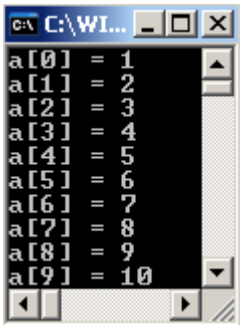
```
int main(void)
{
	int a[10];
	int i;

	init_array(a,10); /* Инициализиране на масива a */
	for( i = 0; i < 10 ; i++ ) /* Отпечатване на масива a */
	{
		printf("a[%d] = %d\n", i, a[i]);
	}

	return 0;
}

void init_array(int p[10], int count)
{
	int i;

	for( i = 0; i < count ; i++ )
	{
		p[i] = i + 1;
	}
}
```



Тъй като декларацията на функция, която приема за параметър масив може да изглежда така **тип име-функция(тип име-масив[размер]);** или така **тип име-функция(тип име-масив[]);**, се създава погрешно впечатление, че ако във функцията се използва конструкцията **sizeof(име-масив)**, може да се определи размера на масива вътре в самата функция. Това е погрешно, защото горните два прототипа са еквивалентни на прототипа **тип име-функция(тип *име-указател);**, от който ясно се вижда че приемащия параметър е указател и конструкцията **sizeof(име-масив)** в действителност ще върне големината на указателя-параметър (броят байтове заделени от компилатора за съхранение на самия указател), а не на масива-аргумент. Например следната програма няма да работи правилно.

Пример 8-9:

```
#include <stdio.h>

void init_array(int p[10]);

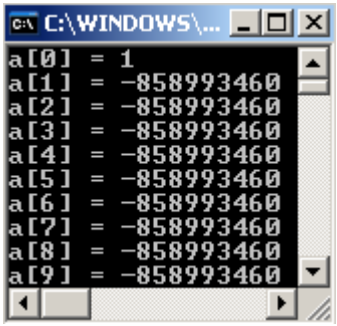
int main(void)
{
	int a[10];
	int i;
```

```
	init_array(a); /* Инициализиране на масива a */
	for( i = 0; i < 10 ; i++ ) /* Отпечатване на масива a */
	{
		printf("a[%d] = %d\n", i, a[i]);
	}

	return 0;
}

void init_array (int p[10])
{
	int i;

	for( i = 0; i < sizeof(p)/sizeof(int) ; i++ )
	{
		p[i] = i + 1;
	}
}
```



Формалният параметър **p** в действителност е указател към int, така че конструкцията sizeof(p) ще върне големината на указателя (примерно 4 байта), а в нашия пример големината на масива е 10 * sizeof (int), например 40 байта ако int заема 4 байта в паметта. Конструкцията sizeof(p) / sizeof (int) ще върне стойност различна от очакваната, например 1 вместо 10 при посочените допускания.

Във всички примери досега размерът на масива се задаваше чрез литерална константа. Това има един голям недостатък. Ако се наложи размера на масива да се промени, това означава, че трябва да се променят всички места в кода където той се използва. Поради тази причина за размер на масива често се използва константа зададена с предпроцесорната директива **#define**.

Пример 8-10:

```
#include <stdio.h>

#define SIZE (10) /* Размерът на масива може да се изменя оттук */

int main(void)
{
	int a[SIZE] = {1,2,3,4,5,6,7,8,9,10};
	int i;

	for( i = 0; i < SIZE ; i++ )
	{
```

```
            printf("a[%d] = %d\n", i, a[i]);
      }

      return 0;
}
```

Ако се наложи да се промени размера на масива, програмата е необходимо да се промени само на едно място, в дефинирането на макро-идентификатора SIZE.

1.1.6 Едномерен масив от указатели

Дефиниране на масив от указатели

```
тип* име-масив[размер];
```

Звездичката пред името на масива показва, че елементите на масива са указатели, а **тип** посочва типа на тези указатели. Единствените стойности, които тези елементи могат да приемат са адреси на променливи от указания тип .

Пример 8-11:

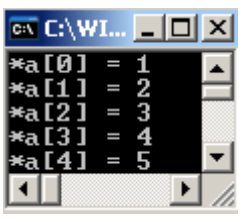
```
#include <stdio.h>

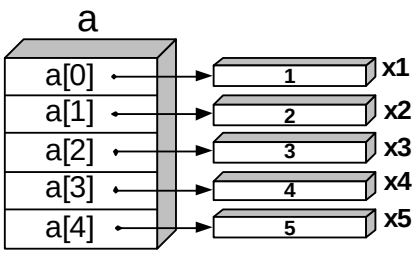
int main(void)
{
      int x1 = 1, x2 = 2, x3 = 3, x4 = 4, x5 = 5;
      int* a[5];
      int  i;

      a[0] = &x1;
      a[1] = &x2;
      a[2] = &x3;
      a[3] = &x4;
      a[4] = &x5;

      for( i = 0; i < 5; i++)
      {
            printf("*a[%d] = %d\n", i, *a[i]);
      }

      return 0;
}
```





Името на масив от указатели се явява указател-към-указател. В горния

приемр **a** се явява указател към указател-към-int, затова ако вместо индексиране се използва указателна аритметика е необходимо конструкцията *a[i] да се замени с **(a + i) както е показано в следващия пример.

Пример 8-12:

```
#include <stdio.h>

int main(void)
{
	int x1 = 1, x2 = 2, x3 = 3, x4 = 4, x5 = 5;
	int* a[5];
	int  i;

	a[0] = &x1;
	a[1] = &x2;
	a[2] = &x3;
	a[3] = &x4;
	a[4] = &x5;

	for( i = 0; i < 5; i++)
	{
		printf("*a[%d] = %d\n", i, **(a + i));
	}

	return 0;
}
```

Нека обясним израза ****(a + i)** по-подробно:

- Конструкцията **(a + i)** избира i-я елемент-указател на масива **a**, т.е елемента **a[i].**

- Конструкцията ***(a + i)** извлича стойността на елемента-указател **a[i]**. Тази стойност в действителност е адреса на променливата $\mathbf{x_{i+1}}$, i = [0..4]. За яснота израза ***(a + i)** може да се замени с израза $\mathbf{\&x_{i+1}}$ (адрес на променливата $\mathbf{x_{i+1}}$).

- Конструкцията $\mathbf{**(a + i) == *(\&x_{i+1})}$ извлича стойността на променливата, която се намира на адрес $\mathbf{\&x_{i+1}}$, т.е. стойността на променливата $\mathbf{x_{i+1}}$.

Масив от char-указатели намира приложение за създаване на низови таблици. Низовата таблица представлява масив от char-указатели към низове.

Пример 8-13:

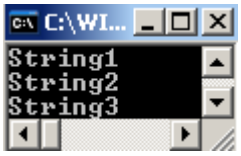
```
#include <stdio.h>

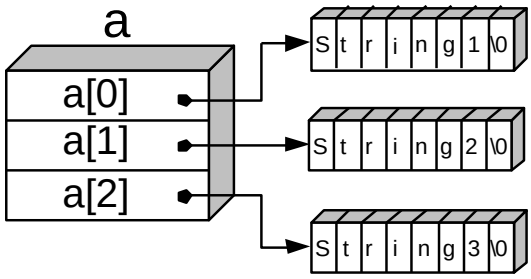
int main(void)
{
```

```
        char *a[3] = { "String1",
                       "String2",
                          "String3" };

        printf("%s\n", a[0]);
        printf("%s\n", a[1]);
        printf("%s\n", a[2]);

        return 0;
}
```





В този пример се дефинира масив **a**, чийто елементи са указатели към char. Тези указатели се инициализират със адресите на първите символи на съответните низове, т.е елементът-указател **a[0]** съдържа адреса на първия символ от низа "**S**tring1", елементът-указател **a[1]** съдържа адреса на първия символ от низа "**S**tring2" и елементът-указател **a[2]** съдържа адреса на първия символ от низа "**S**tring3". Следователно конструкцията **a[i] + j**, където **a[i]** е указател към първия символ на **i**-я низ, а **j** е номера на символ от този низ, е указател към **j**-я символ на **i**-я низ от низовата таблица. Косвеният достъп до този символ става чрез конструкцията ***(a[i] + j)**. От правилото, че конструкция от вида **x[i]** е еквивалентна на конструкцията ***(x + i),** следва че конструкцията ***(a[i] + j)** може да се представи изцяло чрез индексиране или изцяло чрез указателна аритметика, т.е

***(a[i] + j) == *(*(a + i) +j)**

или

***(a[i] + j) == a[i][j]**

Следващият пример дава същия резултат като предходния, но отпечатва низовете символ по символ, използвайки горните разсъждения.

Пример 8-14:

```
#include <stdio.h>
#include <string.h> /* Декларира библиотечната функция strlen() */

int main(void)
{
    char *a[3] = { "String1",
                   "String2",
                      "String3" };

    int arr_index, str_index, str_length;
```

```
    for( arr_index = 0; arr_index < 3;  arr_index++ )
    {
       str_length = strlen(a[arr_index]); /* strlen() връща броя на символите на
                                             низа, който й е подаден като аргумент */
       for (str_index = 0; str_index < str_length ; str_index++)
       {
              printf("%c", *(a[arr_index] + str_index) );
       }

       printf("\n");
    }

       return 0;
}
```

Първият цикъл for обхожда елементите на масива **a**, а вторият отпечатва низа, сочен от елемента-указател, символ по символ. След отпечатване на низа, на дисплея се отпечатва символ за нов ред. Процесът продължава докато се отпечатат всичките низове. За да се определи броят на символите в низовете е използвана библиотечната функция strlen(), декларирана в хедър-файла <string.h>.

Удобството на създаване на низови таблици чрез масив от указатели са две:

- Всеки елемент-указател на масива може да сочи към низ с различна дължина.

- При необходимост низовата таблица да се сортира, не е необходимо да се разменят местата на низовете, а само да се разменят стойностите на елементите-указателите.

Пример 8-15:

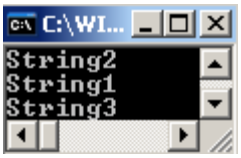
```
int main(void)
{
       char *a[3] = { "String1",
                      "String2",
                        "String3" };

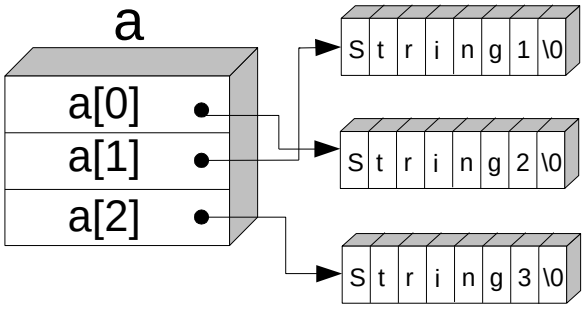
       char* ptr_temp;

       /* Указател a[0] и a[1] си разменят низовете към, които сочат */
       ptr_temp = a[0];
       a[0] = a[1];
       a[1] = ptr_temp;

       printf("%s\n", a[0]);
       printf("%s\n", a[1]);
       printf("%s\n", a[2]);

       return 0;
}
```





1.1.7 Предаване на масив-от-указатели на функция

Следните три декларации са еквивалентни:

- **тип име-функция(тип* име-масив[размер]);**
- **тип име-функция(тип* име-масив[]);**
- **тип име-функция(тип** име-указател-към-указател);**

Когато на функция се предава масив от указатели, в действителност се предава указател-към-указател. Ето защо и приемащия параметър на функцията трябва да е е указател към указател.

Пример 8-16:

```
#include <stdio.h>
#include <string.h>

void print_strings(char *a_ptr[]);

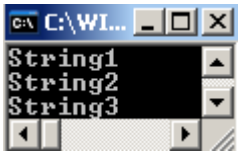
int main(void)
{
	char *a[3] = { "String1",
                       "String2",
                          "String3" };

	print_strings(a);

	return 0;
}

void print_strings(char *a_ptr[])
{
	int i;

	for( i = 0; i < 3; i++)
	{
		printf("%s\n", a_ptr[i] );
	}
}
```



1.1.8 Масив от указатели-към-функции

Дефиниране на масив от указатели към функции

```
тип (*име-масив[размер])(списък-параметри);
```

където

тип
Указва типа на връщания резултат от функцията.

списък-параметри
Съдържа формалните параметри на функцията.

Пример 8-17:

```
#include <stdio.h>
#include <string.h>

void f1(void);
void f2(void);

int main(void)
{
	void (*a[2])(void) = { f1, f2 };

	(*a[0])();
	(*a[1])();

	return 0;
}

void f1(void)
{
	printf ("Call function f1()\n");
}

void f2(void)
{
	printf ("Call function f2()\n");
}
```



1.2 Двумерни масиви

1.2.1 Определение и дефиниране

Двумерният масив по същество е едномерен масив, чийто елементи също са едномерни масиви.

Дефиниране на двумерен масив

```
тип име-масив[размер1][размер2];

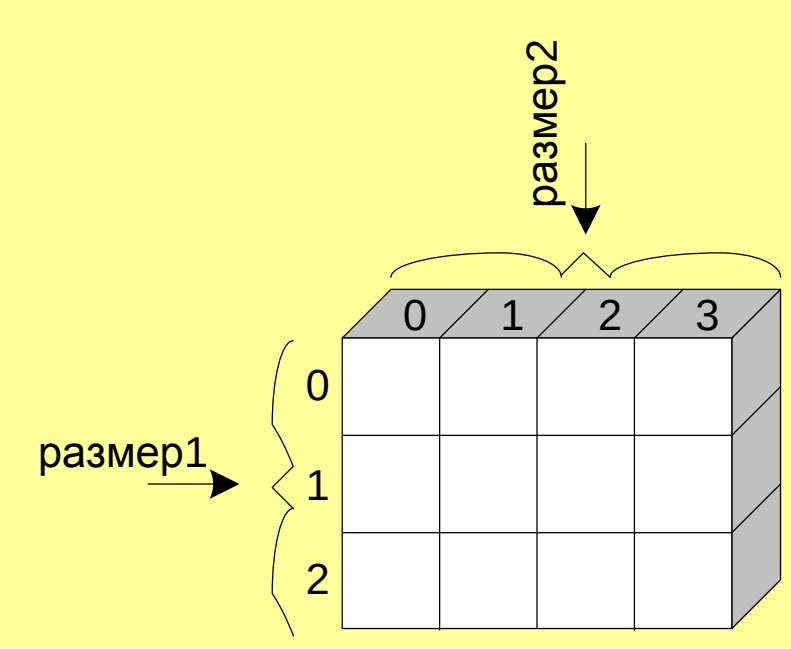
размер1
      Задава броя на елементите на двумерния масив. Всеки елемент представлява
      едномерен масив. Трябва да бъде литерален константен целочислен израз.


размер2
      Задава броя на елементите на едномерните масиви (елементарни елементи).
      Трябва да бъде литерален константен целочислен израз.

тип
      Дефинира типа на елементите на едномерните масиви (типа на елементарните
      елементи на двумерния масив). Елементите на масив могат да бъдат от всеки
      тип с изключение на битови полета и функции.
```

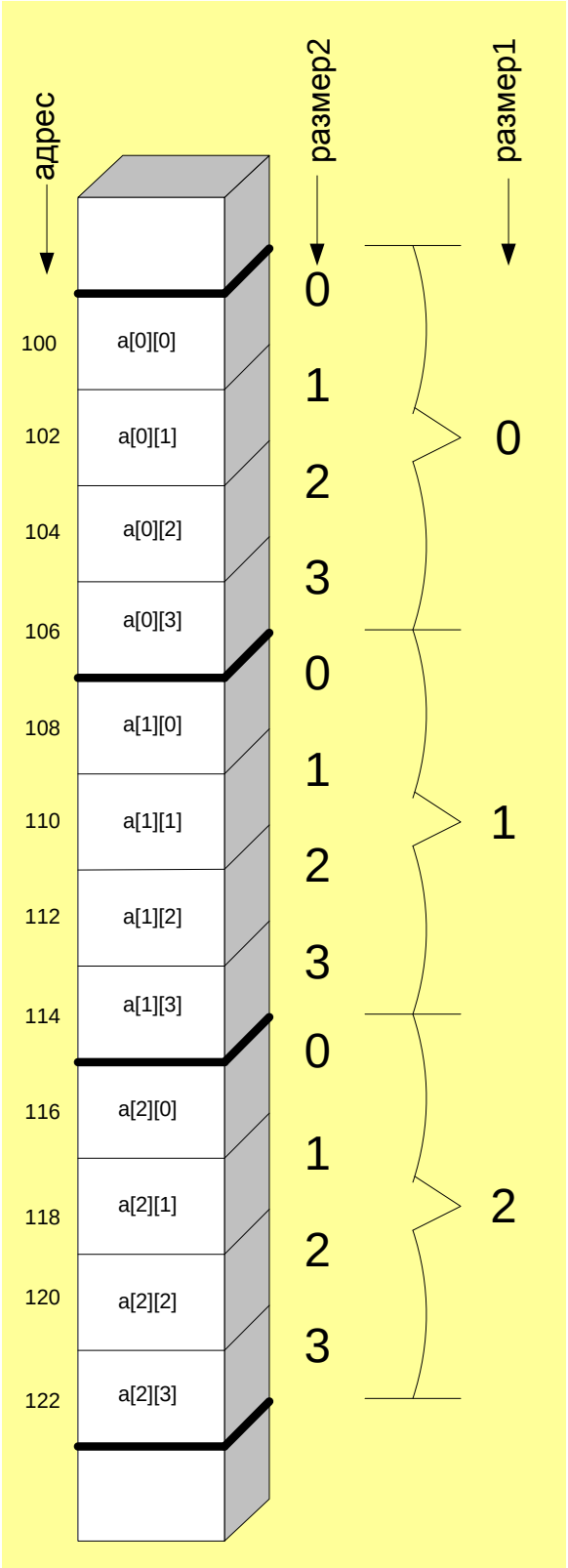
Пример:
```
int a[3][4]; /* Дефиниране на двумерен масив a, от 3 елемента тип
                int[4] (едномерен масив с 4 елемента тип int) */
```

Графично двумерният масив **a** може да се представи като таблица (Фиг. 54), състояща се от редове и колони. Броят на редовете представлява броя на елементите на двумерния масив, които от своя страна са едномерни масиви. Броят на колоните представлява броят на елементите на тези едномерни масиви. Ще ги наричаме елементарни-елементи.



Фиг. 54 Двумерен масив a 3x4

Практически елементарните елементите се съхраняват последователно в паметта като първият елемент на първия едномерен масив (a[0][0]) се съхранява на най-младшия адрес, вторият елемент на първия едномерен масив (a[0][1]) се съхранява веднага след първия елемент т.н. както е показано на Фиг.55 (Адресите са избрани произволно и имат само илюстративен характер. Приема се, че тип int е 2 байта).



Фиг. 55 Разполагане на двумерен масив в паметта

1.2.2 Инициализиране

Инициализиране на двумерен масив

тип име-масив [размер1][размер2] = { списък-инициализатори[1] };

или

тип име-масив [[2]][размер2] = { списък-инициализатори[1] };

където

списък-инициализатори
Литерални константни изрази разделени със запетая.

Забележка[1]: Броят на инициализаторите трябва да е по-малък или равен на броя на елементарните-елементите на масива, получен от произведението на **(размер1*размер2)**. Ако броя на инициализаторите е по-малък от броя на елементарните-елементи, елементите, за които не достигат инициализатори се инициализират с 0 (за целочислени типове), 0.0 (за типове с плаваща запетая) или NULL (указателни типове).

Забележка[2]: Когато масив се инициализира по време на дефинирането си, **размер1** може да се пропусне. Броят на елементите се определя от броя на инициализаторите.

Когато двумерен масив се инициализира по време на дефинирането си, първият размер може да се пропусне.

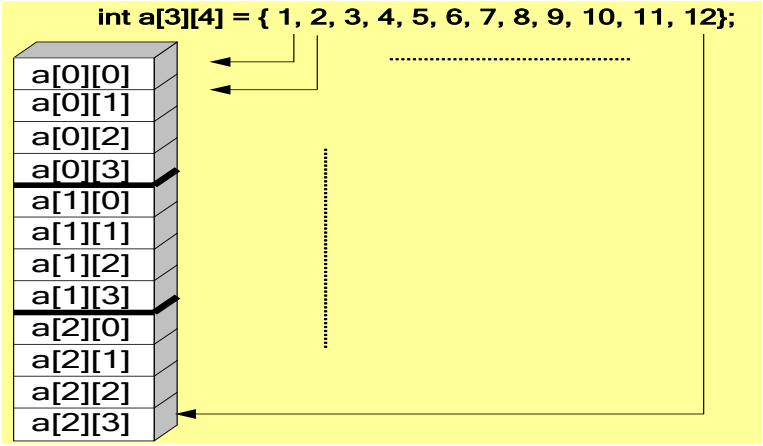
Пример:

```
int a[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

или

```
int a[][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

На Фиг.56 е показано схематично как инициализиращите стойности се присвояват на елементарните-елементи на двумерния масив. Инициализиращите стойности се присвояват последователно на елементите на масива. Ако броят на инициализиращите стойности е по-малък от броя на елементарните-елементи на масива, елементарните-елементи, за които не достигат инициализиращи стойности се нулират.



Фиг. 56 Инициализиране на двумерен масив

За подобряване на нагледността инициализиращите стойности могат да се подредят по следния начин:

```
int a[3][4] =
{
  1, 2, 3, 4,    /* Тези стойности инициализират едномерния насив a[0] */
  5, 6, 7, 8,    /* Тези стойности инициализират едномерния насив a[1] */
  9, 10, 11, 12  /* Тези стойности инициализират едномерния насив a[2] */
};
```

Допълнително около всеки ред с инициализиращи стойности могат да се поставят фигурни скоби.

```
int a[3][4] =
{
  {1, 2, 3, 4},    /* Тези стойности инициализират едномерния насив a[0] */
  {5, 6, 7, 8},    /* Тези стойности инициализират едномерния насив a[1] */
  {9, 10, 11, 12}  /* Тези стойности инициализират едномерния насив a[2] */
};
```

Последната форма на инициализиране на двумерен масив позволява частично инициализиране на елементите-едномерни-масиви с явни инициализатори.

Пример:

```
int a[3][4] =
{
  {1, 2, 3, 4},    /* Инициализиране едномерния насив a[0]             */
  {5, 6},          /* Частично инициализиране на едномерния насив a[1] */
  {0}              /* Нулиране на едномерния масив                a[2] */
};
```

Тази конструкция ще инициализира масива **a** както е показано долу.

1	**2**	**3**	**4**
5	**6**	**0**	**0**
0	**0**	**0**	**0**

1.2.3 Индексиране

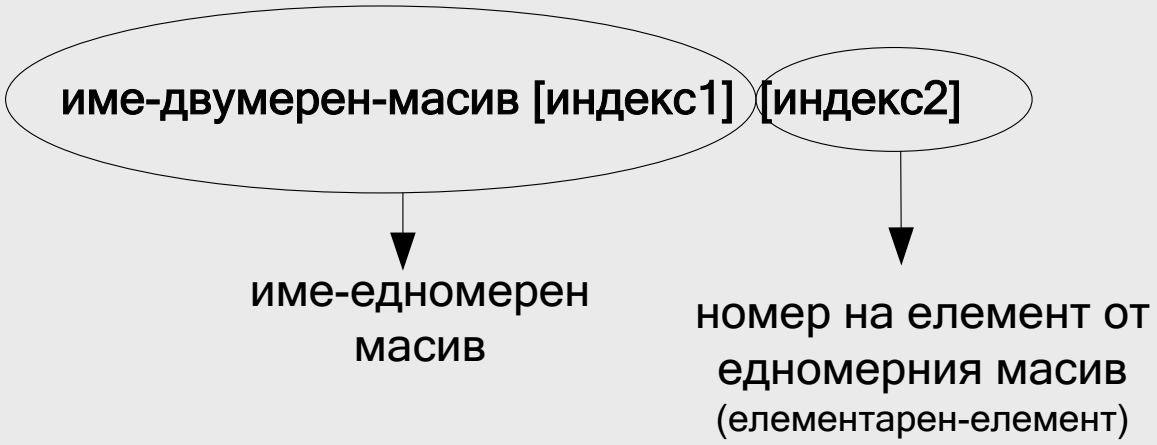
Елементите на двумерния масив както и при едномерен масив се достигат чрез индексиране, т.е. **име-масив[индекс1][индекс2]**. С **индекс1** се избира елемент-едномерен-масив, а ***с*** **индекс2** се избира елемент от този едномерен масив.

За нашия пример елементите на двумерния масив **a** са **a[0]**, **a[1]** и **a[2]**. Всеки от тях представлява едномерен масив от 4 елемента от тип int. Конструкциите **a[0]**, **a[1]** и **a[2]** могат да се разглеждат като имена на тези едномерни масиви. От правилото, че елементите на едномерен масив се достигат чрез името на масива и номера на елемента в квадратни скоби, следва, че за да се достигнат елементите на тези едномерни масиви трябва да се използват следните конструкции:

За масива **a[0]** → **a[0][0], a[0][1], a[0][2], a[0][3]**
За масива **a[1]** → **a[1][0], a[1][1], a[1][2], a[0][3]**
За масива **a[2]** → **a[2][0], a[2][1], a[2][2], a[2][3]**

Обобщено всичко казано до тук може да се напише така:

Индексиране на двумерен масив



индекс1[1]
Може да бъде всеки целочислен израз, чиято стойност изпълнява условието

0 ≤ индекс1 < размер1 , т.е индекс1 Є [0...(размер1-1)]

индекс2[1]
Може да бъде всеки целочислен израз, чиято стойност изпълнява условието

0 ≤ индекс2 < размер2 , т.е индекс2 Є [0...(размер2-1)]

Забележка[1]**:** Отговорност на програмистта е да следи индексите да не излизат извън рамките на масива, в противен случай резултатът е недефиниран.

Конструкцията **име-двумерен-масив[индекс1]** представлява името на

съответния елемент на двумерния масив, т.е. **име-двумерен-масив[индекс1]** е име на едномерен масив. Отделните елементи на този масив се достигат чрез неговото име и номера на елемента в квадратни скоби, т.е. **име-двумерен-масив[индекс1][индекс2]**.

Пример 8-18:

```
#include <stdio.h>

int main(void)
{
       int a[3][4]; /* Дефиниране на двумерен масив 3x4 */
       int row,column;

       /* Нулиране на елементарните-елементи на масива */
       for( row = 0; row < 3; row++ )/* обхожда редовете на масива */
       {
          for( column = 0; column < 4; column++)/* обхожда колоните на масива */
          {
              a[row][column] = 0; /* Нулиране на поредния елементарен-елемент */
          }
       }

       /* Отпечатване на елементарните-елементи на масива */
       for( row = 0; row < 3; row++ ) /* обхожда редовете на масива */
       {
          for( column = 0; column < 4; column++)/* обхожда колоните на масива */
          {
              printf("%d ", a[row][column] ); /* Отпечатване на поредния
                                                 елементарен-елемент */
          }

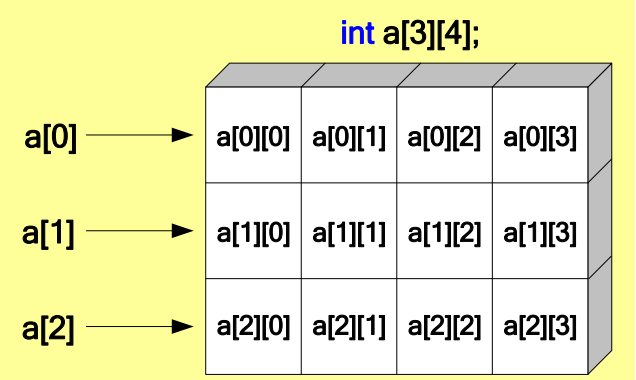
          printf("\n");
       }

       return 0;
}
```



1.2.4 Двумерни масиви и указатели

По дефиниция името на масив е указател към първия елемент. За двумерен масив това означава че името му е указател към първия елемент-едномерен-масив. На Фиг.57 е показано схематично масива **a** с размерност 3x4.

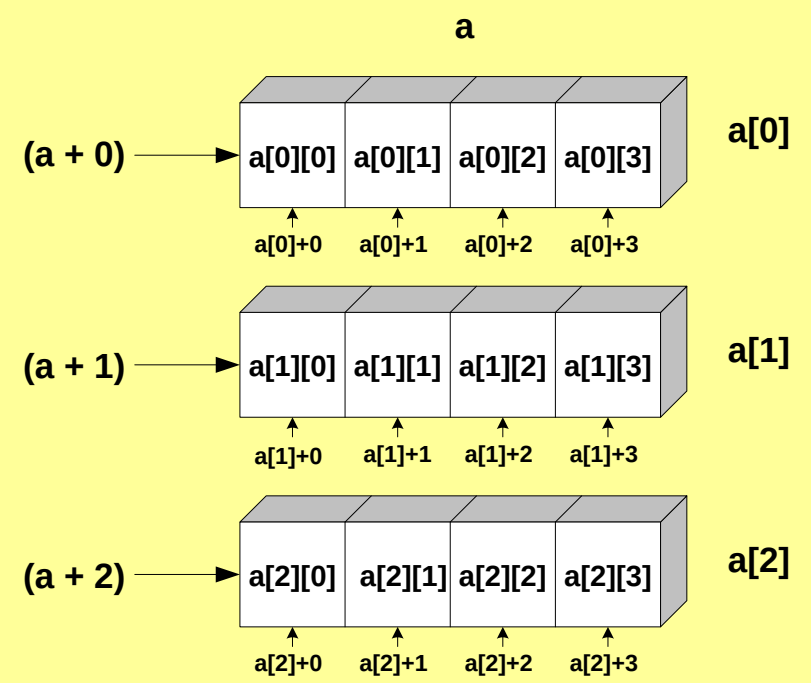


Фиг. 57 Двумерен масив a 3x4

Името **a** е указател към **a[0]**, който от своя страна също е указател към **a[0][0]**. С други думи **a** е указател към цял едномерен масив от тип **int.** Следните изрази са еквивалентни:

```
a + 0 == &a[0]          a[0] == &a[0][0]
a + 1 == &a[1]    и     a[1] == &a[1][0]
a + 2 == &a[2]          a[2] == &a[2][0]
```

Практически погледнато указателите **a** и **a[0]** сочат към един и същ адрес (**&a[0][0]**), но обектите, към които сочат са от различен тип. **a** сочи към цял масив от тип **int**, а **a[0]** сочи към елемент от тип **int**. При използване на адресна аритметика стойностите на тези указатели ще се променят различно: **a + 1** ще доведе до увеличение на указателя така, че той ще сочи към следващия елемент-едномерен-масив, т.е към **a[1]**, докато **a[0] + 1** ще увеличи указателя така, че той ще сочи към следващия елемент на едномерния масив, т.е към **a[0][1]**. Графично това може да се представи както е показано на Фиг. 58.



Фиг. 58

По долу са показани различни начини за достъп до елементарните-елементи на двумерен масив чрез указатели.

- Достъп до елементарните-елементи на двумерен масив чрез указател от типа на елементарните-елементи на масива. Указателят може да използва както адресна аритметика така и индексиране.

Пример 8-19:

```
#include <stdio.h>
int main(void)
{
	int a[3][4];
	int i;
	int *p;

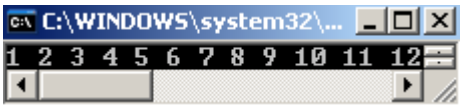
	p = &a[0][0];

	/* Инициализиране на масива */
	for( i = 0; i < 3*4; i++ )
	{
		*(p + i) = i +1; /* p[i] = i + 1; */
	}

	/* Отпечатване на масива */
	for( i = 0; i < 3*4; i++ )
	{
		printf("%d ", *(p + i) ); /* printf("%d ", p[i]); */
	}

	printf("\n");

	return 0;
}
```



Адресът на първия елементарен-елемент на масива **a** може да се присвои на указателя **p** по няколко начина:

- `p = &a[0][0];`

  Тъй като **a[0][0]** по същество е променлива от тип **int** , то нейния адрес може директно да се присвои на указателя **p**, който е също от тип **int**.

- `p = a[0];`

  Тъй като **a[0]** е името на първия елемент-едномерен-масив на двумерния масив **a**, то **a[0]** всъщност представлява и адреса на първия му елемент, т.е **a[0] == &a[0][0]**.

- `p = (int*)a;`

  Тъй като **a** е указател към едномерен масив от 4 int елемента (т.е. int (*)[4]), а **p** е указател към int, то е необходимо указателя **a** да се преобразува в указател-към-int по време на присвояването.

- Достъп до елементите на двумерен масив чрез името на масива като указател

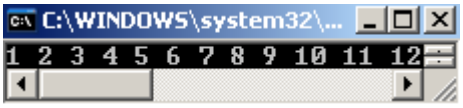
Пример 8-20:

```
#include <stdio.h>
int main(void)
{
	int a[3][4];
	int i;

	/* Инициализиране на масива */
	for( i = 0; i < 3*4; i++ )
	{
		*( (int *)a + i) = i + 1;
	}

	/* Отпечатване на масива */
	for( i = 0; i < 3*4; i++ )
	{
		printf("%d ", *( (int *)a + i) );
	}

	return 0;
}
```



Този пример е подобен на предходния с тази разлика, че вместо да се използва отделен указател, се използва името на масива като указател. Тъй като **a** е указател към едномерен масив, за да се изпълнява правилно адресната аритметика, т.е да се достигат елементарните-елементи на масива, се използва операция за привеждане на типа (int *)a.

- Достъп до елементите на двумерен масив чрез указател към едномерен масив.

Пример 8-21:

```
#include <stdio.h>

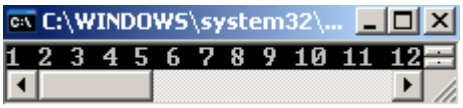
int main(void)
{
	int a[3][4];
	int (*p)[4];
	int i;
```

```
	p = a;

	for( i = 0; i < 3*4; i++ )
	{
		*( (int *)p + i) = i + 1;
	}

	for( i = 0; i < 3*4; i++ )
	{
		printf("%d ", *( (int *)p + i) );
	}

	return 0;
}
```



Този пример е подобен на предходния с тази разлика, че вместо да се използва името на масива се дефинира отделен указател към едномерен масив. Тук също е необходимо този указател да се приведе към указател-към-int когато се достигат елементарните-елементи на масива **a**.

- Достъп до елементите на двумерен масив чрез индексиране и указателна аритметика (смесен достъп)

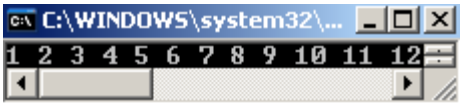
Пример 8-22:

```
#include <stdio.h>

int main(void)
{
	int a[3][4] =
	{
		1 ,  2 ,  3 , 4 ,
		5 ,  6 ,  7 , 8 ,
		9 , 10 , 11 , 12
	};
	int row, column;

	for(row = 0; row < 3; row ++ )
	{
		for(column = 0; column < 4; column ++)
		{
			printf("%d ", *( a[row] + column) );
		}
	}

	return 0;
}
```



Не е трудно да се види, че конструкцията ***(a[row] + column)** е идентична

на **a[row][column]**. Това е следствие от правилото, че всяка конструкция от типа **x[i]** се преобразува от компилатора в ***(x + i)**. За нашия пример x == a[row], а i == column. Като заместим x и i в уравнението x[i] == *(x + i) с еквивалентинте им стойности от горния пример, се получава a[row][column] == *(a[row] + column).

1.2.5 Предаване на двумерен масив на функция

Следните три декларации са еквивалентни:

- `тип име-функция(тип име-масив[размер1][размер2]);`
- `тип име-функция(тип име-масив[][размер2]);`
- `тип име-функция(тип (*име-указател)[размер2]);`

Когато двумерен масив се предава на функция в действителност се предава указател към едномерен масив. Ето защо и приемащия параметър на функцията също представлява указател към едномерен масив. **размер1** може да се пропусне, но **размер2** е задължителен, тъй като именно той указва дължината на едномерния масив, към който сочи указателя.

Пример 8-23:

```
#include <stdio.h>

void init_a(int array[][4]);

int main(void)
{
	int a[3][4];
	int row, column;

	init_a(a);

	for( row = 0; row < 3; row++ )
	{
		for( column = 0; column < 4; column++)
		{
			printf("%d ", a[row][column] );
		}

		printf("\n");
	}

	return 0;
}

void init_a(int array[][4])
{
	int row,column;
```

```
	for( row = 0; row < 3; row++ )
	{
		for( column = 0; column < 4; column++)
		{
			array[row][column] = 5;
		}
	}
}
```



Както и при едномерните масиви, функцията няма информация за броя на елементите-едномерни масиви, затова при необходимост може да се декларира още един параметър, който да указва този брой.

Пример 8-24:

```
#include <stdio.h>

void init(int array[][4], int count);

int main(void)
{
	int a[3][4];
	int row,column;

	init(a,3);

	for( row = 0; row < 3; row++ )
	{
		for( column = 0; column < 4; column++)
		{
			printf("%d ", a[row][column] );
		}

		printf("\n");
	}

	return 0;
}

void init(int array[][4], int count)
{
	int row,column;

	for( row = 0; row < count; row++ )
	{
		for( column = 0; column < 4; column++)
		{
			array[row][column] = 5;
		}
	}
}
```



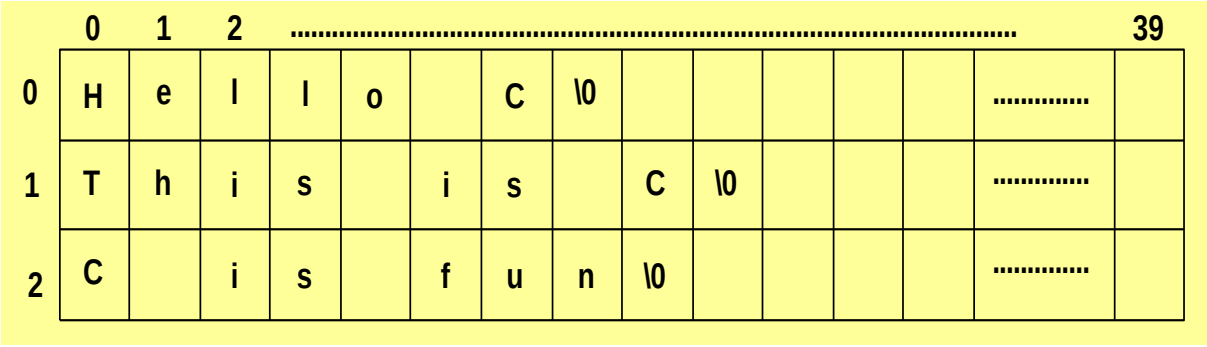
1.2.6 Създаване на низова таблица с помощта на двумерен масив

В следващия пример се дефинира двумерен масив от тип **char** и се инициализира с три низа.

Пример:

```
char  strings[3][40] =
{
      "Hello C"   ,      /* Инициализира масива strings[0] */
      "This is C" ,      /* Инициализира масива strings[1] */
      "C is fun"         /* Инициализира масива strings[2] */
};
```

В графичен вид тази дефиниция изглежда така:



Фиг. 59 Низова таблица с двумерен масив

Вижда се, че не цялата памет заделена за масива се използва ефективно ако низовете са с дължина по-малка от размера на едномерния масив. Освен това ако низовете трябва да се сортират те трябва да разменят местата си. Направете сравнение с низова таблица съставена с помощта на едномерен масив от char-указатели.

1.3 Тримерни масиви

1.3.1 Определение и дефиниране

Тримерният масив по същество е едномерен масив, чийто елементи са двумерни масиви.

Дефиниране на тримерен масив

```
тип име-масив[размер1][размер2][размер3];
```

където

размер1
Задава броя на елементите на тримерния масив. Всеки елемент на тримерния масив представлява двумерен масив. Трябва да бъде литерален константен целочислен израз.

размер2
Здава броя на елементите на двумерните масиви. Всеки елемент на двумерния масив представлява едномерен масив. Трябва да бъде литерален константен целочислен израз.

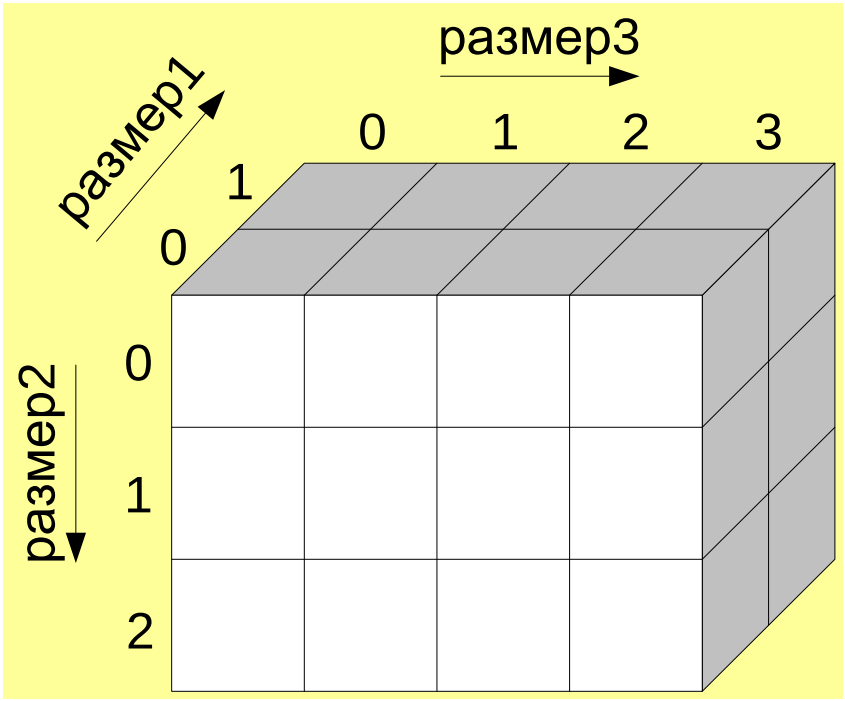
размер3
Задава броят на елементите на всеки едномерен масив. Елементите на едномерния масив представляват елементарните-елементи на тримерния масив. Трябва да бъде литерален константен целочислен израз.

тип
Дефинира типа на елементите на едномерните масиви (типа на елементарните-елементи на тримерния масив). Може да бъде от всеки тип без битово поле и функции.

Пример:

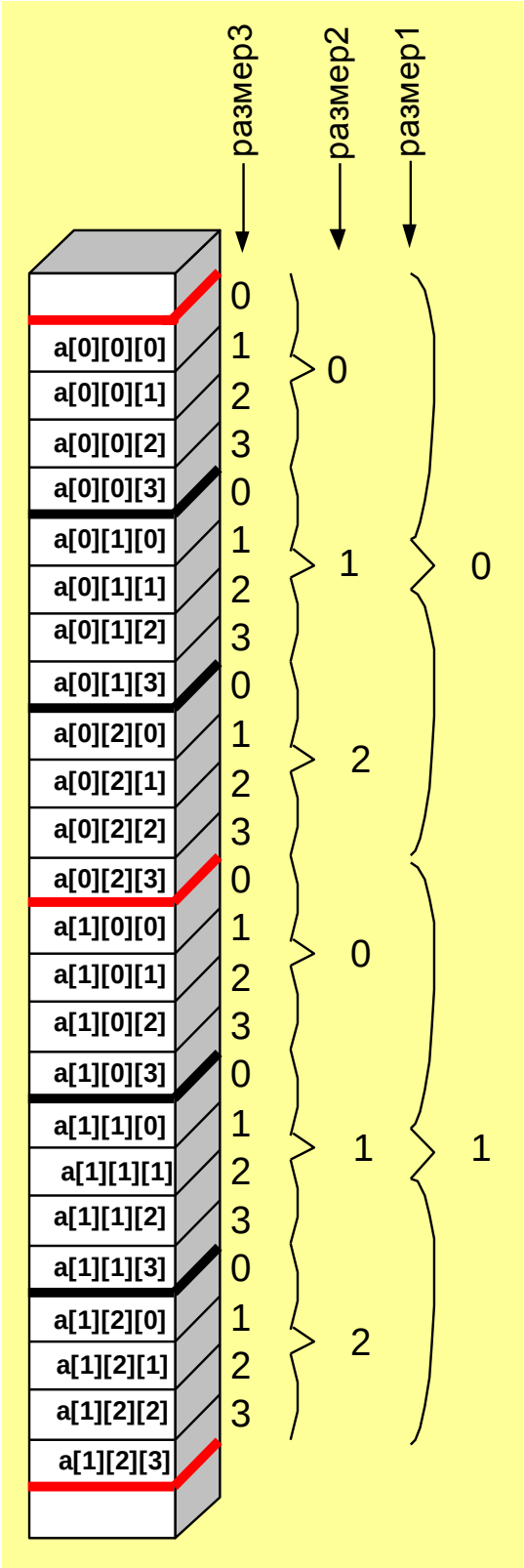
```
int     a[2][3][4]; /* Дефиниране на тримерен масив a, който се състои от 2
                       елемента двумерни масиви. Всеки елемент-двумерен-масив се
                       състои от 3 елемента едномерни масиви. Всеки елемент-
                       едномерен-масив се състои от 4 елемента от тип int      */
```

Графично тримерният масив **a** може да се представи като е показано на Фиг.60.



Фиг. 60 Тримерен масив 2x3x4

Практически елементарните-елементите на тримерен масив се съхраняват последователно в паметта като първия елемент на първия едномерен масив от първия двумерен масив (**a[0][0][0]**) се съхранява на най-младшия адрес, вторият елемент на първия едномерен масив от първия двумерен масив (**a[0][0][1]**) се съхранява веднага след него и т.н. както е показано на Фиг.61.



Фиг. 61 Разполагане на тримерен масив в паметта

1.3.2 Инициализиране

Инициализиране на тримерен масив

тип име-масив [размер1][размер2][размер3] = { списък-инициализатори[1] };

или

тип име-масив [][2][размер2][размер3]] = { списък-инициализатори[1] };

където

списък-инициализатори
Литерални константни изрази разделени със запетая.

Забележка[1]: Броят на инициализаторите трябва да е по-малък или равен на броя на елементарните-елементите на масива, получен от произведението на **(размер1*размер2*размер3)**. Ако броят на инициализаторите е по-малък от броя на елементарните-елементите, елементарните-елементите, за които не достигат инициализатори се инициализират с 0 (за целочислени типове), 0.0 (за типове с плаваща запетая) или NULL (указателни типове).

Забележка[2]: Когато масив се инициализира по време на дефинирането си, **размер1** може да се пропусне. Броят на елементите се определя от броя на инициализаторите.

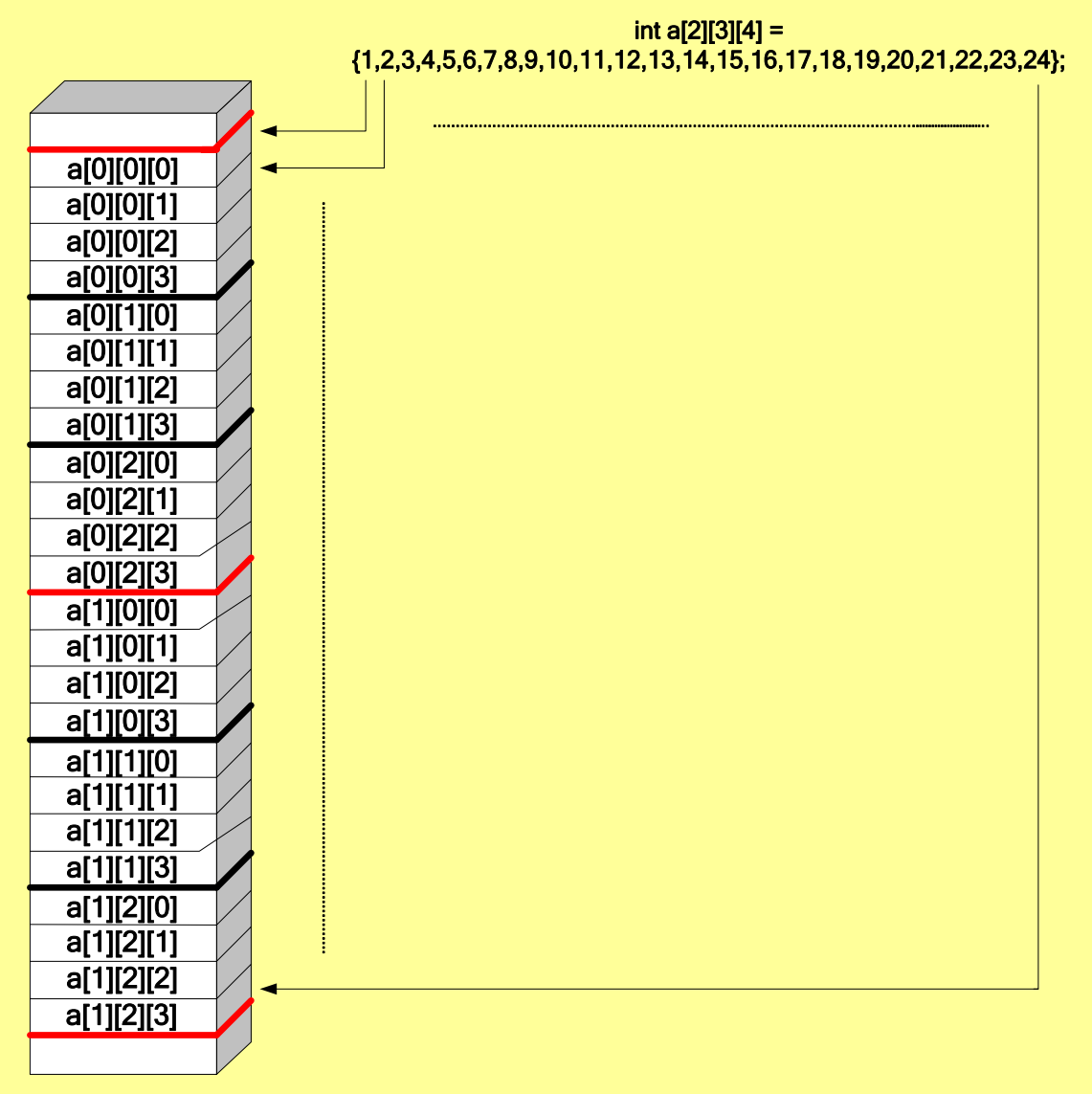
Пример:

```
int a[2][3][4] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24};
```

или

```
int a[][3][4] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24};
```

На Фиг.62 е показано схематично как инициализиращите стойности се присвояват на елементарните-елементи на тримерния масив. Инициализиращите стойности се присвояват последователно на елементите на масива. Ако броят на инициализиращите стойности е по малък от броя на елементарните-елементи на масива, елементарните-елементи, за които не достигат инициализиращи стойности се нулират.



Фиг. 62 Инициализиране на тримерен масив

За подобряване на нагледността инициализиращите стойности могат да се подредят по следния начин:

Пример:

```
int a[2][3][4] =
{
    /* Инициализиране на двумерния насив a[0] */
    {
      1,  2,  3,  4,  /* Тези стойности инициализират едномерния насив a[0][0] */
      5,  6,  7,  8,  /* Тези стойности инициализират едномерния насив a[0][1] */
      9, 10, 11, 12   /* Тези стойности инициализират едномерния насив a[0][2] */
    },

    /* Инициализиране на двумерния насив a[1] */
    {
      13,14,15,16, /* Тези стойности инициализират едномерния насив a[1][0] */
      17,18,19,20, /* Тези стойности инициализират едномерния насив a[1][1] */
      21,22,23,24  /* Тези стойности инициализират едномерния насив a[1][2] */
    }
};
```

1.3.3 Индексиране

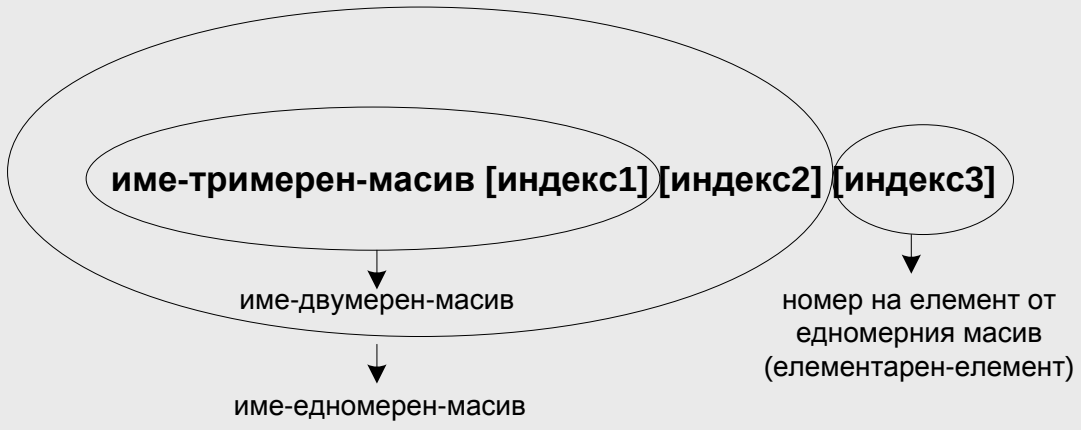
Елементите на тримерния масив се достигат чрез индексиране, т.е. чрез **име-масив[индекс1][индекс2][индекс3]**. С **индекс1** се избира елемент-двумерен-масив, с **индекс2** се избира елемент-едномерен-масив от него, а с **индекс3** се избира елемент от едномерния масив. За нашия пример елементите на тримерния масив **a** са **a[0]** и **a[1]**. Всеки един от тези елементи представлява двумерен масив с рамерност 3x4.

Конструкциите **a[0]** и **a[1]** могат да се разглеждат като имена на двумерните масиви. От правилото че елемент на масива се достига чрез неговото име и номера на елемента в квадратни скоби следва, че конструкциите **a[0][0]**, **a[0][1]**, **a[0][2]** са елементите на първия двумерен масив **a[0]**, а конструкциите **a[1][0]**, **a[1][1]**, **a[1][2]** са елементи на втория двумерен масив **a[1]**. Всеки един от тези елементи представлява едномерен масив от 4 елемента тип int. Конструкциите **a[0][0]**, **a[0][1]**, **a[0][2]** могат да се разглеждат като имена на едномерните масиви, които са елементи на двумерния масив **a[0]**, а конструкциите **a[1][0]**, **a[1][1]**, **a[1][2]** могат да се разглеждат като имена на едномерните масиви, които са елементи на двумерния масив **a[1]**. Като се приложи още един път правилото, че елемент на масив се достига чрез неговото име и номера на елемента в квадратни скоби следва, че елементите на тези едномерни масиви се достигат чрез следните конструкции:

За масив **a[0][0]** → **a[0][0][0]**, **a[0][0][1]**, **a[0][0][2]**, **a[0][0][3]**
За масив **a[0][1]** → **a[0][1][0]**, **a[0][1][1]**, **a[0][1][2]**, **a[0][1][3]**
За масив **a[0][2]** → **a[0][2][0]**, **a[0][2][1]**, **a[0][2][2]**, **a[0][2][3]**
За масив **a[1][0]** → **a[1][0][0]**, **a[1][0][1]**, **a[1][0][2]**, **a[1][0][3]**
За масив **a[1][1]** → **a[1][1][0]**, **a[1][1][1]**, **a[1][1][2]**, **a[1][1][3]**
За масив **a[1][2]** → **a[1][2][0]**, **a[1][2][1]**, **a[1][2][2]**, **a[1][2][3]**

Обобщено всичко казано до тук може да се опише така:

Индексиране на тримерен масив



където

индекс1[1]
Може да бъде всеки целочислен израз, чиято стойност изпълнява условието

0 ≤ индекс1 < размер1 , т.е индекс1 Є [0...(размер1-1)]

индекс2[1]
Може да бъде всеки целочислен израз, чиято стойност изпълнява условието

0 ≤ индекс2 < размер2 , т.е индекс2 Є [0...(размер2-1)]

индекс3[1]
Може да бъде всеки целочислен израз, чиято стойност изпълнява условието

0 ≤ индекс3 < размер3 , т.е индекс3 Є [0...(размер3-1)]

Забележка[1]**:** Отговорност на програмистта е да следи индексите да не излизат извън рамките на масива, в противен случай резултатът е недефиниран.

Конструкцията **име-тримерен-масив[индекс1]** представлява името на съответния елемент на тримерния масив, т.е. **име-тримерен-масив[индекс1]** е име на двумерен масив. Отделните елементи на този двумерен масив се достигат чрез неговото име и номера на елемента в квадратни скоби, т.е. **име-тримерен-масив[индекс1][индекс2]**. От своя страна конструкцията **име-тримерен-масив[индекс1][индекс2]** представлява име на едномерен масив. Елементите на този едномерен масив се достигат чрез неговото име и номера на елемента в квадратни скоби, т.е. **име-тримерен-масив[индекс1][индекс2][размер3]**.

Пример 8-25:

```
#include <stdio.h>

int main(void)
{
	int a[2][3][4];
	int i,j,k;

	/* Нулиране на елементите на масива a */
	for( i = 0; i < 2; i++ )
	{
		for( j = 0; j < 3; j++ )
		{
			for( k = 0; k < 4; k++ )
			{
				a[i][j][k] = 0;
			}
		}
	}

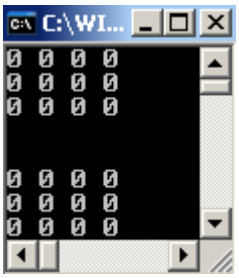
	/* Отпечатване на масива a */
	for( i = 0; i < 2; i++ )
	{
		for( j = 0; j < 3; j++ )
		{
			for( k = 0; k < 4; k++ )
			{
				printf("%d ", a[i][j][k]);
```

```
                }

                printf("\n");
            }

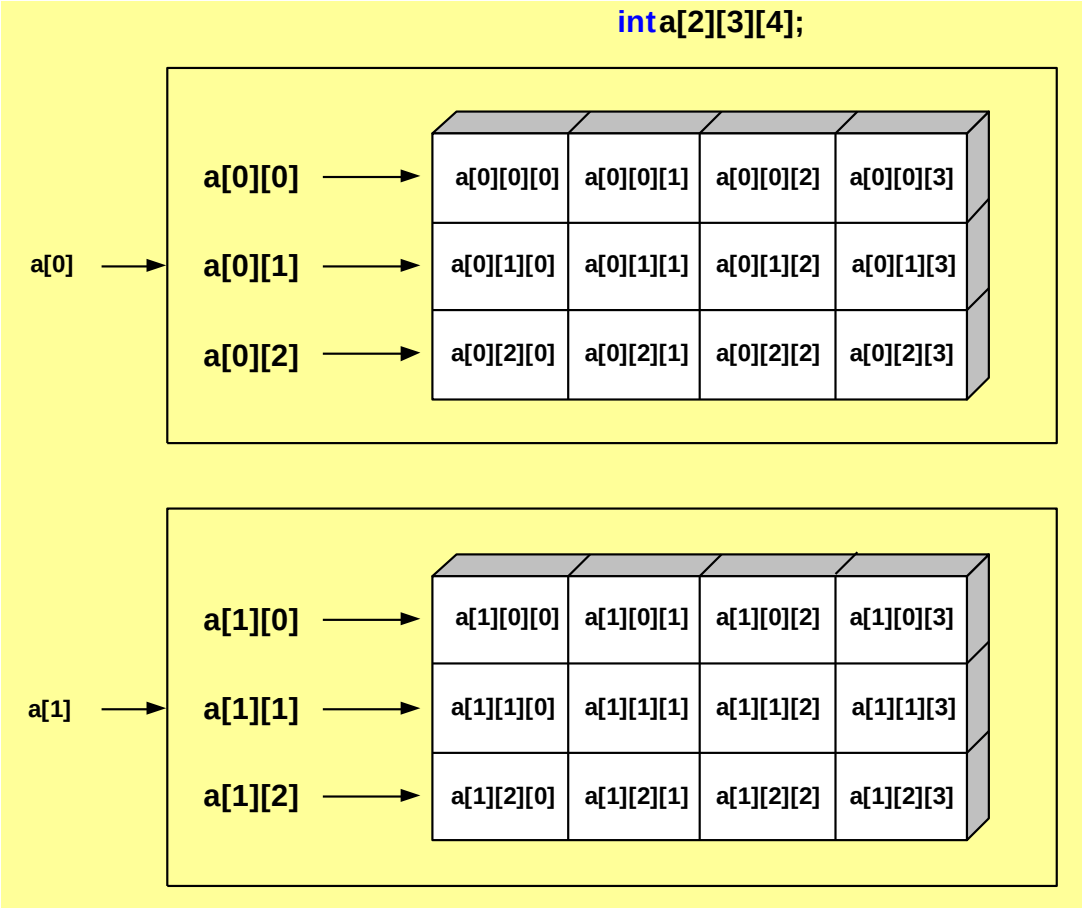
            printf("\n\n");
        }

        return 0;
}
```



1.3.4 Тримерни масиви и указатели

По дефиниция името на масив е указател към първия елемент. За тримерен масив това означава че името му е указател към първия елемент-двумерен-масив. На Фиг.63 е показано схематично масива **a** с размерност 2x3x4.



Фиг. 63 Тримерен масив a 2x3x4

Името **a** е указател към **a[0]**, **a[0]** е указател към **a[0][0]**, а **a[0][0]** е указател към **a[0][0][0]**, т.е.

a – указател към двумерен масив
a[0] – указател към едномерен масив
a[0][0] – указател към елементи от тип int

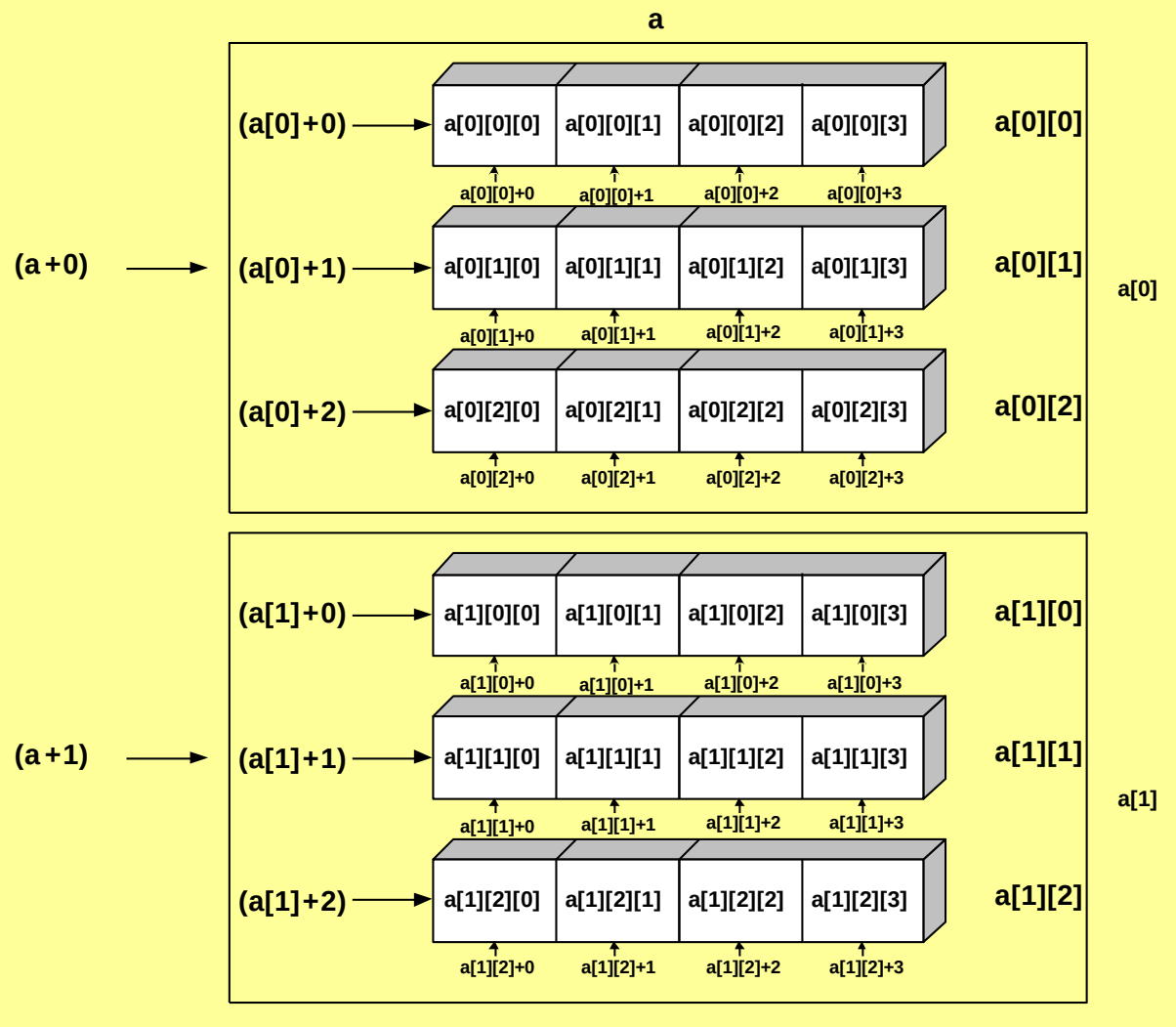
Следните изрази са еквивалентни:

```
a + 0 == &a[0]
a + 1 == &a[1]

a[0] == &a[0][0]
a[1] == &a[1][0]

a[0][0] == &a[0][0][0]
a[1][0] == &a[1][0][0]
```

Практически погледнато указателите **a, a[0], a[0][0]** сочат към един и същи адрес (**&a[0][0][0]**), но обектите, към които сочат са от различен тип, т.е. **a** сочи към двумерен масив, **a[0]** сочи към едномерен масив, а **a[0][0]** към елемент от тип int. При използване на адресна аритметика стойностите на тези указатели ще се променят различно: **a + 1** ще доведе до увеличение на указателя така, че той ще сочи към следващия елемент-двумерен-масив, т.е към **a[1]**, **a[0] + 1** ще увеличи указателя така, че той ще сочи към следващия едномерен масив, т.е към **a[0][1]**, а **a[0][0] + 1** ще сочи към следващия елемент на едномерния масив, т.е към **a[0][0][1]** Графично това може да се представи както е показано на Фиг.64.



Фиг. 64 Тримерен масив и указателна аритметика

По долу са показани различни начини за достъп до елементарните-елементи на тримерен масив чрез указатели.

- Достъп до елементите на тримерен масив чрез указател от типа на елементарните-елементи на масива.

Пример 8-26:

```
#include <stdio.h>

int main(void)
{
	int a[2][3][4];
	int *p,i;

	p = &a[0][0][0];

	/* Инициализиране на масива */
	for( i = 0; i < 2*3*4 ; i++ )
	{
		*(p + i) = i + 1;
	}
```

```
        /* Отпечатване на масива */
        for( i = 0; i < 2*3*4 ; i++ )
        {
                printf("%d ", *(p + i));
        }

        printf("\n");

        return 0;
}
```



Както е известно указателите могат да се индексират, така че вместо конструкцията ***(p + i)** може да се използва **p[i]**.

Адресът на първия елемент на масива може да се присвои на указателя **p** по няколко начина.

- **p = &a[0][0][0];**

Тъй като **a[0][0][0]** по същество е променлива от тип int, то нейния адрес може директно да се присвои на p, който е указател-към-int.

- **p = a[0][0];**

Тъй като **a[0][0]** е името на първия елемент-едномерен масив на първия елемент-двумерен масив на тримерния масив **a**, то **a[0][0]** всъщност представлява адреса на елемента **a[0][0][0]**, т.е. **a[0][0] == &a[0][0][0]**

- **p = (int *)a[0];**

Тъй като **a[0]** е указател към едномерен масив, а **p** е указател-към-int, то е необходимо указателя **a[0]** да се преобразува в указател-към-int по време на присвояването.

- **p = (int *)a;**

Тъй като **a** е указател към двумерен масив, а **p** е указател-към-int, то е необходимо указателя **a** да се преобразува в указател-към-int по време на присвояването.

- Достъп до елементите на тримерен масив чрез името на масив като указател

Пример 8-27:

```c
#include <stdio.h>

int main(void)
{
	int a[2][3][4];
	int i;

	for( i = 0; i < 2*3*4 ; i++ )
	{
		*((int *)a + i) = i + 1;
	}
	for( i = 0; i < 2*3*4 ; i++ )
	{
		printf("%d ", *((int *)a + i));
	}

	printf("\n");

	return 0;
}
```



Този пример е подобен на предходния с тази разлика, че вместо да се използва отделен указател, се използва името на масива като указател. Тъй като то е е указател към двумерен масив, за да се изпълнява правилно адресната аритметика, т.е да се достигат елементарните елементи на масива, се използва операция за привеждане на типа (int *)a.

1.3.5 Предаване на тримерен масив на функция

Следните три декларации са еквивалентни:

- `тип име-функция(тип име-масив[размер1][размер2][размер3]);`
- `тип име-функция(тип име-масив[][размер2][размер3]);`
- `тип име-функция(тип (*име-указател)[размер2][размер3]);`

Когато тримерен масив се предава на функция в действителност се предава указател към двумерен масив. Ето защо и приемащия параметър на функцията също представлява указател към двумерен масив. **размер1** може да се пропусне, но **размер2** и **размер3** са задължителни, тъй като именно те указват размерността на двумерния масив, към който сочи указателя.

Пример 8-28:

```c
#include <stdio.h>

void init_a(int array[][3][4], int value);

int main(void)
{
	int a[2][3][4];
	int i,j,k;

	/* Инициализиране на масива */
	init_a(a, 5);

	/* Отпечатване на масива */
	for( i = 0; i < 2; i++ )
	{
		for( j = 0; j < 3; j++)
		{
			for(k = 0; k < 4; k++)
			{
				printf("%d ", a[i][j][k] );
			}

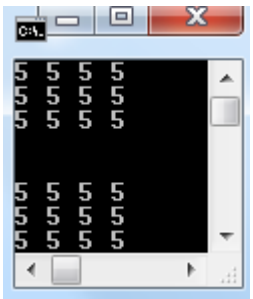
			printf("\n");
		}

		printf("\n\n");
	}

	return 0;
}

void init_a(int array[][3][4], int value)
{
	int i,j,k;

	for( i = 0; i < 2; i++ )
	{
		for( j = 0; j < 3; j++)
		{
			for(k = 0; k < 4; k++)
			{
				array[i][j][k] = value;
			}
		}
	}
}
```



Както и при едномерните и двумерни масиви, функцията няма информация за броя на елементите-двумерни масиви, затова при необходимост може да се декларира още един параметър, който да указва този брой.

1.4 Многомерни масиви

1.4.1 Определение и дефиниране

N-мерен масив (N = 2, 3, 4, …) по същество е едномерен масив, чийто елементи са (N-1)-мерни масиви. От своя страна елементите на тези (N-1)-мерни масиви са (N-2)-мерни масиви. Елементите на (N-2)-мерните масиви са (N-3)-мерни масиви и т.н.

Дефиниране на многомерен масив

тип име-масив[размер1][размер2]...[размерN];

където

размер1
Задава броя на елементите на N-мерния масив. Всеки елемент представлява масив с размерност N-1, т.е. (N-1)-мерни масиви. Трябва да бъде литерален константен целочислен израз.

размер2
Задава броя на елементите на (N-1)-мерните масиви. Всеки елемент представлява масив с размерност N-2, т.е. (N-2)-мерни масиви. Трябва да бъде литерален константен целочислен израз.

....

размерN
Задава броя на елементите в (N-(N-1))-мерните масиви. Трябва да бъде литерален константен целочислен израз.

тип
Дефинира типа на елементарните-елементи на N-мерния масив. Може да бъде всеки тип без битово поле и функции.

Пример:
```
int a[2][3][4][5][6]; /* Дефиниране на 5-мерен масив */
```

1.4.1.1 Инициализиране

Инициализиране на Nмерен масив

тип име-масив [размер1][размер2]...[размерN] = { списък-инициализатори[1] };

или

тип име-масив [][2][размер2]...[размерN]] = { списък-инициализатори[1] };

където

списък-инициализатори
Литерални константни изрази разделени със запетая.

Забележка[1]: Броят на инициализаторите трябва да е по-малък или равен на броя на елементарните-елементите на масива, получен от произведението на

(размер1*размер2...*размерN). Ако броят на инициализаторите е по-малък от броя на елементарните елементите, то елементарните елементите, за които не достигат инициализатори се инициализират с 0 (за целочислени типове), 0.0 (за типове с плаваща запетая) или NULL(указателни типове).

Забележка[2]: Когато масив се инициализира по време на дефинирането си, **размер1** може да се пропусне. Броят на елементите се определя от броя на инициализаторите.

Когато N-мерен масив се инициализира по време на дефинирането си, **размер1** може да се пропусне.

1.4.2 Индексиране

Индексиране на N-мерен масив

име-масив[индекс1][индекс2]...[индексN]

индекс1[1]
Може да бъде всеки целочислен израз, чиято стойност изпълнява условието

0 ≤ индекс1 < размер1 , т.е индекс1 Є [0...(размер1-1)]

индекс2[1]
Може да бъде всеки целочислен израз, чиято стойност изпълнява условието

0 ≤ индекс2 < размер2 , т.е индекс2 Є [0...(размер2-1)]

...

индексN[1]
Може да бъде всеки целочислен израз, чиято стойност изпълнява условието

0 ≤ индексN < размерN , т.е индексN Є [0...(размерN-1)]

Забележка[1]: Отговорност на програмистта е да следи индексите да не излиза извън рамките на масива в противен случай резултата е недефиниран.

1.4.3 N-мерни масиви и указатели

Името на масив представлява указател към първия елемент на масива. За N-мерен масив това означава, че името му е указател към (N-1)-мерен масив.

Дефиниране на указател към Nмерен масив

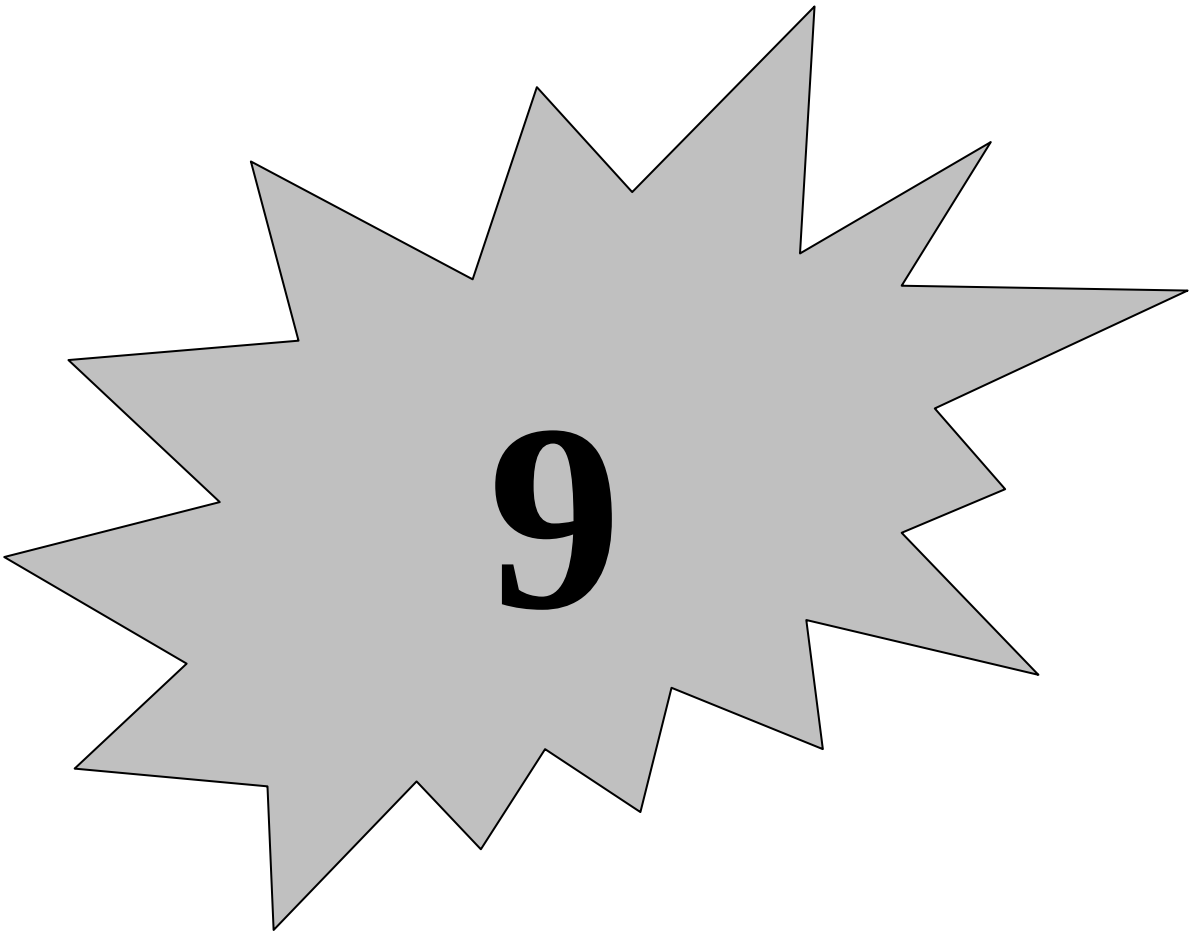
тип (*име-указател)[размер1][размер2]...[размерN];

1.4.4 Предаване на N-мерен масив на функция

Следните три декларации са еквивалентни:

- `тип име-функция(тип име-масив[размер1][размер2]...[размерN]);`
- `тип име-функция(тип име-масив[][размер2]...[размерN]);`
- `тип име-функция(тип (*име-указател)[размер2]...[размерN]);`

Когато N-мерен масив се предава на функция в действителност се предава указател към (N-1)-мерен масив. Ето защо и приемащия параметър на функцията също представлява указател към (N-1)-мерен масив. **размер1** може да се пропусне, но всички други са задължителни. Тъй като функцията приема само указател към първият елемент на N-мерния масив, тя няма информация за техния брой. При необходимост може да се декларира друг параметър, който да указва този брой.



Указатели

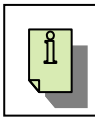
1.1 Определение и свойства

Указателят е променлива, чиято стойност се интерпретира като адрес. Това може да бъде адрес на променлива (обект) или на функция. С други думи указателите са променливи, които могат да сочат към други променливи (обекти) или функции и съгласно това се разделят на:

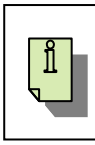
- Указатели към обекти (променливи)
- Указатели към функции

Съществува още един тип указатели, т.нар. **обобщени указатели**. Обобщеният указател е указател към тип void, затова се нарича още void-указател. Void-указателите могат да сочат към обекти от всякакви типове, но за да се достъпи (прочете или запише) обекта, void-указателят трябва да се преобразува в указател към типа на този обект, в противен случай компилаторът ще генерира грешка.

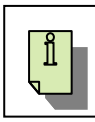
Като всяка променлива указателите също заемат място в паметта. С стандартът обаче не дефинира размера на указателите. Размерът на един указател зависи от имплементацията и в някои случаи зависи дори и от типа на обекта, към който сочи. Например указателите към обекти може да са "по-къси" или "по-дълги" от указателите към функции.



Езикът C не дефинира размера на указателите.



char- и void- указателите могат да бъдат "по-големи" от другите видове указатели и могат да използват вътрешно представяне, което се различава от другите указатели.



Указатели към функции и указатели към обекти могат да имат значителни разлики във вътрешното си представяне и в размера.

1.2 Указатели към обекти

Дефиниране на указател към обект

```
тип   *име-указател;
тип*   име-указател;
тип *  име-указател;
```

където

тип

```
Тип на обектите, към които указателя може да сочи. Може да бъде скаларен
тип, агрегатен тип и union-тип.
```

Пример 9-1:

```
/* Деклариране на структурен тип */
struct mystruct
{
	int x;
	int y;
};

/* Деклариране на union-тип */
union myunion
{
	int x;
	double y;
};

/* Деклариране на enum-тип */
enum myenum
{
	ONE,
	TWO,
	THREE
};

int main(void)
{
	int 		*pi;	/* Указател към int 			*/
	struct mystruct	*ps;	/* Указател към структурен-тип struct mystruct	*/
	union  myunion	*pu;	/* Указател към union-тип union  myunion	*/
	enum myenum	*pe;	/* Указател към enum-тип enum myenum		*/
	int (*pa)[10];		/* Указател към масив от 10 елемента тип int	*/

	/*...*/

	return 0;
}
```

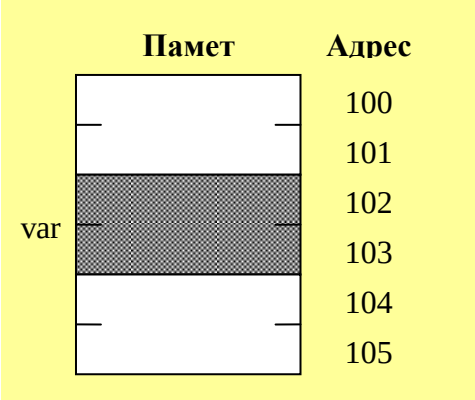
Основните операции за работа с указатели са извличане на адрес на променлива & и дереференциране на указател * (достъп до сочения от указателя обект).

Извличане на адрес на променлива

```
&име-променлива
```

Тази конструкция връща като резултат адреса на променливата, към която е приложена операцията &. Точната стойност на адреса не е от значение в повечето ситуации.

Нека предположим, че имаме променлива тип int, която заема два байта в паметта на адреси 102 и 103 (Фиг.65) .



Фиг. 65 Извличане на адрес на обект

Конструкцията **&var** ще върне като резултат 102, т.е операцията & извлича най-младшия адрес от заеманата памет при многобайтови променливи.

Достъп до променлива чрез указател (косвен достъп)

```
*име-указател
```

Тази конструкция връща като резултат обекта, сочен от указателят и може да стои както отляво така и отдясно на операцията за присвояване или да се използва в изрази. В първия случай променливата сочена от указателя ще бъде записана, а в останалите ще бъде прочетена.

Може да възникне въпроса как указателят знае колко байта да прочете или запише, ако променливата към която сочи е многобайтова, при условие той сочи само към най-младшия адрес. Отговорът е съвсем прост и се крие в дефиницията на указателя. Спомнете си, че дефиницията съдържа типа на обектите, към които този указател е предназначен да сочи. Имайки тази информация, указателят "знае" колко байта трябва да прочете или да запише, започвайки от най-младшия адрес на обекта, към който сочи. При това под внимание се взима също и типа на компютъра (**big-endian** или **little-endian**) (вижте Глава 2 Типове и данни)

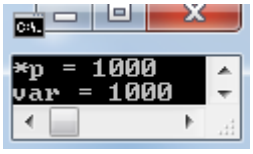
Пример 9-2:

```
#include <stdio.h>

int main(void)
{
	int *p;			/* Дефиниране на указател p към int променлива   */
	int var;		/* Дефиниране на променлива var от тип    int    */

	p = &var;		/* Указателят p сочи към var                     */
	*p = 1000;		/* На var се присвоява 1000 чрез указателя p     */
	printf("*p = %d\n", *p);  /* Отпечатване на var косвено          */
	printf("var = %d\n", var);/* Отпечатване на var директно         */

	return 0;
}
```



1.3 Указатели към функции

При дефинирането си функцията заема определен участък в паметта, където се разполага нейното тяло. В C е напълно допустимо да се дефинира указател към функция и да се инициализира с адреса на дадена функция. Адресът на функцията е началото на блока памет, където функцията се съхранява. Името на функцията в действителност представлява адреса на самата функция.

Дефиниране на указател към функция

```
тип    (*име-указател)(списък-параметри);

където

тип
      Дефинира типа на резултата, връщан от функцията.

списък-параметри
      Дефинира списъка с параметри на функцията.
```

Инициализиране на указател към функция

```
име-указател = име-функция[1];

или

име-указател = &име-функция;
```

Забележка[1]: Името на функция автоматично се преобразува в указател към функцията.

Извикване на функция чрез указател

```
(*име-указател)(списък-аргументи)

или

име-указател(списък-аргументи)
```

Двете конструкции за извикване на функция чрез указател са напълно еквивалентни.



Когато функция се извиква чрез указател за предпочитане е използването на формата `(*име-указател)(списък-аргументи)`, тъй като тя указва ясно, че извикването става чрез указател, а не чрез името на функцията. Това подобрява самодокументирането на кода

Пример 9-3:

```
#include <stdio.h>

void foo(void);

int main(void)
{
	void (*p)(void);	/* Дефиниране на указател към функция, която	*/
				/* не връща резултат и няма параметри		*/
	p = &foo;		/* Инициализиране на указателя с адреса на f() */
	/*p = foo;*/		/* Същото като горната конструкция		*/
	(*p)();			/* Извикване на функцията чрез указателя p	*/
	/* p(); */		/* Същото като горната конструкция		*/

	return 0;
}

void foo(void)
{
	printf("Function pointer test\n");
}
```



Указателите към функции позволяват предаването на функции като параметри на други функции.

Пример 9-4:

```
#include <stdio.h>

typedef void ptr_vfunc(void); /* Деклариране на тип ptr_vfunc */

void test_foo(ptr_vfunc* p);
void func1(void);
void func2(void);

int main(void)
{
	test_foo(func1); /* Подаване на името на func1() като аргумент */
	test_foo(func2); /* Подаване на името на func2() като аргумент */

	return 0;
}

void test_foo(ptr_vfunc* p)
{
	(*p)(); /* звикване на функцията сочена от указателя p */
}

void func1(void)
```

```
{
	printf("Inside func1\n");
}

void func2(void)
{
	printf("Inside func2\n");
}
```



1.4 Обобщени указатели (void-указатели)

C89 въвежда т.нар. **обобщени указатели**. Това са указатели към тип void, затова се наричат още **void-указатели**.

Дефиниране на void-указател

```
void* име-указател;
```

Обобщените указатели могат да сочат само към обектни типове, т.е обобщен указател към функция не съществува.

Указател към void трябва да има същото вътрешно представяне и подравняване като указател към char.

Обобщените указатели не могат да се дереференцират или индексират, нито да бъдат операнди на операциите за събиране и изваждане. Всеки указател към обект или непълен тип (с изключение на функция) може да бъде преобразуван в void-указател и обратно без промяна. Когато обаче void-указател се използва за достъп до обекта, сочен от него, е необходимо той да се преобразува към типа на обекта, в противен случай компилаторът ще генерира грешка, тъй като няма да знае размера на сочения обект и съответно как да го обработи.

Пример 9-5:

```
#include <stdio.h>

int main(void)
{
	void  *pv;   /* Дефиниране на void-указател            */
	int   *pi;   /* Дефиниране на int-указател             */
	float *pf;   /* Дефиниране на float-указател           */
	int    i;    /* Дефиниране на променлива x от тип int   */
	float  f;    /* дефиниране на променлива f от тип float */

	pv = &i;     /* void* = int*                             */
```

```
        pi = pv;     /*  int* = void*                              */

        *(int*)pv = 10; /* Явно преобразуване на void* в int*    */
        printf("%d\n", *(int*)pv );

        pv = &f;     /* void* = float*                           */
        pf = pv;     /* float* = void*                            */

        *(float*)pv = 10.54f; /* Явно преобразуване на void* в float* */
        printf("%f\n", *(float*)pv );

        return 0;
}
```



1.5 Нулеви указатели

На всеки указател може да се присвои една специална стойност, наречена константен нулев указател NULL. Такъв указател се нарича нулев указател (**null-pointer**) и се счита, че не сочи към никакъв обект. В контекст, който изисква булев израз, стойността на нулевия указател се интерпретира като ЛЪЖА (FALSE). Например следната конструкция

```
if (ip) { i = *ip;}
```

е еквивалентна на

```
if (ip != NULL) { i = *ip;}.
```

Макросът NULL е дефиниран в хедър файла <stddef.h> както е показано по-долу.

```
#define NULL  ((void*)0)
```

или

```
#define    NULL   (0)
```

Обикновено нулевите указатели имат вътрешно представяне, при което всички битове са нула, но това не е задължително. Вътрешното представяне на различните указателни типове може да се различава.

Пример 9-6:

```
#include <stdio.h> /* Включва хедър-файла stddef.h */

void foo(int* p);
```

```
int main(void)
{
	int* p = NULL; /* p не сочи към нищо */
	int i;

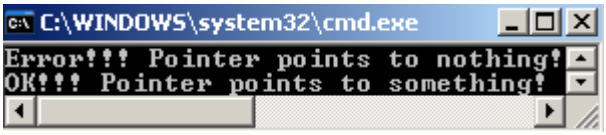
	foo(p);
	p = &i;		/* p сочи към променливата i */
	foo(p);

	return 0;
}

void foo(int* ptr)
{
	if(ptr == NULL) /* Проверка за нулев указател */
	{
		printf("Error!!! Pointer points to nothing!\n");

		return;
	}

	printf("OK!!! Pointer points to something!\n");
}
```





Ако дефинирате указател без да го инициализирате с адрес на променлива, добра практика е да го инициализирате с NULL. Така по-късно в кода може да го проверите дали сочи към валиден адрес или не.

1.6 Указатели и квалификатори на типа

В C89 указателите могат да бъдат квалифицирани с квалификаторите на типа const и volatile.

Указатели и квалификатори const и volatile

```
квалификатор1 тип* квалификатор2 име-указател;
```

където

квалификатор1/квалификатор2
const
volatile
const volatile
volatile const

Квалификатор **квалификатор1** влияе на обекта, сочен от указателя, а **квалификатор2** влияе на самия указател. По долу са дадени всички възможни комбинации.

- `const тип* име-указател;`
- `volatile тип* име-указател;`
- `const volatile тип* име-указател;`

- `тип* const име-указател = &име-променлива;`
- `тип* volatile име-указател;`
- `тип* const volatile име-указател = &име-променлива;`

- `const тип* const име-указател = &име-променлива;`
- `const тип* volatile име-указател;`
- `const тип* const volatile име-указател = &име-променлива;`

- `volatile тип* const име-указател = &име-променлива;`
- `volatile тип* volatile име-указател;`
- `volatile тип* const volatile име-указател = &име-променлива;`

- `const volatile тип* const име-указател = &име-променлива;`
- `const volatile тип* volatile име-указател;`
- `const volatile тип* const volatile име-указател = &име-променлива;`

Ще припомним само, че квалификатора const гарантира само, че една променлива не може да се променя в кода с помощта на програмни инструкции, като присвояване на нова стойност и т.н., но const променливите могат да се променят от някакви външни събития по отношение на програмата. Квалификаторът volatile информира компилатора, че променливата може да се измени неявно, т.е. без намесата на програмни инструкции, например от хардуера.

Пример 9-7:

```
int main(void)
{
  int x = 10, y = 20, z = 30;
  const int* ptr_to_const;
  int* const const_ptr = &x;
  const int* const const_ptr_to_const = &x;

  ptr_to_const = &x;            /* Правилно, ptr_to_const може да сочи към
                                   всякакви обекти тип int */

  /* *ptr_to_const = 10; */     /* !!!Грешка, ptr_to_const не може да изменя
                                   обекта      */

  *const_ptr = 10;             /* Правилно, const_ptr може да изменя обекта     */
  /* const_ptr = &y; */        /* !!!Грешка, const_ptr не може да сочи към друг
                                   обект */

  z = *const_ptr_to_const;      /* Правилно, const_ptr_to_const може да чете
                                   обекта */

  /* *const_ptr_to_const = 10; */ /* !!!Грешка, ptr_to_const не може да изменя
                                   обект */
   return 0;
}
```

1.7 Операции с указатели

1.7.1 Събиране и изваждане на указател с цяло число

За да разберете по-ясно как действа адресната адресната аритметика нека дефинираме масив от int елементи и указател, който сочи към началото на този масив. Приема се, че int променливите са с размер два байта.

Пример 9-8:

```
#include <stdio.h>

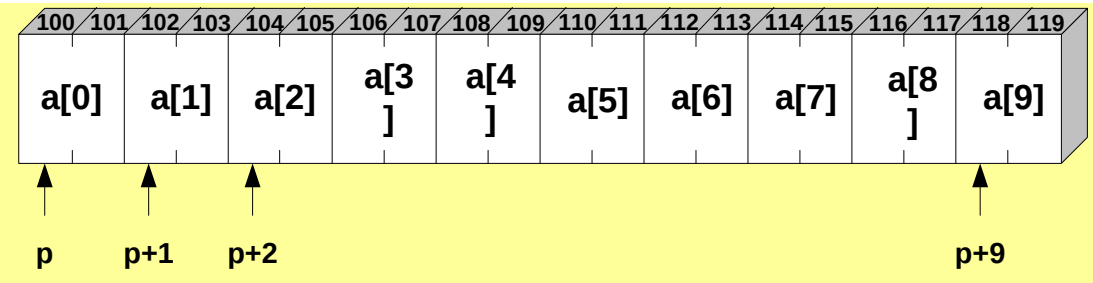
int main(void)
{
      int    a[10];       /* Дефиниране на масив от 10 int елемента      */
      int    *p;          /* Дефиниране на указател към int              */

      p = a;              /* Адресът на елемента a[0]  се присвоява на p  */
      p = p + 1;          /* Увеличаване на указателя с 1                */

      /*...*/

      return 0;
}
```

На Фиг.66 е показано разположението на масива **a** в паметта.



Фиг. 66 Разположение на масива a в паметта

Конструкцията **p = a** присвоява на **p** адреса на първия елемент **a[0],** т.е. 100 (стойностите на адресите са примерни). Конструкцията **p = p +1** увеличава указателя, така че да сочи към следващия елемент на масива, т.е към **a[1]**. Конструкцията **p + i** практически увеличава **p** така че да сочи към **i**-я елемент на масива. С други думи казано при увеличение/намаление на указател с цяло число **i** , указателят се увеличава/намалява с **i*sizeof(тип)** байта, така че да сочи към следващия/предходния елемент от дадения тип. Конструкцията **sizeof(тип)** представлява размера в байтове на обекта, сочен от указателя. За нашия пример конструкцията **p+1** в действителност увеличава **p** с **1*sizeof(int) = 1*2 = 2** байта, т.е от адрес 100 указателя се прехвърля на адрес 102, който е адреса на следващия елемент на масива.

1.7.2 Операциите ++ и -- и указатели

Нека p е указател към някакъв тип данни. Следните конструкции са напълно валидни:

- *p++ == *(p++) -> еквивалентни операции 1) *p 2) p += 1
  *p-- == *(p--) -> еквивалентни операции 1) *p 2) p -= 1

  Първо се извлича обекта, сочен от указателя (неговата стойност ще се използва ако тази конструкция е част от по-голям израз), след което указателят се увеличава/намалява.

- (*p)++ -> еквивалентни операции 1) *p 2) обект += 1
  (*p)-- -> еквивалентни операции 1) *p 2) обект -= 1

  Първо се извлича обекта, сочен от указателя (неговата стойност ще се използва ако тази конструкция е част от по-голям израз), след което стойността на обекта се увеличава/намалява.

- *++p == *(++p) -> еквивалентни операции 1) p += 1 2) *p
  *--p == *(--p) -> еквивалентни операции 1) p -= 1 2) *p

  Първо указателят се увеличава/намалява, след това се извлича стойността на обекта, сочен от указателя (тази стойност ще се използва ако тази конструкция е част от по-голям израз).

- ++*p == ++(*p) -> еквивалентни операции 1) *p 2) обект += 1
  --*p == --(*p) -> еквивалентни операции 1) *p 2) обект -= 1

  Първо се извлича обекта, сочен от указателя, след което неговата стойност се увеличава/намалява (новата стойност ще се използва ако тази конструкция е част от по-голям израз).

Пример 9-9:

```
#include <stdio.h>

int main(void)
{
	int   a[2];
	int     x;
	int    *p;

	a[0] = 10;
	a[1] = 20;
	p = a;
	x = *p++;
	printf("x = %d a[0] = %d a[1] = %d a = %p p = %p\n", x, a[0], a[1], a, p);

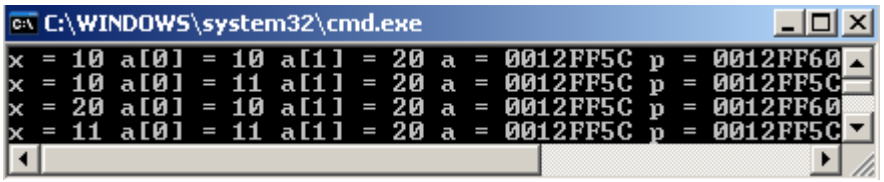
	a[0] = 10;
```

```
        a[1] = 20;
        p = a;
        x = (*p)++;
        printf("x = %d a[0] = %d a[1] = %d a = %p p = %p\n", x, a[0], a[1], a, p);

        a[0] = 10;
        a[1] = 20;
        p = a;
        x = *++p;
        printf("x = %d a[0] = %d a[1] = %d a = %p p = %p\n", x, a[0], a[1], a, p);

        a[0] = 10;
        a[1] = 20;
        p = a;
        x = ++*p;
        printf("x = %d a[0] = %d a[1] = %d a = %p p = %p\n", x, a[0], a[1], a, p);

        return 0;
}
```



1.7.3 Изваждане на указатели от един и същи тип

Изваждането на указатели от един и същи тип дава като резултат броя на елементите между двата указателя. Изваждането на указатели има смисъл само ако сочат към общ обект, например масив.

Пример 9-10:

```
#include <stdio.h>

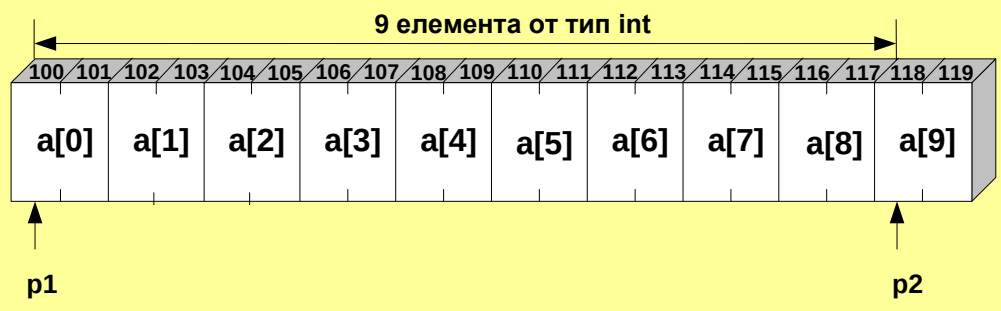
int main(void)
{
        int a[10];    /* Дефиниране на масив a с 10 елемента от тип int  */
        int *p1, *p2; /* Дефиниране на два int указателя                  */
        int number;

        p1 = &a[0];           /* p1 сочи към елемента a[0]                        */
        p2 = &a[9];           /* p2 сочи към елемента a[9]                        */
        number = p2 - p1;      /* number = p2 – p1 = (p1 +9) – p1 = 9             */
        printf("number = %d\n", number);   /* Отпечатване на number на дисплея   */

        return 0;
}
```



Фиг. 67 показва нагледно връзката между указателите p1, p2 и масива a.



Фиг. 67 Изваждане на указатели

p2 - p1 = (p1 + 9) - p1 = p1 + 9 - p1 = 9

Този израз може да се представи и по следния начин:

p1 – p2 = (118 – 100)\sizeof(int) = 18/2 = 9

Обърнете внимание, че void-указателите нямат информация за типа на обектите, към които сочат. По тази причина, те трябва да бъдат преобразувани до указатели от съответния тип.

Пример 9-11:

```
#include <stdio.h>

int main(void)
{
	int a[10];	/* Дефиниране на масив a с 10 елемента от тип int  */
	void *p1, *p2; /* Дефиниране на два void-указателя                */
	int number;

	p1 = &a[0];		/* p1 сочи към елемента a[0]                    */
	p2 = &a[9];		/* p2 сочи към елемента a[9]                    */
	number = (int*)p2 - (int*)p1; /* number = p2 - p1 = (p1 +9) - p1 = 9   */
	printf("number = %d\n", number); /* Отпечатване на number на дисплея   */

	return 0;
}
```



1.7.4 Сравнение на указатели

Указателите от един и същ тип могат да се сравняват. Тази операция има смисъл само ако указателите сочат към общ обект, например масив. Следващият пример използва тази възможност, за да размени символите на низ, т.е "Hello C" -> "C olleH".

Пример 9-12:

```
#include <stdio.h>

void StrRev(char *s);

int main(void)
{
	char str[] = "Hello C";

	printf("%s\n", str);
	StrRev(str);
	printf("%s\n", str);

	return 0;
}

void StrRev( char *s)
{
	char *p1, *p2;
	char t;

	p1 = p2 = s;

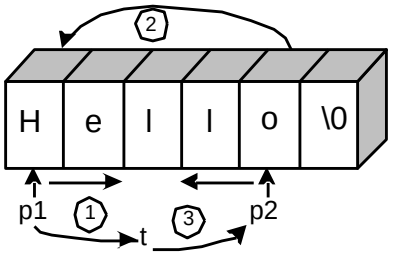
	while(*p2 != '\0')
	{
		p2++;
	}

	p2--;

	while(p2 > p1)
	{
		t = *p1;

		*p1++ = *p2;
		*p2--  = t;
	}
}
```





Всеки указател-към-обект може да се сравнява за равенство (= = и !=) и с void-указател. Обектният указтел ще се преобразува автоматично в void указател преди да се извърши сравнението.

1.7.5 Сравнение на указател с цяло число

Единственото цяло число, с което указател може да се сравнява е 0.

Пример 9-13:

```
#include <stdio.h>

void foo(char *s);

int main(void)
{
	char* str = 0;

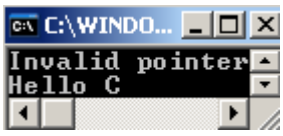
	foo(str);
	str = "Hello C";
	foo(str);

	return 0;
}

void foo(char *s)
{
	if (0 == s) /* Същото като if (NULL == s) */
	{
		printf ("Invalid pointer\n");

		return;
	}

	printf ("%s\n", s);
}
```



В контекста на указателите 0 или (void*)0 се нарича нулев константен указател. Езикът C дефинира специален макрос NULL в <stddef.h>, който представлява нулев константен указател (вижте точка 5 Нулеви указатели)

Нулевият константен указател автоматично се преобразува в типа на другия указател при сравнението.

1.8 Указатели като параметри на функция

Формалните параметри на функция също могат да бъдат дефинирани като указатели.

Указател като параметър на функция

```
тип    име-функция(тип*   име-указател);
```

Използването на указателя-параметър позволява на функцията да променя обекта, сочен от указателя-аргумент, тъй като при извикване на функцията, адресът, който се съдържа в него се копира в указателят-параметър, следователно функцията има достъп до обекта, сочен от указателя-аргумент. Разбира се всяка промяна на стойността на указателя-параметър няма да се отрази на указателя-аргумент, тъй като при извикване на функцията в стека се заделя памет за указателя-параметър, където стойността на указателя-аргумент се копира.

Пример 9-14:

```
#include <stdio.h>

void swap(int *x, int *y);

int  main(void)
{
      int a,b;

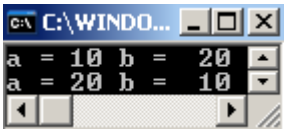
      a = 10;
      b = 20;

      printf("a = %d b = % d\n", a,b);
      swap(&a, &b); /* Размяна на стойностите на a и b */
      printf("a = %d b = % d\n", a,b);

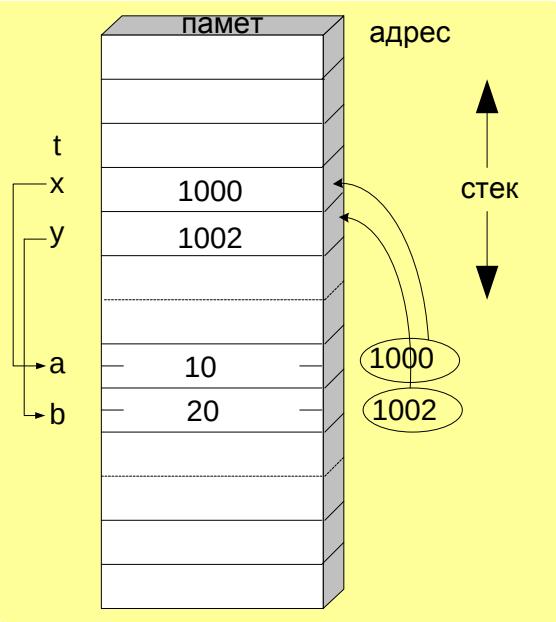
      return 0;
}

void swap(int *x, int *y)
{
      int t;

      t = *x;
      *x = *y;
      *y = t;
}
```



Фиг.68 дава визуална представа как указателите-параметри x, y, сочат към обектите a и b.



Фиг. 68 Указател като параметър на функция

Тъй като функцията swap() приема за параметри указатели, то стойностите които трябва да й се предадат са адреси. Ето защо на нея се предават адресите на променливите a и b чрез конструкциите &a и &b.

1.9 Указатели връщани от функция

Указателите могат да бъдат връщани като резултат на една функция.

Дефиниране на функция, която връща указател

```
тип* име-функция(списък-параметри);
```

Когато указател се връща от функция, обектът, сочен от указателя трябва да съществува и след завършване на функцията, т.е. обектът трябва да има статична продължителност на живот (външна или статична променлива) . Ако се върне адрес на автоматична променлива, паметта заемана от нея ще бъде освободена след завършване на функцията и върнатият адрес ще бъде невалиден.

Пример 9-15:

```
#include <stdio.h>

char* foo(void);

int main(void)
{
	char *s;

	s = foo();
	printf("%s\n", s);
```

```
	return 0;
}

char*  foo(void)
{
	static char* str = "Hello C";

	return str;
}
```



Обърнете специално внимание, че указателя str в foo() е дефиниран с ключовата дума static. Това гарантира, че дори и след излизане от foo(), указателя str ще продължи да съществува.

1.10 Инициализиране на char-указател с низова константа

Указател от тип char може да се инициализира да сочи към низ още по време на неговата дефиниция или по-късно в програмата.

char-указател и низови константи

```
char *име-указател = "низова-константа";

или

char *име-указател;
име-указател = "низова-константа";
```

Когато компилаторът срещне някоя от тези две конструкции се извършват следните действия:

1. Компилаторът заделя памет за указателя.

2. Низовата константа се съхранява в паметта като едномерен символен масив.

3. Адресът на първия символ се връща като резултат и се присвоява на указателя.

Пример 9-16:

```
#include <stdio.h>

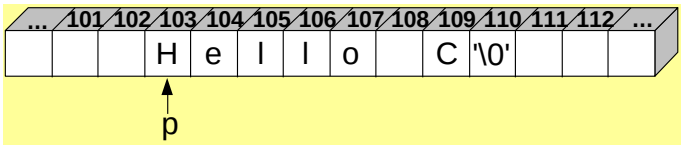
int main(void)
{
	char *p = "Hello C";

	printf("%s\n", p);
```

```
        return 0;
}
```



Фиг.69 дава визуална представа за това как се съхранява низова константа в паметта на компютъра.



Фиг. 69 Съхраняване на низова константа в паметта

1.11 Указател към указател

Указателят също е променлива, която се съхранява някъде в паметта на компютъра на определен адрес. Този адрес може да се присвои на друг указател.

Дефиниране на указател-към-указател

```
тип    **име-указател-към-указател;

където

тип
        Типа на данните, към които сочи указателя, сочен от указателя-към-указател.
```

Пример 9-17:

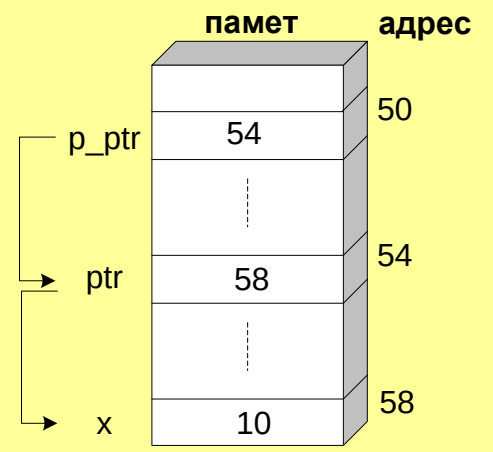
```
#include <stdio.h>

int main(void)
{
        int x;
        int *ptr, **p_ptr;
        ptr = &x;     /* ptr сочи към x                  */
        p_ptr = &ptr; /* p_ptr сочи към ptr              */
        **p_ptr = 10; /* същото като *ptr = 10 и x = 10 */
        printf("x = %d\n", **p_ptr);

        return 0;
}
```



Фиг.70 показава нагледно връзката между, ptr, p_ptr и x.



Фиг. 70 Указател-към-указтел

Нека разгледаме конструкцията **p_ptr = 10 по-отблизо. Тя е еквивалентна на *(*p_ptr) = 10. Конструкцията *p_ptr дава като резултат указателя ptr, т.е *(*p_ptr) ≈ *(ptr). Конструкцията *(ptr) от своя страна дава като резултат променливата x, т.е *(ptr) ≈ x. В крайна сметка числото 10 се присвоява на променливата x , т.е

(**p_ptr ≈ *(*p_ptr) ≈ *(ptr) ≈ x) = 10

В C няма ограничение за вложеността на указателите, но повече от две нива на вложеност е трудно проследима.

1.12 Указател-към-указател и квалификатори на типа

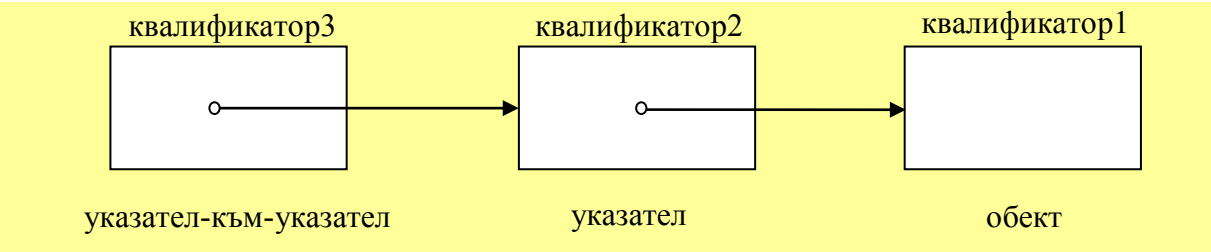
Дефиниране на указател-към-указател и квалификатори

```
тип квалификатор1 * квалификатор2 * квалификатор3 име-указател-към-указател;

където

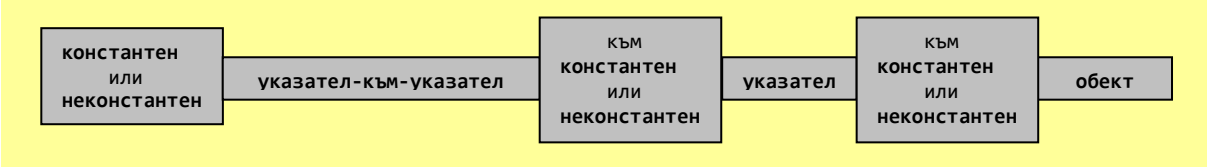
квалификатор1/квалификатор2/квалификатор3
const
volatile
const volatile
volatile const
```

Фиг. 71 показва връзката между **указател-към-указател**, сочения от него **указател**, сочения от указателя **обект** и квалификаторите. Квалификатор1 влияе на сочения от **указател** обект, квалификатор2 влияе на сочения от **указател-към-указател** указател, и **квалификатор3** влияе на самия **указател-към-указател**.



Фиг. 71 Квалификатори на типа и указател към указател

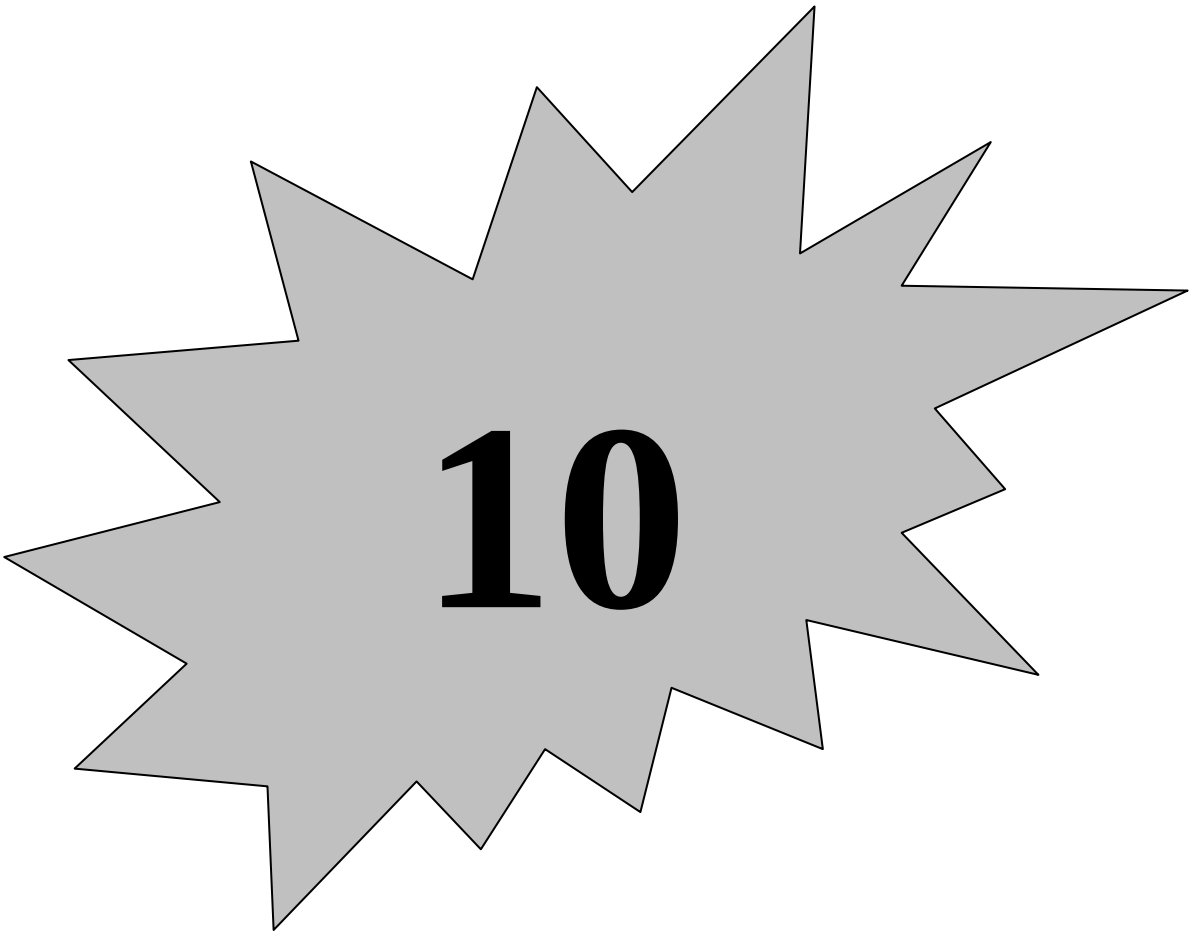
Следващата фигура показва с думи вариантите, които могат да се получат при използването на квалификатора const.



Фиг. 72 Комбинации на квалификатора const в дефиницията на указател-към-указател

1.13 Преобразувания свързани с указатели

Преобразуванията свързани с указатели са разгледани подробно в Глава 5 Преобразувания.



Структури

1.1 Въведение в структурите

1.1.1 Деклариране на структура

Структурата е съвкупност от променливи, които могат да бъдат от различен тип. Декларирането на структура въвежда нов тип данни в програмата, т.е. структурата е комплексен тип данни, създаден от самия програмист.

Деклариране на структура

```
struct име-структура                    } Структурен тип
{                                       }
      тип1   име-променлива1;           }
                                        }
      тип2   име-променлива2;           } Тяло на структурата
      ...                               }
      типN   име-променливаN;           }
};

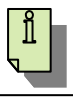
където

тип1/тип2/.../типN
      Може да бъде скаларен тип, агрегатен тип, обединение и битово поле.
```

Декларацията на структурата се въвежда с ключовата дума struct, последвана от името на структурата и тялото на структурата. Забележете, че декларацията на структурата завършва с точка и запетая. Името на структурата се нарича още етикет на структура (**tag**). Променливите, декларирани в тялото на структурата, се наричат членове или полета. Декларирането на структура не води до заделяне на памет за нейните членове. Декларацията представлява просто шаблон, описващ нов тип данни.

Пример:

```
struct mystruct/* Деклариране на структура */
{
      int i;
      double d;
};
```

	**Декларацията на структура завършва с точка и запетая.**
	**Декларацията на структура създава нов тип данни. Компилаторът заделя памет за членовете само когато се дефинира променлива от тази структура.**
	**Членовете на структурата не могат да се инициализират в тялото на структурата. Те могат да се инициализират само при дефиниране на променлива от тази структура.**

Пример:

```
/* Деклариране на структура */
struct mystruct
{
    int i    = 10;  /* !!! Грешка, членовете на структурата не мога да се
                       инициализиратцията в декларацията */
    double d = 2.3; /* !!!Грешка, членовете на структурата не мога да се
                       инициализират в декларацията */
};
```

Всяка структура дефинира нова област на видимост (именовано пространство). Идентификаторите на членовете на структурата трябва да се различават един от друг, но могат да съвпадат с всеки друг идентификатор на променлива, функция, член на друга структура и т.н. без това да създава конфликт. Идентификатор на член на една структура може да съвпада дори с името на структурата и структурна променлива от тази структура.

Пример 10-1:

```
struct mystruct
{
      int i;         /* Съвпада с име на член на  друга структура */
      double d;      /* Съвпада с име на член на  друга структура */
      char mystruct; /* Съвпада с име на:
                        -член на  друга структура,
                        -същата структура,
                        -структурна променлива от тази структура,
                        -локална променлива */
} mystruct;

struct anotherstruct
{
      int i;
      double d;
      char mystruct;
      char anotherstruct; /* Съвпада с името на структурата и име на функция */
} ;

void anotherstruct(void)
{
      /*...*/
}

int main( void )
{
      int mystruct;

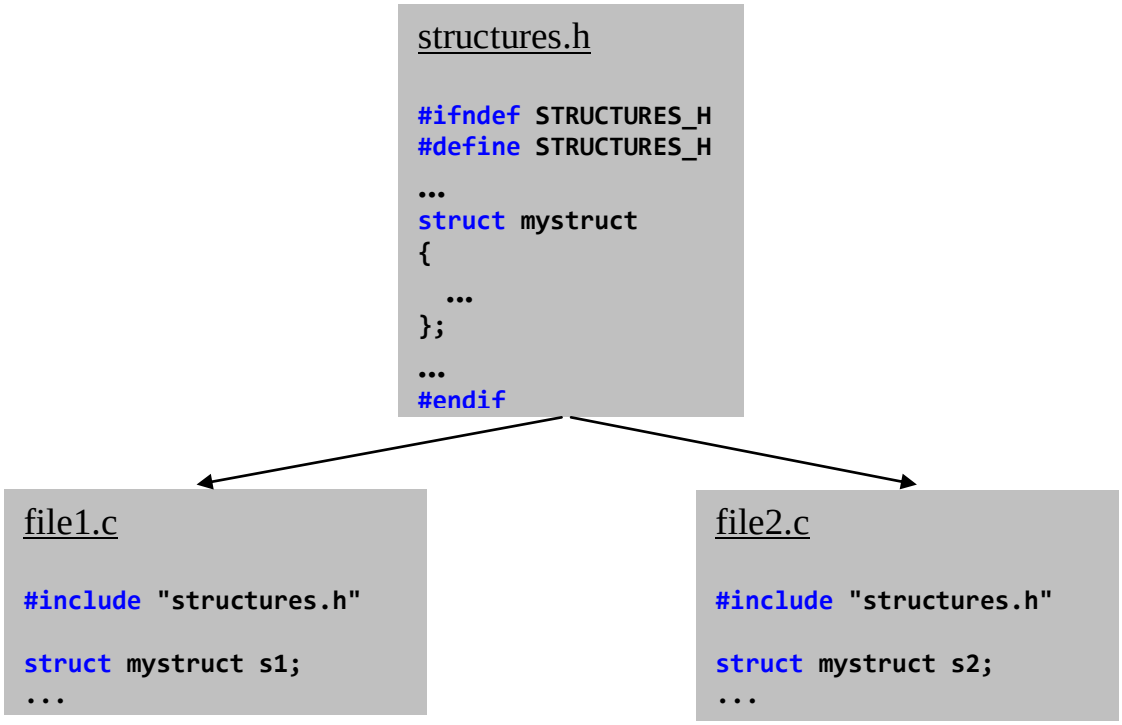
      /*...*/

      return 0;
}
```

Именованите пространства в езика C са описани в Глава 1 Лексикални елементи.

1.1.2 Област на видимост на структура

Декларацията на една структура е видима от точката на деклариране до края на файла или блока, в който се намира декларацията. Всеки файл, който използва структурата трябва да включва и нейната декларация. В такива случай е най-добре структурата да се декларира в хедър-файл и този файл да се включва във всеки сорс-файл, който я използва както е показано по-долу.



1.1.3 Дефиниране на променливи от структурен тип

Веднъж декларирана, структурата може да се използва за дефиниране на променливи. Такива променливи ще наричаме структурни променливи или просто struct-променливи. Комбинацията от ключовата дума struct и името на структурата определя нейния тип. Именно тази комбинация се използва при дефиниране на struct-променливи.

Дефиниране на struct-променливи

```
struct име-структура      име-структурна-променлива;
```

структурен тип

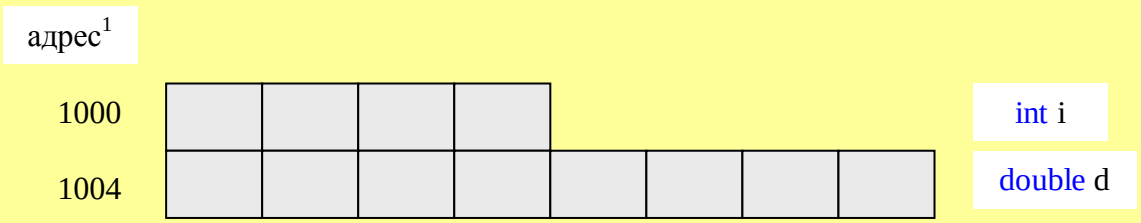
Когато компилаторът срещне подобна конструкция, той заделя памет за всички членове на структурата по реда на появяването им в декларацията на структурата.

Пример:

```
/* Деклариране на структура */
struct mystruct
{
	int i;
	double d;
};

struct mystruct	s; /* Дефиниране на структурна променлива s */
```

Горната конструкция води до заделяне на памет за променливата s, която се състои от едно int поле и едно double поле (Фиг.71). В примера се приема, че тип int е 4 байта, а double 8.



Забележка[1]: Адресите, на които се разполагат членовете на структурната-променлива са произволно избрани в примера и имат само илюстративен характер.

Фиг. 73 Членове на структурната променлива s

Структурните променливи могат да се дефинират и по време на дефиниране на самата структура.

Деклариране на структура и структурни променливи едновременно

```
struct име-структура[1]
{
	тип1	име-променлива1;

	тип2	име-променлива2;
	...
	...
	типN	име-променливаN;
}списък-структурни-променливи;

където

списък-структурни-променливи
	име-структурна-променлива1,...,име-структурна-променливаN
```

Забележка[1]: Ако няма да дефинирате други struct-променливи, освен тези указани в **списък-структурни-променливи, име-структура** може да се изпусне.

Пример:

```
/* Дефиниране на структурни променливи по време на деклариране на структура */
struct mystruct
{
	int i;
	double d;
} s1, s2;

...

/* Дефиниране на допълнителна структурна променлива по-късно в кода */
struct mystruct s3;
```

Променливите s1 и s2 са struct-променливи дефинирани по време на деклариране на самата структура. Променливата s3 се дефинира по-късно в кода по стандартния начин.

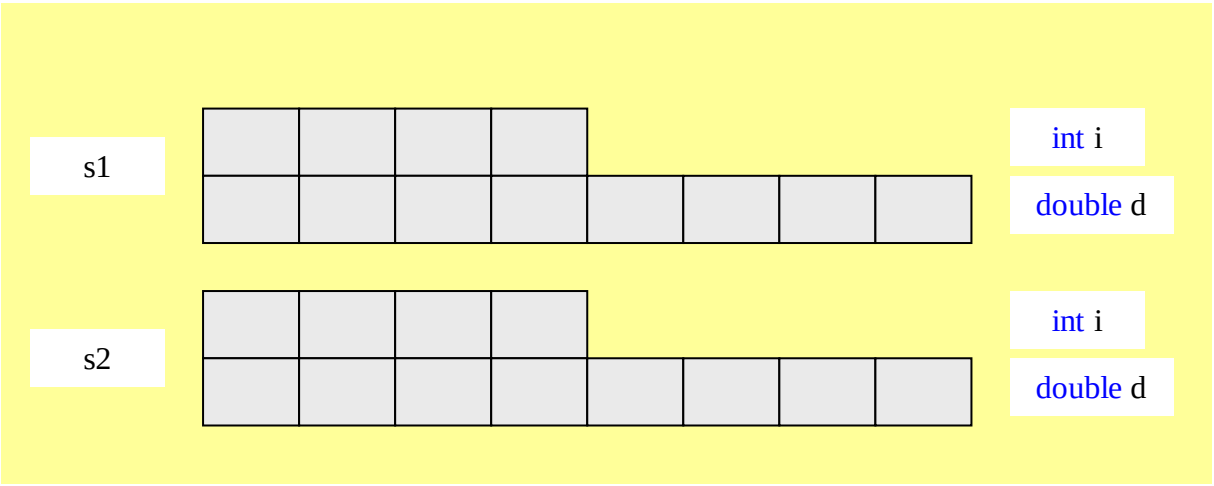
Когато struct-променливите се дефинират по време на деклариране на структурата, името на структурата може да се изпусне.

Пример:

```
/* Деклариране на структура без име */
struct
{
	int i;
	double d;
} s1, s2;
```

Изпускането на името на структурата обаче означава, че повече не може да дефинирате други променливи от този тип по-късно в програмата.

Всяка структурна променлива съдържа свой набор членове. Така например в горния пример структурната променлива s1 има две полета i и d, и s2 има две полета i и d, които са напълно отделни едно от друго (Фиг.72).



Фиг. 74 Членове на s1 и s2

1.1.4 Инициализация на променливи от структурен тип

Структурните променливи могат да се инициализират по време на дефинирането си подобно на инициализиране на елементите на масив. За целта след името на struct-променливата се използва операцията за присвояване, последвана от списък с инициализатори, затворен във фигурни скоби. Всеки инициализатор трябва да бъде литерален константен израз. Типът на инициализатора трябва да съвпада или да е съвместим с типа на съответния член.

Инициализиране на на struct-променливи

```
struct име-структура      име-структурна-променлива = {списък-инициализатори};

където

списък-инициализатори
      инициализатор1, инициализатор2,..., инициализаторN

инициализатор1, инициализатор2,..., инициализаторN
      Всеки инициализатор трябва да бъде литерален константен израз. Типът на
      инициализатора трябва да съвпада или да е съвместим с типа на съответния
      член.
```

Пример:

```
/* Инициализиране на структурни променливи */
struct
{
      int i;
      double d;
} s1 = {10, 2.5}, s2 = {30, 4.5};
```

	i	d
s1	10	2.5
s2	30	4.5

Ако броят на инициализаторите е по-малък от броя на полетата, полетата за които няма инициализатори се инициализират с 0 (за целочислени типове), 0.0 (за типове с плаваща запетая) или NULL (за указателни типове).

Пример:

```
/* Инициализиране на структурни променливи */
struct
{
      int i;
      double d;
} s1 = {10}, s2 = {30, 4.5};
```

	i	d
s1	10	0.0
s2	30	4.5

Ако броят на инициализаторите е по-голям от броя на полетата, компилаторът ще генерира грешка по време на компилиране.

Пример:

```
/* Инициализиране на структурни променливи */
struct
{
	int i;
	double d;
} s1 = {10, 2.3, 20}; /* !!!Грешка, инициализаторите са повече от полетата */
```

Само литерални константни изрази могат да се използват за инициализатори.

Пример:

```
int x    = 10;
double y = 2.3;

/* Инициализиране на структурни променливи */
struct
{
	int i;
	double d;
} s1 = {x, y}; /* !!!Грешка, инициализаторите трябва да са литерални константни
                  изрази */
```

Освен чрез списък с инициализатори една структурна променлива може да се инициализира и чрез друга вече дефинирана структурна променлива.

Пример:

```
struct mystruct s1 = {10, 2.3};/* Използване на литерални константи за
                                  инициализатори */

struct mystruct s2 = s1;       /* Използване на структурна променлива за
                                  инициализатор */
```

1.1.5 Структури и typedef

Както обяснихме в началото, типа на структурата се състои от ключовата дума struct и името на структурата, т.е. **struct име-структура.** Използвайки ключовата дума typedef, може да създадете синоним на структурния тип, който се състои просто от един идентификатор.

Деклариране на синоним на структурен тип

```
/* Деклариране на структура */
struct име-структура
{
	/*...*/
};

/* Деклариране на синоним на типа на структурата  */
typedef  struct име-структура   ново-име-на-тип;
```

Пример 10-2:

```
/* Деклариране на структура mystruct */
struct mystruct
{
      int  i;
      char c;
};

/* Деклариране на синоним blackbox на типа struct mystruct */
typedef struct mystruct   blackbox;

int main(void)
{
      blackbox b;         /* Дефиниране на променлива от тип blackbox */

      struct mystruct s; /* Дефиниране на променлива от тип struct mystruct */

      /*...*/

      return 0;
}
```

Както показва горния пример променливи от структурен тип могат да се дефинират както чрез използване на синонима на структурата, така и чрез ключовата дума struct и името на структурата. Променливите b и s са от един и същ тип.

Ключовата дума typedef може да се приложи и директно върху декларацията на самата структура.

Деклариране на синоним на структурен тип

```
/* Деклариране на структура и синоним на структурата едновременно  */
typedef struct име-структура[1]
{
      /*...*/
} ново-име-на-тип;
```

Забележка[1]: име-структура може да се изпусне. За дефиниране на структурни променливи в този случай обаче може да се използва само синонима, зададен с ново-име-на-тип.

Пример 10-3:

```
/* Деклариране на структура и синоним на структурата едновременно */
typedef struct mystruct
{
      int  i;
      char c;
}blackbox; /* В този контекст blackbox е име на тип, а не име на структурна
              променлива*/

int main(void)
{
```

```
	blackbox b;

	struct mystruct s;

	/*...*/

	return 0;
}
```

Променливите b и s са от един и същ тип. Обект b използва синонима на структурата, а обект s използва класическия начин на задаване на типа.

1.1.6 Непълна(незавършена) декларация на структура

Ако конструкция

```
struct име-структура;
```

се появи преди декларацията, която описва съдържанието на структурата, то конструкцията се нарича непълна декларация на структурата.

Непълната декларация декларира структурен тип, който може да се използва само когато размерът на обект от този тип не е необходим. Размерът на обект от структурен тип не е необходим в следните ситуации:

- Деклариране на typedef-име (синоним) на структурен тип. Непълната декларация на структурата обаче трябва да бъде завършена преди да се създаде обект от нея.

Пример 10-4:

```
/* Непълен тип на структурата mysruct */
typedef struct mystruct blackbox;

/* Пълна декларация на структурата mystruct */
struct mystruct
{
	int  i;
	char c;
};

int main(void)
{
	/* Дефиниране на обект от тип blackbox */
	blackbox s;

	/*...*/

	return 0;
}
```

- Деклариране на указател от структурен тип.

Пример 10-5:

```
/* Непълен тип на структурата mystruct */
typedef struct mystruct blackbox;

/* Дефиниране на указател */
blackbox* ptr; /* Същото като struct mystruct* ptr */

/* Пълна декларация на структурата mystruct */
struct mystruct
{
	int  i;
	char c;
};

int main(void)
{
	/* Дефиниране на обект от тип blackbox */
	blackbox s;

	ptr = &s;

	/*...*/

	return 0;
 }
```

- Деклариране на функция, връщаща структура. Непълната декларация на структурата обаче трябва да бъде завършена преди функцията да се дефинира.

Пример 10-6:

```
/* Непълен тип на структурата mysruct */
typedef struct mystruct blackbox;

/* Деклариране на функция foo() */
blackbox foo(void);

/* Пълна декларация на структурата mystruct */
struct mystruct
{
	int  i;
	char c;
};

int main(void) {

	/* Дефиниране на обект от тип blackbox */
	blackbox s;

	s = foo();

	/*...*/

	return 0;
}
```

```
/* Дефиниране на функцията foo() */
blackbox foo(void)
{
	blackbox b;

	/*...*/

	return b;
}
```

За да се завърши една непълна декларация на структура е необходимо да се добави нова декларация в същата област на видимост, която декларира съдържанието на структурата. Разгледайте горните три примера отново.

Непълната декларация на структура също така скрива всяка друга декларация на структура със същото име, декларирана в обхващащата област на видимост.

Пример 10-7:

```
/* Деклариране на структура */
struct s1
{
	int  i;
	char ch;
};

/* Входна точка на програмата */
int main(void)
{
	/* Деклариране на непълен тип */
	struct s1; /* Скрива предишната декларация на s1 и декларира нова */

	struct s2  { struct s1 *ps1; /*...*/};
	struct s1  { double d; int y; };

	/*...*/

	return 0;
}
```

Указателят ps1 е указател към обект тип struct s1 с членове d и y. Ако се махне непълната декларация struct s1 от функцията main(), ps1 ще бъде указател към обект тип struct s1 с членове i и ch.

1.1.7 Ограничение на структурите

Структурите имат следните ограничения:

- Член на структура не може да има спецификатор за клас памет (auto, register, static, extern).

Пример:

```
struct mystruct
{
      static int ; /* !!!Грешка, спецификатор за клас памет не може да се използва
                      тук */
      double d;
};
```

- Член на структурата не може да бъде променлива от самата структура (но може да бъде указател към същата структура).

Пример:

```
struct mystruct
{
   int ;
   double d;
   struct mystruct s;   /* !!!Грешка, структура не може да има поле от същата
                           структура */
   struct mystruct *ps; /* Допустимо, структура може да има указател към
                           променлива от същата структура */
};
```

- Член на структурата не може да бъде от непълен тип.

Пример:

```
struct s1
{
   struct s2  object2; /* !!!Грешка, не могат да се дефинират обекти от
                          непълен тип*/
};

struct s2
{
   struct s1  object1; /* Допустимо, типът struct s1 е завършен в тази
                          точка  */
};
```

- Член на структура не може да се инициализира по време на декларацията си в тялото на структурата.

Пример:

```
struct mystruct
{
    int    i = 10;  /* !!!Грешка, поле на структура не може да се
                       инициализира тук */
    double d;
};
```

1.2 Операции със структури

Операциите, които могат да се прилагат към променливи от структурен тип са:

- Операция за достъп до член чрез името на структурната променлива '.'.
- Операция за достъп до член чрез указател към структурната променлива '->'.
- Операция за вземане на адрес '&'.
- Операция за присвояване '='.

1.2.1 Достъп до членове на структура

1.2.1.1 Директен достъп с операцията '.'

Достъпът до членовете на една структурна променлива става с помощта на името на структурната променлива последвано от точка и името на члена.

Директен достъп до член на структурна променлива

```
име-структурна-променлива.име-член
```

Пример 10-8:

```
#include <stdio.h>

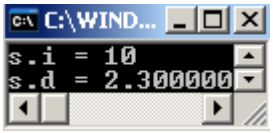
struct mystruct
{
	int i;
	double d;
};

int main(void)
{
	struct mystruct s; /* Дефиниране на структурна променлива s */

	s.i = 10;  /* Дирекетен достъп до i */
	s.d = 2.3; /* Дирекетен достъп до d */

	printf("s.i = %d\n", s.i);
	printf("s.d = %lf\n", s.d);

	return 0;
}
```



1.2.1.2 Косвен достъп с операцията '->'

Косвеният достъп до членовете на структурна променлива става с помощта на указател към структура.

Дефиниране на указател към структура

```
struct име-структура * име-указател;
```

Достъпът до член на структурна променлива чрез указател към нея става с помощта на операцията ->.

Косвен достъп до член на структурна променлива

```
име-указател -> име-член
```

Пример 10-9:

```
#include <stdio.h>

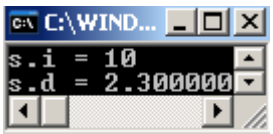
struct mystruct
{
	int i;
	double d;
};

int main(void)
{
	struct mystruct s;   /* Дефиниране на структурна променлива s            */
	struct mystruct *ps; /* Дефиниране на указател към обекти от тип mystruct*/

	ps = &s;     /* Присвояване на адреса на s на указателя ps */
	ps -> i = 10;   /* Косвен достъп до i */
	ps -> d = 2.3;  /* Косвен достъп до d */

	printf("s.i = %d\n",  ps -> i);
	printf("s.d = %lf\n", ps -> d);

	return 0;
}
```



Следните конструкции между структурна променлива и указател към нея са валидни:

```
име-структурна-променлива.име-член == (*име-указател).име-член

име-указател -> име-член == (&име-структурна-променлива) -> име-член
```

Пример:

```
struct mystruct s;   /* Дефиниране на структурна променлива s            */
struct mystruct *ps; /* Дефиниране на указател към обекти от тип mystruct*/

ps = &s;     /* Присвояване на адреса на s на указателя ps */

/* Следните конструкции извършват едни и същи действия, но по различен начин */

s.i = 10;    /* Директен достъп */
s.d = 10;    /* Директен достъп */

ps->i = 10;  /* Косвен достъп */
ps->d = 2.3  /* Косвен достъп */

(*ps).i = 10;  /* Същото като s.i и (&s)->i */
(*ps).d = 2.3; /* Същото като s.d и (&s)->d */

(&s) -> i = 10;  /* Същото като s.i и ps->i */
(&s) -> d = 2.3; /* Същото като s.d и ps->d */
```

1.2.2 Присвояване на структури

Една структурна променлива може да се присвои директно на друга структурна променлива. Единственото условие, което трябва да е изпълнено е двете структурни променливи да са от един и същ тип.

Пример 10-10:

```
#include <stdio.h>

struct mystruct
{
	int i;
	double d;
};

struct anotherstruct
{
	int i;
	double d;
};

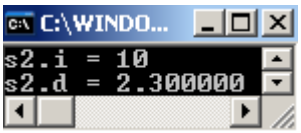
int main(void)
{
	struct mystruct s1, s2;
	struct anotherstruct s3;

	s1.i = 10;
```

```
	s1.d = 2.3;
	s2 = s1; /* s1 се копира в s2 */
	/*s3 = s1*/  /* !!!Грешка s3 и s1 са от от различни типове */

	printf("s2.i = %d\n",  s2.i);
	printf("s2.d = %lf\n", s2.d);

	return 0;
}
```



Както се вижда от примера, дори две структури да са абсолютно идентични, те все пак представляват различни типове.



Дори две структури да са абсолютно идентични, те все пак представляват различни типове.

Както Ви е известно един масив не може да се присвои на друг наведнъж. Това означава, че ако a1 и a2 са два масива от един и същ тип и с еднакъв размер, конструкцията a1 = a2 е невалидна. Това ограничение може да се заобиколи ако масив се капсулира в структура.

Пример 10-11:

```
#include <stdio.h>

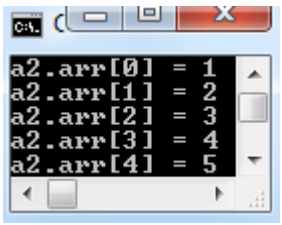
struct mystruct
{
	int arr[5];
};

int main(void)
{
	struct mystruct a1 = {{1,2,3,4,5}};
	struct mystruct a2;
	int i;

	a2 = a1;

	for ( i = 0; i < 5 ; i++){
		printf("a2.arr[%d] = %d\n", i,  a2.arr[i]);
	}

	return 0;
}
```



1.3 Структури и функции

1.3.1 Структури като параметри на функции

Функциите могат да имат параметри от структурен тип. По подразбиране структурните променливи се предават по стойност. Това означава, че всяка промяна на параметъра в тялото на функция не оказва влияние на аргумента.

Структура като параметър на функция

```
тип-резултат име-функция(struct име-структура       име-параметър);
```

Пример 10-12:

```
#include <stdio.h>

/* Деклариране на структура mystruct*/
struct mystruct
{
      int arr[5];
};

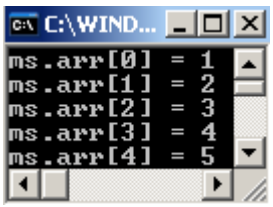
/* Прототип на функцията foo */
void foo(struct mystruct ms);

/* Входна точка на програмата*/
int main(void)
{
      struct mystruct s = {{1,2,3,4,5}};

      foo(s); /* Предаване struct-променливата по стойност */
      return 0;
}

/* Дефиниране на функцията foo */
void foo(struct mystruct ms)
{
      int i;

      for ( i = 0; i < 5 ; i++)
      {
            printf("ms.arr[%d] = %d\n",  i, ms.arr[i]);
      }
}
```



Освен по стойност структурните променливи могат да се предават и по адрес. За целта параметърът на функцията трябва да се декларира като указател към структурата.

Указател към структура като параметър на функция

```
тип-резултат име-функция(struct име-структура       *име-параметър);
```

Пример 10-13:

```
#include <stdio.h>

/* Деклариране на структура mystruct */
struct mystruct
{
      int arr[5];
};

/* Прототип на функцията foo */
void foo(struct mystruct *ps);

/* Входна точка на програмата*/
int main(void)
{
      struct mystruct s;
      int i;

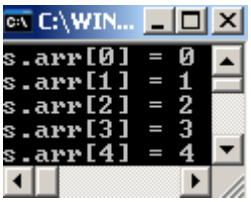
      foo(&s);     /* Предаване на struct-променливата по адрес */

      for ( i = 0; i < 5 ; i++)
      {
            printf("s.arr[%d] = %d\n",  i, s.arr[i]);
      }

      return 0;
}

/* Дефиниране на функцията foo */
void foo(struct mystruct *ps)
{
      int i;

      for ( i = 0; i < 5 ; i++)
      {
            ps->arr[i] = i;
      }
}
```



Предаването по адрес има две преимущества:

- Подобрява се производителността на програмата тъй като се избягва времето за копиране на членовете на аргумента в параметъра. Това е особено осезаемо ако аргументът съдържа голям брой членове или масив с голям брой елементи.

- Избягва се проблема с препълване на стека. Предаването на структурна променлива по стойност означава, че компилаторът трябва да задели място в стека за всеки един член на структурата, в които да се копират членовете на аргумента. Ако структурата съдържа голям брой членове (или масив с много елементи) това лесно може да доведе до препълване на стека.

1.3.2 Структури връщани от функции

Функциите също могат да връщат структури и указатели към структури като резултат.

Връщане на структура от функция

```
struct име-структура      име-функция(списък-параметри);
```

Връщане на указател към структура от функция

```
struct име-структура*     име-функция(списък-параметри);
```

Пример 10-14:

```
#include <stdio.h>

/* Деклариране на структура mystruct*/
struct mystruct
{
	int i;
	double d;
};

/* Прототип на функцията foo */
struct mystruct foo(void);

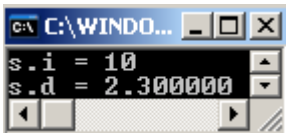
/* Входна точка на програмата*/
int main(void)
{
	struct mystruct s;

	s = foo();
	printf("s.i = %d\n", s.i);
	printf("s.d = %lf\n", s.d);

	return 0;
}

/* Дефиниране на функцията foo */
struct mystruct foo(void)
{
	struct mystruct s = {10, 2.3};

	return s;
}
```



Ако една функция връща указател към структура много е важно да запомните, че обектът, сочен от указателя, трябва да съществува и след като функцията завърши изпълнението си, в противен случай указателят ще сочи към несъществуващ обект и резултатът ще бъде недефиниран. Казано с прости думи една функция не трябва да връща адреса на локална структурна променлива, тъй като този адрес ще бъде невалиден след като функцията завърши изпълнението си. Например следващия пример съдържа логическа грешка.

Пример 10-15:

```
#include <stdio.h>

/* Деклариране на структура mystruct*/
struct mystruct
{
	int i;
	double d;
};

/* Прототип на функцията foo */
struct mystruct* foo(void);

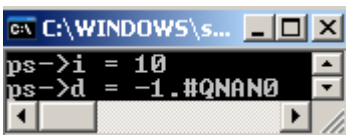
/* Входна точка на програмата*/
int main(void)
{
	struct mystruct* ps;

	ps = foo(); /* !!! foo() връща невалиден адрес */
	printf("ps->i = %d\n", ps->i);
	printf("ps->d = %lf\n", ps->d);

	return 0;
}

/* Дефиниране на функцията foo */
struct mystruct* foo(void)
{
	struct mystruct s = {10, 2.3};

	return &s; /* !!!Грешка, връща се адрес на локална променлива */
}
```



Следващият пример е коригирана версия на горния.

Пример 10-16:

```c
#include <stdio.h>

/* Деклариране на структура mystruct*/
struct mystruct
{
	int i;
	double d;
};

/* Прототип на функцията foo */
struct mystruct* foo(void);

/* Входна точка на програмата*/
int main(void)
{
	struct mystruct* ps;

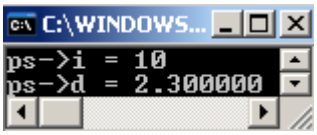
	ps = foo(); /* !!! foo() връща валиден адрес */

	printf("ps->i = %d\n", ps->i);
	printf("ps->d = %lf\n", ps->d);

	return 0;
}

/* Дефиниране на функцията foo */
struct mystruct* foo(void)
{
	static struct mystruct s = {10, 2.3};

	return &s; /* !!!Позволено, връща се адрес на статична локална променлива */
}
```



1.4 Масиви от структури

Структурите също могат да бъдат елементи на масиви.

Масив от структури

```
struct  име-структура   име-масив[размер];
   тип-на-елементите
```

Достъпът до елемент на масива става по стандартния начин:

име-масив[индекс]

Достъпът до конкретен член на елемент на масива става също по стандартния начин:

име-масив[индекс].име-член

име-структурна-променлива

Може също да инициализирате масив от структури по следния начин:

Инициализиране на масив от структури

```
struct  име-структура   име-масив[размер] =
{
      {списък-инициализатори-структура1},
      {списък-инициализатори-структура2},
      …
      {списък-инициализатори-структураN},
};

където

списък-инициализатори-структура1/.../списък-инициализатори-структураN
      Състои се от инициализатор-член1,…, инициализатор-членN
```

Следващият пример демонстрира употребата на масив от структури.

Пример 10-17:

```
#include <stdio.h>

#define CATALOG_SIZE      2

/* Деклариране на структура PersonalInformation */
struct PersonalInformation
{
   char name[40];
   char address[40];
   char profession[20];
   char FamilyStatus[10];
};

/* Дефиниране на масив от структурни-променливи тип PersonalInformation */
struct PersonalInformation  PersonalInformationCatalog[CATALOG_SIZE] =
{
   {"Empty","Empty", "Empty","Empty"}, /* Инициализиране на елемент-структура
                                          PersonalInformationCatalog[0] */
   {"Empty","Empty", "Empty","Empty"}  /* Инициализиране на елемент-структура
                                          PersonalInformationCatalog[1] */
};

/* Прототипи на функции */
void enterInformation(void);
void readCatalog(void);

int main(void)
{
   int choice;

   /* Отпечатване на кратка информация за програмата */
   printf("Personal Catalog\n\n");
```

```
    do
    {
        /* Отпечатване на възможните функции на програмата */
        printf("1 - enter information\n");
        printf("2 - read catalog\n");
        printf("0 - Quit\n\n");
        printf("Choose operation: ");
        scanf("%d", &choice); /* Прочитане на въведения избор */
        getchar(); /* Изчистване на символа за нов ред от входния буфер */
        printf("\n");

        if(choice != 0)
        {
               switch(choice)
               {
               case 1:
                      enterInformation();
                      break;
               case 2:
                      readCatalog();
                      break;
               }
        }
        else
        {
               printf("BYE BYE!\n");
        }

    }while(choice != 0);

    return 0;
}

/* Въвежда информация в каталога */
void enterInformation(void)
{
    int i;

    for(i = 0; i < CATALOG_SIZE ; i++)
    {
        printf("Enter information for Person%d:\n\n", i+1);

        printf("Name: ");
        gets(PersonalInformationCatalog[i].name);
        printf("Address: ");
        gets(PersonalInformationCatalog[i].address);
        printf("Profession: ");
        gets(PersonalInformationCatalog[i].profession);
        printf("Family status: ");
        gets(PersonalInformationCatalog[i].FamilyStatus);

        printf("\n\n");
    }
}

/* Чете информацията от каталога */
void readCatalog(void)
{
    int i;

    for(i = 0; i < CATALOG_SIZE ; i++)
    {
        printf("Person%d:\n", i);
```

```
        printf("Name: %s\n", PersonalInformationCatalog[i].name);
        printf("Address: %s\n", PersonalInformationCatalog[i].address);
        printf("Profession: %s\n", PersonalInformationCatalog[i].profession);
        printf("Family status: %s\n", PersonalInformationCatalog[i].FamilyStatus);
        printf("\n\n");
    }
}
```

1.5 *Вложени структури*

Членовете на една структура могат да бъдат променливи от друга структура.

Пример 10-18:

```
#include <stdio.h>

#define NAME_LENGHT 20

struct  person_name  {
        char first [NAME_LENGHT];
        char middle_initial;
        char last [NAME_LENGHT];
};

struct worker {
        struct person_name name; /* Дефиниране на член от друга структура */
        char               age;
        char               gender;
};

int main(void)
{
  struct worker    w =
  {
        {"Jonh", 'S', "Doe"}, /* Инициализатор за name    */
        30,                   /* Инициализатор за age     */
        'M'                   /* Инициализатор за gender  */
  };

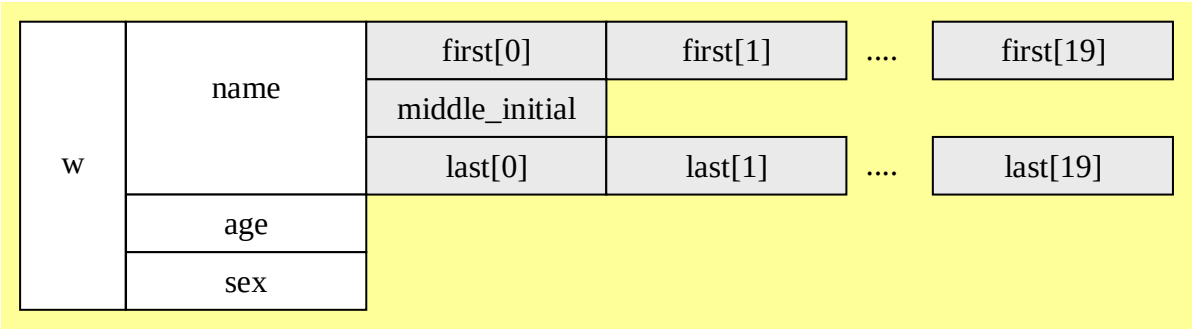
  printf("Name: %s %c. %s\n", w.name.first, w.name.middle_initial, w.name.last);
  printf("Age: %d\n", w.age);

  if(('M' == w.gender) || ('m' == w.gender))
  {
      printf("Gender: male\n");
  }
  else if(('F'== w.gender) || ('f'== w.gender))
  {
      printf("Gender: female\n");
  }
  else
  {
      printf("Gender: wrong gender specified.\n");
  }

  return 0;
}
```



Фиг.73 илюстрира членовете на структурната променлива w.



Фиг. 75 Членове на структурната променлива w

Достъпът до членовете на вложената структурна-променлива става по следния начин:

w.name.first – достъп до масива first
w.name.middle_initial – достъп до члена middle_initial
w.name.last – достъп до масива last

Отделните елементи на масивите first и last се достигат по стандартния начин - чрез индексиране:

Пример: w.name.first[3]

Операцията точка '.' има лява асоциативност, т.е конструкцията w.name.first е еквивалентна на (w.name).first. Операциите индексиране '[]' и точка '.' са с еднакъв приоритет и лява асоциативност.

Не е позволено на структурата да съдържа член от себе си, но може да съдържа указател към променлива от тази структура.

Пример:

```
struct  person_name  {
       char first [NAME_LENGHT];
       char middle_initial;
       char last [NAME_LENGHT];
       struct  person_name     name; /* !!!Непозволена декларация */
       struct  person_name * p_name; /* Позволена декларация      */
};
```

1.6 Разполагане на структурните променливи в паметта

1.6.1 Подравняване (alignment) на базовите типове данни

Единственото изискване, което С стандартът предявавя към членовете на една структура е те да се разполагат в паметта последователно по реда на декларирането им в тялото на структурата. На пръв поглед това може да остави впечатление, че членовете на една структура следват плътно един след друг в паметта. Това обаче не е задължително. Спомнете си, че когато въведохме понятието "променлива" в Глава 2 Типове Данни отбелязохме, че всяка променлива се характеризира и с допълнителен атрибут, наречен подравняване в паметта (**memory alignment**). Подравняването на скаларните типове (char, short, int, указатели и т.н.) не е свързано с езика С, а с конкретната хардуерна платформа, на която се изпълнява кода. Различните хардуерни платформи имат различни изисквания как данните да се разполагат в паметта. Всяка имплементация на С компилатора, трябва да се съобразява с това изискване.

Променливите от скаларен тип трябва да се разполагат в паметта на адреси кратни на модула на подравняване на съответния тип. Като правило подравняването на скаларните типове данни е равно на техния размер определен с sizeof() (но не винаги). Така подравняването на char е 1, на short е 2 и т.н. Една променлива е подравнена ако е изпълнено условието

```
адрес-променлива  %  модул-на-подравняване == 0
```

Типичните стойности на модула на подравняване са: 1, 2, 4, 8 и 16. Модул на подравняване 1 означава всъщност, че променливата може да се разполага на всякакъв адрес.

Следвайки изискванията за подравняване на данните, компилаторът може да вмъкне уплътняващи байтове между отделните членове на една структурна променлива, започвайки след първия член. Допълнително компилаторът може да вмъкне уплътняващи байтове и след последния член на struct-променливата. Поради това размерът на една структура трябва винаги да се определя с оператора sizeof(). Подравняването на променливите в паметта е разгледано Глава 2 Типове Данни.

1.6.2 Подравняване на членове на структура. Основни правила

От предходната точка могат да се направят следните важни изводи:

- Поради голямото разнообразие на микропроцесорните архитектури, не могат да се правят никакви категорични предположения за

подравняването на данните, освен за тип char.

- Разполагането на структурните променливи в паметта е зависимо от конкретната имплементация на компилатора. Дори два различни компилатора, предназначени за една и съща хардуерна платформа могат да се различават в имплементацията си по отношение на подравняването на данните.

- За всяка конкретна имплементация на компилатора разполагането на структурните променливи в паметта обаче е предсказуемо. Всеки компилатор би трябвало да документира модулите на подравняване, които се използват.

- Код, който разчита на конкретно разполагане на членовете на структура в паметта е непреносим.

Следният израз може да се използва, за да се тества дали една скаларна променлива е подравнена.

```
(unsigned long) &име-променлива % модул-на-подравняване == 0
```



C89 стандартът не гарантира, че най-големия целочислен тип може да представи стойността на указател. Размерът в байтове на указателите не е твърдо фиксиран, а зависи от размера на паметта и архитектурата на компютъра. Когато указател се преобразува в целочислен тип, който не може да побере стойността на указателя, резултатът е недефиниран.

Основни правила за разполагане на членовете на структура в паметта:

1. Първият елемент се разполага винаги с отместване 0 от началото на структурата.

2. Изчислява се броя P_I на уплътняващите байтове, които са нобходими да се вмъкнат пред следващия член на структурата.

   $$\mathbf{P_I = ((O_I \ \%\ A_I) == 0)\ ?\ 0 : (A_I - (O_I \ \%\ A_I))};$$

   където

   P_I (Padding) – Брой уплътняващи байтове пред I-тия член
   O_I (Offset) – Текущото отместване, след разполагане на I-1я член
   A_I (Alignment) – Модул на подравняване на I-тия член на структурата

   Правило 2 се прилага рекурсивно докато се разположат всички членове на структурата.

3. След разполагане и на последния член на структурата се изчислява броя на крайните уплътняващите байтове. Те са необходими за да се постигне правилно подравняване на членовете в случай, че се дефинира масив от тази структура и се изчисляват от условието, че размерът на структурата S в байтове, трябва да е кратен на най-големия модул на подравняване A_{MAX} използван в структурата, т.е

   $S \% A_{MAX} == 0$.

   Най-големият модул на подравняване, използван в една структура, се нарича още модул на подравняване на самата структура. Този модул се използва при разполагането на структурна променлива в паметта и при определяне на отместването когато член на структура е променлива от друг структурен тип.

Когато се създава структура, подреждането на членовете по ред на намаляване на модула на подравняване води до минимизиране на уплътняващите байтове.

Нека демонстрираме описаното дотук с няколко примера:

Пример:

```
struct AlignmentDemo
{
                  /* размер      подравняване */
      char c;     /*  1                1     */
      int  i;     /*  2                2     */

};
```

Полето c се разполага на отместване 0.

Изчислява се броя на уплътняващите байтове преди разполагане на полето i.

$P_I = ((O_I \% A_I) == 0) ? 0 : (A_I – (O_I \% A_I))$;

$O_I = 1$ /* Отместване след разполагане на c */
$A_I = 2$ /* Подравняване на полето i */
$P_I = ((1 \% 2) == 0) ? 0 : (2 – (1 \% 2)) = (0) ? 0 : 1 = 1$

След разполагане и на последното поле се изчислява размера на структурата.

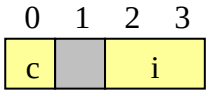
S = 4 байта

Проверява се дали условието **$S \% A_{MAX} == 0$** е изпълнено. В нашия пример

4 % 2 дава нула като резултат, т.е. условието е изпълнено. Това означава, че уплътняващи байтове след последния член не са необходими, а модулът на подравняване на цялата структура е 2.

При посочените условия компилаторът ще добави един уплътняващ байт както е показано по-долу.

```
struct AlignmentDemo
{
      char c;
      char b0;     /* уплътняващ байт */
      int  i;
};
```

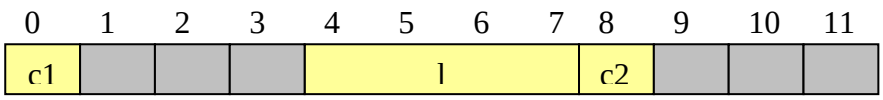


Пример:

```
struct AlignmentDemo
{                    /* размер    подравняване */
      char c1;       /*   1             1      */
      long l;        /*   4             4      */
      char c2;       /*   1             1      */
};
```

При посочените условия компилаторът ще добави уплътняващи байтове както е показано по-долу.

```
struct AlignmentDemo
{
      char c1;
      char b0;     /* уплътняващ байт */
      char b1;     /* уплътняващ байт */
      char b2;     /* уплътняващ байт */
      long l;
      char c1;
      char b3;     /* уплътняващ байт */
      char b4;     /* уплътняващ байт */
      char b5;     /* уплътняващ байт */
};
```



S = 12 байта

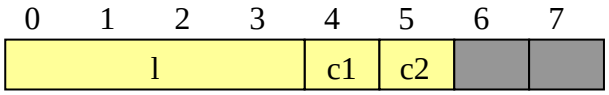
Пример:

```
struct AlignmentDemo
{                    /* размер    подравняване */
      long l;        /*   4             4      */
      char c1;       /*   1             1      */
```

```
        char c2;     /*   1              1     */
};
```

При посочените условия компилаторът ще добави уплътняващи байтове както е показано по-долу.

```
struct AlignmentDemo
{
        long l;
        char c1;
        char c2;
        char b0;     /* уплътняващ байт */
        char b1;     /* уплътняващ байт */
};
```



S = 8 байта

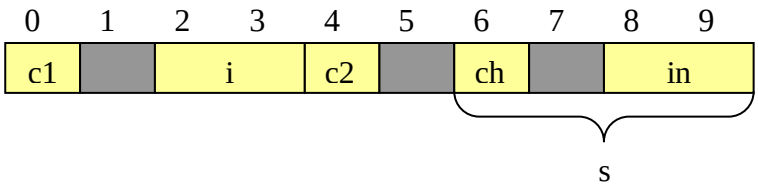
Пример:

```
struct ms
{              /* размер   подравняване */
        char ch; /*    1          1       */
        int  in; /*    2          2       */
};

struct AlignmentDemo
{                    /* размер    подравняване */
        char c1;     /*   1              1     */
        int   i;     /*   2              2     */
        char c2;     /*   1              1     */
        struct ms s; /*   4              2     */
};
```

При посочените условия компилаторът ще добави уплътняващи байтове за структурата AlignmentDemo както е показано по-долу.

```
struct AlignmentDemo
{
        char c1;
        char b0;     /* уплътняващ байт */
        int   i;
        char c2;
        char b1;     /* уплътняващ байт */
        /* struct ms s; */
        char ch;
        char b2;     /* уплътняващ байт */
        int  in;
};
```



S = 10 байта

Следващият пример демонстрира код, който може да се използва за определяне на модула на подравняване на скаларните типове данни. За целта се използва стандартния макрос **offsetof** дефиниран в хедър-файла <stddef.h>.

```
#include <stddef.h>

size_t    offsetof(структурен-тип , име-член);

Макросът offsetoff() връща като резултат отместването в байтове на полето, указано
с име-член, принадлежащо на структурата, указана със структурен-тип.
```

Пример 10-17:

```
#include <stddef.h>

#define short_alignment          (alignment[0])
#define int_alignment            (alignment[1])
#define long_alignment           (alignment[2])
#define float_alignment          (alignment[3])
#define double_alignment         (alignment[4])
#define ldouble_alignment        (alignment[5])
#define pointer_alignment        (alignment[6])

struct Short_
{
      char c;
      short sh;
};

struct Int_
{
      char c;
      int  i;
};

struct Long_
{
      char  c;
      long  l;
};

struct Float_
{
      char  c;
      float f;
};

struct Double_
{
      char  c;
```

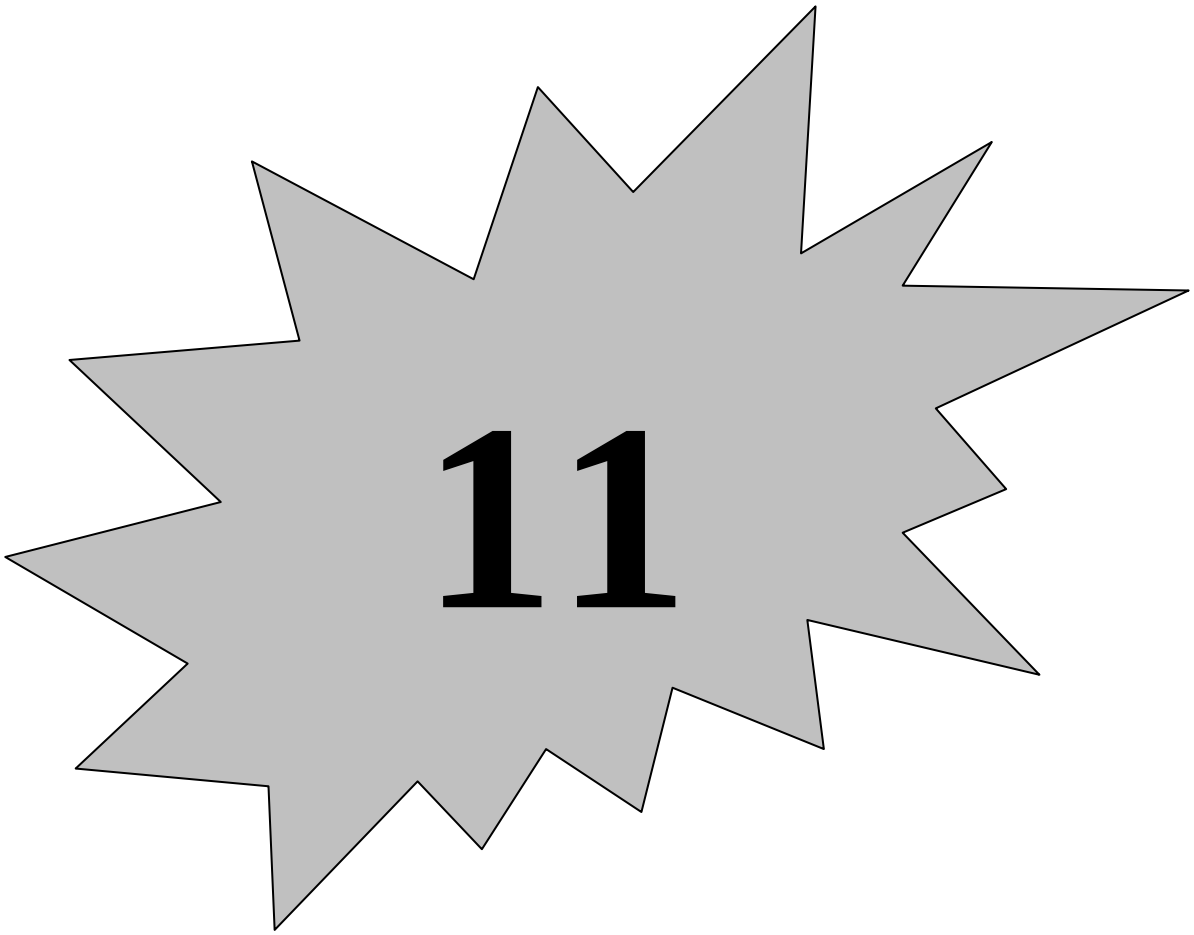
```
        double d;
};

struct LongDouble_
{
        char  c;
        long double ld;
};

struct Ptr_
{
        char  c;
        char* ptr;
};

size_t alignment[7];

int main(void)
{
    short_alignment   = offsetof (struct Short_, sh);
    int_alignment     = offsetof (struct Int_, i);
    long_alignment    = offsetof (struct Long_, l);
    float_alignment   = offsetof (struct Float_, f);
    double_alignment  = offsetof (struct Double_, d);
    ldouble_alignment = offsetof (struct LongDouble_, ld);
    pointer_alignment = offsetof (struct Ptr_, ptr);
    return 0;
}
```



Обединения

1.1 Въведение в обединенията

1.1.1 Деклариране на обединение

Обединенията много приличат външно на структурите. Разликата, е че всички членове на обединението използват обща памет, т.е всички членове се разполагат на един и същ адрес. Казано с други думи обединението е средство, чрез което една и съща област от паметта може да се използва от различни типове променливи. Това подразбира, че в даден период от време се използва само една от променливите, но както ще видим по-нататък е възможно прилагането на различни техники като например паметта се записва, използвайки един тип променлива, а се чете чрез друг тип променлива.

Деклариране на обединение

```
union име-обединение                    } union-тип
{                                       }
      тип1   име-променлива1;           }
                                        }
      тип2   име-променлива2;           } Тяло на обединение
      ...                               }
      ...                               }
      типN   име-променливаN;           }
};

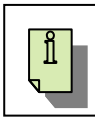
където

тип1/тип2/.../типN
      Може да бъде скаларен тип, агрегатен тип, обединение и битово поле.
```

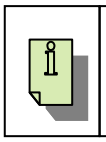
Декларацията на обединението се въвежда с ключовата дума union, последвана от името на обединението и тялото на обединението. Забележете, че декларацията на обединението завършва с точка и запетая. Името на обединението се нарича още етикет на обединението (**tag**). Променливите, декларирани в тялото на обединението, се наричат членове или полета. Декларирането на обединение не води до заделяне на памет за нейните членове. Декларацията представлява просто шаблон, описващ нов тип данни.

Пример:

```
/* Декларация на обединение */
union myunion
{
      int i;
      double d;
};
```



Декларацията на обединение завършва с точка и запетая.



Декларацията на обединение създава нов тип данни. Компилаторът заделя памет, достатъчна да побере най-големия член на обединението, само когато се дефинира променлива от това обединение.

Всяко обединение дефинира нова област на видимост (именовано пространство). Идентификаторите на членовете на обединението трябва да се различават един от друг, но могат да съвпадат с всеки друг идентификатор на променлива, функция, член на друго обединение, структура и т.н. без това да създава конфликт. Идентификатор на поле на едно обединение може да съвпада дори с името на обединението и променлива от този тип.

Пример 11-1:

```
union myunion
{
      int i;          /* Съвпада с име на член на  друго обединение */
      double d;       /* Съвпада с име на член на  друго обединение */
      char myunion;   /* Съвпада с име на:
                           -член на  друго обединение,
                           -същото обединение,
                           -union-променлива от това обединение,
                           -локална променлива */
} myunion;

union anotherunion
{
      int i;
      double d;
      char myunion;
      char anotherunion; /* Съвпада с име на функция */
};

void anotherunion(void)
{
      /*...*/
}

int main( void )
{
      int myunion;

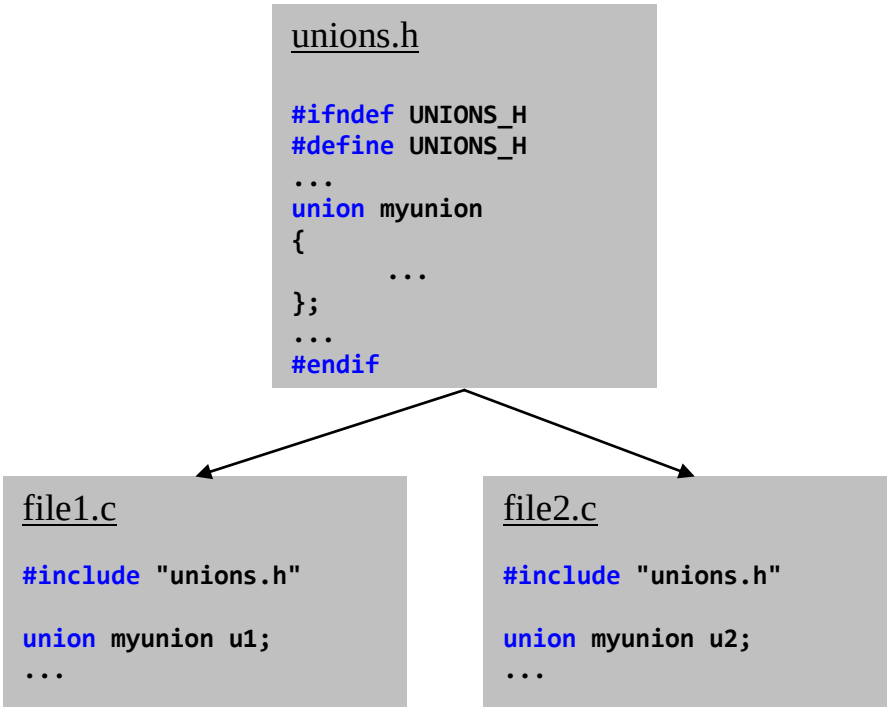
      /*...*/

      return 0;
}
```

1.1.2 Област на видимост на обединение

Декларацията на едно обединение е видима от точката на декларация до края на файла и блока, в който се намира декларацията. Всеки файл, който използва обединението трябва да включва и неговата декларация. В такива случай е най-добре обединението да се декларира в хедър-файл и този

файл да се включва във всеки сорс-файл, който го използва както е показано по-долу.



1.1.3 Дефиниране на union-променливи

Веднъж декларирано, обединението може да се използва за дефиниране на променливи. Такива променливи ще наричаме union-променливи. Комбинацията от ключовата дума union и името на обединението определя неговия тип. Именно тази комбинация се използва при дефиниране на union-променливи.

Дефиниране на union-променливи

```
union име-обединение  име-union-променлива;
```

union-тип

Когато компилаторът срещне подобна конструкция, той заделя памет достатъчна да побере най-големия член на обединението. Всички членове на обединението се разполагат от началото на обединението.

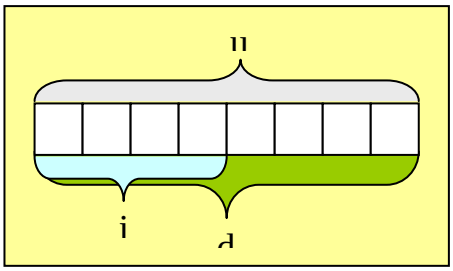
Пример:

```
/* Деклариране на обединение */
union myunion
{
      int i;
      double d;
```

```
};

/* Дефиниране на union-променлива */
union myunion      u;
```

Горната конструкция води до заделяне на памет за променливата u, достатъчно голяма да побере тип double (Фиг.73). В примера се допуска, че тип int заема 4 байта, а double 8.



Фиг. 76 Членове на union-променливата u

Както се вижда от Фиг.73 променливите i и d започват от началото на обединението, т.е. те се припокриват.

Union-променливи могат да се дефинират и по време на деклариране на самото обединение.

Деклариране на обединение и union-променливи едновременно

```
union име-обединение[1]
{
      тип1   име-променлива1;

      тип2   име-променлива2;
      ...
      ...
      типN   име-променливаN;
}списък-union-променливи;

където

списък-union-променливи
      име-union-променлива1,...,име-union-променливаN
```

Забележка[1]: Ако няма да дефинирате други union-променливи, освен тези указани в **списък-union-променливи**, **име-обединение** може да се изпусне.

Пример:

```
/* Дефиниране union-променливи по време на деклариране на обединение */
union myunion
{
      int i;
      double d;
} u1, u2;
```

...

```
/* Дефиниране на допълнителна union-променлива по-късно в кода */
union myunion      u3;
```

Променливите u1 и u2 са union-променливи дефинирани по време на деклариране на самата структура. Променливата u3 се дефинира по-късно в кода по стандартния начин.

Когато union-променливите се дефинират по време на деклариране на обединението, името на обединението може да се изпусне.

Пример:

```
/* Деклариране на обединение без име */
union
{
      int i;
      double d;
} s1, s2;
```

Изпускането на името на обединение обаче означава, че повече не може да дефинираме други променливи от този тип в програмата.

1.1.4 Инициализация на union-променливи

Подобно на структурите, union-променливите могат да се инициализират с помощта на списък с инициализатори. Разликата е, че този списък се състои само от един инициализатор. Друга важна особеност, е че може да се инициализира само първия член на обединението. Инициализаторът може да бъде само константен литерален израз.

Инициализиране на на union-променливи

```
union име-обединение      име-union-променлива = {списък-инициализатори};
```

където

списък-инициализатори
инициализатор[1]

Забележка[1]: Инициализаторът трябва да е литерален константен израз, чийто тип съвпада или е съвместим с типа на първия член на обединението.

Пример:

```
/* Инициализиране на union-променливи */
```

```
union
{
	int i;
	double d;
} u1 = {10}, u2 = {30.4};
```

	i	d
u1	10	-
u2	30	-

```
union
{
	double d;
	int i;
} u3 = {2.5}, u4 = {5};
```

	d	i
u3	2.5	-
u4	5.0	-

```
union
{
	int i;
	char* p;
} u5 = {"Hello C"}; /* !!!Грешка, компилаторът не може да преобразува тип char*
                       в тип int */
```

1.1.5 Обединения и typedef

Както обяснихме в началото, типът на обединението се състои от ключовата дума union и името на обединението, т.е. **union име-обединение**. Използвайки ключовата дума typedef, може да създавате синоним на union-тип, който се състои просто от един идентификатор.

Деклариране на синоним на union тип

```
/* Декларация на обединение */
union име-обединение
{
	/*...*/
};

/* Деклариране на синоним на union-тип */
typedef  union име-обединение   ново-име-на-тип;
```

Пример 11-2:

```
/* Деклариране на обединение myunion */
union myunion
{
	int  i;
	double d;
};

/* Деклариране на синоним blackbox на типа union myunion */
typedef union myunion     blackbox;

int main(void)
{
	blackbox b;     /* Дефиниране на променлива от тип blackbox */

	union myunion u; /* Дефиниране на променлива от тип union myunion */
```

```
	/*...*/

	return 0;
}
```

Както показва горния пример union-променливи могат да се дефинират както чрез използване на синонима на обединението, така и чрез ключовата дума union и името на обединението. Променливите b и u са от един и същ тип.

Ключовата дума typedef може да се приложи и директно върху декларацията на обединението.

Деклариране на синоним на union-тип

```
/* Декларация на обединение */
typedef union име-обединение[1]
{
	/*...*/
} име-union-тип;
```

Забележка[1]: име-обединение може да се изпусне. За дефиниране на union-променливи в този случай обаче може да се използва само псевдонима, зададен с **име-union-тип.**

Пример 11-3:

```
/* Деклариране на обединенние и синоним на обединението едновременно */
typedef union myunion
{
	int i;
	doubel d;
}blackbox; /* В този контекст blackbox е име на тип, а не на union-
		променлива*/

int main(void)
{
	blackbox b;

	union myunion u;

	/*...*/

	return 0;
}
```

Променливите b и u са от един и същ тип. Обектът b използва синонима на обединението, а обекта u използва класическия начин на задаване на типа.

1.1.6 Непълна (незавършена) декларация на обединение

Ако конструкция

```
union име-обединение;
```

се появи преди декларацията, която описва съдържанието на обединението име-обединие, то конструкцията се нарича непълна декларация на обединението.

Непълната декларация декларира union-тип, който може да се използва само когато размера на обект от този тип не е необходим. Размерът на обект от union-тип не е необходим в следните ситуации:

- Деклариране на typedef име (синоним) на union-тип. Непълната декларация на обединението обаче трябва да бъде завършена преди да се създаде обект от него.

Пример 11-4:

```
/* Непълен тип на обединение myunion*/
typedef union myunion blackbox;

/* Пълна декларация на обединението myunion */
union myunion
{
   int    i;
   double d;
};

int main(void)
{
   /* Дефиниране на обект от тип blackbox */
    blackbox u;

   /*...*/

   return 0;
}
```

- Деклариране на указател от union-тип.

Пример 11-5:

```
/* Дефиниране на указател  към обект от тип union myunion */
union myunion* ptr;

/* Пълна декларация на обединението myunion */
union myunion
{
   int    i;
   double d;
};

int main(void)
{

   /* Дефиниране на обект от тип union myunion */
    union myunion u;
```

```
    ptr = &u;

   /*...*/

   return 0;
}
```

- Деклариране на функция, връщаща обединение. Непълната декларация на обединението обаче трябва да бъде завършена преди функцията да се дефинира.

Пример 11-6:

```
/* Деклариране на функция foo() */
union myunion foo(void);

/* Пълна декларация на обединението myunion */
union myunion
{
   int    i;
   double d;
};

int main(void)
{
    /* Дефиниране на обект от тип union myunion */
    union myunion u;

    u = foo();

    /*...*/

    return 0;
}

/* Дефиниране на функцията foo() */
union myunion foo(void)
{
   union myunion b;

   /*...*/

   return b;
}
```

За да се завърши една непълна декларация на обединение е необходимо да се добави нова декларация в същата област на видимост, която декларира съдържанието на обединението. Разгледайте горните три примера отново.

Непълната декларация на обединение също така скрива всяка друга декларация на обединение със същото име, декларирана в обхващащата област на видимост.

Пример 11-7:

```
/* Деклариране на обединение */
union u1
```

```
{
	int  i;
	char ch;
};

/* Входна точка на програмата */
int main(void)
{
	/* Деклариране на непълнен тип */
	union u1; /* Скрива предишната декларация на u1 и декларира нова */

	union u2  { union u1 *pu1; /*...*/};
	union u1  { double d; int y; };

	/*...*/

	return 0;
}
```

Указателят pu1 е указател към обект тип union u1 с членове d и y. Ако се махне непълната декларация union u1 от функцията main(), pu1 ще бъде указател към обект union u1 с членове i и ch.

1.1.7 Ограничение на обединенията

Обединениеята имат следните ограничения:

- Членовете на обединенеията не могат да имат спецификатор за клас памет (auto, register, static, extern).

  Пример:

  ```
  union myunion
  {
  	static int i; /* !!!Грешка, спецификатор за клас памет не може да се
  	                  използва тук */
  	double d;
  };
  ```

- Член на обединение не може да бъде променлива от самото обединение (но може да бъде указател към същото обединение).

  Пример:

  ```
  union myunion
  {
  	int    i;
  	double d;
  	/*union myunion u;*/ /* !!!Грешка, обединение не може да има поле от
  	                          същото обединение */
  	union myunion *pu;   /* Допустимо, обединение може да има указател
  	                          към променлива от същото обединение */
  };
  ```

- Член на обединение не може да бъде от непълен тип.

  Пример:

```
union u1
{
      union u2  object2; /* !!!Грешка, не могат да се дефинират обекти от
                            непълен тип*/
};

union u2
{
      union u1  object1; /* !!!Допустимо, типът union u1 е завършен в тази
                            точка  */
};
```

- Член на обединение не може да се инициализира по време на декларацията си в тялото на обединението.

  Пример:

```
union myunion
{
      int    i = 10;  /* !!!Грешка, поле на обединение не може да се
                         инициализира тук */
      double d;
};
```

1.2 Операции с обединения

Операциите, които могат да се прилагат към union-променливи са:

- Операция за достъп до член чрез името на union-променлива '.'.
- Операция за достъп до член чрез указател към union-променлива '->'.
- Операция за взимане на адрес '&'.
- Операция за присвояване '='.

1.2.1 Достъп до членове на обединение

1.2.1.1 Директен достъп с операцията '.'

Достъпът до членовете на една union-променлива става с помощта на името на union-променливата, последвано от точка и името на члена.

Директен достъп до член на union-променлива

```
име-union-променлива.име-член
```

Пример 11-8:

```
#include <stdio.h>

union myunion
{
	int i;
	double d;
};

int main(void)
{
	union myunion u;

	u.i = 10;
	printf("u.i = %d\n", u.i);

	u.d = 2.3;
	printf("u.d = %lf\n", u.d);

	return 0;
}
```



1.2.1.2 Косвен достъп с операцията '->'

Косвеният достъп до членовете на union-променлива става с помощта на указател към обединение.

Дефиниране на указател към обединение

```
union име-обединение * име-указател;
```

Достъпът до член на union-променлива чрез указател към нея става с помощта на операцията ->.

Косвен достъп до член на union-променлива

```
име-указател -> име-член
```

Пример 11-9:

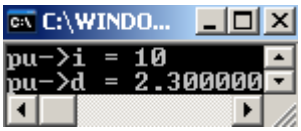
```
#include <stdio.h>

union myunion
{
	int i;
	double d;
};

int main(void)
{
	union myunion u, *pu;

	pu = &u;
	pu->i = 10;
	printf("pu->i = %d\n", pu->i);
	pu->d = 2.3;
	printf("pu->d = %lf\n", pu->d);

	return 0;
}
```



Следните конструкции между union-променлива и указател към нея са валидни:

```
име-union-променлива.име-член == (*име-указател).име-член

име-указател -> име-член == (&име-union-променлива) -> име-член
```

Пример:

```
union myunion u;   /* Дефиниране на union променлива u                      */
union myunion *pu; /* Дефиниране на указател към обекти от тип union myunion */

pu = &u;     /* Присвояване на адреса на u на указателя pu */

/* Следните конструкции извършват едни и същи действия, но по различен начин */

u.i = 10;    /* Директен достъп */
u.d = 10;    /* Директен достъп */

pu->i = 10;  /* Косвен достъп */
pu->d = 2.3  /* Косвен достъп */

(*pu).i = 10;  /* Същото като u.i и (&u)->i */
(*pu).d = 2.3; /* Същото като u.d и (&u)->d */

(&u) -> i = 10;  /* Същото като u.i и pu->i */
(&u) -> d = 2.3; /* Същото като u.d и pu->d */
```

1.2.2 Присвояване на обединения

Една union-променлива може да се присвои директно на друга union променлива. Единственото условие, което трябва да е изпълнено е двете union-променливи да са от един и същ тип.

Пример 11-10:

```
#include <stdio.h>

union myunion
{
	int i;
	double d;
};

union anotherunion
{
	int i;
	double d;
};

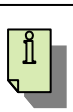
int main(void)
{
	union myunion u1, u2;
	union anotherunion u3;

	u1.i = 10;
	u2 = u1; /* u1 се копира в u2 */
	/*u3 = u1*/  /* !!!Грешка u3 и u1 са от от различни типове */
	printf("u2.i = %d\n",  u2.i);

	return 0;
}
```



Както се вижда от примера, дори две обединения да са абсолютно идентични, те все пак представляват различни типове.



Дори две обединения да са абсолютно идентични, те все пак представляват различни типове.

Както ви е известно един масив не може да се присвои на друг наведнъж. Това означава, че ако a1 и a2 са два масива от един и същ тип и с еднакъв размер, конструкцията a1 = a2 е невалидна. Това ограничение може да се заобиколи ако масив се капсулира в обединение.

Пример 11-11:

```
#include <stdio.h>
```

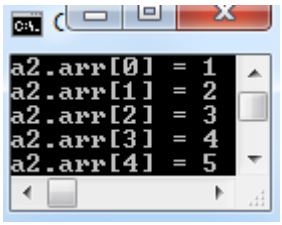
```
union myunion { int arr[5]; };

int main(void)
{
	union myunion a1 = {{1,2,3,4,5}};
	union myunion a2;
	int i;

	a2 = a1;

	for ( i = 0; i < 5 ; i++)
	{
		printf("a2.arr[%d] = %d\n", i, a2.arr[i]);
	}

	return 0;
}
```

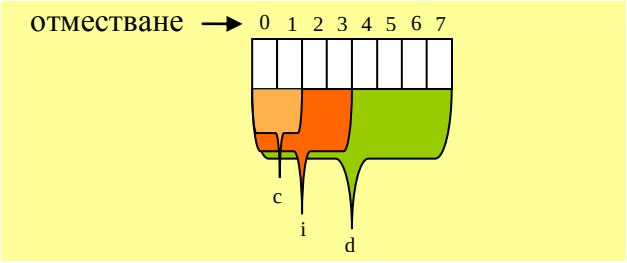


1.3 Разполагане на обединенията в паметта

Както вече споменахме всички членове на едно обединение се разполагат от началото на обединението, т.е имат отместване 0. Компилаторът ще задели толкова памет, колкото е необходимо, за да се побере за най-големия член на обединението.

Пример:

```
union U
{                   /* Размер    Подравняване */
	double d;   /*   8              8      */
	char c[2];  /*   2              1      */
	int i;      /*   4              4      */
};
```



Фиг. 77 Отместване на членовете на обединение

Както се вижда от Фиг.74 независимо от типа на членовете, всички те се разполагат от началото на обединението, т.е имат един и същ адрес в

паметта.

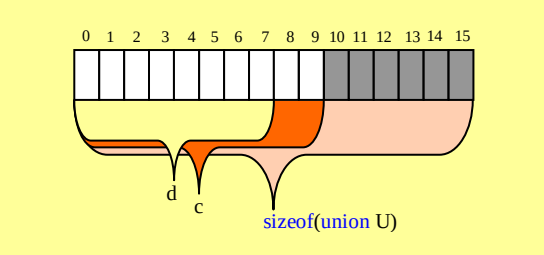
Допълнително компилаторът може да вмъкне уплътняващи байтове в края на обединението. Това се прави с цел ако се дефинира масив от този тип, членовете на отделните union-елементи на масива да бъдат подравнени правилно. Ето защо размерът на едно обединение трябва да се определя винаги с оператора sizeof. Броят на уплътняващите байтове се определя от правилото, че размерът на обединението в байтове трябва да е кратен на най-големия модул на подравняване, използван от членовете на обединението.

Пример:

```
union U
{                   /* Размер    Подравняване */
      double d;     /*   8             8      */
      char c[10];   /*   10            1      */
};
```

При дадените условия компилаторът ще добави уплътняващи байтове както е показано по-долу.

```
union U
{
      double d;
      char c[10];
      char b0;      /* уплътняващ байт   */
      char b2;      /* уплътняващ байт   */
      char b3;      /* уплътняващ байт   */
      char b4;      /* уплътняващ байт   */
      char b5;      /* уплътняващ байт   */
      char b6;      /* уплътняващ байт   */
};
```



Фиг. 78 Разполагане на членове на обединение в паметта

1.4 Използване на обединения

Обединението може да се използва за определяне на формата на съхранение на многобайтовите целочислени променливи: **little-endian** или **big-endian**.

Пример 11-12:

```
#include <stdio.h>

union U
{
	unsigned int i;
	unsigned char c[sizeof(int)];
};

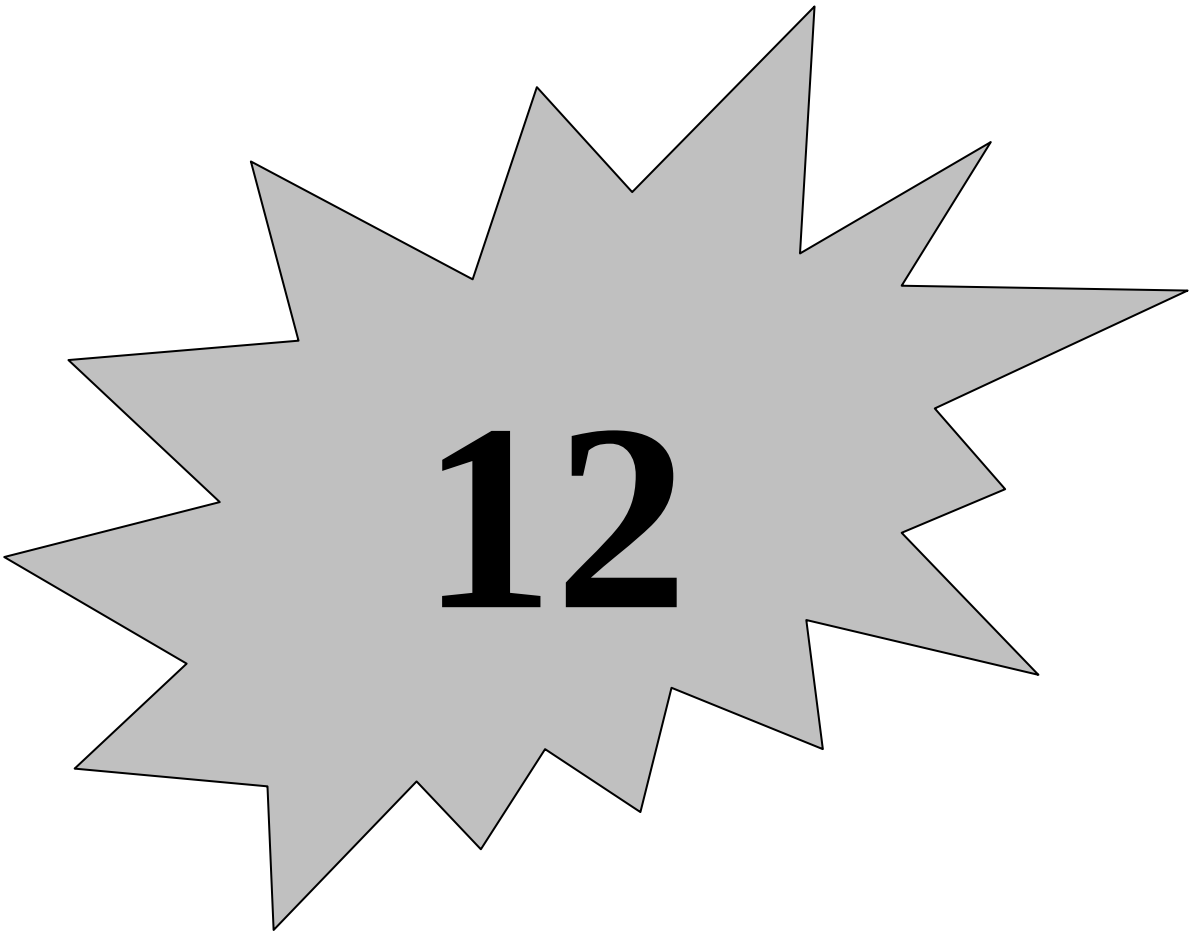
int main( void )
{
	union U endianess = {0x1234};

	if (0x34 == endianess.c[0])
	{
		printf("The computer uses little endian.\n ");
	}
	else
	{
		printf("The computer uses big endian.\n ");
	}

	return 0;
}
```



Различните компютри ще дадат различни резултати.



Битови полета

1.1 Дефиниране на битово поле

Битовото поле е целочислена променлива, която се състои от определен брой битове. Едно битово поле може да бъде само член на структура или обединение. Извън този контекст дефинирането на битово поле не е позволено.

Дефиниране на битово поле

```
union-или-struct   име-тип
{
      тип   име-битово-поле: широчина;
};
```

където

тип
Може да бъде int[1], signed int или unsigned int

Забележка[1]: В контекстта на битовите полета тип int може да бъде signed или unsigned в зависимост от компилатора.

широчина
Определя броя на битовете в битовото поле. Може да бъде само литерален неотрицателен целочислен израз, чиято стойност не трябва да превишава броя на битовете на специфицирания тип, т.е от 0 до броят на битовете в тип int.

Едно n-битово поле може да представи 2^n различни стойности, от 0 до 2^n-1 ако е тип unsigned, и -2^{n-1} до 2^{n-1}-1 ако е от тип signed и отрицателните числа се представят в код с допълнение до 2 (**2's complement**). Забележете, че знаково битово поле с дължина 1 бит в код с допълнение до 2 може да има само две стойности: 0 и -1.

Пример:

```
struct BitFieldsDemo
{
      unsigned b0 : 3; /* стойности от 0 до 8 */
      signed   b1 : 3; /* стойности от -4 до 3 ако се използва 2's complement */
      signed   b2 : 1  /* стойности  0 и -1 ако се използва 2's complement */
};
```

За достъп до битовите полета се използват същите конструкции както и за достъп до обичайните членове на структура или обединение:

```
име-структурна-променлива . име-битово-поле

име-указател-към-структура -> име-битово-поле

(&име-структурна-променлива) -> име-битово-поле

(*име-указател-към-структура) . име-битово-поле
```

Пример 12-1:

```
/* Деклариране на структура с битови полета */
typedef struct BitFieldsDemo
{
	unsigned b0 : 3;
	unsigned b1 : 8;
	unsigned b2 : 5;
	unsigned b3 : 16;
}blackbox;

int main(void)
{
	blackbox b, *p;

	p = &b;

	b.b0	= 1;
	p->b1	= 4;
	(&b)->b2 = 10;
	(*p).b3  = 10;

	/*...*/

	return 0;
}
```

Една структура може да съдържа битови полета и обичайни членове едновременно.

Пример:

```
struct StructDemo
{
	int i;
	double d;
	unsigned b0 : 3;
	unsigned b1 : 8;
};
```

В такъв случай е препоръчително битовите полета да се декларират след обичайните членове.

Битовите полета могат да се използват навсякъде където целочислени променливи могат да се използват. Когато битово поле е част от израз (с изключение като ляв операнд на операцията присвояване), то се преобразува в тип int или unsigned int съгласно обичайните унарни преобразувания (вижте Глава 5 Преобразувания).



Битовите полета могат да се използват навсякъде където целочислени променливи могат да се използват. Когато битово поле е част от израз (с изключение като ляв операнд на операцията присвояване), то се преобразува в тип int или unsigned int съгласно правилата за целочислено повишаване на типовете.

1.2 Битови полета със специално предназначение

С стандартът позволява също така и неименовани (безименни) битови полета. Неименованите битови полета с широчина различна от 0 играят ролята на уплътняващи битове и не могат да се достъпват от кода.

Пример:

```
struct S
{
      unsigned a : 4;
      unsigned   : 2; /* уплътняващи битове */
      unsigned b : 6;
};
```

Неименовано битово поле с дължина 0 бита принуждава компилатора да спре пакетирането на битовите полета в текущия сегмент за съхранение (**storage unit**), в който е пакетирал предходните битови полета и да започне пакетирането от началотото на следващия сегмент. Разполагането (пакетирането) на битовите полета в паметта е разгледано в отделна точка (вижте точка 5 Разполагане на битовите полета в паметта)

Пример:

```
struct S
{
      unsigned a : 4;
      unsigned   : 0;
      unsigned b : 6;
};
```

1.3 Инициализиране на битови полета

Битовите полета могат да се инициализират по същия начин както и обичайните полета на структурата/обединение.

Пример 12-2:

```
struct S
{
      unsigned a : 4;
      unsigned b : 6;
};

int main(void)
{
      struct S  bf = {3, 10}; /* bf.a = 3, bf.b = 10 */

      /*...*/

      return 0;
```

```
}
```

Ако структурата/изброяването съдържа неименовани битови полета, инициализитори за тях не се изискват. Ако се опитате да добавите инициализатор за неименовано битове поле, компилаторът ще генерира грешка.

Пример:

```
union bitfileids          struct bitfileids
{                         {
  int   :2;                  int   :2;
  int b1:2;                  int b1:2;
  int b2:3;                  int b2:3;
}u = {1}; /* b1 = 1 */    }s = {1, 2}; /* b1 = 1, b2 = 2 */
```

Пример 12-3:

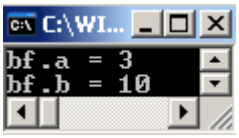
```
#include <stdio.h>

struct bitfields
{
      unsigned a : 4;
      unsigned   : 4;
      unsigned b : 6;
};

int main(void)
{
      struct bitfields  bf = {3, 10}; /* bf.a = 3, bf.b = 10 */

      printf("bf.a = %d\n", bf.a);
      printf("bf.b = %d\n", bf.b);

      return 0;
}
```



1.4 Ограничения на битовите полета

Битовите полета имат следните ограничения:

- Битово поле не може да съществува самостоятелно извън тялото на структура или обединение.

Пример 12-4:

```
int main(void)
```

```
{
      int b:3;   /* !!!Грешка, битово поле може да е само член на структура
                    или обединение */
      return 0;
}
```

- Не може да се извлича адреса на битово поле

Пример 12-4:

```
struct S
{
      int a : 3;
      int b : 5;
}bf;

int main(void)
{
    int *p;

    p = &bf.a; /* !!!Грешка, не може да се извлича адреса на битово поле */

    return 0;
}
```

- Не може да се дефинира указател към битово поле

Пример:

```
int:5  *p; /* !!!Грешка, това е невалидна конструкция в C */
```

- Не може да се дефинира масив от битови полета

Пример:

```
int:5  arr[10]; /* !!!Грешка, това е невалидна конструкция в C */
```

1.5 Разполагане на битовите полета в паметта

Битовите полета се пакетират в т.нар сегмент за съхранение (**storage unit**). Стандартът C дава голяма свобода на компилаторите, по отношение на рамера на тези сегменти и начина на пакетиране на битовите полета в тях.

C89 дефинира, че една имплементация може да задели всеки адресируем сегмент за съхранение достатъчно голям да съдържа битовото поле. Ако след разполагане на битовото поле, в сегмента за съхранение остава място, в което може да се добави следващото битово поле, то се пакетира плътно след предходното. Ако оставащото място от сегмента за съхранение е недостатъчно, дали битовото поле ще се пакетира в следващия сегмент, или ще се пакетира в същия и ще продължи в следващия сегмент за

съхранение, зависи от имплементацията. Посоката на пакетиране на битовите полета също не е дефинирана. Една имплементация може да пакетира битовите полета започвайки от младшия бит на сегмента (отдясно-наляво), докато друга може да ги пакетира, започвайки от старшия бит (отляво-надясно). Подравняването на сегментите за съхранение също не е специфицирано от стандарта.

От горното описание се вижда, че големината на т.нар. сегмент за съхранение не е дефинирана от стандарта. Няма изискване към имплементациите да заделят най-малкият адресируем сегмент за съхранение (байт). Това означава, че големината на сегмента за съхранение може да бъде 8, 16, 32 и т.н бита. Най-често тя е равна на големината на машинната дума на компютъра, която е с размера на тип int (8-битовите микропроцесори са изключение от това правило).

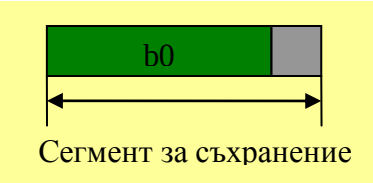
Ще изясним описаното по-горе с няколко хипотетични сценария. Нека приемем, че един сегмент се състои от **m** бита и съвпада с големината на машинната дума на процесора (т.е може да е 8, 16, 32 и т.н. бита) и битовите полета се съхраняват отляво-надясно.

Сценарии 1

int b0 : n

където n < m

Фиг.76 показва пакетирането на битовото поле b0.



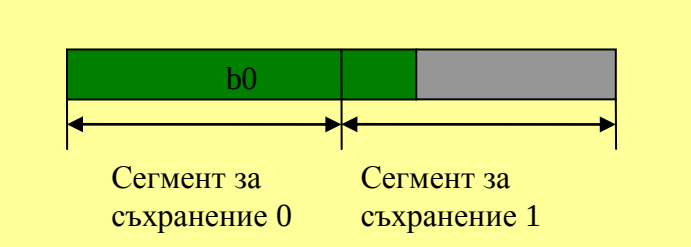
Фиг. 79 n < m

Сценарии 2

int b0 : n

където n > m

Фиг.77 показва пакетирането на битовото поле b0.



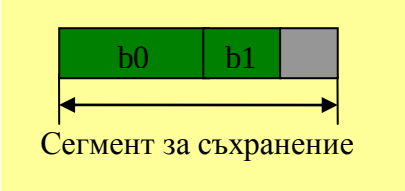
Фиг. 80 n > m

Сценарии 3

int b0 : n0
int b1 : n1

където n0 + n1 < m

Фиг.78 показва пакетирането на битовите полета b0 и b1.



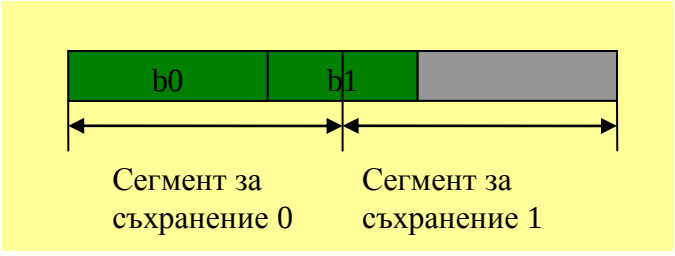
Фиг. 81 n0 + n1 < m

Сценарии 4

int b0 : n0
int b1 : n1

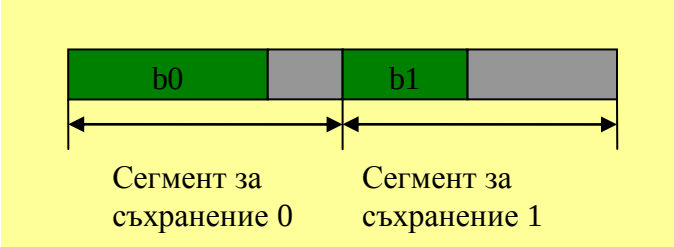
където n0 + n1 > m

Фиг.79 показва пакетирането на битовите полета b0 и b1 за имплементации, които позволяват битово поле да пресича границата на сегмент за съхранение.



Фиг. 82 n0 + n1 > m вариант 1

Фиг.80 показва пакетирането на битовите полета b0 и b1 за имплементации, които не позволяват битово поле да пресича границата на сегмент за съхранение.



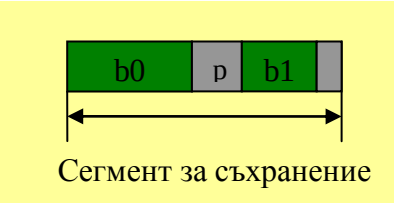
Фиг. 83 n0 + n1 > m вариант 2

Сценарии 5

```
int b0 : n0
int    : p
int b1 : n1
```

където p ≠ 0 и n0 + p + n1 < m

Фиг.81 показва пакетирането на битовите полета b0 и b1



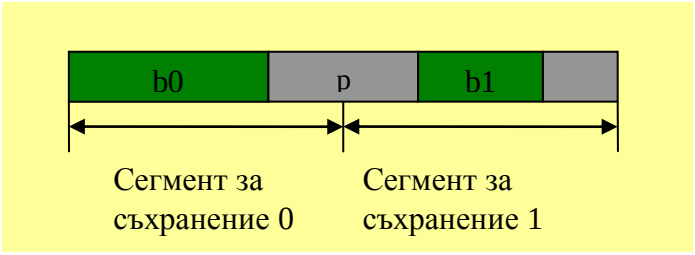
Фиг. 84 p ≠ 0 и n0 + p + n1 < m

Сценарии 6

```
int b0 : n0
int    : p
int b1 : n1
```

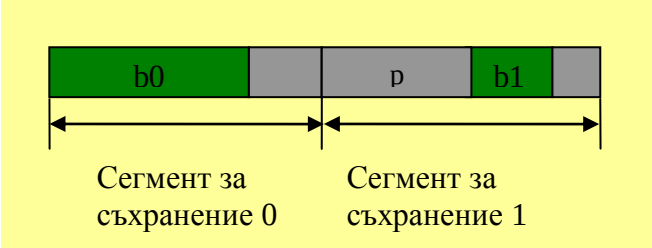
където p ≠ 0 и n0 + p > m

Фиг.82 показва пакетирането на битовите полета b0 и b1 за имплементации, които позволяват битово поле да пресича границата на сегмент за съхранение.



Фиг. 85 p ≠ 0 и n0 + p > m

Фиг.83 показва пакетирането на битовите полета b0 и b1 за имплементации, които не позволяват битово поле да пресича границата на сегмент за съхранение.



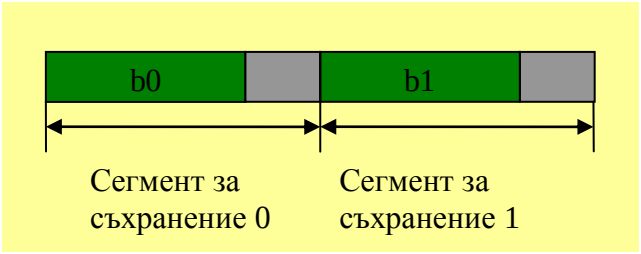
Фиг. 86 p ≠ 0 и n0 + p > m

Сценарии 7

```
int b0 : n0
int    : 0
int b1 : n1
```

където n0 < m и n1 < m

Фиг.84 показва пакетирането на битовите полета b0 и b1.



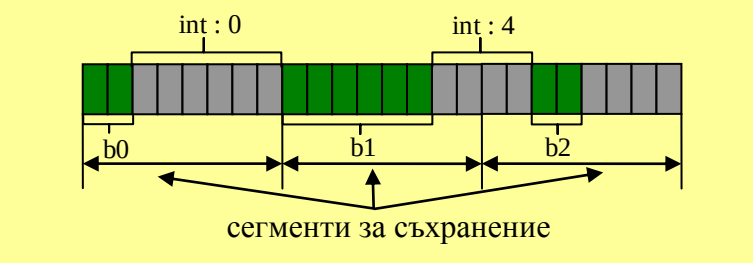
Фиг. 87 n0 < m и n1 < m

Следващият пример е по-практичен като се приема, че използваният сегмент за съхранение е 8 бита. Приема се, че имплементацията позволява битово поле да пресича границите на сегмента за съхранение.

Пример:

```
struct S
{
      int b0 : 2;
      int    : 0;
      int b1 : 6;
      int    : 4;
      int b2 : 2;
};
```

Фиг.85 показва пакетирането на битовите полета.



Фиг. 88 Пакетиране на битовите полета на struct S

Пример 12-5 показва как може да се определи големината на сегмента за съхранение, използван от конкретната имплементация.

Пример 12-5:

```
#include <stddef.h>

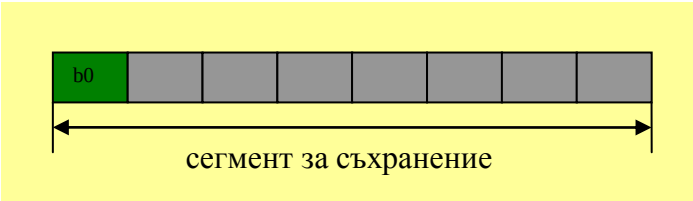
struct TestStorageUnit
{
  int b0 : 1;
  int    : 0;
};

int main(void)
{

  size_t su = sizeof(struct TestStorageUnit);

  return 0;
}
```

Компилаторът ще разположи битовото поле b0 в началото на сегмента за съхранение (Фиг.86). Следващото неименовано поле с нулева дължина принуждава компилаторът да прескочи оставащите битове от текущия сегмент. Това гарантира, че големината на структурата се състои от точно един сегмент за съхранение.

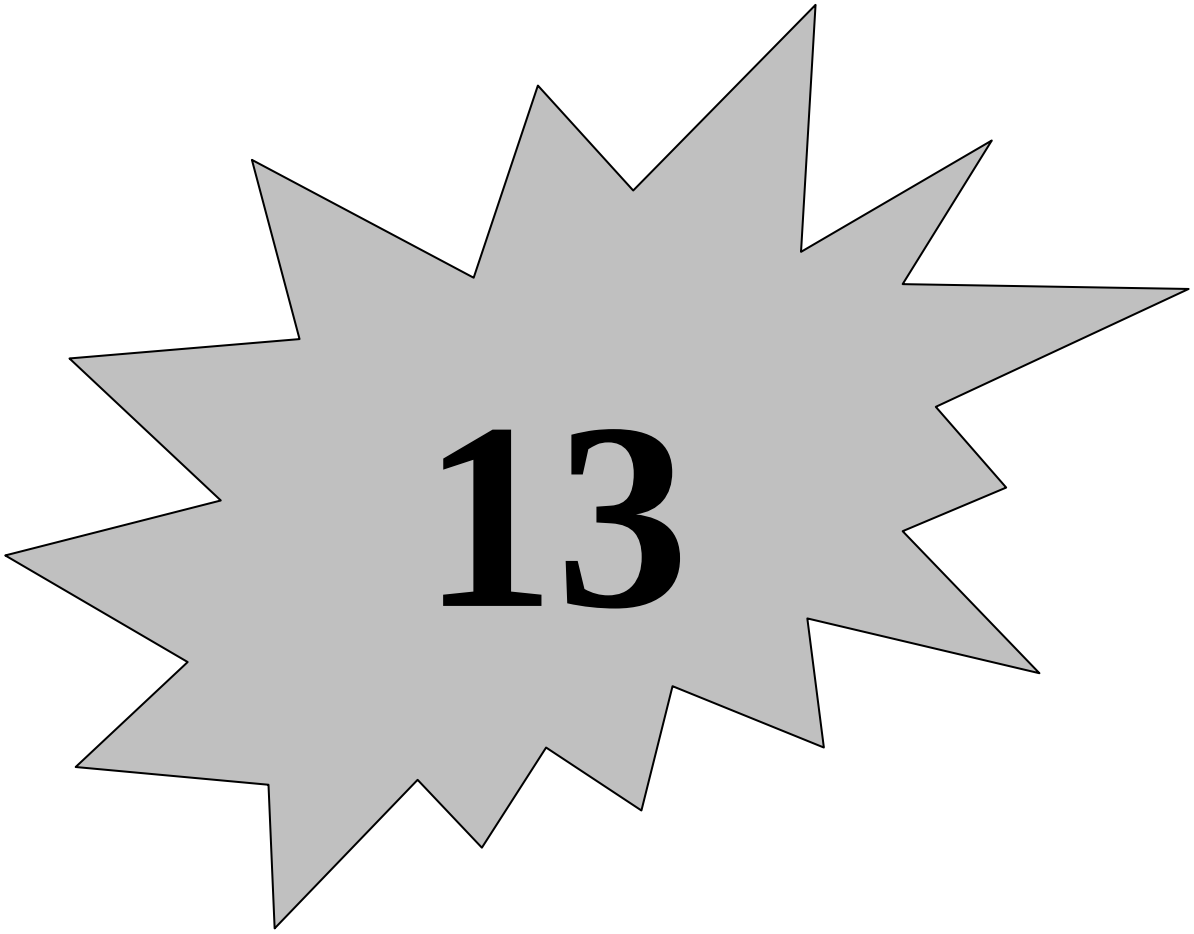


Фиг. 89 Определяне на големината на сегмент за съхранение

Конструкцията

```
sizeof(struct TestStorageUnit)
```

връща броя байтове, заети от цялата структура, а това означава заети и от сегмента за съхранение, използван от конкретната имплементация на компилатора.



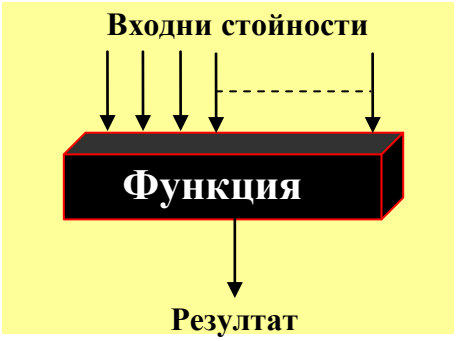
Функции

1.1 Въведение

Функциите са основен градивен елемент на всяка С програма. Една функция капсулира код, който извършва специфична задача. Така вместо програмата да се състои от една голяма функция, която изпълнява всички задачи, тя може да се разбие на по-малки функции, всяка от които изпълнява специфична задача. След това тези функции могат да се изпълнят в желания ред за постигане на крайната цел. Разделянето на програмата на отделни функции има няколко предимства:

- Всяка функция може да се компилира и тества отделно.
- Функциите могат да бъдат разпределени между няколко програмисти, което значително би ускорило развойния процес.
- Капсулирането на задачата във функция позволява един и същ код да се създаде и тества веднъж, след което може да се преизползва толкова пъти колкото е необходимо.
- Една функция може да се преизползва и в други проекти изискващи същата функционалност.
- Разделянето на програмата на отделни функции улеснява поддръжката й и бъдещото й разширяване.

Функцията може да се разглежда още като черна кутия. За потребителя е достатъчно да знае какво точно прави функцията, а не как го прави. Функциите могат да приемат входни стойности и да връщат резултат (Фиг.87).



Фиг. 90 Функция като черна кутия

В зависимост от това кой е създал функциите, те се разделят на две категории: **библиотечни** и **потребителски.**

- Библиотечни функции

Това са функции, които са създадени от производителя на компилатора. Библиотечните функции от своя страна могат да бъдат:

- ✓ Стандартни библиотечни функции

  Това са функции, които са дефинирани от езика С.

- ✓ Нестандартни библиотечни функции

  Това са функции, доставени от производителя на компилатора, но не са част от езика С.

- Потребителски функции

  Това са функции създадени от самия програмист.

Всяка функция е свързана със следните действия:

- Дефиниране на функция

  Дефинирането на функция е процес, при който се изгражда функционалността, извършвана от тази функция, с помощта на последователност от С програмни инструкции (вижте точка 2 Дефиниране на Функция). Функцията се дефинира само веднъж.

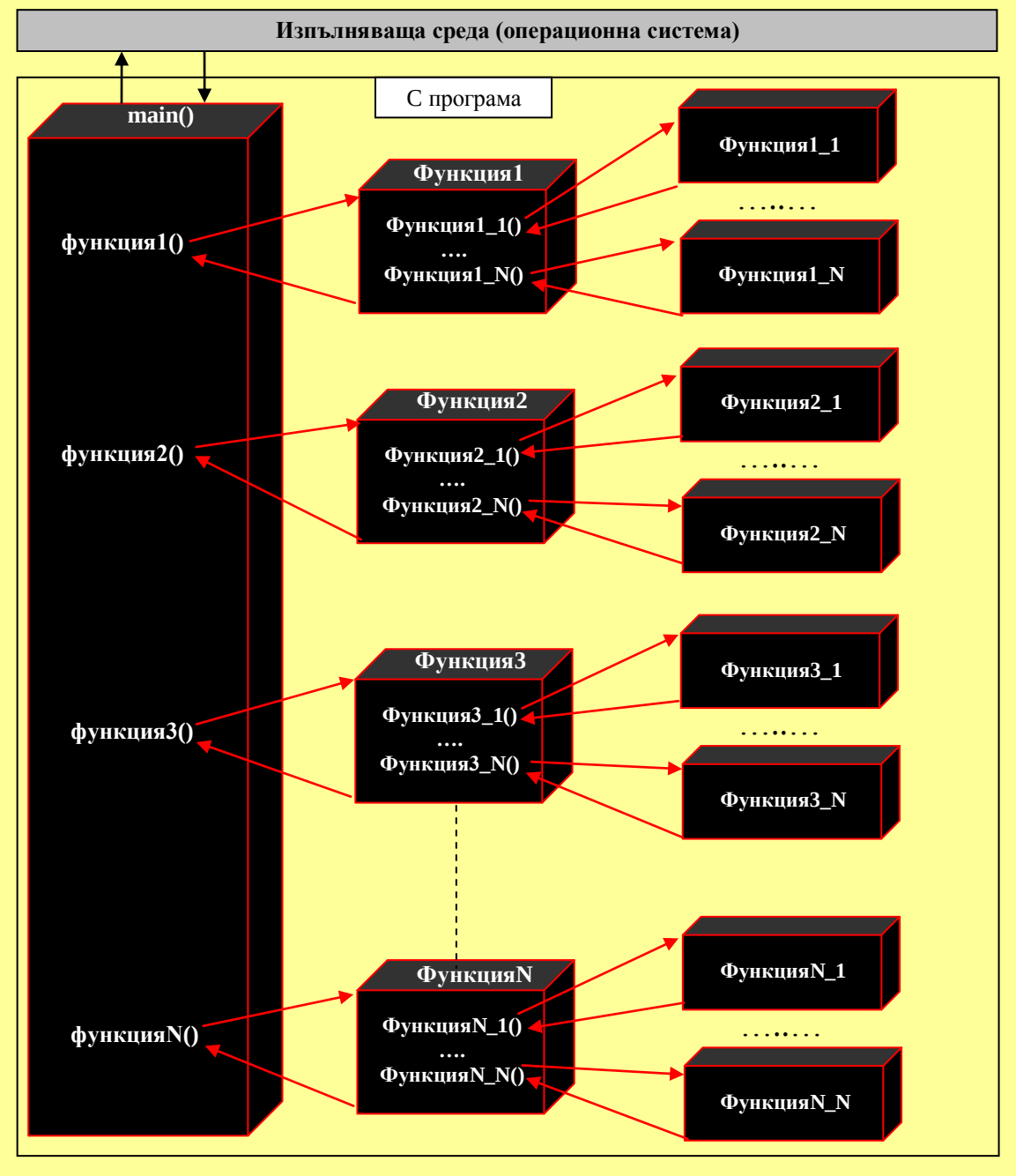
- Извикване на функция

  За да се изпълни кода на дадена функция тя трябва да се извика. Ако функцията използва входни стойности, те трябва да й се подадат. Функцията може да бъде извиквана многократно.

- Декларация на функция

  Декларацията не една функция играе поддържаща роля при нейното извикване. Тя позволява на компилатора да следи дали функцията се използва правилно, например дали й се подават правилния брой и тип входни стойности. Ако това не е така компилаторът ще генерира грешка. Също така компилаторът следи дали типа на резултата, връщан от функцията, е съвместим с този указан в декларацията (вижте точка 4 Деклариране на функция). Функцията може да се декларира многократно.

Фиг.88 показва обобщената структура на една С програма .



Фиг. 91 Обобщена структура на С програма

При стартиране на програмата изпълняващата среда (най-често операционна система) извиква главната функция main() (вижте точка 3 Функцията main), която е входна точка за всяка С програма. Тази функция от своя страна извиква всички останали функции в определен ред. Тези функции също могат да извикват други функции и т.н. При завършване на програмата управлението отново се връща в изпълняващата среда.

1.2 Дефиниране на функция

Общата форма на дефиниране на една функция изглежда така:

Дефиниране на функция

```
тип-резултат      име-функция(списък-параметри)
{
      /* Декларации на променливи */
      extern тип име-променива1;
      ...
      extern тип име-променливаN;

      /* Декларации на функции */
      extern тип име-функция1(списък-параметри);
      ...
      extern тип име-функцияN(списък-параметри);

      /* Декларации на типове с typedef, struct, union, enum */
      typedef декларация-на-тип ново-име-на-тип;
      struct StructName {...};
      union  UnionName {...};
      enum   EnumName {...};
      ...

      /* Дефиниране на променливи */
      тип име-променлива1;
      ...
      тип име променливаN;

      /* Оператори */
      оператор1;
      ...
      операторN;

      return резултат;
}
```

Декларации

Оператори

където

тип-резултат
Може да бъде скаларен тип, структура, обединение, изброяване и void.

списък-параметри
тип параметър1, тип параметър2,..., тип параметърN

Частта, обозначена като

```
тип-резултат   име-функция(списък-параметри)
```

се нарича още заглавие на функцията.

Отварящата и затварящата фигурна скоба {}, декларациите и операторите между тях изграждат тялото на функцията.

Както се вижда заглавието на функцията е изградено от три компонента:

- **тип-резултат** – функциите могат да връщат стойност след завършване на изпълнението си. Този компонент определя типа на стойността връщана от функцията. Той може да бъде всеки един от типовете данни в C без масиви и функции, т.е една функция не може

да върне като резултат от изпълнението си цял масив или функция. Върната стойност може да бъде присвоена на променлива, да се използва в някакъв израз или просто да се игнорира. Ако една функция не връща стойност, се използва тип **void**.

Функциите връщат резултат с помощта на оператора **return**. Той има следния синтаксис.

```
/* За функции, които не връщат стойност */
return;

/* За функции, които връщат стойност */
return резултат;
```

където

резултат
Може да бъде константа, променлива или всеки израз в С, чийто тип е може да се преобразува към връщания от функцията тип.

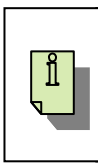
Типът на **резултат** трябва да е съвместим с дефинирания в заглавието на функцията **тип-резултат**. Ако типът е различен, компилаторът автоматично ще се опита да го преобразува към връщания тип. Ако преобразуването е невъзможно компилаторът ще генерира грешка.

Една функция може да има повече от един оператор return в различни точки от изпълнението си. Операторът return води до прекратяване на изпълнението на кода на функцията и връщане на програмата непосредствено след точката на извикване на функцията.

По отношение връщания резултат функциите се делят на такива, които не връщат резултат (наричат се още void-функции) и функции, които връщат резултат. За void-функциите затварящата фигурна скоба в края на функцията изпълнява ролята на оператора return, т.е. за такива функции оператора return в края на функцията не е задължителен. За всички други функции конструкцията.

```
return резултат;
```

е задължителна при завършване на изпълнението си.



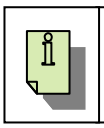
С89 стандартът позволява функции с тип, различен от void да използват оператор return, който не връща резултат, но поведението в този случай е недефинирано.

- **име-функция** – името на функцията се задава с идентификатор. То се използва за извикване на функцията. Извикването на функция

става по следния начин:

```
име-функция(списък-аргументи);
```

списък-аргументи представлява стойности, разделени със запетаи, които се подават на функцията. Ако функцията не приема стойности **списък-аргументи** е празен. Стойностите, които се подават при извикване на функцията се наричат още **фактически аргументи** или **фактически параметри**. За простота в тази книга се използва просто термина аргумент.



Извикването на функция е оператор и затова завършва с точка и запетая.

Когато една функция се извика, програмата се прехвърля за изпълнение на операторите вътре в тялото на функцията.

- **списък-параметри** – този списък представлява декларации на променливи, разделени със запетая, в които се копират входните стойности, подавани на функцията при нейното извикване, т.е

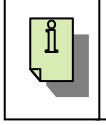
```
тип име-параметър1, тип име-параметър2,..., тип име-параметърN
```

```
където
```

```
тип
      Може да бъде скларен тип, структура, обединение, изброяване
```

Ако функцията не приема аргументи, списък-параметри се заменя с ключовата дума void. Параметрите на функцията се наричат още **формални аргументи** или **формални параметри**. За простота в тази книга се използва просто термина параметър.

Аргументът и съответния параметър трябва да са от съвместими типове.



Аргументите могат да бъдат константи, променливи или всеки допустим израз, чиято стойност може да се преобразува към типа параметъра.

Следващият пример демонстрира дефинирането на функция, която няма входни параметри и не връща резултат.

Пример 13-1:

```
#include <stdio.h>

void foo(void)
```

```
{
	printf("This is a function without parameters and return value\n");

	return; /* може да се пропусне */
}

/* Входна точка на програмата */
int main(void)
{
	foo(); /* Извикване на foo() */

	return 0; /* Връщане в извикващия процес */
}
```



Пример 13-2 демонстрира дефинирането на функция, която има един входен параметър и не връща резултат.

Пример 13-2:

```
#include <stdio.h>

void foo(int a)
{
	printf ("a = %d\n", a);
}

int main(void)
{
	foo(10); /* Извикване на foo() и подаване на един аргумент */

	return 0;
}
```



Пример 13-3 демонстрира дефинирането на функция, която има два входни параметъра и връща тяхната сума като резултат.

Пример 13-3:

```
#include <stdio.h>

int sum(int a, int b)
{
    return a + b;
}

int main(void)
{
    int res;

    res = sum(10,20); /* Извикване на foo() и подаване на два аргумента */
```

```
    printf ("res = %d\n", res); /* Отпечатване на резултата от foo() */

    return 0;
}
```



1.3 Функцията main

Всяка C програма задължително включва една специална функция наречена **main()**. Тази функция е входна точка за всяка C програма, т.е. при стартиране на C програмата, изпълнението й започва от функцията main(), която се извиква от изпълняващата среда (най-често операционна система). Функцията main() от своя страна може да извиква други функции, които също могат да извикват функции и т.н. Функцията main() е дефинирана от стандарта C по следните два начина:

```
int main(void);
```

и

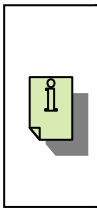
```
int main(int argc, char* argv[]);
```

Втората форма позволява подаване на аргументи при стартиране на програмата от командния ред.

Описание на параметрите на main:

- **argc** - съдържа броя на всички аргументи подадени от командия ред, в това число и името на програмата. Ако няма подадени входни аргументи, параметърът argc съдържа стойност 1, тъй като името на програмата се брои за първи аргумент по подразбиране.

- **argv** – масив от указатели към char. Всички аргументи от командния ред представляват символни низове. Първият елемент argv[0] сочи към името на програмата, втория елемент argv[1] сочи към първия аргумент, следващ името на програмата, и т.н. Масивът argv[] винаги съдържа един елемент повече от броя на аргументите от командния ред (в това число влиза и името на програмата). По дефиниция този елемент е нулев указател, т.е argv[argc] = = NULL.



Запомнете, че името на програмата, въведено в командия ред, както и всички входни аргументи, се разглеждат като символни низове. Ако входните аргументи са числа, то преди да се използват в програмата, те трябва да се преобразуват от низове в съответния им числов еквивалент.

Пример 13-4 демонстрира отпечатва името на програмата и всички входни аргументи, ако има такива.

Пример 13-4:

```
#include <stdio.h>

int main(int argc, char* argv[])
{
	int i;

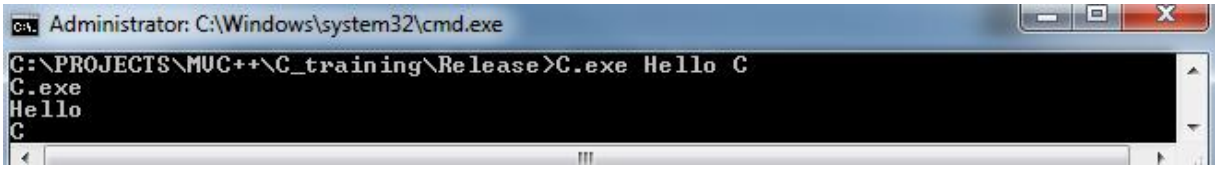
	if (1 == argc)
	{
		printf ("There are no command line arguments\n");
	}
	else
	{
		for (i = 0 ; i < argc ; i++)
		{
			printf ("%s\n", argv[i]);
		}
	}

	return 0;
}
```

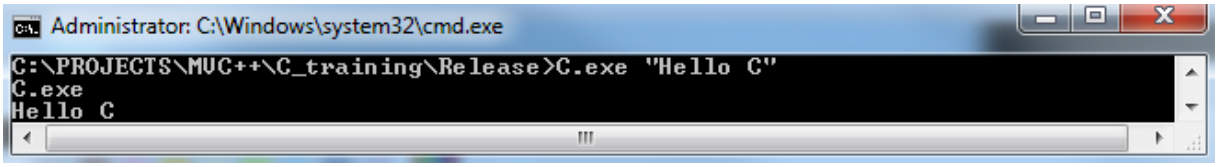
Стартирането на програмата, компилирана с Microsoft Visual Studio 2010 Express, е показано по-долу. Името на програмата е C.exe



Както се вижда от резултата, програмата отпечатва името си и всички допълнителни входни аргументи. Входните аргументи трябва да са разделени с поне един интервал. Ако въведете входен аргумент, който съдържа в себе си интервали, за да се възприеме като един аргуметър, той трябва да се постави в кавички. Например сравнете долните два примера на подаване на аргументи на същата програма.



и



В първия случай последователността HELLO C се възприема като два отделни аргумента (HELLO и C), докато във втория случай "HELLO C" се възприема като един аргумент.

Имената на входните параметри на **main()** не са фиксирани до показните по-горе **argc** и **argv**. Може да използвате имена по ваш вкус, но показаните тук са общоприети. Също така, освен показаните форми на main(), различните имплементации на компилатора могат да поддържат и други допълнителни форми. За повече подробности се обърнете към документацията на Вашия компилатор.

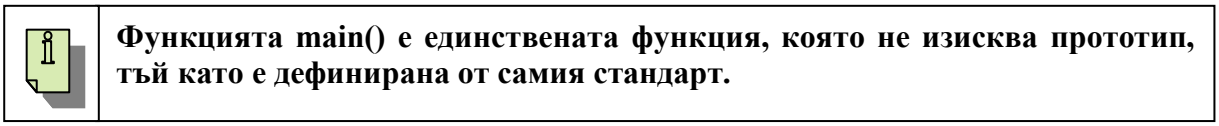
Стойността връщана от **main()** се подава на извикващата я среда. Връщаната стойност може да бъде 0 или **EXIT_SUCCESS**, което е идникация, че програмата е завършила нормално, или **EXIT_FAILURE**, което е индикация за някакъв проблем. Макросите EXIT_SUCCESS и EXIT_FAILURE са дефинирани в хедъра <stdlib>. Използването на други стойности е имплементационно-зависимо. Някои среди игнорират връщаната стойност.

1.4 Деклариране на функция

1.4.1 Прототип и област на видимост на прототипа

Декларацията на функция се нарича още прототип. Прототипът на една функция се състои от заглавието на функцията, последвано от точка и запетая.

```
/* Декларация/прототип на функция  */
тип-резултат        име-функция(списък-параметри);
```



Функцията main() е единствената функция, която не изисква прототип, тъй като е дефинирана от самия стандарт.

Прототипът трябва да предшества извикването на функцията. Единствената ситуация, в която прототипът може да се изпусне е когато функцията е дефинирана преди първото си извикване в един файл. Това не винаги обаче е достатъчно. Ако функцията е дефинирана в един файл, но

се използва в друг, то втория файл трябва да включва и прототипа.

Пример 13-5:

```
#include <stdio.h>

/* Декларация на функцията foo */
void foo(void);

int main(void)
{
	foo(); /* Извикване на foo() */

	return 0;
}

/* Дефиниране foo() */
 void foo(void)
{
	printf ("Hello C\n");
}
```



В следващия пример, дефинирането на foo() е преместено преди main(). В този случай дефиницията играе роля и на декларация и декларацията може да се изпусне.

Пример 13-6:

```
#include <stdio.h>

/* Дефиниране на foo() */
void foo(void)
{
	printf ("Hello C\n");
}

int main(void)
{
	foo();

	return 0;
}
```

Областта на видимост на декларацията се определя от мястото където е поставена. Ако е поставена извън всички функции, декларацията е видима от точката на декларация до края на файла. Декларациите могат имат и локална област на видимост (област на видимост във функция или блок).

В **Пример 13-7** декларацията на foo() се вижда в само в тялото на функцията main().

Пример 13-7:

```
#include <stdio.h>

int main(void)
{
	/* Локална декларация на foo() в main */
	void foo(void);

	foo();

	return 0;
}

/* Дефиниране foo */
void foo(void)
{
	printf ("Hello C\n");
}
```

В **Пример 13-8** декларацията на foo() се вижда в само в тялото на оператора if.

Пример 13-8:

```
#include <stdio.h>

int main(void)
{
	int x;

	x = 1;

	if(1 == x)
	{
		/* Локална декларация на foo() в if */
		void foo(void);

		foo();
	}

	foo(3); /* !!!Грешка, прототипът не се вижда извън if */

	return 0;
}

/* Дефиниране foo */
void foo(void)
{
	printf ("Hello C\n");
}
```

В **Пример 13-9** функцията foo() е дефинирана във сорс-файл2, но се извиква в сорс-файл1.

Пример 13-9:

```
/* сорс-файл1*/

/* Декларация на foo() */
void foo(void);

int main(void)
{
      foo();

      return 0;
}
```

```
/* сорс-файл2*/

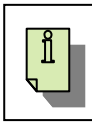
#include <stdio.h>

/* Дефиниция на foo() */
void foo(void)
{
      printf ("Hello C\n");
}
```

Още едно нещо, което трябва да знаете когато задавате прототип, е че компилаторът игнорира имената на параметрите в прототипа. Това означава, че имената на параметрите могат се изпуснат в прототипа или пък да не съвпадат с тези в дефиницията на функцията. Например следните прототипи са абсолютно еквивалентни за компилатора:

```
/* Прототип на foo */
void foo(int x);    /* Прототипът съвпада със заглавието на функцията */
void  foo(int  y);   /* Името на параметъра в прототипа се различава от това в
                        дефиницията */
void foo(int);      /* Името на параметъра е изпуснато */
...

/* Дефиниране на foo */
void foo(int x)
{
   ...
}
```



Имената на параметрите в прототип на функция се игнорират от компилатора.

Все пак добрият стил на програмиране изисква прототипът да съвпада точно със заглавието на функцията.

1.4.2 Използване на прототипи

Декларацията (прототипът) на една функция не е задължителна в C89. Преди да разгледаме поведението при извикване на функция, която е покрита с прототип, ще разгледаме поведението, когато липсва прототип.

Поведение на функции с предвариателна декларация

Преди C89 вместо прототип на функция се използваше т.нар

предварителна декларация на функцията.

Предварителна декларация на функция

```
тип-резултат  име-функция();
```

Предварителната декларация дава информаця на компилатора само за типа на резултата, но не и за параметрите. Това означава, че правилността на извикване на една функция лежи изцяло на програмистта. Освен това компилаторът извършва следните автоматични преобразувания:

- Аргументите от тип char (signed или unsigned), short (signed или unsigned) и enum-типове и битови полета се подлагат на целочислено повишаване на типа преди да бъдат подадени на функцията (вижте Глава 5 Преобразувания).

- Аргументите от тип float се преобразуват в тип double.

Следващата програма ще се компилира, но няма да работи правилно, тъй като се извиква неправилно.

Пример 13-10:

```
#include <stdio.h>

/* Предварителна декларация на foo() */
int sum();

int main(void)
{
	int res;

	res = sum(); /* !!!Грешка, sum() изисква 2 аргумента */
	printf("res = %d\n", res);

	return 0;
}

/* Дефиниция на sum() */
int sum(int x, int y)
{
	 return x+y;
}
```



Поведение на функции без декларация

Когато една функция не е покрита с прототип или предварителна декларация, първото извикване на функцията във файла служи за декларация. Аргументите на функцията се подлагат на същите

преобразувания както и при наличие на предварителна декларация. По подразбиране компилаторът допуска, че връщания от функцията тип е int. Ако в дефиницията на функцията типът на резултата е различен от int, компилаторът ще генерира грешка. Например следната програма няма да се компилира.

Пример 13-11:

```
#include <stdio.h>

int main(void)
{
	foo(); /* Компилаторът допуска, че типа на резултата е int */

	return 0;
}

/* Дефиниция на foo() */
void foo(void) /* В дефиницията типът на резултата е void */
{
	 /*...*/

	return;
}
```

При липса на прототип компилаторът не прави никакви допускания относно типа и броя на параметрите на функцията. Това означава, че правилността на извикване на една функция лежи изцяло на програмистта. Например следващата програма ще се компилира, но няма да работи правилно.

Пример 13-12:

```
#include <stdio.h>

int main(void)
{
	int res;

	res = divideByTwo(); /* !!!divide() има един параметри, но се извиква без
				аргумент */
	printf("res = %d\n", res);

	return 0;
}

/* Дефиниция на foo() */
int divideByTwo(int i)
{
	return i/2;
}
```



По подразбиране компилаторът допуска, че връщания от divideByTwo()

резултат е int. В дефиницията връщания тип също int, така че кодът ще се компилира без проблем. Компилаторът обаче не прави никакви предположения относно параметрите на функцията. Ето защо извикването на divideByTwo() с неправилен тип и/или брой аргументи няма да бъде засечено от компилатора и резултатът от изпълнението на програмата ще бъде недефиниран.

Поведение на функции с прототип

Когато една функция е покрита с прототип, типовете на аргументите се преобразуват към типовете на параметрите по същите правила както при присвояване (вижте Глава 5 Преобразувания). Подразбиращи се преобразувания на типа на аргументите не се извършват.

Когато една функция е извикана в кода, компилаторът проверява дали типа и броя на подадените й аргументите съвпадат с тези указани в прототипа. Ако броят на аргументите и параметрите не съвпада, компилаторът ще генерира грешка. Например следващата програма няма да се компилира.

Пример 13-13

```
#include <stdio.h>

void foo(int a, int b);

int main(void)
{
	foo(1); /* !!!Грешка, foo изисква два аргумента */

	return 0;
}

void foo(int a, int b)
{
	/* някакъв код */
}
```

Ако броят на аргументите и параметрите съвпада, компилаторът сравнява техните типове. Ако име несъответствие, компилаторът ще се опита да преобразува типа на аргумента в типа на съответния параметър. Ако преобразуването е невъзможно, компилаторът ще генерира грешка. Например следващата програма ще се компилира без проблем.

Пример 13-14:

```
#include <stdio.h>

void foo(int a);

int main(void)
{
	unsigned char i = 10;
```

```
	foo(i); /* Компилаторът автоматично преобразува  i в int */

	return 0;
}

void foo(int a)
{
	 printf ("a = %d\n", a);
}
```



Не всяко преобразуване обаче е безопасно. Някои проебразувания на типове могат да доведат до загуба на данни. Следващата програма например ще се компилира, но няма да работи правилно.

Пример 13-15:

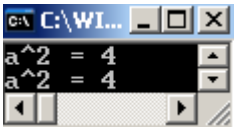
```
#include <stdio.h>

void sqr(int a);

int main(void)
{
	sqr(2);
	sqr(2.3); /* !!!Внимание */

	return 0;
}

void sqr(int a)
{
	printf ("a^2 = %d\n", a * a);
}
```



Извикването на sqr() с аргумент 2 ще отпечата правилен резултат. Второто извикване обаче, с аргумент 2.3, ще доведе до грешен резултат. Причината, е че дробната част на аргумента се отрязва поради автоматичното преобразуване в типа на параметъра. Повечето компилатори ще генерират предупреждение при подобни преобразувания.



С не е строго типизиран език за разлика от езици като Java и C#. Това означава, че компилаторите ще позволят автоматични преобразувания между типове, които могат да доведат до загуба на данни, или дори до недефинирани резултати. Отговорност на програмистта е да гарантира, че това няма да се случи.

Следващата програма няма да се компилира поради пълното несъответсвие

на типовете на аргумента и параметъра.

Пример 13-16:

```
#include <stdio.h>

void foo(int a);

int main(void)
{
	int i = 10;

	foo(&i); /* !!!Грешка, изисква се целочислена стойност, а се подава адрес */

	return 0;
}

void foo(int a)
{
	/* някакъв код */
}
```

Освен че следи типа на аргументите, подавани на функцията, компилаторът следи и типа на резултата, връщан с оператора return. Ако той не съвпада с указания тип в прототипа, предварителната деклрация или подразбиращата се декларация, компилаторът ще се опита да го преобразува. Ако преобразуването е невъзможно ще се генерира грешка при компилиране.

Пример 13-17:

```
#include <stdio.h>

int foo();

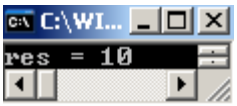
int main(void)
{
	int res;

	res = foo();
	printf ("res = %d\n", res);

	return 0;
}

int foo()
{
	unsigned char i = 10;

	return i; /* Преобразува типа на i в int, т.е. */
}		  /* return (int)i;                    */
```



Следващата програма няма да се компилира поради пълна несъвместимост между типа връщания резултат и типа на операнда на оператора return.

Пример 13-18:

```
#include <stdio.h>

int foo();

int main(void)
{
	int res;

	res = foo();
	printf ("res = %d\n", res);

	return 0;
}

int foo()
{
	unsigned char i = 10;

	return &i; /* Грешка, изисква се тип int, а се връща адрес (int*) */
}
```

1.5 Механизми на подаване на аргументи на функция

Аргументите могат да се подават на функциите по два начина:

- **чрез копие** – този начин се нарича още и подаване по стойност. Стойността на аргумента се копира в параметъра. Всяка по-нататъшна промяна на параметъра в тялото на функцията не влияе на стойността на аргумента.

- **чрез адрес** – при този начин на функцията се предава не стойността, а адреса на аргумента. Това означава, че функцията може да изменя неговата стойност.

  Подаването на адрес на аргумент е начин една функция да върне повече от един резултат в извикващата я програма. Технически операторът return може да върне само една стойност.

  Подаването чрез адрес е особено полезно ако е необходимо на функцията да се подаде голямо количество данни. Вместо тези данни да се копират в параметъра, могат да се подадат адресите им в паметта. Това значително увеличава производителността на програмата и избягва проблеми с препълване на стека.

1.5.1 Подаване на аргументи чрез копие

Всички примери от началото на тази глава досега използваха именно този метод. Следващият пример демонстрира факта, че изменението на параметъра в тялото на функцията не влияе по никакъв начин на аргумента.

Пример 13-19:

```
#include <stdio.h>

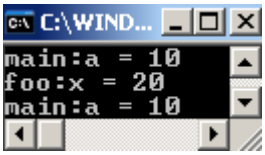
void foo(int x);

int main(void)
{
	int a = 10;

	printf ("main:a = %d\n", a);
	foo(a);
	printf ("main:a = %d\n", a);

	return 0;
}

void foo(int x)
{
	x = 20; /* Промяна на параметъра в тялото на функцията */
	printf ("foo:x = %d\n", x);
}
```



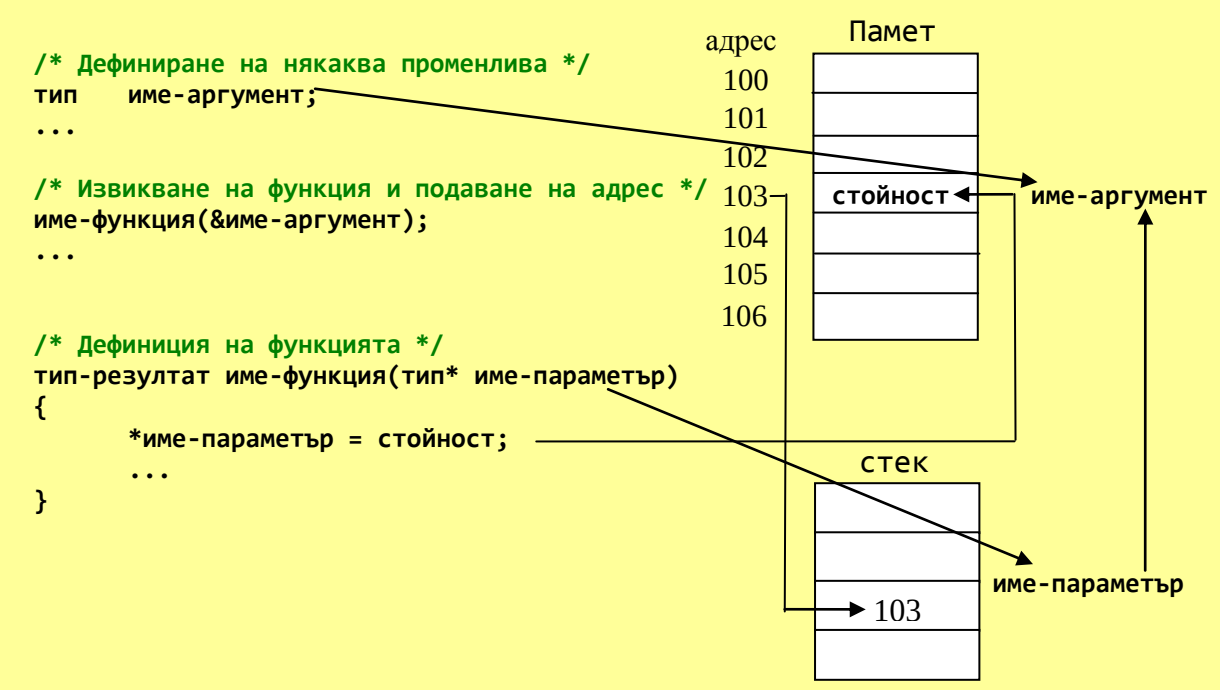
Стойността на аргумента a се копира в параметъра x. Всякакви по нататъшни действия върху x, се извършват върху копираната в него стойност и не засягат аргумента a.

1.5.2 Подаване на аргументи чрез адрес

Подаването на аргумент чрез указател (по адрес) схематично е показано на Фиг.89.

За да се подаде аргумент чрез указател е необходимо параметъра да бъде дефиниран като указател към съответния тип. При извикване на функцията се подава адреса на аргумента, а не неговата стойност, с помощта на опеацията за взимане на адрес &. Преди влизане в тялото на функцията, адреса се копира в указателя-параметър, за който компилаторът е заделил памет в стека. В тялото на функцията дереференцирането на указателя

осъществява достъп до аргумента.



Фиг. 92 Подаване на аргумент чрез указател

Пример 13-20:

```
#include <stdio.h>

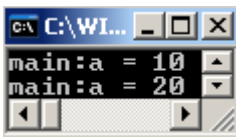
void foo(int* p);

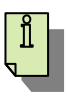
int main()
{
	int a = 10;

	printf ("main:a = %d\n", a);
	foo(&a); /* Подава се адреса на a, а не стойността */
	printf ("main:a = %d\n", a);

	return 0;
}

void foo(int* p)
{
	*p = 20; /* Аргументът, сочен от указателя p, се променя */
}
```





Технически погледнато един указател-аргумент също се подава чрез копие на указателя-параметър, т.е. адреса съхранен в указателя-аргумент се копира в указателя-параметър. Изменението на стойността на указателя-параметър в тялото на функцията (а не на сочената от него променлива), не изменя указателя-аргумент.

Пример 13-21:

```
#include <stdio.h>

void foo(int* ptr);

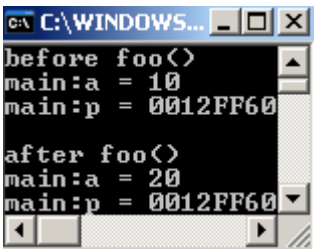
int main()
{
	int  a = 10;
	int* p = &a;

	printf ("before foo()\n");
	printf ("main:a = %d\n", a);
	printf ("main:p = %p\n\n", p);
	foo(p);
	printf ("after foo()\n");
	printf ("main:a = %d\n", a);
	printf ("main:p = %p\n\n", p);

	return 0;
}

void foo(int* ptr)
{
	*ptr = 20;

	/* "разкачане" на ptr от сочения от него обект */
	ptr = NULL;
}
```



Квалификаторите **const** и **volatile** също могат да се прилагат към декларациите на параметри на функция. Спецификаторите за клас памет не са позволени.

Следващият пример показва как да защитим обекта, сочен от указателя-параметър, от изменение в тялото на функцията. Това е полезно ако аргументът съдържа голямо количество данни, например структурна променлива, които са достъпни само за четене, но не и за изменение. Подаването на адреса на структурната променливата ще ускори изпълнението на програмата, а декларирането на параметъра като указател към const ще защити сруктурната променлива от случайно изменение в тялото на функцията.

Пример 13-22:

```
#include <stdio.h>
```

```
void foo(const struct mystruct* ptr);

struct mystruct
{
	int a[5];
};

int main(void)
{
	struct mystruct s = {1,2,3,4,5};

	foo(&s);

	return 0;
}

void foo(const struct mystruct* ptr)
{
	ptr->a[0] = 20; /* !!!Грешка, обекта, сочен от ptr не може да се изменя */
	/*...*/
}
```

1.6 Глобални (външни) и локални (статични) функции

Функциите могат да бъдат направени достъпни за цялата програма (глобални функции) или само за файла, в който са дефинирани (локални функции).

1.6.1 Глобални функции

Ако една функция ще се извиква от различни файлове, то всеки файл, трябва да включва и прототипа на функцията преди точката, от която функцията се извиква. Освен това прототипът трябва да бъде видим в тази точка (вижте точка 4.1 Прототип и област на видимост на прототипа). По подразбиране имената на функциите са външни идентификатори. Това означава, че следните прототипи са еквивалентни:

```
	тип-резултат	име-функция(списък-параметри);

extern тип-резултат	име-функция(списък-параметри);
```

Подобно на глобалните променливи, глобалните функции имат външно свързване.

1.6.2 Локални функции

Ако една функция ще се извиква само във файла, в който е дефинирана, тя

може да се направи "невидима" за всички останали файлове (подобно на глобалните статични променливи). За целта пред декларацията и дефиницията трябва да се използва ключовата дума **static**.

Пример 13-23:

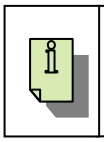
```
#include <stdio.h>

static  void foo(void);

int main(void)
{
	foo();

	return 0;
}

static  void foo(void)
{
	/* някакъв код */
}
```



Няколко сорс файла могат да дефинират статична функция с едно и също име. Всяка версия на статичната функция е локална за файла, в който е дефинирана, и не се вижда от другите файлове.

Подобно на статични променливи, локалните функции имат вътрешно свързване.

1.7 Функции с променлив брой аргументи

С позволява дефинирането на функции, които могат да приемат променлив брой аргументи. Такива функции се наричат още **variadic-функции**. Variadic-функциите изискват наличието на поне един фиксиран аргумент, последван от променлив брой аргументи, които могат да бъдат от различен тип. Броят на променливите аргументи може да се определи или по стойността на някой от фиксираните аргументи или чрез специална стойност, с която завършва списъка на променливите аргументи.

Класически пример за такива функции са библиотечните функции print() и scanf(). Тези функции имат един задължителен аргумент, който представлява низ. Форматните спецификатори в низа се използват, за да се определи броя и типа на променливите аргументи.

Дефиниране на функция с променлив брой аргументи:

```
тип-резултат име-функция(списък-фиксирани-параметри, ...) { /* Код */  }
```

Трите точки **...** след запетаята указват, че функцията приема променлив брой аргументи.

За работа с variadic-функции се използват следните макроси дефинирани в хедъра <stdarg.h>:

- ```
  void va_start(va_list argptr, lastparam);
  ```

  va_list - специален тип, който представлява указател към променлив брой аргументи.

  ```
  typedef   ...   va_list
  ```

  argptr - указател от тип **va_list**, към списъка с променлив брой аргументи.

  lastparam - името на последния фиксиран параметър.

  Макросът va_start инициализира указателя argptr с позицията на най-десния параметър от списъка с фиксирани параметри. Този макрос се извиква винаги първи преди да се използват променливите аргументи.

- ```
  type va_arg(va_list argptr, type);
  ```

  argptr - указател от тип **va_list**, към списъка с променлив брой аргументи.

  type - тип на поредния аргумент от списъка с променлив брой аргументи.

  Първото извикване на макроса va_arg() връща стойността на първия аргумент от списъка с променлив брой аргументи (първия аргумент след този указан с lastparam). Всяко последващо извикване на va_arg() връща стойността на поредния аргумент и указателят argptr се увеличава към следващия аргумент от списъка. Типът на поредния аргумент (след прилагане на целочисленото повишение) трябва да се прилага, за да може va_arg() да изчисли правилно неговия размер в стека.

- ```
  void va_end(va_list argptr);
  ```

  argptr - указател от тип **va_list**, към списъка с променлив брой аргументи.

Макросът va_end деинициализира указателя argptr. Този макрос трябва да
  ```

се извиква веднъж за всяко извикване на va_start. Ако va_end се извика преди да се извика va_start или ако функцията завърши изпълнението си без да се извика va_end, поведението е недефинирано.

Следващият пример демонстрира функция с променлив брой аргументи, която може да сумира произволен брой числа с плаваща запетая и връща сумата.

Пример 13-24:

```
#include <stdio.h>
#include <stdarg.h>

double sum(int n,...);

int main(void)
{
	printf ("%.1lf\n", sum(2,1.4,2.5));
	printf ("%.1lf\n",sum(3,1.2, 2.8,3.1));

	return 0;
}

/*************************************************************************
* Сумира аргументи от тип double и връща резултата.                     *
* Параметърът n съдържа броя на аргументите, с които се извиква функцията *
*************************************************************************/
double sum(int n,...)
{
	int	i   = 0;
	double	sum = 0.0;

	va_list argptr; /* Дефиниране на указател от тип va_list */

	va_start(argptr, n); /* Инициализиране на указателя argptr */

	for (i = 0; i < n; ++i) /* Сумиране на аргументите */
	{
		/* Четене на стойността на поредния аргумент, сочен от argptr */
		double temp = va_arg(argptr, double);

		sum += temp;
	}

	va_end(argptr); /* Деинициализиране на argptr */

	return sum;
}
```



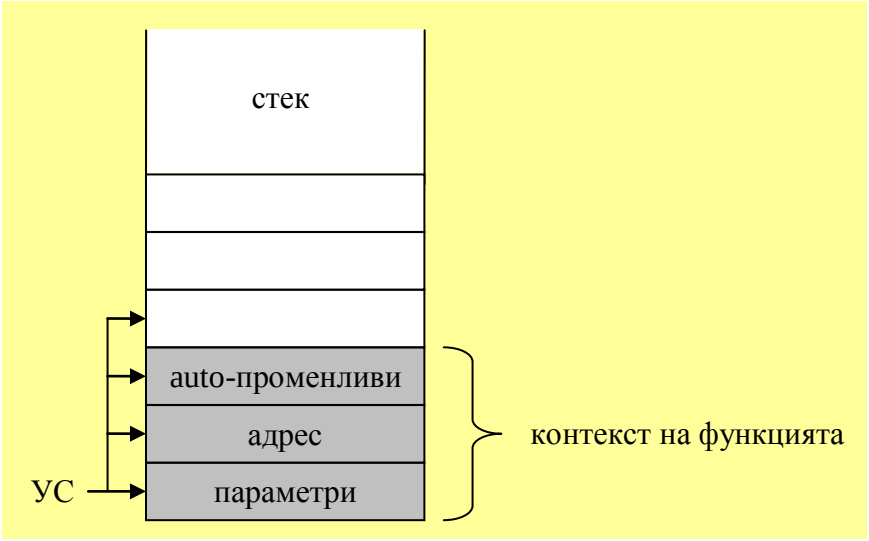
Пример 13-24 е прост пример на variadic-функция, която работи само с променлив брой аргументи от тип double. Ако искате да дефинирате по-универсална функция, която да работи с аргументи от различен тип ще трябва да измислите някакъв свой подход, чрез който да определяте типа

на аргументите (за съжаление C89 не предоставя програмни средства за определяне на типа на данните). Един такъв подход е да използвате фиксиран параметър, който е указател към char. При извикване на функцията, първия фиксиран аргумент може да бъде низ, който подобно на функцията printf() съдържа някакви определени от вас типове форматни спецификатори. Броят на форматните спецификатори трябва да отговаря на броя на променливите аргументите, а типа на спецификатора трябва да отговаря на типа на съответния аргумент.

1.8 Стек

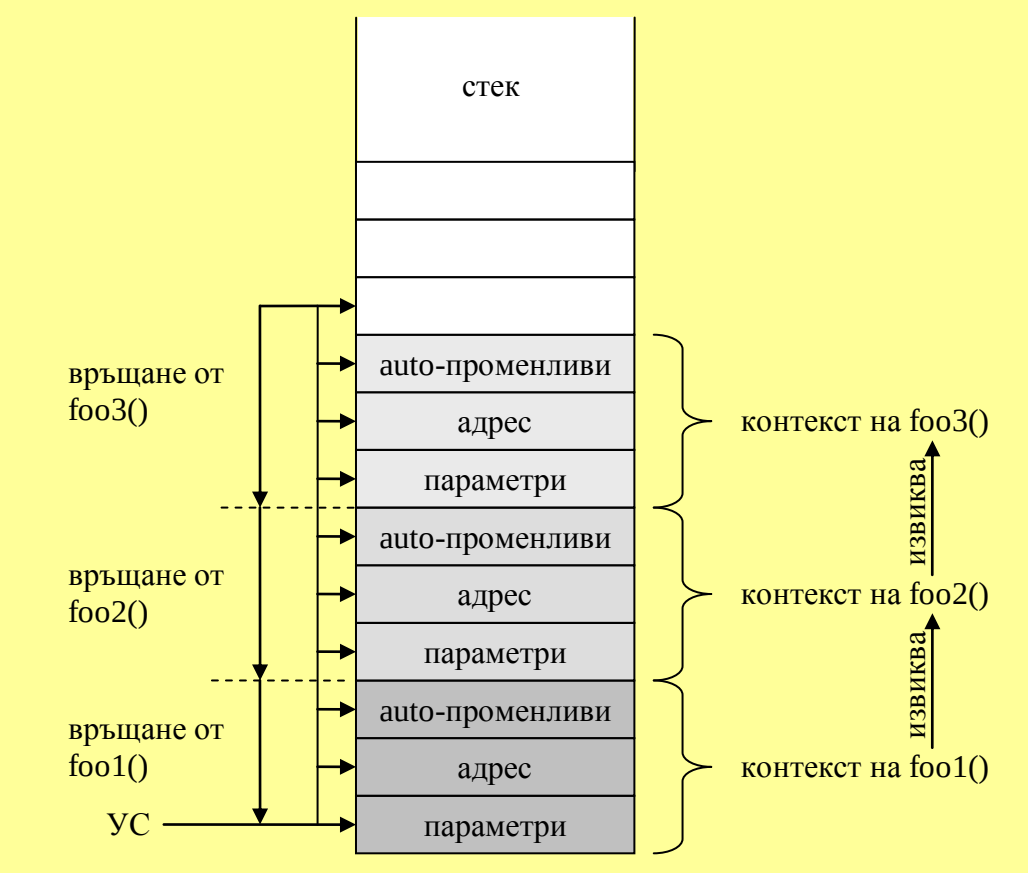
В тази точка ще разгледаме какво точно е стек и обобщения принцип на работа на стека. Когато програмирате на език от по-високо ниво, какъвто е езика C, обикновено няма да Ви се налага да се занимавате със стека. Компилаторът се грижи за това. Познаването на общите принципи обаче ще Ви помогне да разберете по-добре как работят функциите.

Стекът е част от RAM паметта на компютъра. Когато компилирате една програма, компилаторът автоматично заделя една част от паметта за стек. Големината на стека може да се конфигурира. Всеки процесор съдържа един специален регистър, наречен **указател към стека - УС** (**SP – Stack Pointer**). Както самото име подсказва УС сочи към стека и обикновено към поредната свободна клетка от него. Когато извикате една функция, компилаторът заделя памет за параметрите й (ако има такива) в стека и УС се увеличава така, че да сочи към следващата свободна клетка. Компилаторът също генерира асемблерен код, който копира аргументите в съответните параметри, след което се прави преход към тялото на функцията (с помощта на специална асемблерна инструкция). Адресът на инструкцията, която се намира след точката на извикване на функцията също се запомня в стека. Това е необходимо за да може след връщане от функцията програмата да продължи с изпълнението на тази инструкция. Ако функцията съдържа дефиниции на локални auto-променливи, компилаторът също заделя памет за тях в стека. Ако функцията съдържа съставни оператори, в които също са дефинирани локални auto-променливи, то при влизане в такъв оператор, компилаторът заделя памет за тях в стека, а при излизане я освобождава. Процесът на освобождаване се състои просто в намаляване на УС така, че да сочи към адреса преди заделяне на памет за тези локални променливи. Този процес се повтаря за всеки съставен оператор във функцията. При завършване на изпълнението на функцията (при достигане на оператор return или затварящата фигурна скоба на функцията), УС се намалява така, че цялата памет от стека, използвана при извикването на тази функция се освобождава. Информацията съхранявана в стека при извикване на функция се нарича **контекст на функцията** (Фиг. 90).



Фиг. 93 Стек и указател към стека

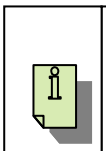
Ако една функция извиква друга функция, която пък извиква трета функция и т.н., то контекстите на функциите се съхраняват в стека последователно. При излизане от функция, паметта заемана от нейния контекст се освобождава (Фиг. 91).



Фиг. 94 Развиване и навиване на стека

Като обобщение можем да кажем, че при изпълнение на програмата използваната памет от стека ту се увеличава (нарича се развиване на стека), ту се намалява (нарича се навиване на стека). Обърнете внимание на

следния проблем, който може да възникне. Ако имате дълга последователност от извикване на функция, която извиква друга функция, която извиква трета функция и т.н. или функцията има голям брой параметри (например масив с много елементи или структура с много членове), или голям брой локални променливи, или комбинация от трите, е възможно препълване на стека. Резултатът обикновено е аварийно завършване на програмата.



Технически, организацията на контекста на функцията, който се записва в стека при нейното извикване, и посоката на изменение на указателя към стека (дали се увеличава или намалява), зависи от компютъра и имплементацията на компилатора.

1.9 Рекурсивни функции

Рекурсивна функция е тази, която извиква себе си, косвено или директно. Директно означава, че тялото на функцията съдържа оператор за извикване на функция към себе си. Рекурсивната функция трябва да съдържа условие за излизане от рекурсията, в противен случай ще се получи безкрайно извикване на функцията, което ще доведе до препълване на стека и аварийно завършване на програмата.

```
тип foo(списък-параметри)
{
	...

	if(условие-за-излизане-от-рекурсията)
	{
		foo(списък-аргументи); /* Директно извикване */
	}
	...
}
```

Следващият пример изисква от потребителя въвеждане на цяло положително число num и сумира на всички цели числа от 1 до num с помощта на директна рекурсия.

Пример 13-25:

```
#include <stdio.h>

int sum(int n);

int main(void)
{
	int num, add;

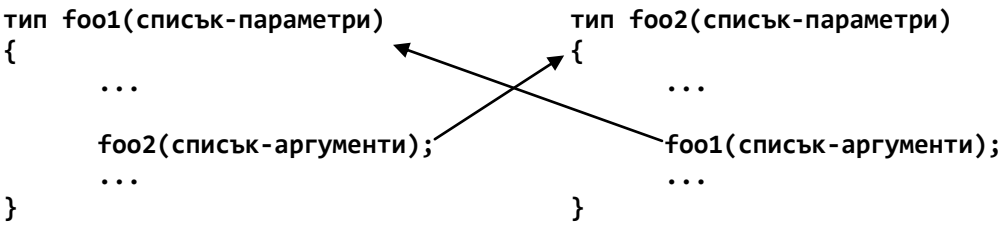
	printf("Enter a positive integer: ");
	scanf("%d",&num);
	add=sum(num); /* 1+2+ ... +num */
```

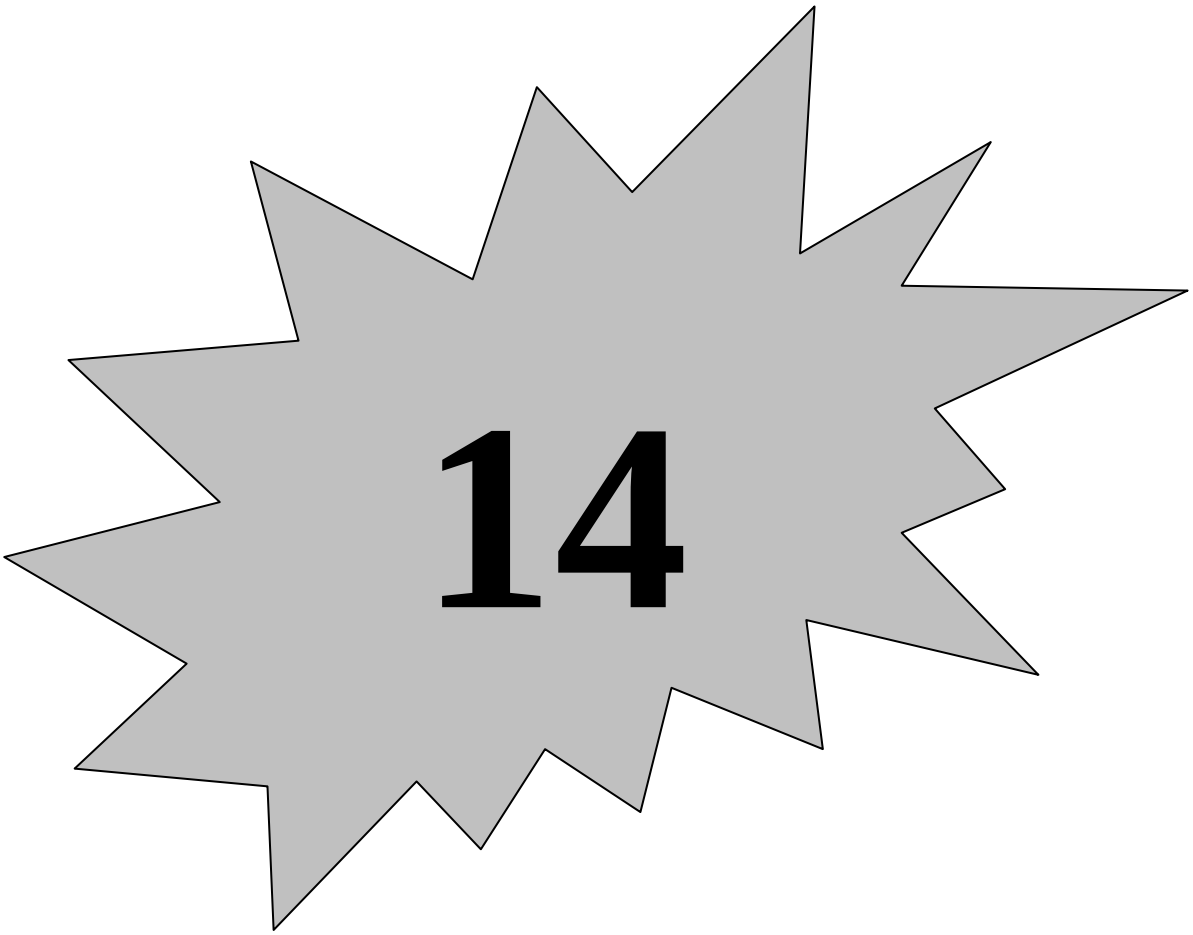
```
	printf("sum=%d",add);

	return 0;
}

int sum(int n)
{
	if(0 != n) /* Условие за излизане от рекурсията */
	{
		return n+sum(n-1);
	}
	else
	{
		return n;
	}
}
```

Косвена рекурсия означава, че функцията извиква друга функция (която може да извиква трета функция и т.н.), която извиква първата функция.





Съвместими и композитни типове

1.1 Съвместими типове

Тази клауза преди всичко се занимава със съвместимостта на типове измежду различни сорс-файлове. Най-общо ANSI/ISO-9899:1990 стандартът дефинира, че два типа са съвместими ако те са еднакви. Всички валидни декларации в една област на видимост, които сочат към един и същ обект или функция, трябва да имат съвместими типове. Следващите точки описват по-подробно понятието "съвместимост" за различните типове данни в C.

1.1.1 Съвместимост на аритметични типове

Два аритметични типа са съвместими само ако са еднакви. Следните аритметични са типове са еднакви.

short	≈ signed short	≈ short int	≈ signed short int
unsigned short	≈ unsigned short int		
signed	≈ signed int	≈ int[1]	
unsigned	≈ unsigned int		
long	≈ long int	≈ signed long	≈ signed long int
unsigned long	≈ unsigned long int		

Забележка[1]: В контекста на битовите полета простия тип int може да бъде както signed, така и unsigned.

Добавянето на квалификатор (const и/или volatile) променя типа. Например const int е различен от int и съответно не е съвместим с него.

1.1.2 Съвместимост на изброявания

Всяка декларация на enum-тип въвежда нов целочислен тип данни в програмата. Типът на enum-променливите е имплемнтационно-зависим. В зависимост от стойностите на enum-константите компилаторът може да избере най-подходящия от целочислените типове char (signed, unsigned), short (signed, unsigned) или int. В един сорс-файл не може да има две еднакви декларации на изброяване в една и съща област на видимост. Две декларации на изброяване в различни сорс-файлове са съвместими ако са изпълнени следните условия:

1. Имат еднакъв брой членове.

2. Имат еднакви имена на членовете.

3. Членовете имат еднакви стойности.

1.1.3 Съвместимост на структури

Всяка декларация на структура въвежда нов тип данни, който не е еднакъв или съвместим с всеки друг тип в същия сорс-файл. Две декларации на структури в различни сорс-файлове са съвместими, ако са изпълнени следните условия:

1. Имат еднакъв брой членове.

2. Имат еднакви имена на членовете.

3. Членовете са декларирани в еднакъв ред.

4. Членове имат съвместими типове.

5. Битовите полета имат еднаква ширина.

1.1.4 Съвместимост на обединения

Всяка декларация на обединение въвежда нов тип данни, който не е еднакъв или съвместим с всеки друг тип в същия сорс-файл. Две декларации на обединения в различни сорс-файлове са съвместими, ако са изпълнени следните условия:

1. Имат еднакъв брой членове.

2. Имат еднакви имена на членовете.

3. Членове имат съвместими типове.

4. Битовите полета имат еднаква ширина.

1.1.5 Съвместимост на масиви

Два еднакво квалифицирани масива са съвместими само ако типовете на техните елементи са съвместими. Ако са указани размерите и на двата

масива, те трябва да бъдат еднакви. Ако е указан размера само на единия масив или размерът не е указан и за двата масива, то двата масиви са съвместими.

Пример:

```
extern int a[];   /* масивът a е съвместим с масивите b и c    */
       int b[5];  /* масивът b е съвместим с масива a          */
       int c[10]; /* масивът c е съвместим с масива a          */
const  int d[10]; /* масивът d не е съвместим с другите масиви */
```

1.1.6 Съвместимост на типове на функции

1.1.6.1 Съвместимост на типове на функции без прототип

Два типа на функции са съвместими ако типовете на връщания резултат са съвместими.

Пример:

```
signed int foo();
int        foo();
```

1.1.6.2 Съвместимост на типове на функции с прототип

Две функции, декларирани в прототипна форма, са съвместими, ако следните условия са спазени:

1. Типовете на връщаният резултат трябва да са съвместими.

2. Броят на параметрите и използването на многоточие (...) трябва да съвпадат.

3. Съответните параметри трябва да имат съвместими типове. Имената на параметрите не е задължително да съвпадат.

Пример:

```
signed  foo(int x, double y,...);
int     foo(signed a, double b,...);
```

1.1.6.3 Съвместимост на типове на функции с и без прототипна форма

Ако само едната функция е в прототипна форма, то два типа са съвместими

ако следните условия са изпълнени:

1. Връщаните типове са еднакви.

2. Прототипът не трябва да включва многоточие (...).

3. Типът на всеки параметър в прототипа трябва да бъде съвместим с резултатния тип на аргументите след прилагането на обичайните преобразувания.

1.1.7 Съвместимост на указатели

Два еднакво квалифицирани указателя са съвместими, ако сочат към съвместими типове.

Пример:

```
short     * p;
short int * p;
```

1.2 Съставни (композитни) типове

Концепцията за композитни типове е необходима, тъй като езикът C позволява множество декларации на един и същ обект или функция. Две декларации на един и същ идентификатор трябва да бъдат съвместими. Няма изискване декларациите да бъдат лексикално (символ по символ) идентични. На практика композитните типове се конструират за съвместими типове, появяващи се в един и същ сорс-файл. Композитните типове не се конструират за съвместими типове, намиращи се в различни сорс-файлове. Следните правила са валидни за различните категории типове:

- Композитният тип на два съвместими аритметични типове е самия аритметичен тип.

Пример:

```
short
short int
```

Типовете short и short int са съвместими (защото са еднакви) и техният композитен тип е short int (или само short).

- Композитният тип на две съвместими изброявания е типа на изброяването.

- Композитният тип на два съвместими масиви е масив със композитен тип на елементите. Ако единия от масивите специфицира константен размер, композитният тип има този размер, в противен случай размерът е неспецифициран.

Пример:

```
extern int    a[];
extern signed a[42];
```

```
signed int a[42]; /* Композитен тип */
```

- Композитният тип на два съвместими типове функции (без прототипна форма) е тип на функция с композитен връщан тип.

Пример:

```
signed  f();
int     f();
```

```
signed int f(); /* Композитен тип */
```

- Композитният тип на два съвместими типове функции (в прототипна форма) е тип на функция, чийто параметри имат композитен тип, със същата употреба на многоточие (...), и със композитен връщан тип.

Пример:

```
int f(int (*) (), double (*) [3],...);
int f(int (*) (char *), double (*)[],...);
```

```
int f(int (*) (char *), double (*)[3],...); /* Композитен тип */
```

- Композитният тип на два съвместими типове функции (в и без прототипна форма) е тип на функция в прототипната форма и със композитен връщан тип.

Пример:

```
signed f();
int    f(int (*) (char *), double (*)[]);
```

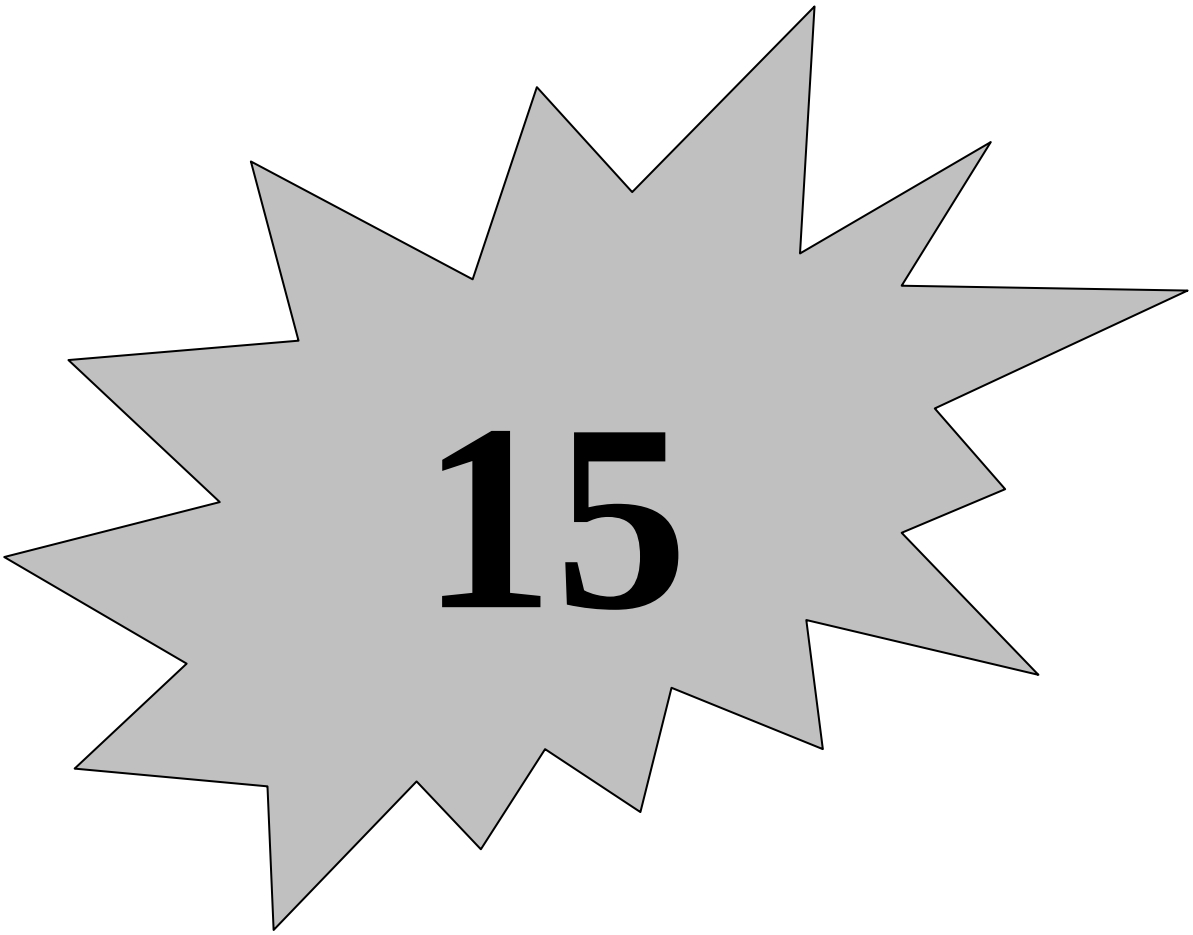
```
signed int f(int (*) (char *), double (*)[3]); /* Композитен тип */
```

- Композитният тип на два съвместими указателни типове е указателен тип към композитен тип.

Пример:

```
extern int    * p;
extern signed * p;
_____________

signed int * p /* Композитен тип */
```



Декларации

1.1 Определение

Тази глава е обобщение на понятието декларация. Декларацията играе главна роля в C програмирането. Тя определя значението и свойствата на един или повече идентификатори. Идентификаторите в декларацията могат да обозначават следните неща:

- имена на обекти (променливи от всички допустими C типове)
- имена на функции
- етикети на структури, обединения, и изброявания (**tags**)
- typedef имена на типове (синоними на типове).
- имена на членове на структури, обединения и изброявания
- етикети на оператори (**labels**)
- имена на макроси.
- имена на параметри в макро-функции

Освен етикети на оператори, макроси и параметри на макро-функции, всички други идентификатори трябва да бъдат декларирани (преди да мога да се използват за първи път) или извън всички функции (външни декларации) или в началото на блок (вътрешни декларации).

Етикетите използвани като цел на оператора goto могат да бъдат поставяни пред всеки оператор. Те се декларират по-подразбиране в мястото където се появяват.

Макросите и параметрите в макро-функциите се декларират чрез предпроцесорната директива #define.

След като един идентификатор е деклариран, той може да се използва във всички изрази до края на областта си на видимост.

1.2 Обобщен синтаксис на декларации

Декларациите могат да се появяват на няколко места в една C програма. Мястото на декларацията влияе на нейните свойства.

Обобщен синтаксис на декларация

```
клас-памет  квалификатор-тип  спецификатор-тип  декларатор  =  инициализатор;

където

клас-памет
      auto
      register
      static
      extern
      typedef

квалификатор-тип
      const
      volatile
      const volatile
      volatile const

спецификатор-тип
      void
      signed char,
      unsigned char
      short, signed short, short int, signed short int
      unsigned short, unsigned short int
      int, signed, signed int
      unsigned int, unsigned
      long, signed long, long int, signed long int
      unsigned long, unsigned long int
      float
      double
      long double
      struct
      union
      enum
      typedef-име

декларатор
      прост-декларатор
      декларатор-на-указател
      декларатор-на-масив
      декларатор-на-функция

инициализатор
      Задава първоначална стойност на идентификатора, съдържан в декларатора.
      Формата на инициализатора зависи от вида на декларацията. Само деклларации,
      които дефинират променливи могат да имат инициализатори.
```

1.3 Декларация и дефиниция

Един идентификатор може да има много декларации, но само една дефиниция. Декларацията се превръща в дефиниция в следните случаи:

- Декларация на обект е дефиниция, ако декларацията води до заделяне на памет за обекта. Декларации, които включват инициализатори винаги са дефиниции. Всички декларации във функция или в съставен оператор

са дефиниции, освен когато съдържат спецификатор за клас памет extern.

Пример:

```
int a = 10;              /* Дефиниция на a */
extern int i = 10;       /* Дефиниция на i */
extern double b[];       /* Декларация на масив, дефиниран някъде в програмата */

void func( )
{
  extern char c;         /* Декларация на c, дефинирана някъде в програмата */
  static short d;        /* Дефиниция на d */
  float e;               /* Дефиниция на e */
  /* ... */
}
```

- Декларация на обект, разположена извън функции, без инициализатор, и без спецификатор за клас памет extern, се нарича временна дефиниция (**tentative definition**).

Пример:

```
int i, v[];                /* Временни дефиниции на  i, v and j. */
static int j;
```

Ако сорс-файлът, който съдържа такава дефиниция, съдържа друга дефиниция на същия идентификатор, то временната дефиниция се превръща просто в декларация. В противен случай, компилаторът преобразува временната дефиниция в дефиниция, добавяйки подразбиращ се инициализатор 0. Например горните временни дефиниции са еквивалентни на следните дефиниции:

```
int i = 0, v[1] = 0;
static int j = 0;
```

Декларация на функция е дефиниция ако декларацията включва и тялото на функцията.

Пример:

```
long sum (int a, int b);  /* Декларация на sum */

long sum (int a, int b)   /* Дефиниция на sum */
{
      return (a + b);
}
```

1.4 Свързване

1.4.1 Въведение

Свързването е процес, който определя кои декларации, съдържащи еднакъв идентификатор, сочат към едно и също нещо. Терминът "свързване" отразява факта, че идентификаторите декларирани в различни файлове трябва да бъдат свързани, ако сочат към общо нещо.

Идентификаторите в C могат да имат следните видове свързване:

- външно (**external linkage**)
- вътрешно (**static linkage**)
- без свързване (**no linkage**)

Видът на свързването на идентификатора се определя от мястото на декларацията и класа памет ако има такъв.

Съгласно стандарта C две декларации, които сочат към едно и също нещо, трябва да са съвместими, в противен случай поведението е недефинирано.

Процесът на свързване се извършва от програма наречена Линкер (Linker).

1.4.2 Външно свързване

Декларации на функции и глобални обекти, включващи спецификатор за клас памет extern имат външно свързване.

Пример:

```
extern int x;
extern long sum (int a, int b);
```

Това правило има едно изключение: ако един идентификатор е бил вече деклариран с вътрешно свързване, втора декларация на идентификатора в същата област на видимост като първата не може да промени свързването на идентификатора да е външно.

Пример 15-1:

```
static int x; /* Променливата x е с вътрешно свързване */

int main(void)
```

```
{
      extern int x;

      x = 10;

      return 0;
}
```

Идентификаторите с външно свързване могат да имат много декларации, разположени в различни файлове, но могат да имат само само една дефиниция.

Функции и глобални обекти декларирани без спецификатор за клас памет се третират от компилатора, все едно че имат клас памет extern, и имат външно свързване.

Пример:

```
int x; /* Същото като extern int x */
long sum (int a, int b); /* Същото като extern long sum (int a, int b); */
```

1.4.3 Вътрешно свързване

Функция или обект декларирани със спецификатор за клас памет static имат вътрешно свързване. Идентификаторите с вътрешно свързване не са видими за Линкера и са достъпни само във файла, в който са декларирани. Това означава, че други файлове също могат да декларират идентификатори със същите имена при условие, че те също са със вътрешно свързване. Казано с други думи не може да декларирате в един файл идентификатор с вътрешно свързване, а в друг файл същия идентификатор с външно свързване.

Пример 15-2:

```
/* сорс-файл1 */
#include <stdio.h>

static int x;

extern void foo(void);

int main(void)
{
    x = 10;

    printf("main::x = %d\n", x);
    foo();
    printf("main::x = %d\n", x);

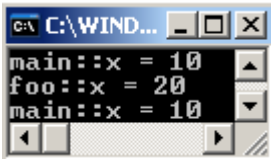
    return 0;
}
```

```
/* сорс-файл2 */
#include <stdio.h>

static int x;

void foo(void)
{
    x = 20;
    printf("foo::x = %d\n", x);

    return;
}
```



1.4.4 Без свързване

Идентификатори, които се намират в следните декларации нямат свързване:

- декларация на тип (**struct**, **union** и **enum**)
- декларации на етикети на оператори
- typedef декларации
- декларации на макроси
- декларации на параметри на функции
- декларации на обекти във функция или съществен оператор без спецификатор за клас памет или със спецификатр за клас памет auto.

Пример 15-3:

```
struct MyStruct
{
	int a;
	int b;
};

union Myunion
{
	int a;
	double b;
};

enum MyEnum
{
	ONE,
	TWO
};

#define MyMACRO    (10)

typedef unsigned char uint8;

int main(void)
{
	uint8 local1;
	auto uint8 local2;

	goto label;

	/*...*/

label:
	/*..*/

	return 0;
}
```

1.5 Декларатори

Деклараторът декларира идентификатор и добавя допълнителна информация за типа. Декларациите биват четири вида:

- прост-декларатор
- декларатор-на-указател
- декларатор-на-масив
- декларатор-на-функция

1.5.1 Прости декларатори

Простите декларатори се използват да дефинират променливи от аритметични типове, изброявания, структури, обединения, или typedef типове, синоними на някой от предходните типове. Спецификаторът за тип на един прост декларатор съдържа цялата необходима информация за типа. По тази причина простия декларатор се състои само от идентификатор.

Прост декларатор

```
идентификатор
```

Ако T е спецификатор за тип на някой от изброените по-горе типове и id е идентификатор, то декларацията

T id;

указва, че id е от тип T и id представлява прост декларатор.

Пример:

```
int                     x;
enum   {ON, OFF}        x;
struct {int a, double b} x;
union  {int a, double b} x;
typedef int integer;
integer                 x;
```

1.5.2 Декларатори на указатели

Указателните декларатори се използват за деклариране на променливи от

указателни типове.

Декларатор на указател

```
* квалификатор-тип_опц  идентификатор

където

квалификатор-тип
      const
      volatile
      const volatile
      volatile const
```

Ако T е спецификатор за тип и id е идентификатор, то декларацията

T *id;

указва, че id е указател към T, а конструкцията *id представлява декларатор на указател.

Пример:

```
int   *x;
int * const x;
void  *x
```

1.5.3 Декларатори на масиви

Декларациите на масиви се използват за деклариране на променливи, които са масиви.

Декларатор на масив

```
идентификатор [ константен-израз_опц ]

където

константен-израз
      Целочислен константен израз със стойност по-голяма от 0, определящ броя на
      елементите на масива, който може да се изпусне ако декларацията включва и
      инициализатор.
```

Ако T е спецификатор за тип и id е идентификатор, а E е целочислен константен израз, то декларацията

T id[E];

указва, че id е масив от T, а конструкцията id[E] представлява декларатор на масив.

Пример:

```
int    x[5];
```

1.5.4 Декларатори на функции

Декларатор на функция

```
идентификатор (списък-параметри)

където

списък-параметри
      Съдържа декларации на формални параметри.
```

Ако T е спецификатор за тип и id е идентификатор, а P е списък с формални параметри, то декларацията

T id (P);

указава, че id е функция, която връща тип T и има списък с параметри P, а конструкцията id(P) се нарича декларатор на функция

Пример:

```
int    x (int i, double d);
```

1.5.5 Дешифриране на сложни декларатори

Декларациите в C могат да комбинират *, () и [], вследствие на което могат да се получат изключително сложни декларации. В такива сложни декларации не винаги е лесно да се определи веднага дали идентификаторът е "указател към...", "масив от...", или "функция, връщаща...". В тази точка се разглеждат правилата, по които да дешифрирате (също може и да създавате) сложни декларации. Най-общо казано дешифрирането се състои в зигзагообразно интерпретиране на символите вдясно и вляво на идентификатора в декларацията. Езикът C не поддържа декларации, които се интерпретират като:

- неуказателна променлива тип void
- масив от функции или масив от void
- функция, връщаща масив

- функция връщаща като резултат друга функция

Когато се дешифрира декларация, която съдържа сложен декларатор, следните символи се дешифрират като:

* - "указател към..." и винаги е разположен вляво на идентификатора

[] - "масив от..." и винаги е разположен вдясно на идентификатора

() - "функция, връщаща..." и винаги е разположен вдясно на идентификатора

Друго важно нещо, което трябва да се вземе предвид са приоритетите на декларatорите. Декларатори на масив и функция имат еднакъв приоритет (асоциативност отляво надясно), който е по-висок от този на декларатор на указател. Приоритетът може да се променя с помощта на скоби.

Разгледайте следните примери:

```
int *x[10];
```

За един начинаещ C програмист идентификаторът x в декларатора *x[10] може да бъде както "указател към..." така и "масив от...". От правилата за приоритетите следва, че декларatорът е еквивалентен на *(x[10]), т.е x е масив от 10 елемента. Типът на елементите е указател.

Нека сега променим горната декларация както е показано по-долу.

```
int (*x)[10];
```

Скобите около идентификатора x в случая ясно дефинират приоритета, т.е декларatорът е *x и се дешифрира като указател към тип int [10], т.е. едномерен масив от 10 елемента int.

Същата е ситуацията и в следните примери:

```
int *x(void);
```

Декларatорът в тази декларация е *x(void). От правилата за приоритет следва, че това е еквивалентно на *(x(void)), т.е x е функция, която не приема параметри и връщаща указател към int.

Нека сега променим горната декларация както е показано по-долу.

```
int (*x)(void);
```

Скобите около идентификатора x ясно дефинират приоритета, т.е

декларaторът е *x и се дешифрира като указател към функция, която не приема параметри и връща int.

Точката и запетая, с която завършва всяка декларация не са част от нея и не участва в процеса на дешифриране.

Тъй като правилата за дешифиране на декларации е много трудно да се опишат с думи, по-долу е дадено само опростено описание. Примерите след това би трябвало да изяснят тези правила на практика.

Правила за дешифриране на декларации:

1. Намерете идентификатора в декларацията.

2. Намерете символа вдясно от идентификатора и го дешифрирайте. Продължете така докато изчерпите всички символи или стигнете затваряща скоба).

    a) Ако първият символ е [, идентификаторът е "масив от...".

    b) Ако първият символ е (, идентификаторът е "функция връщаща...".

    c) Ако първият символ е) или няма нищо, се преминава към първия символ вляво на идентификатора. Ако той е *, идентификаторът е "указател към...".

3. Намерете символа вляво от идентификатора и го дешифрирайте. Продължете така докато изчерпите всички символи или стигнете отваряща скоба (.

4. Повторете стъпки 2 и 3 докато дешифрирате цялата декларация

Пример:

```
int (*x)(void);
```

1) Намира се идентификатора, int (*x)(void), **x е…**
2) Намира се първия символ вдясно, int (*x)(void)
3) Намира се първия символ вляво, int (*x)(void), **x е указател към...**
4) Намира се следващия символ вляво, int (*x)(void)
5) Намира се следващия символ вдясно, int (*x)(void), и се дешифрира всичко докато се срещне затваряща скоба или нищо, **x е указател към функция, която не приема аргументи...**
6) Намира се следващи символ вляво, int (*x)(void), **x е указател към функция, която не приема аргументи и връща int**

Следващите примери използват стъпките посочени в горния пример за да дешифрират декларацията.

Примери:

1) `int ident;`

   ident е int.

2) `int ident [10];`

   ident е масив от 10 елемента int.

3) `int ident [3][10];`

   ident е масив от 3 едномерни масива всеки с 10 елемента int. Казано по друг начин ident е двумерен масив.

4) `int * p;`

   ident е указтел към int.

5) `int * ident [10];`

   ident е масив от 10 указатели към int.

6) `int * ident [3][10];`

   ident е масив от 3 едномерни масива от 10 указатели към int, т.е iedent е двумерен масив от указатели към int.

7) `int ( * ident ) [10];`

   ident е указател към масив от 10 елемента int.

8) `int ( * ident ) ( double x );`

   ident е указател към функция от вида int foo(double).

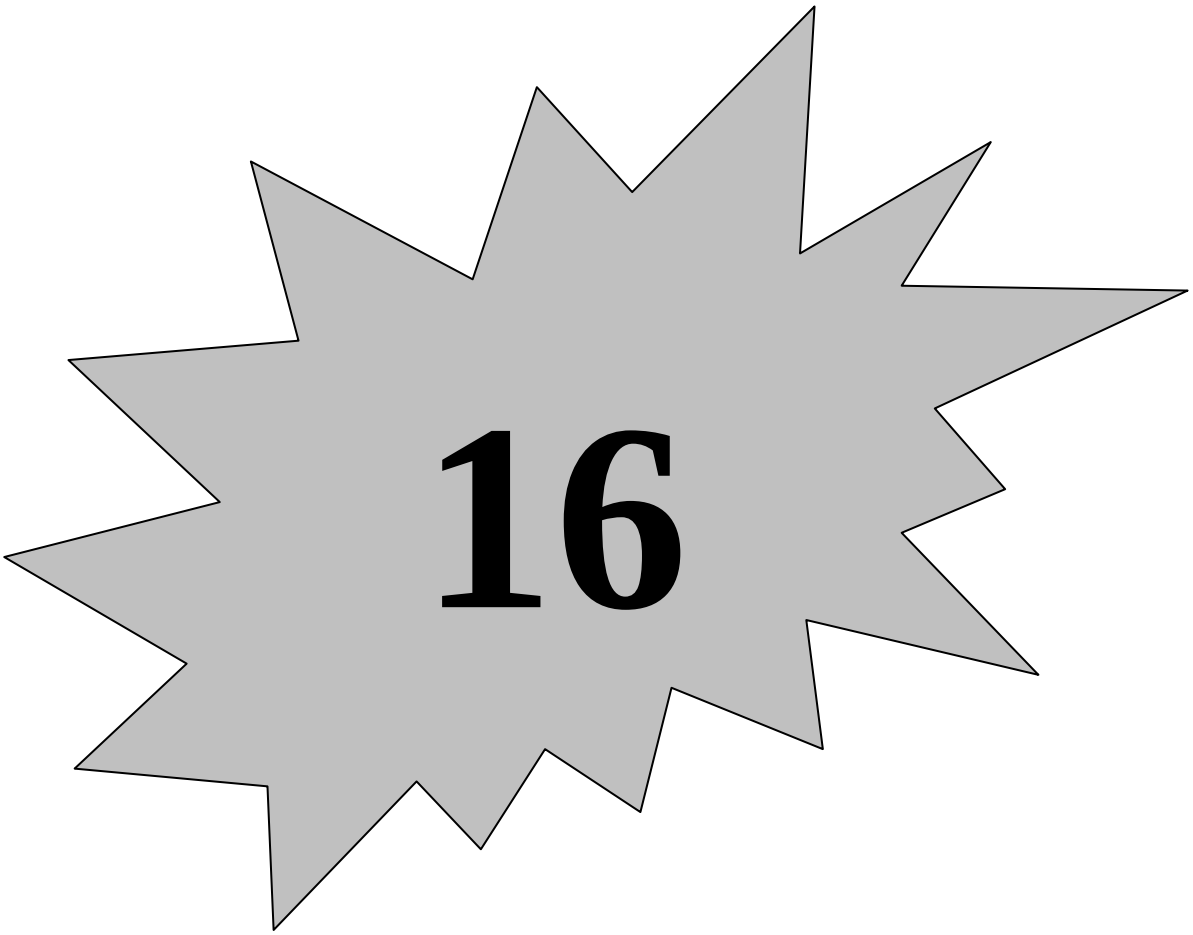
9) `float ( * ident (void) ) [3][10];`

   ident е функция, която връща указател към двумерен масив и не приема аргументи.

10) `char *(*(**ident[2][4])(void))[5];`

    ident е масив от 2 едномерни масиви. Всеки едномерен масив съдържа 4 елемента , които са указатели, които сочат към указател-към-функция, която не приема аргументи и връща указател към

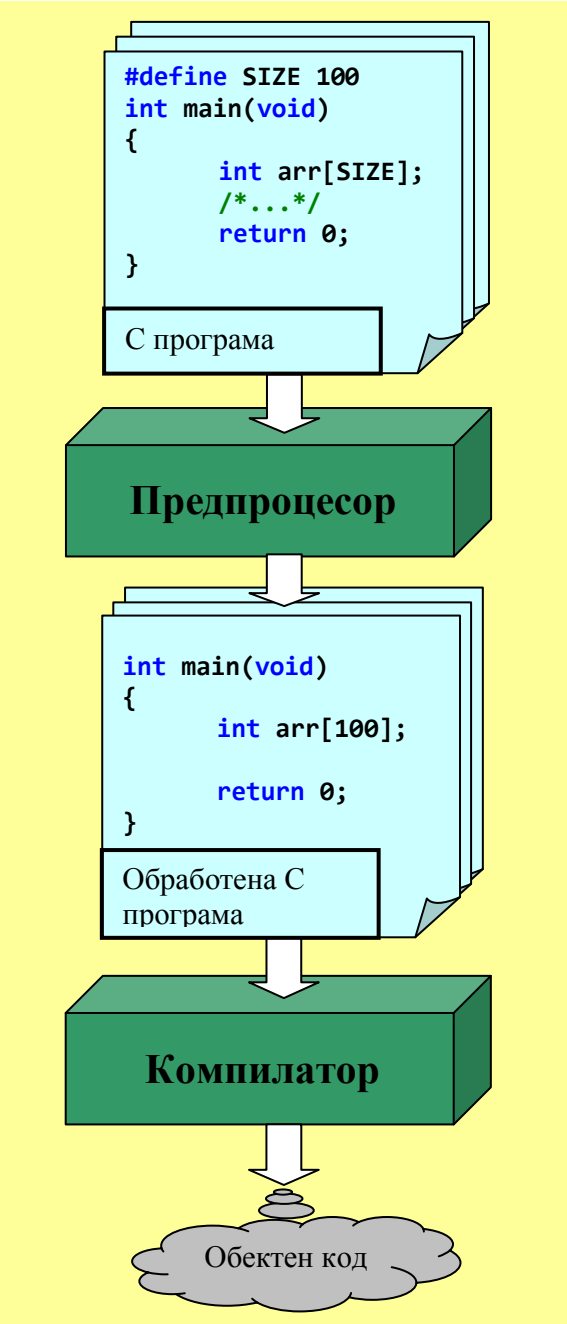
масив от 5 елемента, които са указатели към тип char.



Предпроцесор

1.1 Общи сведения

Предпроцесорът е програма, която обработва текста на сорс-файловете преди компилатора. Фиг. 92 описва този процес.



Фиг. 95 Предпроцесор

Предпроцесорът се управлява от предпроцесорни команди, наречени още предпроцесорни директиви, които започват със символа #.

Предпроцесорна команда

#предпроцесорна-команда

Освен предпроцесорни команди предпроцесорът включва и три

предпроцесорни оператора: defined, # и ##.

Табл.48 показва всички предпроцесорни команди и оператори.

№	Команда	Описание
1	**#define**	Дефиниране на макрос
2	**#undef**	Отдефиниране на макрос
3	**#include**	Вмъкване на текст от друг файл
4	**#if**	Директиви за условно компилиране
5	**#ifdef**	
6	**#ifndef**	
7	**#else**	
8	**#endif**	
9	**#elif**	
10	**defined**	Предпроцесорен оператор за условно компилиране
11	**#line**	Задаване на номер на ред
12	**#error**	Генериране на съобщение за грешка по веме на компилиране
13	**#pragma**	Имплементационно-зависими директиви
14	#	Предпроцесорен оператор за създаване на низ
15	##	Предпроцесорен оператор за съединяване на предпроцесорни лексеми

Табл. 48 Предпроцесорни команди и оператори

Всяка предпроцесорна команда се намира самостоятелно на един ред. Символът за начало на командата # може да се предшества само от символи за интервали и хоризонтална табулация. Краят на директивата се отбелязва със символ за нов ред. Между символа # и директивата също може да има символи за интервали и хоризонтална табулация, т.е

```
        #предпроцесорна-команда
```

или

```
#       предпроцесорна-команда
```

1.2 Нулева директива (#)

Самостоятелното използване на символа # се нарича нулева директива. Нулевата директива няма никакъв ефект и се игнорира от компилатора.

1.3 Директива за включване на файл (#include)

Директивата **#include** инструктира компилатора да вмъкне съдържанието на специфицирания файл в мястото на извикване на директивата. Тя има три форми.

Директива #include

```
#include  <име-файл>                1
#include  "име-файл"                 2
#include  предпроцесорни-лексеми     3
```

При първата форма компилаторът търси файла в стандартни места, дефинирани от имлементацията на компилатора. Типично това са директориите, съдържащи библиотечните хедър-файлове на компилатора като <stdio.h>, <math.h> и т.н. Тази форма по правило се използва от програмистите за включване на библиотечни хедър-файлове.

При втората форма компилаторът първо търси файла в места дефинирани от имлементацията на компилатора, но различни от тези дефинирани с първата форма. Типично тези места са директориите, съдържащи сорс файловете на програмата. Ако файлът не бъде намерен в тях, компилаторът претърсва стандартните места както при първата форма. Тази форма по правило се използва от програмистите за включване на хедър-файлове създадени от самите тях.

Третата форма се подлага на макро разширяване и резултатът трябва да е някоя от първите две форми.

Пример 16-1:

```
#ifdef _DEBUG_
  #define MY_HEADER "myProject_dbg.h"
#else
  #define MY_HEADER "myProject.h"
#endif

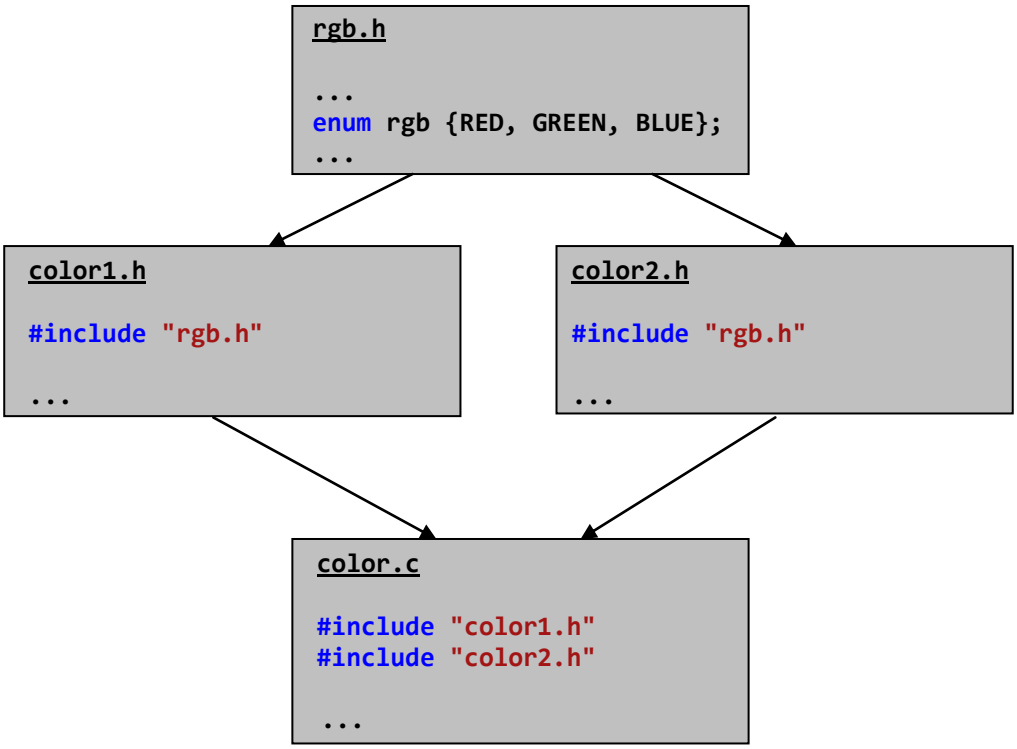
#include MY_HEADER
```

Синтаксисът за **име-файл** е имплементационно зависим, но стандтартът C изисква всички имплементации да позволяват име, състоящо се от букви и цифри (започващо с буква), последвано от точка и единичен символ. C89 изисква само първите 6 символа от името на файла да са значими. Отчитането на регистъра на буквите също зависи от имплементацията.

Файлът, който се добавя с директивата #include, може от своя страна също да добавя друг файл. C89 допуска поне 8 нива на вложеност на

директивата #include.

Може да се случи така, че един хедър-файл да бъде включен повече от веднъж. Например в следващия пример файловете color1.h и color2.h включват файла rgb.h. Сорс-файлът color.c включва и двата файла color1.h и color2.h. По този начин rgb.h файлът се оказва два пъти включен в color.c (Фиг.93).



Фиг. 96 Многократно включване на хедър-файл

Това ще доведе до грешка при компилиране на файла color.c, тъй като типа enum rgb се оказва деклариран два пъти, което не е позволено.За да се избегне това, всеки хедър-файл трябва да се защити от многократно включване по следния начин.

include_file.h

```
#ifndef INCFILE_H
#define INCFILE_H

...

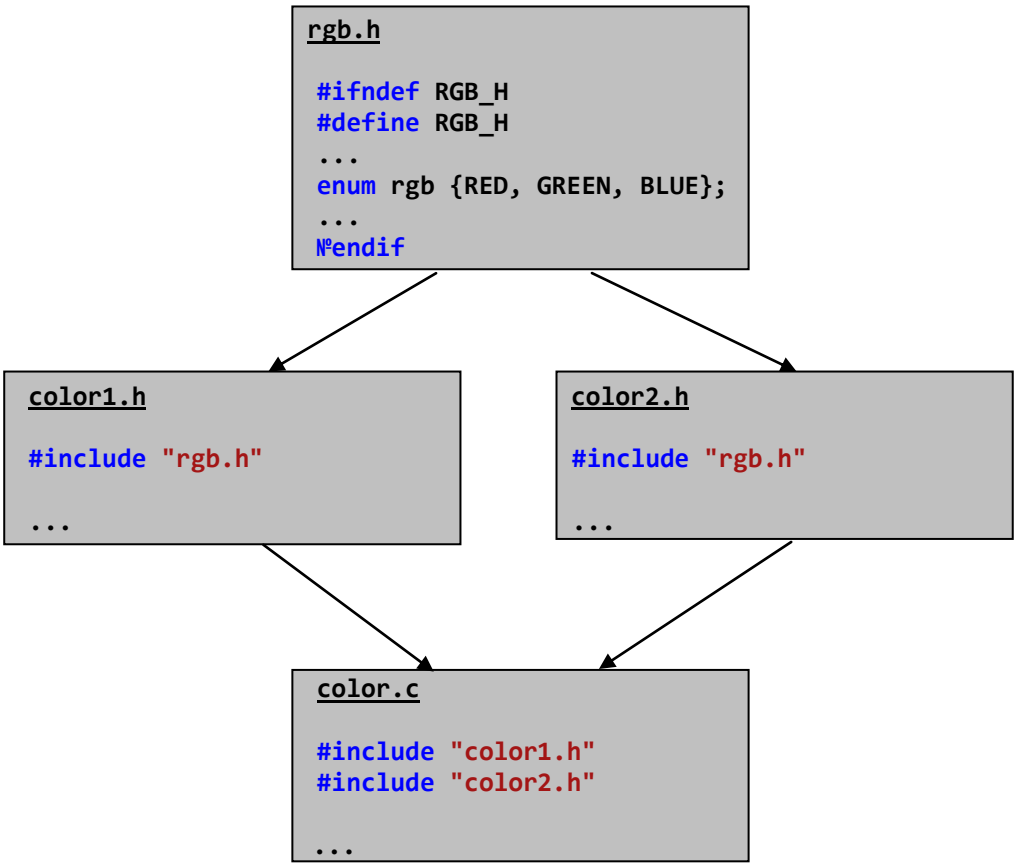
/* ...Съдържанието на хедър-файла се добавя тук... */

...

#endif
```

Първото включване на хедър-файла include_file.h с директивата #include дефинира макроса INCFILE_H. При всяко по нататъшно включване на същия файл (косвено или директно) неговото съдържание повече не се добавя към текущия файл, тъй като условието #ifndef INCFILE_H вече не

е изпълнено. Директивите #ifndef и #endif са описани по-надолу в настоящата глава. Фиг.94 представя коригирана версия на предходния пример.



Фиг. 97 Защита на хедър-файлове от многократно включване

1.4 Директива за дефиниране на макроси #define

1.4.1 Описание

Директивата #define се използва за задаване на име на някакъв текст, например константа или оператор. Името (идентификатора, разположен непосредствено след #define), с което се идентифицира текста, се нарича макрос. Когато предпроцесорът срещне макрос някъде в сорс-кода, той го заменя със свързания с него текст, наричан още тяло на макроса. Заменението текст след това отново се сканира от предпроцесора за наличието на други макроси, които трябва да бъда заменени. Вмъкването на име на макрос в кода се нарича извикване на макрос, а процеса на замяна на макрос с неговото тяло се нарича разгъване (или разширяване) на макрос.

Директивата #define има две форми:

1. Първата форма се нарича макрос без параметри или макро-обект (**object-like macro**)

Дефиниране на макрос без параметри (макро-обект)

```
#define име-макрос заместващ-текстопц

където

заместващ-текстопц

      Може да се пропусне или да бъде всеки текст, който след замяната формира
      валидна С конструкция.
```

Когато предпроцесорът срещне **име-макрос** в кода, той го заменя с **заместващ-текст**. Заместващият текст може да бъде всеки текст, стига след замяната да се получава валидна С конструкция. Между **име-макрос** и **заместващ-текст** трябва да има поне един интервал за разделител. Интервалите преди и след **заместващ-текст** не са част от тялото на макроса. Типичната употреба на макро-обектите е задаване на подходящи имена на литерални константи, за избягване на т.нар. "магически числа" [1] в кода.

Забележка[1]: Понятието "магически числа" означава използването на числови стойности в кода с неясно значение.



Избягвайте използването на "магически числа" в кода. Вместо това използвайте макроси с подходящи имена, подсказващи значението на числото.

Пример 16-2:

```
#include <stdio.h>

#define PI 3.14159 /* Дефиниране на макрос */

int main(void)
{
      double radius;

      printf("Enter the circle radius: ");
      scanf("%lf", &radius);

      printf("The circle fase is %f\n", PI*radius*radius);

      return 0;
}
```

Извикване и разгъване на макроса:

⇩

3.14159*radius*radius

Пример 16-3:

```
#include <stdio.h>

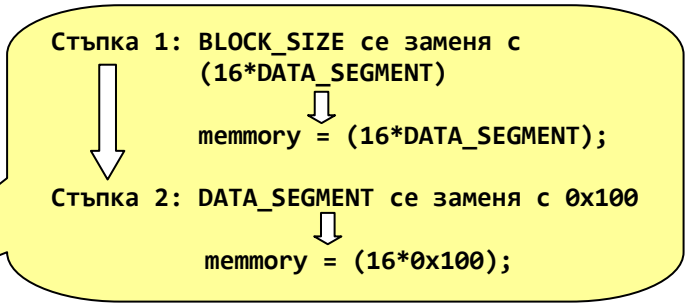
#define DATA_SEGMENT 0x100
#define BLOCK_SIZE (16*DATA_SEGMENT)

int main(void)
{
	unsigned long memmory;

	memmory = BLOCK_SIZE;

	/*...*/

	return 0;
}
```



Друга типична употреба на макро-обектите е задаването на размер на масив. Ако е необходимо после да се промени този размер, това ще стане само в дефиницията на макроса. Промяната ще се отрази автоматично навсякъде където се използва размера на масива.

Пример 16-4:

```
#define ARRAY_SIZE 100

int main(void)
{
	int data[ARRAY_SIZE];
	int i;

	for (i = 0; i < ARRAY_SIZE ; i++)
	{
		data[i] = 0;
	}

	/*...*/

	return 0;
}
```

Тялтото на макрос може да бъде празно. В този случай директивата #define дефинира идентификатор, който може да се използва от директивите за условно компилиране (директивите за условно компилиране са разгледани по-надолу в тази глава).

2. Втората форма се нарича макрос с параметри или макро-функция (**function-like macro**)

Дефиниране на макрос с параметри (макро-функция)

```
#define име-макрос(списък-параметри_опц) заместващ-текст_опц
```

където

списък-параметри$_{опц}$
Идентификатори, представляващи имена на формални параметри, разделени със запетая. Може да бъде и празен.

заместващ-текст$_{опц}$
Може да се пропусне или да бъде всеки текст, който след замяната формира валидна C конструкция. Параметрите от **списък-параметри** могат да се използват многократно в заместващия текст (или може въобще да не се използват).

Лявата скоба **(** трябва да следва непосредствено името на макроса, в противен случай скобата и всичко след нея става част от заместващия текст, а не от името на макроса. Макросите с параметри наподобяват функции. Извикването на макроса става като името на макроса се последва от списък с действителни аргументи, разделени със запетая, затворен в скоби. Когато предпроцесорът срещне такава конструкция в сорс-кода, той заменя макроса с неговото тяло. Ако в тялото се използват формални параметри, те се заменят с действителните аргументи преди разгъването на макроса. Видимостта на формалните параметри е ограничена до тялото на макроса.

Пример 16-5:

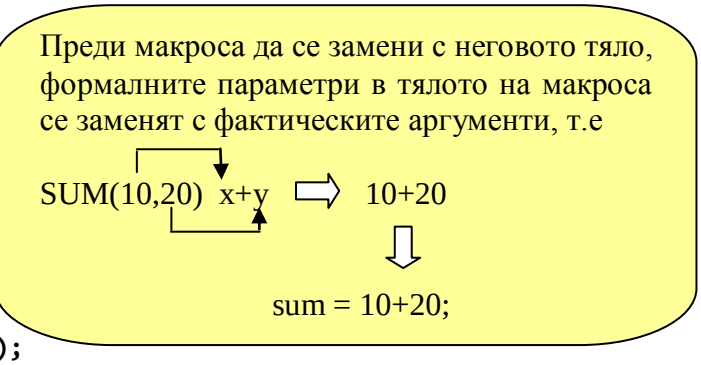
```
#include <stdio.h>

#define SUM(x,y) x+y

int main(void)
{
	int sum;

	sum = SUM(10,20);
	printf("sum = %d\n", sum);

	return 0;
}
```





Когато макро-функцията се извиква, между името на макроса и лявата скоба може да има интервал, т.е следните две конструкции са еквивалентни:

```
sum = SUM(10,20);
```

И

```
sum = SUM   (10,20);
```

Формалните аргументи могат да бъдат и изрази. В този случай обаче е

възможно действителния резултат да се различава от очаквания. Например разгледайте следния код:

Пример 16-6:

```
#include <stdio.h>

#define MUL(x,y) x*y

int main(void)
{
        int mul;

        mul = MUL(2+3,2);
        printf("mul = %d\n", mul);

        return 0;
}
```

Преди макросът да се замени с неговото тяло, формалните параметри в тялото на макроса се заменят с фактическите аргументи, т.е

MUL(2+3,2) x+y ⇨ 2+3*2

⇩

mul = 2+3*2; /* 8 */



Резултатът от предходната програма е 8, а вероятно сте очаквали 10. За да се избегне този проблем се препоръчва всички формални параметри в заместващия текст да се обграждат със скоби включително и целия заместващ текст.

Пример 16-7:

```
#include <stdio.h>

#define MUL(x,y) ((x)*(y))

int main(void)
{
        int mul;

        mul = MUL(2+3,2);
        printf("mul = %d\n", mul);

        return 0;
}
```

Преди макросът да се замени с неговото тяло, формалните параметри в тялото на макроса се заменят с фактическите аргументи, т.е

MUL(2+3,2) ((x)+(y)) ⇨ ((2+3)*(2))

⇩

mul = ((2+3)*(2)); /* 10 */





Винаги заграждайте формалните параметри в заместващия текст в скоби, включително и целия заместващ текст.

Списъкът с параметри на една макро-функция може да бъде празен. Извикването на такъв макрос наподобява извикване на функция без параметри.

Пример 16-8:

```
#include <stdio.h>

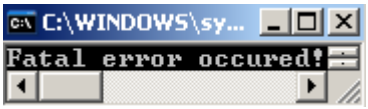
#define FatalErrorMessage() printf("Fatal error occured!\n")

int main(void)
{
	/*...*/

	FatalErrorMessage();

	/*...*/

	return 0;
}
```



Използването на макро-функция вместо нормална функция има следните предимства:

- **Увеличава се бързодействието на програмата**

  Извикването на макро-функция избягва всички необходими действия, като съхраняване на адреса на връщане, копиране на аргументите в параметрите и т.н, които се извършват при извикване на нормалните функции.

- **Макросите са по-универсални**

  Формалните параметри на макро-функцията, за разлика от тези на една нормална функция, нямат тип. Като резултат една макро-функция може да се извика с аргументи от всякакъв тип при условие, че след замяна се получава валидна C конструкция. Например следния макрос

  ```
  #define max(x,y) (((x)>(y))?(x):(y))
  ```

  може да се използва за намирането на по-голямото от две числа от всички целочислени типове, double и float.

Използването на макро-функция вместо нормална функция има следните недеостатъци:

- **Увеличава се кода на програмата**

  Многократното извикване на една макро-функция води до

многократно вмъкване на нейното тяло в програмата, което води до увеличаване на кода.

- **Макро-функциите не са безопасни**

  Когато една функция се извика, компилаторът проверява типа на всеки аргумент дали съответства на типа на параметъра. Ако не, типът на аргумента се преобразува до този на парамтъра, а ако това е невъзможно компилаторът генерира грешка. Параметрите на макро-функциите нямат тип и съответно не се извършва никаква проверка и преобразуване на типове. Това може да доведе до непредвидени резултати.

- **Не е възможно дефинирането на указател към макрос**

  Концепцията с указател към функция, която може да е много полезна в определени ситуации, е неприложима към макро-функциите.

- **Аргумент на макро-функция може да се изчисли повече от веднъж**

  Една нормална функция изчислява аргументите си само веднъж, докато една макро-функция може да изчисли аргументите си повече от един път. Това може да доведе до непредвидени резултати, ако аргументите включват странични ефекти. Например следния израз

```
n  = max(i++,  j);
```

  се разгъва до

```
n = (((i++)>(j))?(i++):(j));
```

  Ако i е по-голямо от j (например j = 1, i = 2), i ще се увеличи два пъти и на n ще се присвои стойност различна от очакваната. Също така на пръв поглед може да се очаква, че i ще бъде 3 след изпълнение на този израз, докато на практика поради двойното увеличение i е 4.

- **Макро-функиите са по-трудни за проследяване и откриване на грешки (дебъгване) в сравнение с нормалните функции**



Въпреки че не задължително, широко възприето е имената на макросите да се пишат с главни букви. Това се прави с цел по-лесно да се различават от имена на променливи (за макро-обекти) и от функции (за макро-функции).

1.4.2 Предефиниране на макроси

Технически погледнато заместващият текст на макрос се състои от т.нар. предпроцесорни лексеми (вижте Глава 1 Лексикални елементи). Предпроцесорните лексеми са отделни "думи", разделени с един или повече интервали. С позволява един макрос да бъде дефиниран повече от веднъж при условие, че всяка дефиниция се състои от еднакъв брой идентични лексеми. Например:

Следните двойки дефиниции не са еднакви:

```
#define FOUR 2*2   /* Заместващият текст се състои от една лексема: 2*2     */
#define FOUR 2 * 2 /* Заместващият текст се състои от три лексеми: 2, * и 2 */

#define FUNC(b) ( a ) /* Заместващият текст се състои от 3 лексеми: (, a и ) */
#define FUNC(b) ( b ) /* Заместващият текст се състои от 3 лексеми: (, b и ) */
```

Следните двойки дефиниции са еднакви:

```
#define FOUR 2 *    2  /* Заместващият текст се състои от три лексеми: 2, * и 2 */
#define FOUR 2 * 2     /* Заместващият текст се състои от три лексеми: 2, * и 2 */

#define FUNC(b) ( a )    /* Заместващият тескст се състои от 3 лексеми:(, a и ) */
#define FUNC(b) (  a   ) /* Заместващият тескст се състои от 3 лексеми:(, a и ) */
```

За защита от многократно дефиниране на един и същ макрос може да се използва директивата #ifndef.

```
#ifndef име-макрос
#define име-макрос заместващ-текст_опц
#endif
```

Пример:

```
#ifndef PI /* Проверка дали макроса PI е вече дефиниран с директивата #define */
#define PI 3.14159 /* Ако не, макроса PI се дефинира */
#endif /* Край на директивата #ifndef */
```

1.4.3 Повторно сканиране и замяна на макроси

Когато предпроцесорът срещне име на макрос в кода, той го заменя с неговото тяло. Ако макросът е макро-функция, първо параметрите в тялото се заменят с аргументите и тогава макро-функцията се заменя с нейното тяло. След това предпроцесорът сканира отначало резултатната последователност от лексеми за наличието на други имена на макроси, които да бъдат заменени. Процесът се повтаря докато резултатната последователност от лексеми не съдържа повече макроси. Например нека имаме следните две макро дефиниции:

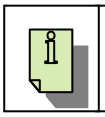
```
#define plus(x,y) add(y,x)
#define add(x,y)  ((x)+(y))
```

Някъде в кода първия макрос се извиква по следния начин:

```
plus(plus(a,b),c)
```

Последователността на разгъване на макроса е показана по-долу:

```
1. plus(plus(a,b),c) /* Първоначално извикване на макроса            */
2. add(c,plus(a,b))  /* Първоначално разгъване на макроса            */
3. ((c)+(plus(a,b))) /* Повторно сканиране и разгъване  на макроса   */
4. ((c)+(add(b,a)))  /* Последващо сканиране и разгъване  на макроса */
5. ((c)+(((b)+(a)))) /* Край на разгъване на макроса                 */
```



Имена на макроси в тяло на друг макрос се разпознават само след разгъване на тялото за някое специфично извикване на този макрос.

C89 не поддържа рекурсивни макроси, т.е макрос, чието тяло косвено или директно съдържа извикване на същия макрос.

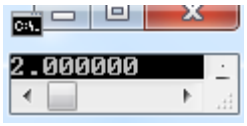
Пример 16-9:

```
#include <stdio.h>
#include <math.h>

#define sqrt(x) ((x)<0 ? sqrt(-(x)) : sqrt(x))

int main(void)
{
	printf("%f\n", sqrt(-4));

	return 0;
}
```



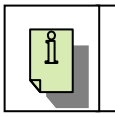
Макросът sqrt(-4) се разгъва по следния начин:

```
1. ((-4)<0 ? sqrt(-(-4)) : sqrt(-4)) /* Параметрите се заменят с аргументите  */

2. printf("%f\n", ((-4)<0 ? sqrt(-(-4)) : sqrt(-4))); /* Макроса се заменя с
                                                         неговото тяло      */

3. printf("%f\n", sqrt(-(-4) );
```

Обърнете внимание, че ако макросите се извикваха рекурсивно, то това би довело до безкрайно разгъване на макроса при неговото извикване.



C89 не поддържа рекурсивни макроси.

1.4.4 Област на видимост на макро-дефинициите

Макросите могат да бъдат дефинирани навсякъде в кода, извън всички функции, вътре във функция или дори в блок с код. Независимо от това къде е дефиниран, един макрос е видим от реда на дефиниране до реда, в който макроса се отдефинира с директивата #undef или до края на файла.

1.4.5 Разполагане на макрос на няколко реда

Тялото на макрос може да бъде разположено на няколко реда с помощта на знака \ в края на всеки ред (без последния). Между края на реда и знака \ може да има интервали.

Пример 16-10:

```
#include <stdio.h>

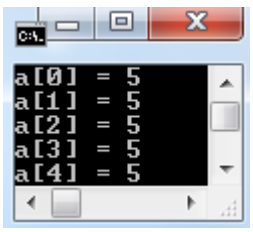
#define InitArray(x)            \
{                               \
      int i;                    \
      for(i = 0; i<5 ; i++)     \
      {                         \
            a[i] = x;           \
      }                         \
}

#define PrintArray()\
{\
      int i;\
      for(i = 0; i<5 ; i++)\
      {\
            printf("a[%d] = %d\n", i, a[i]);\
      }\
}

int main(void)
{
      int a[5];

      InitArray(5) /* Инициализиране на масива с 5        */
      PrintArray() /* Отпечатване на елементите на масива */

      return 0;
}
```



1.4.6 Предефинирани макроси

Всяка имплементация трябва задължително да предоставя дефинициите на следните макро-обекти:

- __LINE__

  Десетична константа представляваща номера на текущия ред.

- __FILE__

  Низ представляващ текущото име на сорс-файла.

- __DATE__

  Низ представляващ датата на компилиране на сорс-файла, изразена във формата "Mmm dd yyyy", където имената на месеците са като тези генерирани от функцията asctime(). Ако датата на компилиране не е налична, имплементационно-дефинирана валидна стойност трябва да бъде предоставена.

- __TIME__

  Низ представляващ веремето на компилиране на сорс-файла, изразено във формата "hh:mm:ss" (както при функцията asctime()).Ако времето на компилиране не е налично, имплементационно-дефинирана валидна стойност трябва да бъде предоставена.

- __STDC__

  Десетична константа 1, указваща, че компилаторът е ANIS/ISO C компилатор.

Никой от тези макроси не може да бъде отдефиниран с #undef или предефиниран от програмистта.

Пример 16-11:

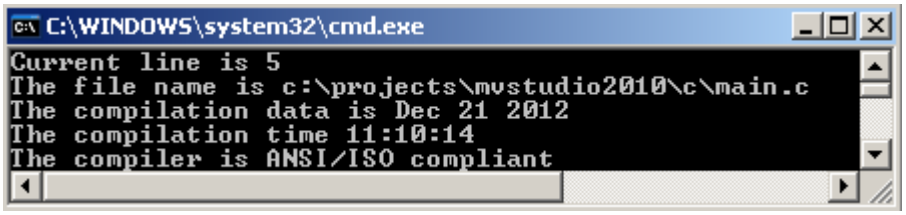
```
#include <stdio.h>

int main(void)
{
	printf("Current line is %d\n", __LINE__);
	printf("The file name is %s\n", __FILE__);
	printf("The compilation data is %s\n", __DATE__);
	printf("The compilation time %s\n", __TIME__);

	if (1 == __STDC__)
```

```
	{
		printf("The compiler is ANSI/ISO compliant\n");
	}
	else
	{
		printf("The compiler is non-ANSI/ISO compliant\n");
	}

	return 0;
}
```



1.4.7 Особености на макросите

Предпроцесорът разпознава име на макрос на всички места с изключение на следните:

- коментар
- низ
- символна константа
- #include <име-файл>

Предпроцесорът не различава ключови думи от други идентификатори. Това означава, че е възможно използването на ключова дума като име на макрос, но се счита за лоша програмна практика.



Не използвайте ключова дума като име на макрос.

Предпроцесорните директиви се разпознават преди разгъването на макроси. Следователно ако един макрос се разгъва до предпроцесорна директива, тя няма да бъде разпозната от предпроцесора в C89.

1.5 Директива за отдефиниране на макроси #undef

Директивата #undef отдефинира макрос.

Отдефиниране на макрос

```
#undef име-макрос
```

Име-макрос може да бъде както име на макро-обект, така и име на макро-функция. Ако име-макрос съществува, след изпълнението на тази директива то ще бъде невалидно. Ако макрос с такова име не съществува, директивата просто се игнорира.

Дрективата #undef позволява един макрос да бъде предефиниран с друго тяло.

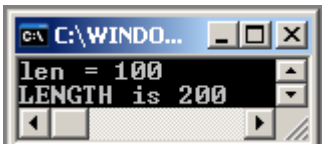
Пример 16-12:

```
#include <stdio.h>

#define LENGTH 100 /* Дефиниране на макрос LENGTH */
int len = LENGTH;
#undef LENGTH      /* Отдефиниране на макроса LENGTH */
#define LENGTH 200 /* Предефиниране на макрос LENGTH */

int main(void)
{
      printf("len = %d\n", len);
      printf("LENGTH is %d\n", LENGTH);

      return 0;
}
```



1.6 Директиви за условно компилиране (#if, #ifdef, #ifndef, #else, #elif, #endif)

Тези директиви управляват кои части от кода да бъдат подадени към компилатора и кои да се игнорират в зависимост от определени условия.

1.6.1 #if

Директива #if

1

```
#if управляващ-израз
    програмен-текст
#endif
```

2

```
#if управляващ-израз
    програмен-текст
#else
    програмен-текст
#endif
```

3

```
#if управляващ-израз1
    програмен-текст
#elif управляващ-израз2
    програмен-текст
...

#elif управляващ-изразN
    програмен-текст
#else
    програмен-текст
#endif
```

където

управляващ-израз, управляващ-израз1, ... управляващ-изразN
Предпроцесорен управляващ израз, който се изчислява до константна целочислена аритметична стойност.

програмен-текст
Програмен текст от всякакъв вид. Може да съдържа също и директиви, включително да бъде и празен.

Както се вижда от синтаксиса, директивата #if винаги завършва с директивата #endif. Директивите #else и #elif могат да се пропуснат.

Тази директива работи по следния начин:

1. Изчислява се управляващия израз на директивата #if.

2. Ако резултатът е ненулева стойност, програмния текст, заключен между #if и #endif (за форма 1), или #if и #else (за форма 2), или #if и първия #elif (за форма 3), се подава на компилатора. Всички останали програмни текстове се игнорират.

3. Ако резултатът е 0, се извършват следните действия в звисимост от използваната форма на директивата:

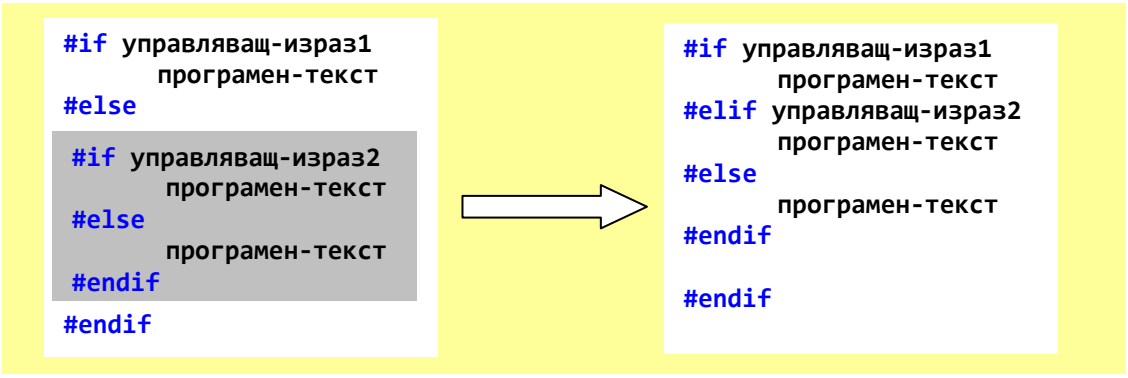
    a. Форма 1: Програмният текст между #if и #endif се игнорира.

    b. Форма 2: Програмният текст между #if и #else се игнорира, а програмния текст между #else и #endif се подава на компилатора.

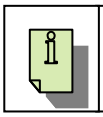
    c. Форма 3: Програмният текст между #if и първият #elif се игнорира и се преминава към изчисляване на управляващия израз на първия #elif. Ако резултатът е ненулев стойност, програмният текст, заключен между първия #elif и следващия #eilif, се подава на компилатора, а останалите програмни текстове се игнорират. Ако резултатът е 0, се преминава към

изчисляване на управляващия израз на следващия #elif и т.н.. Ако всички управляващи изрази се изчисляват до 0, програмният текст, заключен между #else и #endif се подава на компилатора, а всички останали програмни тесктове се игнорират. Ако няма #else, нито един програмен текст не се подава на компилатора.

Директивата #elif всъщност е съкратен запис на последователността #else-#if. Фиг.95 показва вложена #if-#else-#endif конструкция в #else частта на друга #if-#else-#endif конструкция и как това може да се пренапише с помощта на директивата #elif.



Фиг. 98 Директива #elif



C89 стандартът изисква една имплементация да поддържа поне 8 нива на вложеност на директивата #if.

Управляващия израз на #if директивата има следните особености:

- Управляващият израз се изчислява по време на транслационна фаза 4 (вижте Глава 1 Лексикални елементи) и трябва да бъде целочислен константен израз.

- Управляващият израз може да съдържа само целочислени и символни константи, и оператора defined (вижте точка 10.1 Предпроцесорен оператор defined). Дали числовите стойности на символните константи, използвани в управляващия израз, съвпадат с числовите стойности на същите символни константи в изпълнимия C код е имплементационно-зависимо.

- Управляващият израз може да използва всички целочислени аритметични, битови, логически операции и операциите за отношение и равенство (вижте Глава 4 Операции и изрази).

- При изчисление на управляващия израз, числови изрази тип int и unsigned int се третират като long и unsigned long съответно.

- Управляващият израз не може да използва операцията за преобразуване на типове и оператора sizeof.

- Ако управляващият израз съдържа макроси, те се разгъват преди изчислението на израза. Всички идентификатори, които остават след разгъването на всички макроси и изпълнението на оператора defined, се заменят с 0, а всяка предпроцесорна лексема се преобразува в лексема.

Компилаторите са свободни да приемат и допълнителни форми на управляващия израз, но използването им намалява преносимостта на програмата.

Например някои компилатори ще компилират следващия код, докато други (като Microsoft Visual Studio C++) ще генерират грешка от вида:

invalid integer constant expression

Пример 16-13:

```
#include <stdio.h>

#define MACRO_NAME

int main(void)
{

#if MACRO_NAME
	printf("This compiler uses a non-standard usage of #if directive\n");
#else
	printf("This compiler uses a standard usage of #if directive\n");
#endif

	return 0;
}
```

Пример 16-14:

```
#include <stdio.h>

#define TABLE_SIZE 200

#if TABLE_SIZE>200
#undef TABLE_SIZE
#define TABLE_SIZE 200
#elif TABLE_SIZE<50
#undef TABLE_SIZE
#define TABLE_SIZE 50
#else
#undef TABLE_SIZE
#define TABLE_SIZE 100
#endif

int table[TABLE_SIZE];
```

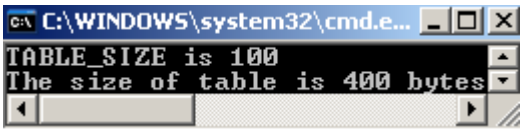
програмен-текст1

програмен-текст2

програмен-текст3

```
int main(void)
{
	printf("TABLE_SIZE is %d\n", TABLE_SIZE);
	printf("The size of table is %d bytes\n", sizeof(table));

	return 0;
}
```



От всички тези особености, свързани с управляващия израз, може да се направи извода, че резултатът от изчислението на управляващия израз от предпроцесора може да се различава от резултата на същия израз, изчислен по време на изпълнение на програмата.

Пример 16-15:

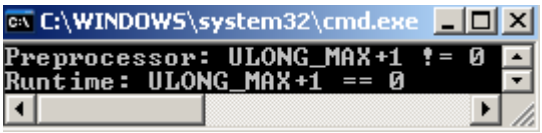
```
#include <limits.h>
#include <stdio.h>

int main(void)
{

#if ULONG_MAX+1 != 0
	printf("Preprocessor: ULONG_MAX+1 != 0\n");
#else
	printf("Preprocessor: ULONG_MAX+1 == 0\n");
#endif

	if(ULONG_MAX+1 != 0){
	   printf("Runtime: ULONG_MAX+1 != 0\n");
	}
	 else {
	   printf("Runtime: ULONG_MAX+1 == 0\n");
	 }

	return 0;
}
```



1.6.2 #ifdef

Директива #ifdef

1

```
#ifdef идентификатор
    програмен-текст
#endif
```

2

```
#ifdef идентификатор
    програмен-текст
#else
    програмен-текст
#endif
```

3

```
#ifdef идентификатор
    програмен-текст
#elif управляващ-израз1
    програмен-текст
...

#elif управляващ-изразN
    програмен-текст
#else
  програмен-текст
#endif
```

където

идентификатор
Може да бъде всеки идентификатор.

програмен-текст
Програмен текст от всякакъв вид. Може да съдържа също и директиви, включително да бъде и празен.

управляващ-израз1, ... управляващ-изразN
Предпроцесорен управляващ израз, който се изчислява до константна целочислена аритметична стойност.

Както се вижда от синтаксиса, директивата #ifdef винаги завършва с директивата #endif. Директивите #else и #elif могат да се пропуснат.

Тази директива работи по следния начин:

1. Предпроцесорът проверява дали идентификаторът е дефиниран с директивата #define.

2. Ако иденитификаторът е дефиниран с #define, програмния текст, заключен между #ifdef и #endif (за форма 1), или между #ifdef и #else (за форма 2), или между #ifdef и първия #elif (за Форма 3), се подава на компилатора. Всички останали програмни текстове се игнорират.

3. Ако иденитификаторът не е дефиниран с #define или е отдефиниран с #undef, се извършват следните действия в звисимост от използваната форма на директивата:

    a. Форма 1: Програмния текст, заключен между #ifdef и #endif се игнорира.

    b. Форма 2: Програмния текст, заключен между #ifdef и #else се игнорира, а програмния текст, заключен между #else и #endif се подава на компилатора.

    c. Форма 3: Програмния текст между #ifdef и първия #elif се игнорира и се преминава към изчисляване на управляващия

израз на първия #elif. Ако резултатът е ненулева стойност, програмния текст, заключен между първия #elif и следващия #eilif, се подава на компилатора, а останалите програмни текстове се игнорират. Ако резултатът е 0, се преминава към изчисляване на управляващия израз на следващия #elif и т.н.. Ако всички управляващи изрази се изчисляват до 0, програмния текст, заключен между #else и #endif се подава на компилатора, а всички останали програмни тесктове се игнорират. Ако няма #else, нито един програмен текст не се подава на компилатора.

Пример 16-16:

```
#include <stdio.h>

#define TABLE_SIZE 10

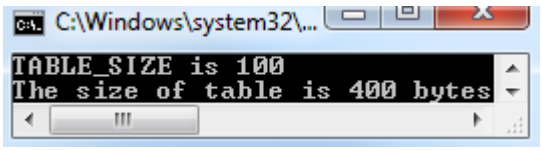
#ifdef TABLE_SIZE
#undef TABLE_SIZE
#define TABLE_SIZE 100
#endif

int table[TABLE_SIZE];

int main(void)
{
	printf("TABLE_SIZE is %d\n", TABLE_SIZE);
	printf("The size of table is %d bytes\n", sizeof(table));

	return 0;
}
```

Програмен текст



Конструкцията **#ifdef идентификатор** е еквивалентна на **#if defined (идентификатор)** (вижте точка 10.1 Предпроцесорен оператор defined).

1.6.3 #ifndef

Директива #ifndef

1

```
#ifndef идентификатор
    програмен-текст
#endif
```

2

```
#ifndef идентификатор
    програмен-текст
#else
    програмен-текст
#endif
```

3

```
#ifndef идентификатор
    програмен-текст
#elif управляващ-израз1
    програмен-текст

...

#elif управляващ-изразN
    програмен-текст
#else
  програмен-текст
#endif
```

където

идентификатор
Може да бъде всеки идентификатор.

програмен-текст
Програмен текст от всякакъв вид. Може да съдържа също и директиви, включително да бъде и празен.

управляващ-израз1, ... управляващ-изразN
Предпроцесорен управляващ израз, който се изчислява до константна целочислена аритметична стойност.

Както се вижда от синтаксиса директивата #ifndef винаги завършва с директивата #endif. Директивите #else и #elif могат да се пропуснат.

Тази директива работи по следния начин:

4. Предпроцесорът проверява дали идентификаторът е дефиниран с директивата #define.

5. Ако иденитификаторът не е дефиниран с #define, или е бил отдефиниран с #undef, програмния текст, заключен между #ifndef и #endif (за форма 1), или между #ifndef и #else (за форма 2), или между #ifndef и първия #elif (за Форма 3), се подава на компилатора. Всички останали програмни текстове се игнорират.

6. Ако иденитификаторът е дефиниран с #define, се извършват следните действия в звисимост от използваната форма на директивата:

    a. Форма 1: Програмният текст, заключен между #ifndef и #endif се игнорира.

    b. Форма 2: Програмният текст, заключен между #ifndef и #else се игнорира, а програмния текст, заключен между #else и #endif се подава на компилатора.

    c. Форма 3: Програмният текст между #ifndef и първия #elif се

игнорира и се преминава към изчисляване на управляващия израз на първия #elif. Ако резултатът е ненулева стойност, програмния текст, заключен между първия #elif и следващия #eilif, се подава на компилатора, а останалите програмни текстове се игнорират. Ако резултатът е 0, се преминава към изчисляване на управляващия израз на следващия #elif и т.н.. Ако всички управляващи изрази се изчисляват до 0, програмният текст, заключен между #else и #endif се подава на компилатора, а всички останали програмни тесктове се игнорират. Ако няма #else, нито един програмен текст не се подава на компилатора.

Пример 16-17:

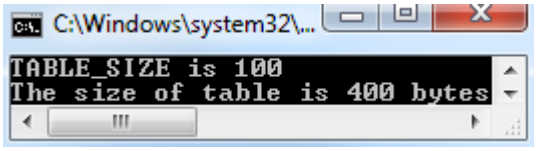
```
#include <stdio.h>

#ifndef TABLE_SIZE
#define TABLE_SIZE 100
#endif

int table[TABLE_SIZE];

int main(void)
{
	printf("TABLE_SIZE is %d\n", TABLE_SIZE);
	printf("The size of table is %d bytes\n", sizeof(table));

	return 0;
}
```



Конструкцията **#ifndef идентификатор** е еквивалентна на **#if !defined (идентификатор)** (вижте точка 10.1 Предпроцесорен оператор defined).

1.7 Директива за номериране на редове (#line)

Директивата #line променя стойността на предефинирания макрос __LINE__ и/или __FILE__ (вижте точка 4.6 Предефинирани макроси). Макросът __LINE__ съдържа номера на текущия компилиран ред от кода, а __FILE__ съдържа името на сорс-файла.

Директива #line

```
#line номер-на-ред
```

1

```
или
```

```
#line номер-на-ред "име-файл"        2

или

#line предпроцесорни-лексеми        3

където

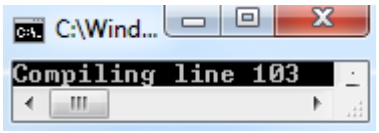
номер-на-ред
      Положително цяло число в интервала 1 ÷ 32767.

предпроцесорни-лексеми
      Може да се състои от макроси, които след разгъване образуват една от първите
      две форми на директивата #line.
```

Следващият сорс-ред след директивата #line получава номера, указан в директивата.

Пример 16-18:

```
1  #include <stdio.h>                          /* ред 1   */
2                                              /* ред 2   */
3  #line 100                                   /* ред 3   */
4  int main(void)                              /* ред 100 */
5  {                                           /* ред 101 */
6                                              /* ред 102 */
7      printf("Compiling line %d\n", __LINE__);/* ред 103 */
8                                              /* ред 104 */
9      return 0;                               /* ред 105 */
10 }                                           /* ред 106 */
11
```



Закоментирайте ред номер 3 и сравнете получения резултат с предишния.

Сорс-редовете, които следват #line се допуска, че идват от файл, указан с "име-файл", а номер-на-ред указва номера на реда, от който се взимат тези сорс-редове.

1.8 Директива за грешка (#error)

Директивата #error принуждава компилатора да генерира диагностично съобщение, указано в самата директива, по време на компилиране на кода.

Директива за грешка #error

```
#error съобщение
```

Тази директива типично се използва в комбинация с някоя от директивите

за условно компилиране.

Пример 16-19:

```
#include <stdio.h>

#if (__STDC__ != 1)
#error The compiler is non-compliant with the standard C!!!
#endif

int main(void)
{
  /*.....*/
  return 0;
}
```

1.9 Имплементационни директиви (#pragma)

Директивите #pragma могат да се използват за специфициране на имплементационно-зависимо поведение на предпроцесора.

Директиви #pragma

```
#pragma предпроцесорни-лексеми
```

Стандартът C89 не дефинира никакви стандартни #pragma директиви. Директиви #pragma, които не се поддържат от конкретна имплементация на компилатора, просто се игнорират.

1.10 Предпроцесорни оператори (defined, # и ##)

1.10.1 Предпроцесорен оператор defined

Предпроцесорният оператор defined е алтернатива на директивата #ifdef. Той също може да се използва, за да се определи дали даден идентификатор е дефиниран с директивата #define.

Предпроцесорен оператор defined

```
defined идентификатор

или

defined (идентификатор )
```

Ако **идентификатор** е дефиниран с #define, резултатът е 1, а ако не е дефиниран или е бил отдефиниран с #undef, резултатът е 0.

Предпроцесорният оператор се използва типично в управляващия израз на директивите #if и #elif (вижте точка 6.1 #if).

Следните двойки конструкции за еквивалентни:

```
#if defined  (име-макрос) ≈ #ifdef име-макрос

#if !defined (име-макрос) ≈ #ifndef име-макрос
```

Пример 16-20:

```
#include <stdio.h>

#define TABLE_SIZE 10

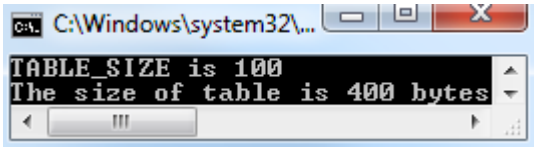
#if defined TABLE_SIZE
#undef TABLE_SIZE
#define TABLE_SIZE 100
#endif

int table[TABLE_SIZE];

int main(void)
{
	printf("TABLE_SIZE is %d\n", TABLE_SIZE);
	printf("The size of table is %d bytes\n", sizeof(table));

	return 0;
}
```

Програмен текст



Предимството на оператора defined пред директивата #ifdef, е че с негова помощ могат да се проверят няколко идентификатора наведнъж.

Пример 16-21:

```
#include <stdio.h>

#define MACRO1
#define MACRO2

#if defined (MACRO1) && defined (MACRO2)
#define TABLE_SIZE 100
#endif

int table[TABLE_SIZE];

int main(void)
{
	/*...*/

	return 0;
}
```

1.10.2 Предпроцесорен оператор # (stringizing operator)

Този оператор е предназначен да се използва само в заместващия текст на макрос с праметри (вижте точка 4 Директива за дефиниране на макроси #define).

Предпроцесорен оператор #

```
#define име-макрос(параметър1, параметър2,...параметърN)   заместващ-текст

където

заместващ-текст
      Съдържа конструкции от вида #операнд.

операнд
      Може да бъде всеки един от формалните параметри параметър1,...параметърN.
```

Предпроцесорният оператор # преобразува операнда в низ, затворен в двойни кавички.

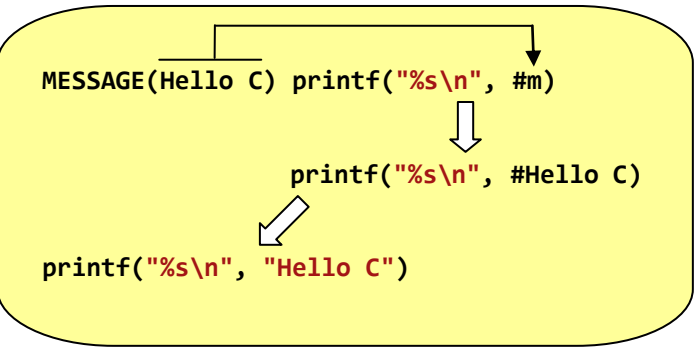
Пример 16-22:

```
#include <stdio.h>

#define MESSAGE(m) printf("%s\n", #m)

int main(void)
{
      MESSAGE(Hello C);

      return 0;
}
```





Всяка последователност от интервали между предпроцесорните лексеми на аргумента се заменя с единичен интервал. Интервалите пред първата и след последната лексеми, изграждащи аргумента, се изтриват.

Пример 16-23:

```
#include <stdio.h>

#define MESSAGE(m) printf("%s\n", #m)

int main(void)
{
      MESSAGE(     Hello         C           );

      return 0;
}
```

Всички интервали между лексемите се заменят с един интервал.

Всички интервали след последната лексема се изтриват.

Всички интервали пред първата лексема се изтриват.



Също така ако аргументът съдържа двойни кавички " или обратно наклонена черта \, предпроцесорът автоматично вмъква \ пред тях, за да запази тяхното значение в низа. Празен аргумент се преобразува в празен низ "".

Пример 16-24:

```
#include <stdio.h>

#define MESSAGE(m) printf("%s\n", #m)

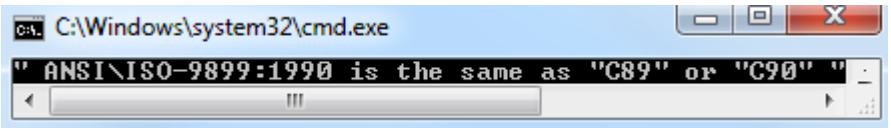
int main(void)
{
	MESSAGE(" ANSI\ISO-9899:1990 is the same as "C89" or "C90" ");

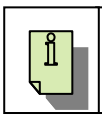
	return 0;
}
```

```
" ANSI\ISO-9899:1990 is the same as "C89" or "C90" "
```

⇩

```
"\" ANSI\\ISO-9899:1990 is the same as \"C89\" or \"C90\" \""
```





Редът на изчисление на предпроцесорните оператори # и ## е недефиниран.

1.10.3 Предпроцесорен оператор ## (pasting operator)

Този оператор е предназначен да се използва само в заместващия текст на макрос (виж точка т 4 Директива за дефиниране на макроси #define).

Предпроцесорен оператор ##

```
#define име-макрос заместващ-текст
или

#define име-макрос(параметър1, параметър2,...параметърN)  заместващ-текст

където

заместващ-текст
	Съдържа конструкции от вида операнд1##операнд2.
```

```
операнд1, операнд2
      Могат да бъдат всякакви лексеми, включително и формалните параметри
      параметър1,...параметърN.
```

Предпроцесорният оператор ## съединява двата си операнда в една лексема. Ако операндите са формални параметри, те се заместват с аргументите си преди да се съединят. Ако формираната лексема е невалидна, поведението е недефинирано. Операторът ## не трябва да се появява в началото или в края на заместващия текст на макро-дефиниция.

Пример 16-25:

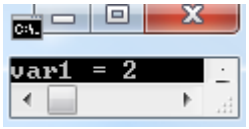
```
#include <stdio.h>

#define VAR var##1

int main(void)
{
      int var1;

      VAR = 2; /* VAR се заменя с var1 от предпроцесора */
      printf ("var1 = %d\n", var1);

      return 0;
}
```



Пример 16-26:

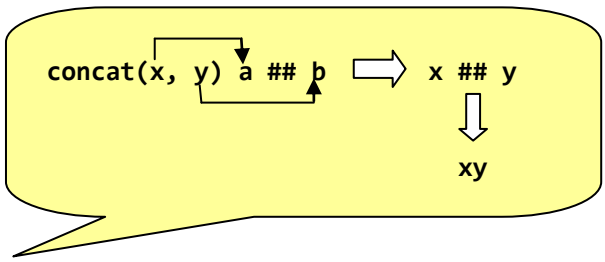
```
#include <stdio.h>

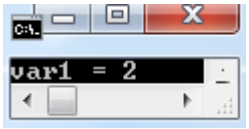
#define concat(a, b) a ## b

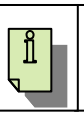
int main(void)
{
      int xy = 10;

      printf("xy = %d\n", concat(x, y));

      return 0;
}
```



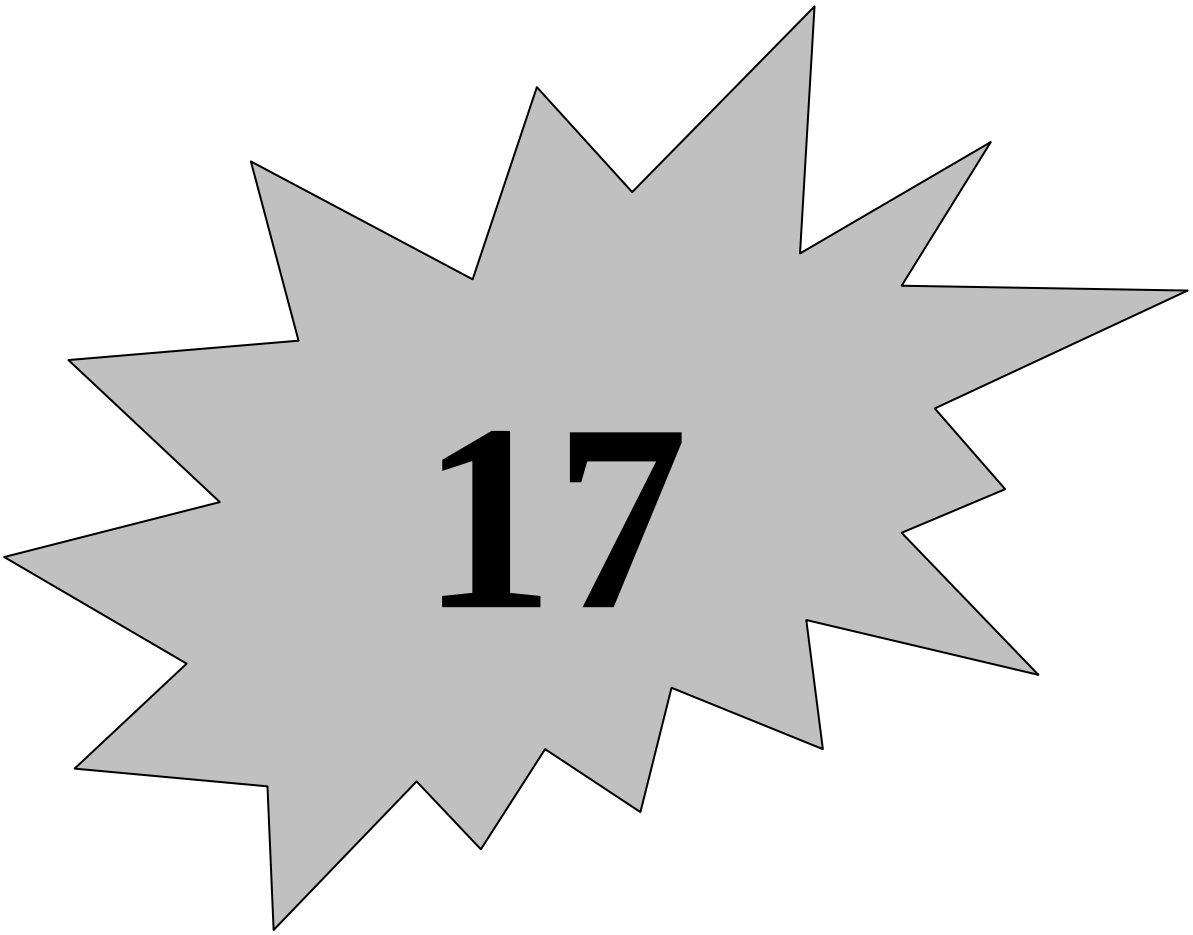




Редът на изчисление на предпроцесорните оператори # и ## е недефиниран.

Част II: C99





Въведение в C99

1.1 Нововъведения в C99

C99 добавя много нови програмни елементи към C89 и стандартната библиотека. Въпреки нововъведенията, C99 не променя коренно езика. За разлика от C89, C99 не е широко възприет. Повечето компилатори поддържат само частично тази версия на стандарта. Нововъведенията в C99 включват:

- добавяне на коментар в стил C++ (//)
- смесване на декларации и код
- добавяне на нови целочислени типове
- добавяне на булев тип (**<stdbool.h>**)
- назначени инициализатори (**designated initializer**)
- съставни литерали (**compound literal**)
- масиви с променлива дължина (**VLA-масиви**)
- вложени-функции (**inline-функции**)
- ограничени указатели (**restricted pointers**)
- гъвкави масиви (**flexible arrays**)
- static и квалификатори на типа в декларатор на масив
- добавяне на комплексни типове
- поддръжка на макроси за математика с обобщени типове (**<tgmath.h>**)
- надеждно целочислено деление
- универсални символни имена (**\u** и **\U**)
- разширени идентификатори
- шестнадесетични константи с плаваща запетая и **%a** и **%A** printf/scanf форматни спецификатори
- разширени целочислени типове и библиотечни функции

(**<inttypes.h>** и **<stdint.h>**)

- премахване на подразбиращата се декларация на функции
- предпроцесорна аритметика извършвана с типове **intmax_t/uintmax_t**
- нова област на видимост за операторите за избор и цикъл
- макроси с променлив брой аргументи
- завършваща запетая в enum декларация
- спецификатор **%lf** в printf()
- нови правила за съвместимост на структури
- нови предефинирани макроси
- **_Pragma** предпроцесорен оператор
- стандартни pragma-директиви
- предефиниран идентификатор **__func__**
- макрос **va_copy**
- преобразуването на масив в указател не е ограничено до L-стойности
- намаляване на ограниченията при инициализация на агрегатните и union- типове
- return без израз не е позволен за функции, връщащи резултат
- добавяне на нови хедъри и библиотечни функции

1.2 Ключови думи в C99

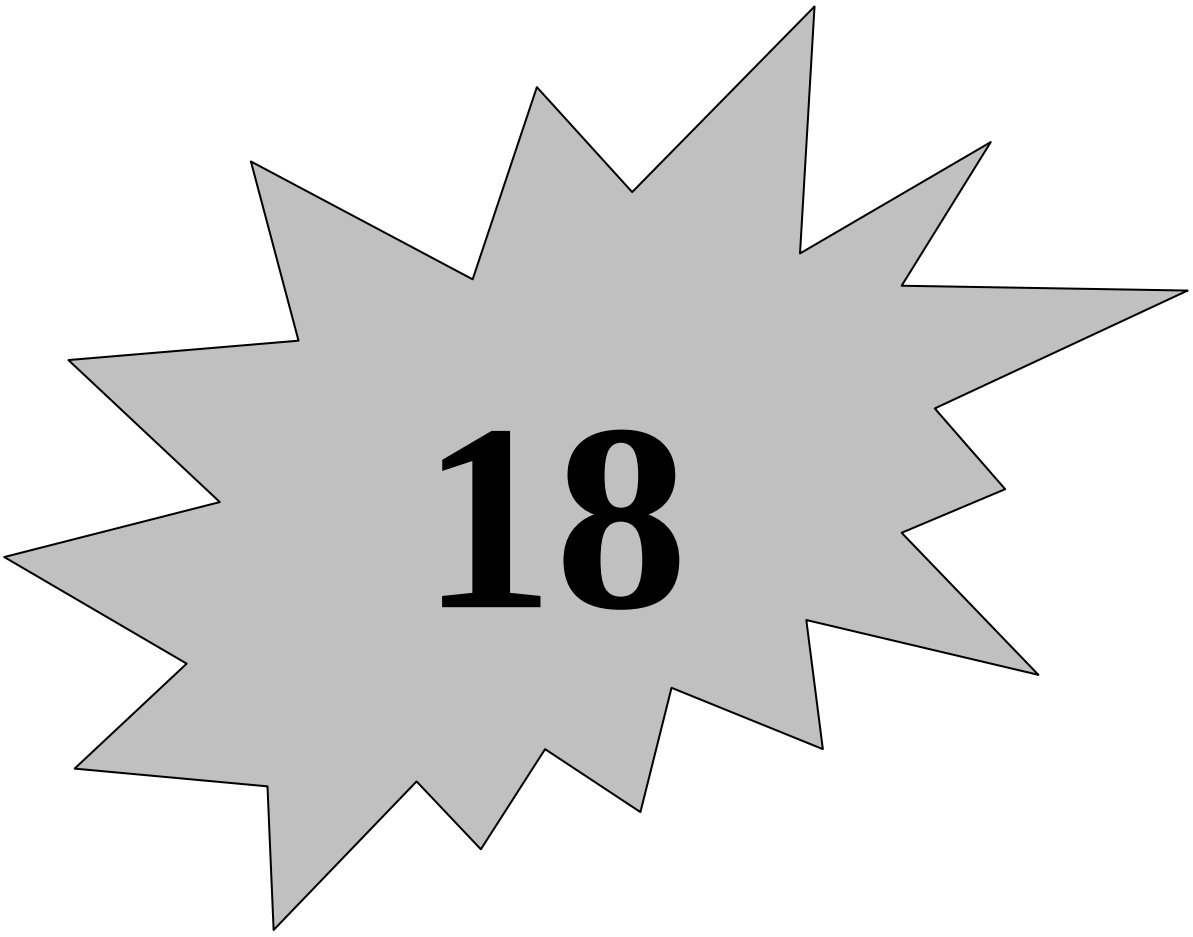
C99 дефинира 37 ключови думи (Табл.49).

auto	**enum**	**restrict** C99	**unsigned**
break	**extern**	**return**	**void**
case	**float**	**short**	**volatile**
char	**for**	**signed**	**while**

const	**goto**	**sizeof**	**_Bool** C99
continue	**if**	**static**	**_Complex** C99
default	**inline** C99	**struct**	**_Imaginary** C99
do	**int**	**switch**	-
double	**long**	**typedef**	-
else	**register**	**union**	-

Табл. 49 Ключови думи в C99

Горният индекс C99 показва ключовите думи добавени в C99.



Описание на нововъведенията в C99

1.1 Коментар в стил C++

C99 добавя нов вид коментар, който е възприет от C++. Този коментар се нарича още едноредов и се обозначава с //. Всички символи разположени след него до края на реда се игнорират от компилатора.

Пример 1-1:

```
//Включване на библиотечни хедър-файлове
#include <stdio.h>

//Входна точка на програмата
int main(void)
{
	//Отпечатай съобщение на дисплея
	printf("First C program\n");

	//Връщане на резултат от изпълнение на функцията main()
	return 0;
}
```

1.2 Смесване на декларации и изпълним код

C89 изисква всякакви декларации в един блок да се разполагат преди всички изпълними оператори. C99 премахва това ограничение. Това означава, че можем да дефинираме променливи навсякъде в блок.

Пример 1-2:

```
#include <stdio.h>

int main(void)
{
	int x = 1; // Дефиниране на променлива в начало на блок
	printf("x is %d\n", x);

	int y = 2; // Дефиниране на променлива в средата на блок
	printf("y is %d\n", y);

	return 0;
}
```

Област на видимост на x

Област на видимост на y

В тази програма променливата y е дефинирана след извикване на изпълним оператор, какъвто е извикването на функцията printf(). Такива променливи са видими от точката си на дефиниране до края на блока, в който се намира дефиницията. Въпреки тази възможност много програмисти предпочитат да разполагат дефинициите на променливи в началото на блок, и по-точно в началото на функцията.

C99 позволява дефиниране на променлива в **израз1** на цикъла for. Областта

на видимост и времето на живот на променливата е тялото на цикъла.

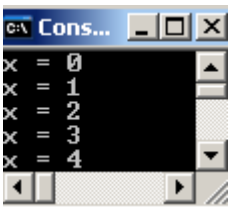
Пример 1-3:

```
#include <stdio.h>

int main(void)
{
	for(int x = 0; x < 5 ; x++)
	{
		printf("x = %d\n", x);
	}

	/* x = 20; */ /*!!!Грешка, x не се вижда извън for */

	return 0;
}
```



1.3 Нови целочислени типове

Към целочислените типове в C89 C99 добавя два нови: **unsigned long long int** и **signed long long int**. Следните означения са еквивалентни:

long long ≈ long long int ≈ signed long long ≈ signed long long int
unsigned long long ≈ unsigned long long int

Табл.50 описва минималната ширина в битове и диапазона от стойности за тези типове.

тип	ширина в битове	диапазон
signed long long	64	–9 223 372 036 854 775 807 ÷ +9 223 372 036 854 775 807
unsigned long long	64	0 ÷ 18 446 744 073 709 551 615

Табл. 50 Диапазон от стойности на long long и unsigned long long

За да извеждате/въвеждате стойности тип long long и unsigned long long с printf()/scanf() използвайте форматните спецификатори **%ll** и **%ull** съответно.

1.4 Булев тип

C99 въвежда беззнаков целочислен тип **_Bool**, който може да съдържа само две стойности: 0 и 1. За удобство хедър-файла <stdbool.h> дефинира макроса **bool**, който е синоним на **_Bool** и макросите **false** и **true**, които имат стойност 0 и 1 съответно.

C99 позволява да извършвате преобразувания между аритметични и указателни типове и булев тип. Когато аритметична стойност се преобразува в тип _Bool, резултатът е 0 ако стойността е нула, иначе резултатът е 1. Когато булев тип се преобразува до аритметичен тип, резултатът е 0 или 1, преобразувани до аритметичния тип. Когато указател се преобазува в тип _Bool, резултатът е 0, ако стойността на указателя е NULL, иначе резултатът е 1.

Следващият пример отпечатва на екрана битовото представяне на 8-битова променлива.

Пример 1-4:

```
#include <stdio.h>
#include <stdbool.h>

#define BYTE_LENGTH       8
#define BIT_MASK       0x80

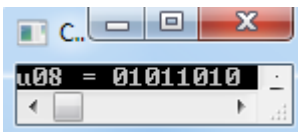
int main(void)
{
	unsigned char u08 = 0x5a;
	bool bit;

	printf("u08 = ");
	for(int i = 0 ; i < sizeof(char)*BYTE_LENGTH ; i++)
	{
		bit = (bool)((u08 << i) & BIT_MASK);

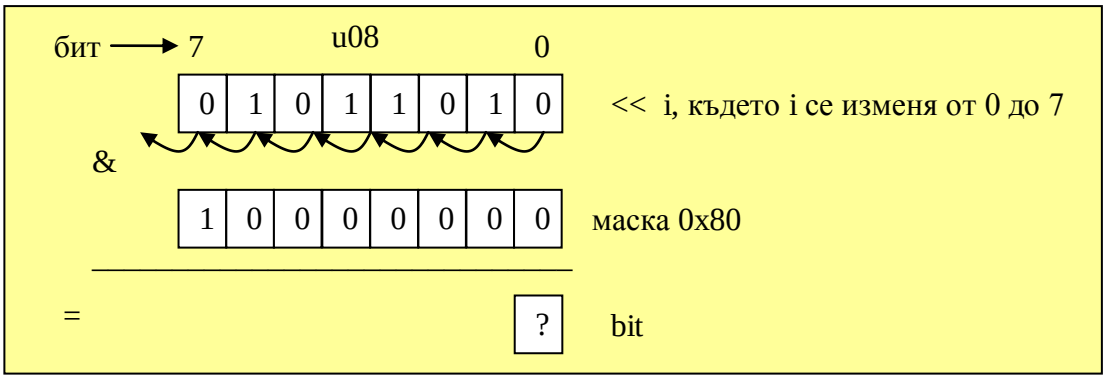
		if(true == bit)
		{
			putchar('1');
		}
		else
		{
			putchar('0');
		}
	}

	putchar('\n');

	return 0;
}
```



Фиг.97 илюстрира Пример 1-4. За простота на илюстарацията не са отчетени автоматичните преобразувания, дължащи се на целочисленото повишение на типовете.



Фиг. 99 Илюстрация към Пример 3-4

Битовете на променливата u08 се извличат с помощта на израза **bit = (bool)((u08 << i) & BIT_MASK);**. Едно извъртане на цикъла извлича и отпечатва поредния бит, започвайки от най-старшия. При всяко преместване на променливата u08 наляво с една позиция всички битове от 0-ия до 6-тия се нулират и се извлича стойността на старшия бит с помощта на маската BIT_MASK. Ако старшият бит е 0 и резултатът от битовата операция & е 0, в противен случай е различен от нула. Този резултат се присвоява на булевата променлива bit.

Може да дефинирате и битово поле тип _Bool.

Пример 1-5:

```
#include <stdbool.h>

typedef struct _flags /* Деклариране на структура с 8 флага */
{
	bool f0;
	bool f1;
	bool f2;
	bool f3;
	bool f4;
	bool f5;
	bool f6;
	bool f7;
}tFlags;

int main(void)
{
	tFlags flags; /* Дефиниране на структурна променлива */

	/*...*/

	flags.f0 = true; /* Установяване на f0 в true */

	/*...*/

	if (false == flags.f0) /* Проверка на f0 */
```

```
	{
		/*...*/
	}

	return 0;
}
```

1.5 Назначени инициализатори

Назначеният инициализатор (**designated initializer**) е средство, което Ви позволява да инициализирате конкретен елемент на масив, член на структура или обединение. Освен това, назначеният инициализатор позволява да инициализирате отделните компоненти в произволен ред.

1.5.1 Назначен инициализатор на масив

Назначен инициализатор на масив

```
[индекс] = инициализатор

където

индекс
	Може да бъде само целочислен неотрицателен константен израз.

инициализатор
	За локални масиви може да бъде всеки израз, чийто тип е съвместим с типа на
	масива. За глобални и статични масиви инициализаторът трябва да е константен
	израз.
```

Назначеният инициализатор на масив има следните свойства:

Ако размерът на масива не е дефиниран, **индекс** може да бъде всяка неотрицателна стойност и назначеният инициализатор с най-големия индекс определя размера на масива.

Пример:

```
                                /* индекс -> 0   1   2   3   4   5 */
int a1[] = {[1] = 20, 30, 40, 50, 60}; /* 0, 20, 30, 40, 50, 60 */
```

Ако позиционен инициализатор (има се предвид обичаен инициализатор) се намира непосредствено след назначен инициализатор, тогава позиционният инициализатор инициализира следващия елемент след назначения.

Пример:

```
                                /* индекс -> 0  1   2   3   4   5 */
int a2[6] = {10, [2] = 30, 40, 50, 60}; /* 10, 0, 30, 40, 50, 60 */
```

Назначеният инициализатор позволява реинициализиране на елемент на масив.

```
                                        /* индекс -> 0   1   2   3   4   5 */
int a3[6] = {10, 30, 40, 50, 60, [2] = 20, -4}; /* 10, 30, 20, -4, 60, 0 */
```

Елементи 2, 3 и 4 първо се инициализират с 40, 50 и 60 съответно, но после елемент 2 се реинициализира с 20, а от предходното правило следва, че елемент 3 се инициализира с -4. Тъй като за елемент 5 няма инициализатор, той се инициализира с 0.

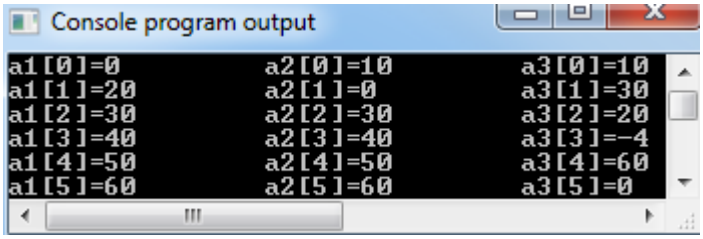
Пример 1-6:

```
#include <stdio.h>

int main(void)
{
  int a1[]  = {[1] = 20, 30, 40, 50, 60};          /*  0, 20, 30, 40, 50, 60 */
  int a2[6] = {10, [2] = 30, 40, 50, 60};          /* 10,  0, 30, 40, 50, 60 */
  int a3[6] = {10, 30, 40, 50, 60, [2] = 20, -4}; /* 10, 30, 20, -4, 60,  0 */

  for(int i = 0; i < 6; i++)
  {
    printf("a1[%d]=%d \ta2[%d]=%d \ta3[%d]=%d\n", i, a1[i], i, a2[i], i, a3[i]);
  }

  return 0;
}
```



1.5.2 Назначен инициализатор на структурна-променлива

Назначен инициализатор на структурна-променлива

```
.име-член = инициализатор

където

име-член
      Име на член на стуктурата.

инициализатор
      За локални структурни-променливи може да бъде всеки израз, чийто тип е
      съвместим с типа на члена. За глобални и статични структурни променливи
      инициализаторът трябва да е константен израз.
```

Назначеният инициализатор на структурна-променлива има следните

свойства:

Ако позиционен инициализатор (има се предвид обичаен инициализатор) се намира непосредствено след назначен инициализатор, тогава позиционният инициализатор инициализира следващия член след назначения.

Пример:

```
struct book {char title[40]; char author[40]; double price;};

struct book b = {.author = "Unknown", 0.0}; /* "", "Unknown", 0.0 */
```

Назначеният инициализатор позволява реинициализиране на член на структурата.

```
struct book b = {"Unknown", .author = "Agatha Christie", 25.3, .title = "Ten
Little Indians"}; /* title = "Ten Little Indians", author = "Agatha Christie",
                     price = 25.3 */
```

Пример 1-7:

```
#include <stdio.h>

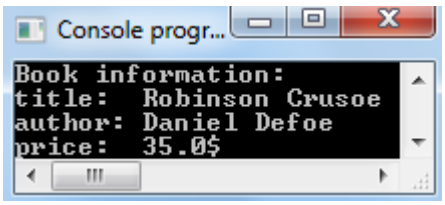
struct book {char title[40]; char author[40]; double price; };

int main(void)
{

  struct book b = {.price = 35.0, .title = "Robinson Crusoe", .author = "Daniel
Defoe"};

  printf("Book information:\n");
  printf("title:  %s\n", b.title);
  printf("author: %s\n", b.author);
  printf("price:  %.1f$\n", b.price);

  return 0;
}
```



1.5.3 Назначен инициализатор на обединение

Назначеният инициализатор на union-променлива има същия синтаксис като на структурна-променлива. Той позволява union-променливата да бъде инициализирана чрез всеки един от своите членове, а не само с

първия както е при C89.

Пример:

```
union U {int a; float b; char c[4]};

union U u1 = {.c = "abc"}; /* u1.c = "abc" */
union U u2 = {.a = 15};    /* u2.a = 15    */
union U u3 = {.b = 3.14};  /* u3.b * 3.14  */
```

1.6 Съставни литерали

Съставните литерали Ви позволяват да дефинирате безименни инициализирани обекти (променливи) от всеки обектен тип, използвайки следния синтаксис:

Съставен литерал

```
(тип) {списък-инициализатори}

където

тип
      Указва типа на литерала и може да бъде от всеки обектен тип (базов тип,
      указателен тип, масив, структура, обединение, изброяване)

списък-инициализатори
      Следва същите правила като инициализаторите на конкретния тип.
```

Пример:

```
(int []) {1, 2, 3, 4, 5} /* Създава масив от 5 елемента тип int и ги инициализира
                            с указаните стойности */

struct ms {int a; double b; char* p;}; /* Деклариране на структура */
(struct ms) {10, 3.2, "Hello"} /* Създава структурна-променлива от тип struct ms и
                                  инизиализира членовете с указаните стойности */
```

Списък-инициализатори може да съдържа и назначени инициализатори.

Пример:

```
(int []) {[3] = 2, 3, 4, 5} /* литерал-масив 0,0,0,2,3,4,5 */

struct ms {int a; double b; char* p;}; /* Деклариране на структура */
(struct ms) {.p = "Hello"} /* литерал-структура 0, 0.0, "Hello" */
```

Ако използвате съставен литерал извън функция (глобален съставен литерал), инициализаторите задължително трябва да са константи. Ако използвате съставен литерал в блок с код (локален съставен литерал) инициализаторите може да са всякакви изрази.

Съставните литерали се използват за създаване най-вече на литерали-масиви и литерали-структури и могат да се използват за инициализиране на указатели и структурни-променливи съответно, като аргументи на функции и изобщо навсякъде където може да се използва променлива от този тип.

Пример 1-8:

```
#include <stdio.h>

struct ms {int a; double b; char* p;};

void foo(int* p, int n, struct ms s);

int main(void)
{
	int a = 10;
	int b = 20;
	int *p;

	p = (int []){10,20}; /* Указателят p сочи към първия елемент на литерал-
                                масив */
	 struct ms s = (struct ms) {10, 3.2, "Hello"}; /* Обектът s се инициализира
                                                          с литерал-структура */
	 /* Използване на съставни литерали като аргументи на функция */
	 foo((int []){1,2,3,4,5,}, 5, (struct ms) {100, 2.3, "Say something."});

	/* .. */

	return 0;
}

void foo(int* p, int n, struct ms s)
{
	/* ... */
}
```

Съставните литерали са l-стойности, т.е имат адрес в паметта. Това позволява да прилагате към тях операцията за извличане на адрес & по същия начин както и при обичайни променливи от същия тип.

Пример 1-9:

```
#include <stdio.h>

struct book {char title[40]; char author[40]; double price; };

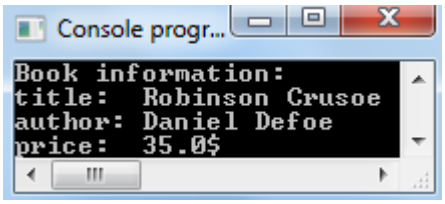
void print_book_info(struct book * p_book);

int main(void)
{
  print_book_info(&(struct book){.price = 35.0, .title = "Robinson Crusoe",
.author = "Daniel Defoe"});

  return 0;
}

void print_book_info(struct book * p_book)
```

```
{
  printf("Book information:\n");
  printf("title:  %s\n", p_book->title);
  printf("author: %s\n", p_book->author);
  printf("price:  %.1f$\n",p_book->price);
}
```



Може да прилагате квалификатора const към типа на съставния литерал.

Пример: `(const int []){1,2,3,4,5};`

Пример 1-10:

```
#include <stdio.h>

#define SIZE 5

void print_array(const int* p);
void clear_array(int* p);

int main(void)
{
	int* a = (int [SIZE]){1,2,3,4,5};
	const int* b = (const int [SIZE]){5,6,7,8,9};

	print_array(a);
	clear_array(a);
	print_array(a);

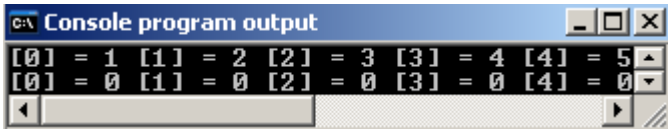
	/*clear_array(b);*/ /*!!!Грешка, b сочи към константен масив */

	return 0;
}

void clear_array(int* p)
{
	for(int i = 0 ; i < SIZE ; i++)
	{
		p[i] = 0;
	}
}

void print_array(const int* p)
{
	for(int i = 0 ; i < SIZE ; i++)
	{
		printf("[%d] = %d ", i, p[i]);
	}

	putchar('\n');
}
```



1.7 VLA-типове

1.7.1 Масиви с променлива дължина

Масивите с променлива дължина (**VLA - variable-length array**) са подобни на обичайните масиви с тази разлика, че размерът им може да бъде всякакъв целочислен израз, а не само константен.

VLA-масив

```
тип име-масив[n];

където

n
      Променлива от целочислен тип или неконстантен целочислен израз.
```

Пример:

```
int n = 5;
int a[n];
int b[5+n];
```

VLA-масивите имат следните ограничения:

- Могат да бъдат само локални auto-масиви. Глобалните и статичните масиви не могат да бъдат VLA-масиви.
- Членове на структури и обединения не могат да бъдат VLA-масиви.
- VLA-масивите не могат да се инициализират.

При всяко влизане в областта на действие на масива, компилаторът изчислява неговия размер и заделя памет в стека. При излизане от блока, в който масивът е дефиниран, тази памет се освобождава. Веднъж създаден VLA-масива не може да променя дължината си.

Пример 1-11:

```
#include <stdio.h>

void foo(int n);
void print_array(const int* p, int size);
```

```c
int main(void)
{
	foo(1);
	foo(2);
	foo(3);
	foo(4);

	return 0;
}

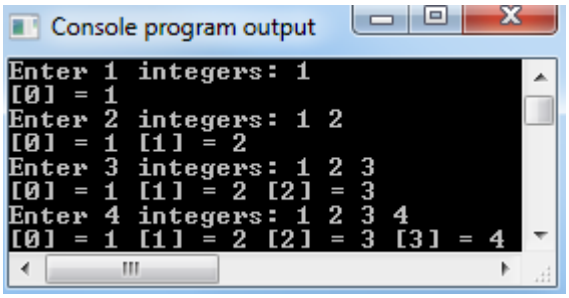
void foo(int n)
{
	int a[n]; /* Дефиниране на VLA-масив */

	printf("Enter %d integers: ", n);
	for(int i = 0 ; i < n ; i++)
	{
		scanf("%d",&a[i]);
	}

	print_array(a, n);
}

void print_array(const int* p, int size)
{
	for(int i = 0 ; i < size ; i++)
	{
		printf("[%d] = %d ", i, p[i]);
	}

	putchar('\n');
}
```



1.7.2 VLA-модифицирани типове

Тип, който е свързан с VLA-масив, се нарича VLA-модифициран тип. Такива са например указатели към VLA-масиви.

VLA-типовете имат следните ограничения:

- Могат да бъдат само локални auto-типове.
- Членове на структури и обединения не могат да бъдат VLA-типове.

Пример 1-12:

```
int main(void)
{
	int n = 5;
	int a[n];
	int (*p)[n];

	/*...*/

	return 0;
}
```

За разлика от VLA-масивите, VLA-типовете могат да бъдат статични. Въпреки че стойностите на VLA-типовете се запазват при излизане от областта им на видимост (блока с код, където се намира дефиницията), размерът на свързания с тях VLA-масив може да се променя при всяко последващо влизане в блока.

Пример 1-13:

```
#include <stdio.h>

void foo(void);

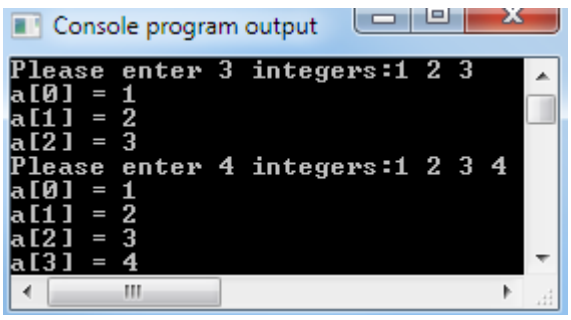
int main(void)
{
	foo();
	foo();

	return 0;
}

void foo(void)
{
	static int size = 2;
	int a[size];
	static int (*p)[size++] = &a;

	printf("Please enter %d integers:", size);
	for(int i = 0; i < size ; i++)
	{
		scanf("%d", ((int*)p + i));
	}

	for(int i = 0; i < size ; i++)
	{
		printf("a[%d] = %d\n",i,a[i]);
	}
}
```



1.7.3 VLA-масиви и typedef

Ако VLA-масив се използва в typedef-декларация, стойността на размера се изчислява само веднъж, когато се срещне декларацията.

Пример 1-14:

```
int main(void)
{
	int n = 5;
	typedef int vector[n];

	vector a; //  Еквивалентно е на int a[5]
	n++;
	vector b; //  Еквивалентно е на int b[6]

	//...

	return 0;
}
```

1.7.4 VLA-типове като параметри на функции

VLA-типовете могат да се използват и като параметри на функции. Когато размерът на масив е също параметър на функцията, той трябва да се появява преди VLA-типа в списъка с параметри. Обърнете внимание, че ако размерът също е параметър, той задължително трябва да стои преди масива в списъка с параметри на функцията.

Пример:

```
void f(int r, int c, int a[c][r]){...} /* Правилно   */
void f(int a[c][r], int r, int c){...} /* Неправилно */
```

В прототипа на функцията размерът на VLA-масив може да се замени с *, т.е. [*]. Всеки неконстантен израз, който се появява в квадратните скоби на масив в прототип на функция се третира като еквиваленетен на [*]. Следните прототипи са еквивалентни:

Пример:

```
void foo(int, int [*][*]);
void foo(int n, int [*][m]);
void foo(int n, int [n][n]);
void foo(int n, int [][*]);
```

Забележете, че най-левият размер при многомерни VLA-масиви, като параметри, може да се пропусне.

1.8 Вложени функции

Извикването на обикновена функция принуждава компютърът да запомни адреса на следващата инструкция, да копира аргументите в параметрите, да извърши преход към функцията, да я изпълни и да се върне. Всички тези действия оказват влияние върху производителността на програмата. За да се справи с този проблем C99 добавя т.н. вложени функции (**inline-функции**). Една функция става вложена като пред декларацията или дефиницията й се добави спецификатора **inline**. Ключовата дума inline е заявка към компилатора да вмъкне кода на функцията в мястото на извикването й подобно на макро-функция. По този начин всички изброени по-горе действия по извикване на обикновена функция се избягват. Подобряването на производителността на програмата обаче става за сметка на увеличаване на кода. Запомнете, че ключовата дума inline е само заявка към компилатора, която може да бъде отхвърлена по различни причини. Типични кандидати за вложени функции са малки функции, които често се извикват в кода.

За да може компилаторът да вмъкне кода на функцията в мястото на извикването й, дефиницията на функцията трябва да бъде в същия файл, в който се извиква. Обикновено тя се поставя преди първото извикване на функцията във файла. В този случай дефиницията играе ролята и на декларация (прототип).

Пример 1-12:

```
#include <stdio.h>

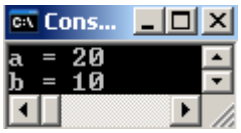
/* Дефиниране на вложена функция */
inline void swapf( int *p1, int *p2 )
{
	int tmp = *p1;

	*p1 = *p2;
	*p2 = tmp;
}

int main(void)
{
	int a = 10, b = 20;

	swapf(&a, &b); /* Размени стойностите на a и b */
	printf("a = %d\n", a);
	printf("b = %d\n", b);

	return 0;
}
```



Ако компилаторът приеме заявката, тялото на функцията swapf() ще бъде вмъкнато в мястото на извикването й както е показано по-долу.

```
#include <stdio.h>

/* Дефиниране на вложена функция */
inline void swapf( int *p1, int *p2 )
{
   int tmp = *p1; *p1 = *p2; *p2 = tmp;
}

int main(void)
{
      int a = 10, b = 20;

      /* Вмъкване на кода на функцията swapf() в мястото й на извикване */
      /* swapf(&a, &b); */ /* Размени стойностите на a и b */
      {
            int *p1;
            int *p2;

            p1 = &a;
            p2 = &b;

            int tmp = *p1;
            *p1 = *p2;
            *p2 = tmp;
      }

      printf("a = %d\n", a);
      printf("b = %d\n", b);

      return 0;
}
```

Всяка статична функция може да бъде направена вложена. По подразбиране вложените функции са с вътрешно свързване. Ако функцията е външна, нещата са малко по-сложни. Общото правило е, че ако всички декларации на една функция в един сорс-файл включват спецификатора inline, но не и extern, тогава дефиницията на функцията в този сорс-файл се нарича inline дефиниция. Ако функцията се извиква в други сорс-файлове, тогава е необходимо да се осигури външна дефиниция в някой друг сорс-файл. Когато функцията се извика, компилаторът може да избере дали да извърши обичайно извикване (използвайки външната дефиниция на функцията) или вложено извикване (използвайки inline дефиницията на функцията). Двете дефиниции на функцията може да се различават.

Най-често дефиницията на вложената функция се поставя в хедър-файл, който се добавя във всеки сорс-файл, който я използва. Следващият пример демонстрира това.

Пример:

```c
/* add.h */
#ifndef ADD_H
#define ADD_H

inline int add(int i, int j) { return i + j; }

#endif
```

```c
/* file1.c */
#include "add.h"

void foo1(void)
{
      add(1,2);
}
```

```c
/* file2.c */
#include "add.h"

void foo2(void)
{
      add(3,4);
}
```

Inline дефиниция на функция с външно свързване има следните ограничения:

- функцията не може да съдържа модифицируеми статични локални променливи.
- функцията не може да съдържа референции към променливи с вътрешно свързване.

1.9 Ограничени указатели

C99 въвежда нов квалификатор на типа – **restrict**. Той може да се прилага само към указатели към обекти. Такива указатели се наричат ограничени указатели (**restricted pointers**).

Ограничени указатели

```
квалификатор1опц тип * квалификатор2 опц restrictопц име-указател;
```

където

квалификатор1 / квалификатор2
- const
- volatile
- const volatile

Ключовата дума restrict е указание за компилатора, че сочения от указателя обект се достъпва само чрез този указател. Това позволява на компилатора да приложи определени оптимизационни техники. Ключовата дума restrict обаче не е задължаваща и компилаторът може да я игноира. Важно е да запомните, че отговорност на програмистта е да гарантира, че даден обект се достъпва само чрез ограничен указател. Например ако програмата

модифицира един обект чрез ограничен указател, тя не трябва да го достъпва по име или чрез друг указател докато ограничения указател съществува.

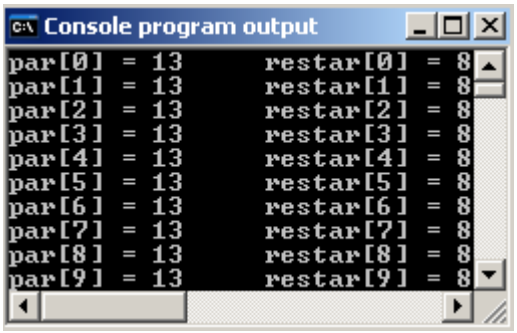
Пример 1-13:

```
#include <stdlib.h>

int main(void)
{
	int ar[10];
	int * restrict restar = (int *)malloc(10 * sizeof(int));
	int * par = ar;

	for (int n = 0; n < 10; n++)
	{
		par[n] += 5;
		restar[n] += 5;
		ar[n] *= 2;
		par[n] += 3;
		restar[n] += 3;
	}

	return 0;
}
```



Знаейки че указателят restar е единственото средство за достъп до соченатa от него памет, компилаторът може да извърши следната оптимизация:

Следните два реда

```
restar[n] += 5;
restar[n] += 3;
```

могат да бъдат заменени с един ред: `restar[n] += 8;`

1.10 Гъвкави масиви

В C99 последният елемент на структура може да бъде масив, чийто размер не е указан (масив от непълен тип). Такъв масив се нарича **гъвкав масив (flexible array)**. Структурата трябва да съдържа поне един обикновен член.

Гъвкав масив

```
struct име-структура_опц
{
	тип име-член1;
	...
	тип име-членN;

	тип име-гъвкав-масив[]; // Декларация на гъвкав масив
};
```

Особеност на гъвкавия масив е, че при дефиниране на променлива от структурата, компилаторът не заделя памет за него, т.е при дефинирането на обичайни структурни-променливи гъвкавият масив се игнорира. Ако S е структура, чийто последен елемент е гъвкав масив F, чийто елементи са от тип T, то операцията sizeof(struct S) ще върне като резултат броя на байтовете за обичайните членове, без гъвкавия масив. В това число влизат и уплътняващите байтове, включително и тези между предпоследния член и гъвкавия масив. Казаното дотук може да се изрази по следния начин:

```
sizeof(struct S) == offsetof(struct S, F)[1]
```

Забележка[1]: Във версията 7.00.35 на средата **Pelles C for Windows** това условие не е спазено.

Пример 1-14:

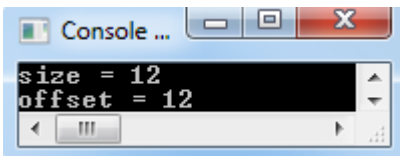
```
#include <stdio.h>
#include <stddef.h>

struct S
{
	int	L
	int	F[]; // Деклариране на гъвкав масив
};

int main(void)
{
	int	size;
	int	offset;

	size   = sizeof (struct S); // Определяне на размера на структурата
	offset = offsetof(struct S, F); // Определяне на отместването на F
	printf("size = %d\n", size);
	printf ("offset = %d\n", offset);

	return 0;
}
```



Когато достъпвате гъвкав масив с операцията '.' или '->', трябва да сте сигурни, че обектът в паметта е достатъчно голям да съдържа стойностите на елементите на гъвкавия масив. Това може да се постигне например чрез заделяне на динамична памет с malloc().

Пример:

```
/* Заделяне на памет достатъчно голяма да побере членовете на структурата S плюс елементите на гъвкавия масив F */
struct S* ps = malloc ( sizeof(struct S) + 10*sizeof(int) );
```

Заделя памет за L.	Заделя памет за 10 елемента на масива F.

Достъпът до елементите на гъвкавия масив става по същия начин като нормалните членове на структурата.

Пример:

```
for(int i = 0; i < 10 ; i++)
{
	ps->flex[i] = i; // Инициализиране на елементите на масива flex
}
```

Следващият пример показва друг начин за създаване на обект в паметта достатъчно голям да побере елементите на гъвкавия масив.

Пример 1-15:

```
#include <stdio.h>

struct S
{
	int	L;
	int	F[];
};

union U
{
	int objects[ ( sizeof(struct S) + 10*sizeof(int)) ];
	struct S  flex_s;
} u = {.flex_s = {.L = 10}};

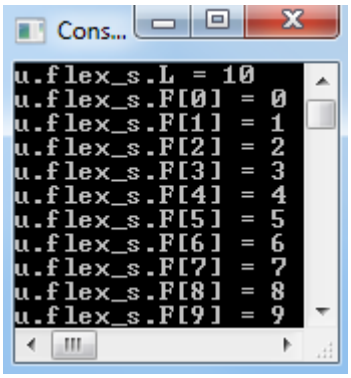
int main(void)
{
	printf("u.flex_s.L = %d\n", u.flex_s.L);

	// Инициализиране на елементите на масива F
	for (int i = 0; i < 10 ; i++)
	{
		u.flex_s.F[i] = i;
	}

	// Отпечатване на елементите на масива F
	for (int i = 0; i < 10 ; i++)
```

```
	{
		printf("u.flex_s.F[%d] = %d\n",i,u.flex_s.F[i]);	}
	}

	return 0;
}
```



Казаното по-горе може да се обобщи по следния начин: обикновено когато декларирате структура с гъвкав масив, се предполага, че няма да дефинирате променлива от тази структура, а указател към нея. Ако обектът в паметта е с големина Lo, то следното условие трябва да е спазено:

```
sizeof(struct S) + L*sizeof(T) <= Lo
```

Друга важна особеност на структурите с гъвкави масиви, е, че ако присвоявате една структурна променлива на друга, се копират само обикновените членове на структурата, т.е. . гъвкавият масив се игнорира. Същия е резултатът, ако използвате структурна-променлива като аргумент на функция.

Независимо дали обектът в паметта е достатъчно голям да побере елементите на гъвкавия масив, конструкцията &F[0] винаги е валидна.

1.11 static и квалификатори на типа в декларатор на масив

C99 разширява синтаксиса на деклариране на масив като параметър на функция (вижте 1.1.5 Предаване на масив на функция) като допуска квалификаторите const, volatile и restrict, както и спецификатора static да се появяват в квадратните скоби на декларацията на масива.

```
тип име-функция(тип име-масив[списък-квалификатори_опц       размер_опц])
{
	/* тяло на функцията */
}

тип име-функция(тип име-масив[static списък-квалификатори_опц      размер])
{
	/* тяло на функцията */
```

```
}

тип име-функция(тип име-масив[списък-квалификатори static размер])
{
      /* тяло на функцията */
}

където

списък-квалификатори
      const
      volatile
      const volatile
      volatile const
      restrict
```

Декларацията

```
тип име-масив[списък-квалификатори     размер]
```

се преобразува от компилатора в

```
тип * списък-квалификатори име-масив
```

Както забелязвате размерът на масива отново се игнорира както и в C89. Следващите примери ще Ви изяснят горните конструкции.

Пример 1-16:

```
#include <stdio.h>

void foo(int a[3]); /* Същото като void foo(int a[]) или void foo int * a) */

int main(void)
{
      int i[5]; /* Дефиниране на масив от 5 int елемента */

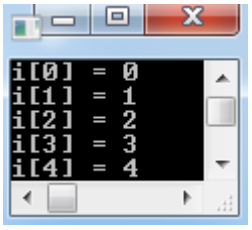
      foo(i);

      for(int x = 0; x < 5 ; x++) /* Отпечатване на масива */
      {
            printf("i[%d] = %d\n",x, i[x]);
      }

      return 0;
}

void foo(int a[3]) /* Обърнете внимание, че указаният размер не се взима под
                      внимание */
{/* Същото като void foo(int a[]) или void foo int * a) */

      for(int x = 0; x < 5 ; x++)
      {
            a[x] = x;
      }
}
```



Горният пример е напълно съвместим със C89 и демонстрира факта, че размерът, указан в скобите, не се взима под внимание.

Пример 1-17:

```
#include <stdio.h>

void foo(int a[const 3]); /* Същото като void foo(int a[const]); */
                          /* или като void foo(int * const a)*/

int main(void)
{
	int i[5];

	foo(i);

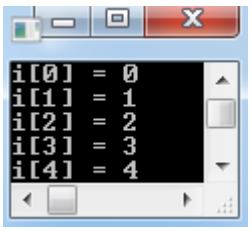
	for(int x = 0; x < 5 ; x++)
	{
		printf("i[%d] = %d\n",x, i[x]);
	}

	return 0;
}

void foo(int a[const 3])
{/* Същото като void foo(int a[const]) или void foo(int * const a)*/

	for(int x = 0; x < 5 ; x++)
	{
		a[x] = x;
	}

	/*
	int y;
	a = &y; */ /*!!!Грешка, a е константен указател */
}
```



Ключовата дума static не променя поведението на програмата, а указва на компилатора, че аргументът ще съдържа минимум на брой елементи, указани в дефиницията на функцията. Това позволява на компилатора да оптимизира кода за достъп до елементите на масива.

Пример 1-18:

```
#include <stdio.h>

void foo(int a[static 3]);

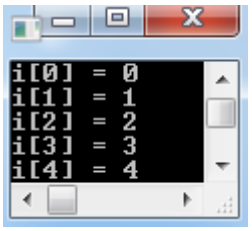
int main(void)
{
	int i[5]; /* Дефиниране на масив от 5 int елемента */

	foo(i);

	for(int x = 0; x < 5 ; x++) /* Отпечатване на масива */
	{
		printf("i[%d] = %d\n",x, i[x]);
	}

	return 0;
}

void foo(int a[static 3])
{
	for(int x = 0; x < 5 ; x++)
	{
		a[x] = x;
	}
}
```



Ако параметърът е многомерен масив, ключовата дума static може да се приложи само към първата (най-лявата) димeнсия.

Пример 1-19:

```
#include <stdio.h>

void foo(int a[static 3][3]);

int main(void)
{
	int i[3][3]; /* Дефиниране на двумерен масив от 3x3 int елемента */

	foo(i);

	/*...*/

	 return 0;
}

void foo(int a[static 3][3])
{
	/*...*/
```

```
}
```

Рамерът на масива в прототипа може да се замени с *. Това е указание, че декларирате VLA-масив като параметър. В действителност всеки неконстантен израз, използван за рамер на масива в прототипа, се третира все едно е заменен с *.

Пример 1-20:

```
#include <stdio.h>

void foo(int n, int a[n]); /* Същото като void foo(int a[*]); */

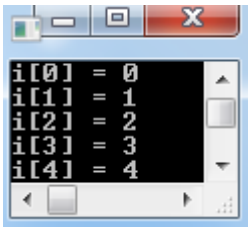
int main(void)
{
	int i[5]; /* Дефиниране на масив от 5 int елемента */

	foo(5, i);

	for(int x = 0; x < 5 ; x++) /* Отпечатване на масива */
	{
		printf("i[%d] = %d\n",x, i[x]);
	}

    return 0;
}

void foo(int n, int a[n])
{
	for(int x = 0; x < n ; x++)
	{
		a[x] = x;
	}
}
```



От всичко казано дотук, изводът, който може да се направи, е, че използването на ключовата дума static е може би единствената съществена новост, която заслужава да се запомни.

1.12 Макроси с променлив брой аргументи

Дефиниране на макрос с променлив брой аргументи

```
#define име-макрос(списък-параметри_опц,...) заместващ-текст_опц
```

където

списък-параметри$_{опц}$
Идентификатори, представляващи имена на формални параметри, разделени със запетая. Може да бъде и празен.

заместващ-текст$_{опц}$
Може да се пропусне или да бъде всеки текст, който след замяната формира валидна C конструкция. Параметрите от **списък-параметри** могат да се използват многократно в заместващия текст (или може въобще да не се използват).

Когато такъв макрос се извика, всички допълнителни аргументи, включително и запетаите между тях се комбинират в един общ аргумент. В заменящ-текст използването на специалния идентификатор **__VA_ARGS__** се заменя с този общ аргумент.

Пример 1-21:

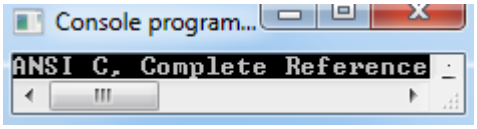
```
#include <stdio.h>

#define CREATE_STRING(...)      #__VA_ARGS__

int main(void)
{
      char string_buffer[] = CREATE_STRING(ANSI C, Complete Reference);

      printf("%s\n", string_buffer);

      return 0;
}
```



Пример 1-22:

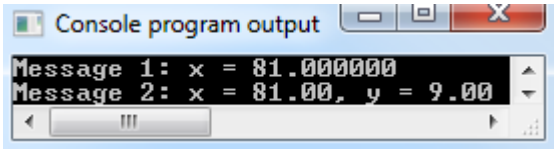
```
#include <stdio.h>
#include <math.h>

#define my_printf(X, ...) printf("Message " #X ": " __VA_ARGS__)

int main(void)
{
      double x = 81;
      double y;

      y = sqrt(x);
      my_printf(1, "x = %f\n", x);
      my_printf(2, "x = %.2f, y = %.2f\n", x, y);

      return 0;
}
```



C99 допуска един или всички аргументи в макрос с параметри (с фиксиран или променлив брой аргументи) да бъде празен. Съответните параметри се заменят с нищо, т.е. . се изтриват от заменящ-текст.

Пример 1-23:

```
#include <stdio.h>

#define SUM(y,x)  x+y

int main(void)
{
	int sum;

	sum = SUM(5,4); // sum = 5+4
	printf("sum = %d\n", sum);

	sum = SUM(5,); // sum = 5+
	printf("sum = %d\n", sum);

	sum = SUM(,4); // sum = +4
	printf("sum = %d\n", sum);

	return 0;
}
```

Ако празен аргумент е операнд на предпроцесорния оператор #, резултатът е нулев низ.

Пример:

```
#define CREATE_STRING(x)   #x
char string_buffer[] = CREATE_STRING(); //char string_buffer[] = "";
```

Ако празен аргумент е операнд на предпроцесорния оператор ##, той се заменя с невидим маркер. Свързването на обичайна лексема с такъв маркер дава като резултат лексемата, т.е. . маркерът се маха. Ако и двата операнда на ## са маркери, резултатът е един маркер. Когато разгъването на макроса завърши, всички маркери се махат.

Пример:

```
#define JOIN(x, y, z)     x##y##z

int JOIN(a,b,c); // int abc;
int JOIN( ,b,c); // int bc;
int JOIN( , ,c); // int c;
int JOIN(a, ,c); // int ac;
```

1.13 Нови предефинирани макроси

C99 добавя следните нови предефинирани макроси:

- __STDC_VERSION__

  Този макрос има стойност 199409L, ако комилаторът поддържа стандарта C95 (C89 с поправка 1). Ако компилаторът поддържа стандартът C99, този макрос има стойност 199901L. Ако компилаторът не спазва нито един от тези стандарти, стойността на макроса е недефинирана.

- __STDC_HOSTED__

  1 - хостед[1] имплементация
  0 - фрийстендинг[2] имплементация

  Забележка[1]: хостед имплементация (**hosted implementation**) се нарича всяка имплементация на компилатора, която поддържа езика C и цялата стандартната библиотека.

  Забележка[2]: фрийстендинг имплементация (**freestanding implementation**) се нарича всяка имплементация на компилатора, която поддържа езика C, без стандартната библиотека с изключение на средствата осигурени в хедър-файловете **float.h**, **iso646.h**, **limits.h**, **stdarg.h**, **stdbool.h**, **stddef.h** и **stdint.h**.

- __STDC_IEC_559__

  1 - имплементацията на числата с плаваща запетая спазва стандарта IEC 60559, в противен случай стойността на макроса е недефинирана.

- __STDC_IEC_559_COMPLEX__

  1 - имплементацията на комплексната аритметика спазва стандарта IEC 60559, в противен случай стойността на макроса е недефинирана.

- __STDC_ISO_10646__

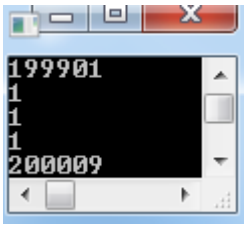
  Константа тип long int от вида **yyyymmL** (**yyyy**-година, **mm**-месец, **L**-суфикс, обозначаващ типа на константата), обозначаваща, че кодирането на широки символи тип wchar_t спазва стандарта ISO/IEC 10646, включително всички добавки и корекции добавени до годината и месеца, указани в стойността на макроса.

Пример 1-24:

```
#include <stdio.h>

int main(void)
{
	printf("%d\n",__STDC_VERSION__);
	printf("%d\n",__STDC_HOSTED__);
	printf("%d\n",__STDC_IEC_559__);
	printf("%d\n",__STDC_IEC_559_COMPLEX__);
	printf("%d\n",__STDC_ISO_10646__);

	return 0;
}
```



1.14 Стандартни pragma-директиви

C99 въвежда няколко стандартни #pragma-директиви.

Стандартни #pragma-директиви

```
#pragma  STDC  FP_CONTRACT       on_off_switch
#pragma  STDC  FENV_ACCESS       on_off_switch
#pragma  STDC  CX_LIMITED_RANGE  on_off_switch

където

on_off_switch
	Може да бъде ON, OFF или DEFAULT
```

Всички стандартни #pragma-директиви започват с STDC. Всяка имплементация на предпроцесора може да добавя допълнителни #pragma-директиви. Ако предпроцесорът не разпознае дадена #pragma-директива, тя се игнорира и никакви действия не се извършват.

Значението на стандартните #pragma-директиви е описано в описанието на хедър-файловете <math.h>, <fenv.h> и <complex.h> съответно.

1.15 _Pragma предпроцесорен оператор

_Pragma-оператор

```
_Pragma ( "низова-константа" )
```

_Pragma-операторът работи по следния начин: низовата константа в скобите се преобразува в предпроцесорни лексеми. За целта кавичките, заграждащи низовата-константа, се премахват; всяка последователност от вида \" се заменя с "; всяка последователност от вида \\ се замнея с \; предпроцесорът интерпретира така получените лексеми по същия начин както в #pragma-директива.

Пример:

```
_Pragma ( "STDC  FENV_ACCESS    ON" )
```

се третира като

```
#pragma      STDC  FENV_ACCESS  ON
```

_Pragma-операторът позволява формиране на #pragma-директиви чрез разширяване на макрос. #pragma-директивата може да се появава сама на ред и предпроцесорните й лексеми не се подлагат на макро-разширяване.

Пример:

```
#define  STR(s)  #s

_Pragma ( STR(STDC  FENV_ACCESS ON) )
```

Пример:

```
#define  DO_PRAGMA(x)     _Pragma ( #x )

DO_PRAGMA(STDC  FENV_ACCESS     ON)
```

1.16 Предефиниран идентификатор __func__

Идентификаторът __func__ представлява името на функцията, в която се използва, под формата на низова-константа. Компилаторът автоматично добавя следната дефиниция в началото на всяка функция:

```
тип име-функция(списък параметри)
{
      static const char __func__[] ="име-функция"; // автоматична дефиниция
      //...
}
```

Пример 1-25:

```
#include <stdio.h>

void foo(char *s);

int main(void)
{
	foo(NULL);

	//...

	return 0;
}

void foo(char *s)
{
	if( NULL == s)
	{
		printf( "%s: received null pointer argument!\n", __func__ );
		return;
	}

	/* ... */

	return;
}
```



1.17 Други

- Намаляване на ограниченията при инициализация на агрегатните и union- типове

  C99 позволява инициализаторите на локални структурни-променливи, union-променливи и масиви да бъдат всякакви изрази, чийто тип е съвместим с типа на инициализирания член. В C89 инициализаторите трябва да са константни изрази.

  Пример 1-26:

```
struct mystruct
{
	int i;
	double d;
};

int main(void)
{
	int	i = 10;
```

```
	double d = 2.5;

	struct mystruct s = {i, d}; // Позволено в C99, но забранено в C89

	//...

	return 0;
}
```

Пример 1-27:

```
int main(void)
{
	int a = 1, b = 2, c = 3;

	 int buf[] = {a, b, c}; // Позволено в C99, но забранено в C89

	 //...

	 return 0;
}
```

- Надеждно целочислено деление

  В C89 ако единия от операндите е отрицателно число, резултатът зависи от имплементацията (вижте 1.11.1.6 Операция деление). В C99 резултатът е ясно дефиниран и винаги се закръглява към нулата, т.е.. резултатът винаги е цялата част без закръгляване.

Пример 1-28:

```
#include <stdio.h>

int main(void)
{
	int x = -10/3;

	printf("x = %d\n", x);

	return 0;
}
```



C89 компилатор може да даде резултат -4 (закръгляване нагоре от нулата).

- Премахване на подразбиращата се декларация на функции

  В C99 добавянето на прототип на функция е задължително.

Пример 1-29:

```
//int foo(void);

int main(void)
{
    foo();

    return 0;
}

int foo(void)
{
  //..
}
```

Този код ще се компилира от C89 компилатор, докато C99 компилатор ще генерира грешка.

- Добавяне на завършваща запетая в enum декларация

  C99 позволявя поставянето на запетая след последната enum-константа.

Деклариране на изброяване

```
enum име-изброяване_опц
{
      име-enum-константа1,
      име-enum-константа2,
      ...
      име-enum-константаN,
};
```

- return без израз не е позволен за функции, връщащи резултат

  C89 допуска използването на оператор return без израз, дори за функции, връщащи резултат. В C99 това е забранено и ще доведе до генериране на грешка при компилиране.

Пример 1-30:

```
int foo(void);

int main(void)
{
    foo();

    return 0;
}

int foo(void)
{
  //..
```

```
    return; // Позволено в C89, но забранено в C99
}
```

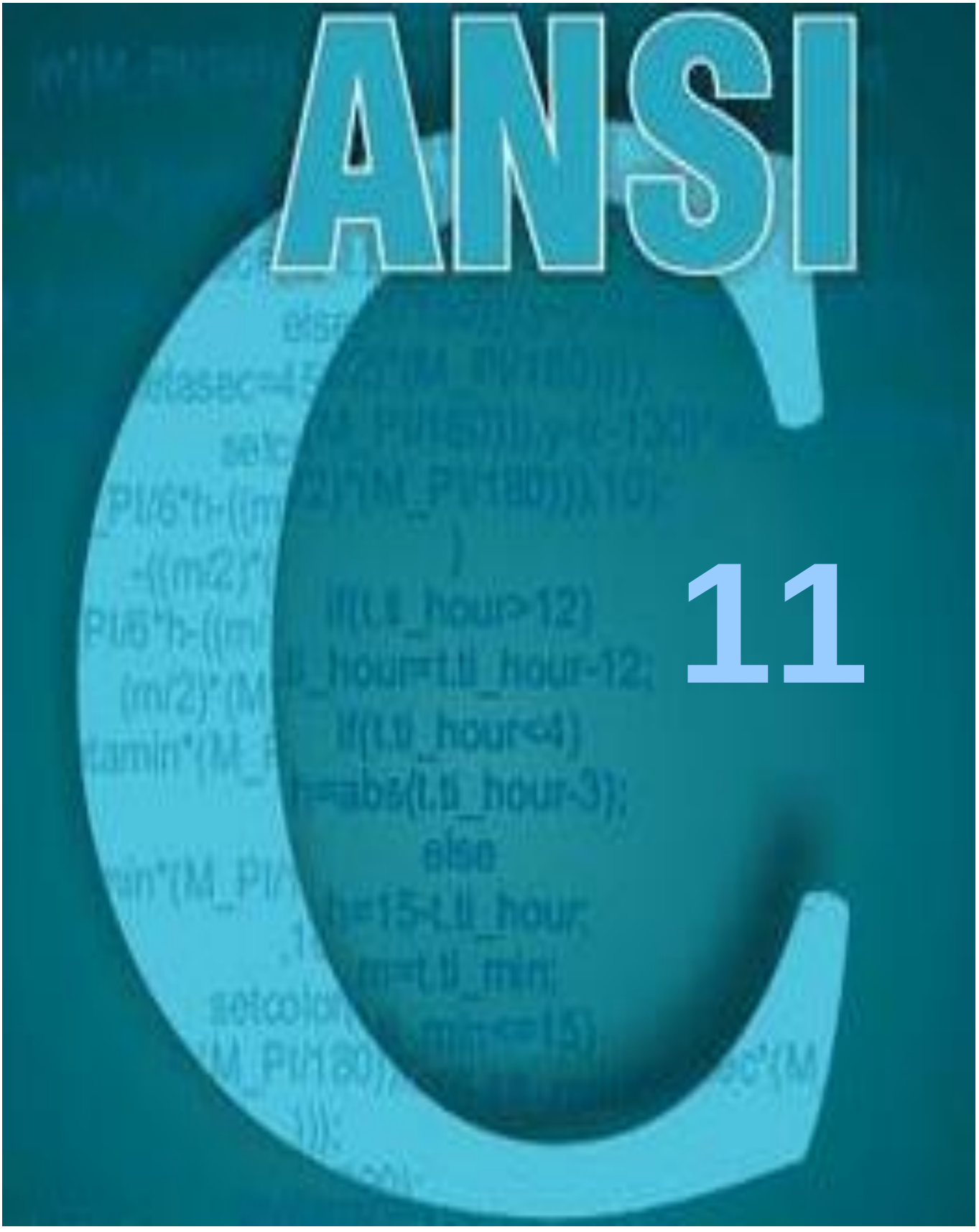
- Нови правила за определяне на типа на целочислени константи

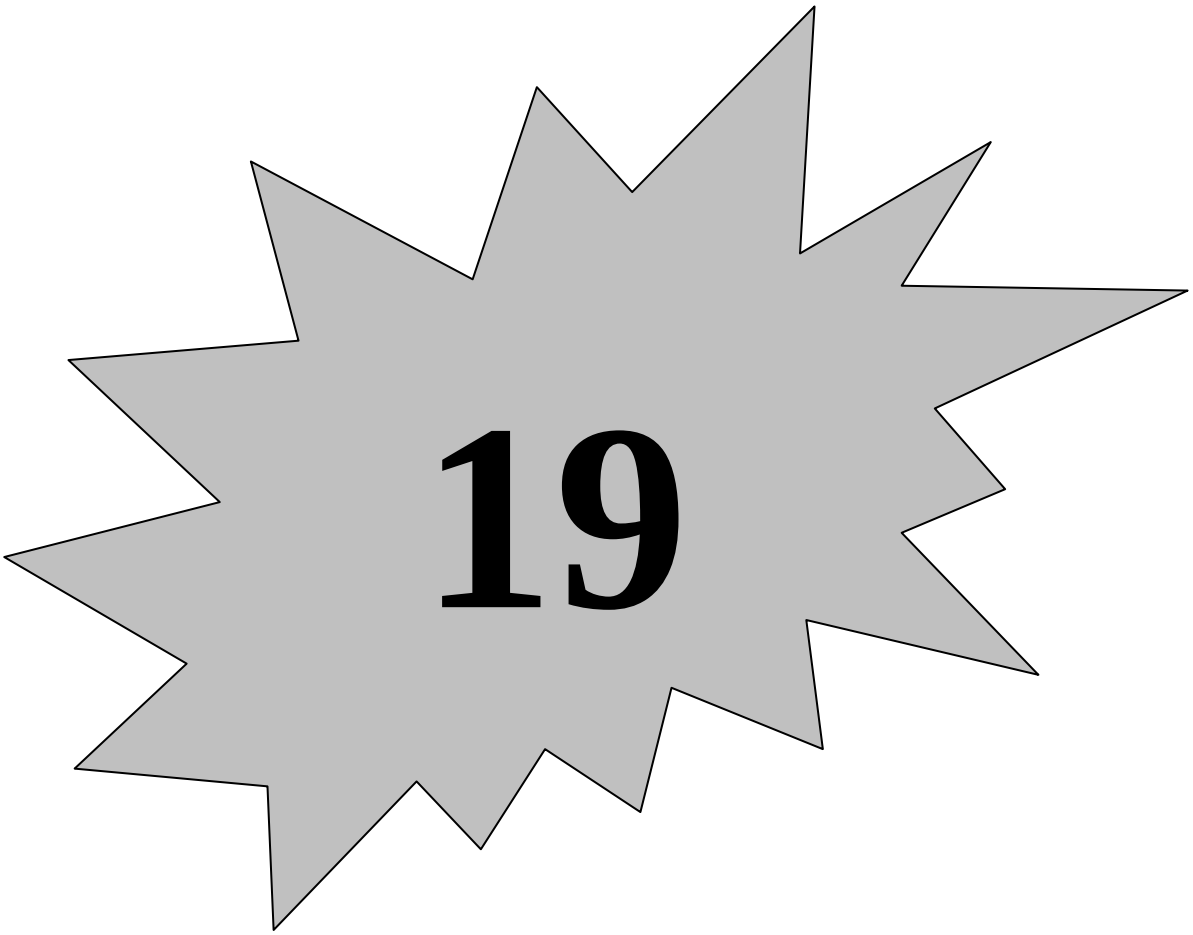
  Табл.51 показва какъв тип се назначава на константите в зависимост дали имат суфикс или не.

суфикс	десетични		осмични/шестнайсетични	
	C89	C99	C89	C99
-	int long unsigned long	int long long long	int unsigned long unsigned long	int unsigned long unsigned long long long unsigned long long
u или **U**	unsigned unsigned long	unsigned unsigned long unsigned long long	unsigned unsigned long int	unsigned unsigned long int unsigned long long
l или **L**	long unsigned long	long long long	long unsigned long	long unsigned long long long unsigned long long
ul или **UL**	unsigned long	unsigned long unsigned long long	unsigned long	unsigned long unsigned long long
ll или **LL**	-	long long int	-	long long unsigned long long
ull или **uLL**	-	unsigned long long	-	unsigned long long

Табл. 51 Тин на константи в C89 и C99

Част III: C11





Въведение в C11

1.1 Нововъведения в C11

C11 е текущия стандарт на езика C. Той добавя много нови програмни елементи към C99 и стандартната библиотека. Въпреки нововъведенията, C11 не променя коренно езика. Този стандарт е сравнително нов по време на писане на тази книга. Едва няколко компилатора го поддържат. Нововъведенията в C11 включват:

- добавяне на спецификатор _Atomic(тип) за тип
- добавяне на квалификатор _Atomic за тип
- добавяне на спецификатор _Noreturn за функция
- добавяне на спецификатор _Alignas за подравняване на променливи в паметта
- добавяне на оператор _Alignof
- добавяне на анонимни структури и обединения
- добавяне на поддръжка за многопоточност (**multithreading**)
- изрази с обобщени типове
- премахване на библиотечната функция getc()
- добавяне на нови хедъри и библиотечни функции

1.2 Ключови думи в C11

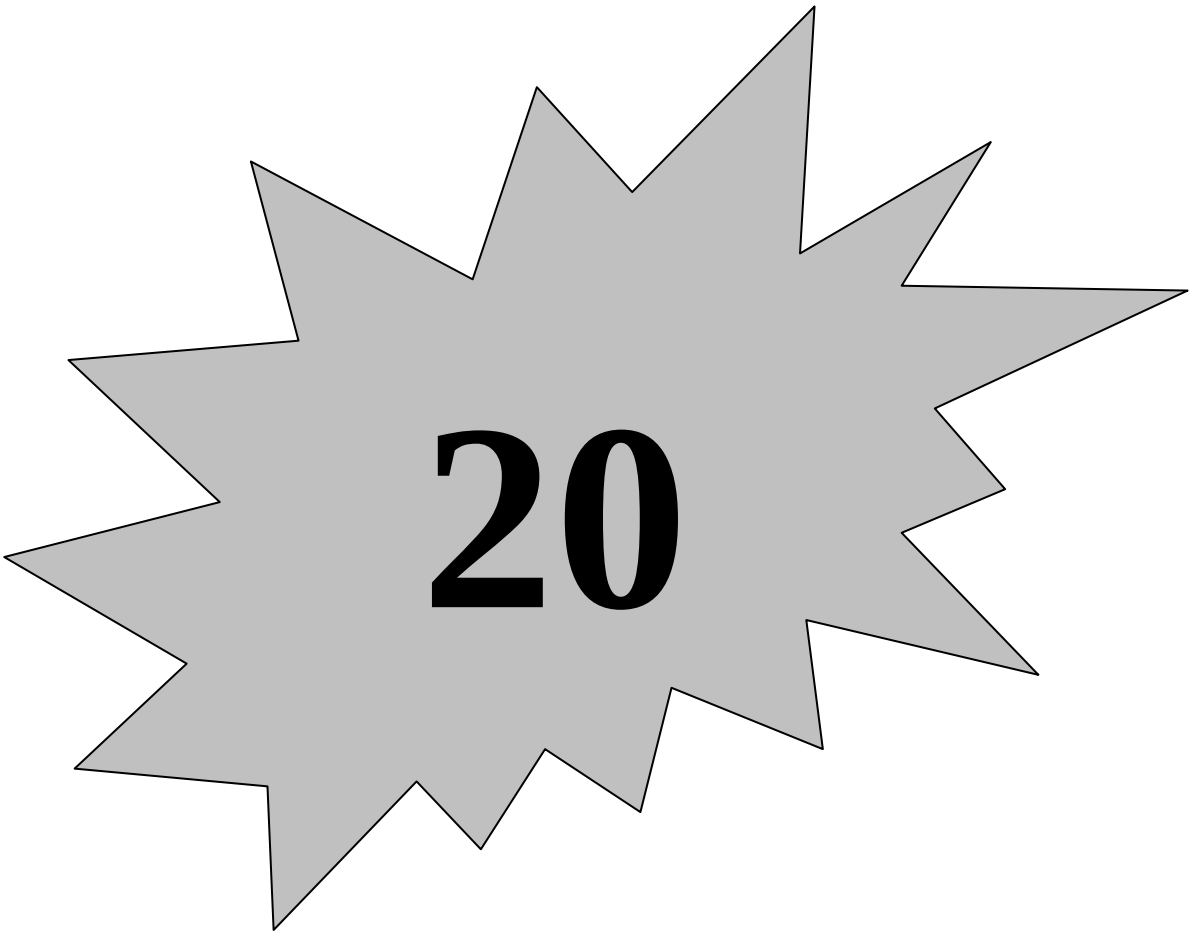
C11 дефинира 44 ключови думи (Табл.52).

auto	**if**	**unsigned**
break	**inline** C99	**void**
case	**int**	**volatile**
char	**long**	**while**
const	**register**	**_Alignas** C11
continue	**restrict** C99	**_Alignof** C11
default	**return**	**_Atomic** C11
do	**short**	**_Bool** C99
double	**signed**	**_Complex** C99

else	**sizeof**	**_Generic** C11
enum	**static**	**_Imaginary** C99
extern	**struct**	**_Noreturn** C11
float	**switch**	**_Static_assert** C11
for	**typedef**	**_Thread_local** C11
goto	**union**	-

Табл. 52 Ключови думи в C11

Горните индекси C99 и C11 показват ключовите думи добавени в C99 и C11 съответно.



Описание на нововъведенията в C11

1.1 Спецификатор _Atomic(тип)

Спецификаторът _Atomic се използва за дефиниране на атомични (неделими) типове. Той има следния синтаксис:

_Atomic(тип)

където

тип

Може да бъде всеки тип с изключение на масив, функция, атомичен тип или квалифициран тип.

Пример: `_Atomic(int) counter;`

1.2 Квалификатор _Atomic

Ключовата дума _Atomic може да се използва и като квалификатор на тип. C89 добавя квалификаторите const и volatile, C99 добавя квалификатора restrict (не е разгледан в тази книга), а C11 добавя към тях _Atomic. Резултатът от него е същия като спецификатора _Atomic(тип). Масиви и функции не могат да се квалифицират с _Atomic.

Пример: `_Atomic int counter;`

За съжаление, от спецификацията не става ясно каква е разликата при използване на ключовата дума _Atomic като квалификатор и спецификатор на тип.

1.3 Оператор _Alignof

Операторът _Alignof връща модула на подравняване на своя операнд. Той има следния синтаксис:

_Alignof(тип)

Следващият пример показва модулите на подравняване на базовите типове.

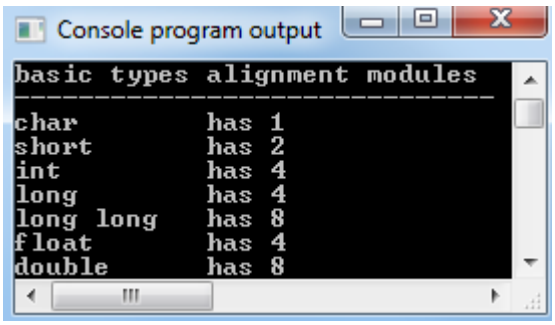
Пример 1-1:

```
#include <stdio.h>

int main(void)
```

```
{
	printf("basic types alignment modules\n");
	printf("-----------------------------\n");
	printf("char		has %d\n", _Alignof(char));
	printf("short		has %d\n", _Alignof(short));
	printf("int		has %d\n", _Alignof(int));
	printf("long		has %d\n", _Alignof(long));
	printf("long long	has %d\n", _Alignof(long long));
	printf("float		has %d\n", _Alignof(float));
	printf("double		has %d\n", _Alignof(double));

	return 0;
}
```



1.4 Спецификатор за подравняване _Alignas

Този спецификатор позволява да променяте естествения модул на подравняване на променливите. Той има следния синтаксис:

_Alignas(тип)

или

_Alignas(константен-израз)

Спецификаторът _Alignas има следните ограничения:

1. Ако един тип има подравняване m, не може да му зададете подравняване n, където n<m.

Пример: приема се, че $m_{int} = 4$, $m_{long} = 8$

```
_Alignas(8)    int  y; /* Правилно   */
_Alignas(long) int  y; /* Правилно, на y се задава подравняване като на long */
_Alignas(2)    long x; /* Неправилно, 2<m_long */
```

2. Не може да се прилага в typedef декларация към битово поле, параметър на функция или register-променлива.

Пример:

```
typedef _Alignas(int) unsigned short uint16; /* Неправилно */

struct ms
{
      _Alignas(2) _Bool b:1;    /* Неправилно */
};

void foo(_Alignas(4) short s)   /* Неправилно */
{

}

_Alignas(4) register short s;   /* Неправилно */
```

3. **константен-израз** трябва да се изчислява до валиден модул на подравняване.

Пример:

```
_Alignas(3) short x; /* Неправилно */
_Alignas(4) short y; /* Правилно   */
```

Следващият пример демонстрира влиянието на _Alignas върху размера памет, заеман от структура.

Пример 1-2:

```
/* m_int = 4, m_long = 8 */

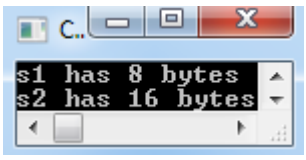
#include <stdio.h>

struct s1
{
      short s; /* s има подравняване 2 */
      int   i; /* i има подравняване 4 */
};

struct s2
{
      _Alignas(int)       short s; /* s има подравняване като тип int       */
      _Alignas(long long) int   i; /* i има подравняване като тип long long */
};

int main(void)
{
      printf("s1 has %d bytes\n", sizeof(struct s1));
      printf("s2 has %d bytes\n", sizeof(struct s2));

      return 0;
}
```



Изменението на естественото подравняване на членовете на структурата s2

влияе на начина, по-който те се разполагат в паметта. Компилаторът е принуден да разположи членовете на адреси кратни на зададения модул на подравняване, което води до увеличаване на паметта, заемана от тази структура.

1.5 Спецификатор _Noreturn

C99 въвежда спецификатор inline за функции. C11 добавя нов спецификатор за функции, наречен _Noreturn. Функция, декларирана със спецификатор _Noreturn, не трябва да се връща в извикващото я ниво. Ако функцията е възможно да се върне в извикващото я ниво, компилаторът ще генерира диагностично съобщение за това.

Пример:

```
_Noreturn void f(void)
{
      abort(); /* abort() води до прекратяване на програмата */
}

_Noreturn void g(int i)
{
      if (i > 0) { abort() };
}
```

Функцията abort() (декларирана е в <stdlib.h>) е библиотечна функция, която води до принудително прекратяване изпълнението на програмата. Следователно функцията f(), която извиква abort(), никога няма да се върне в точката си на извикване. Функцията g() обаче може да се върне в точката си на извикване в зависимост от стойността на аргумента, с който се извиква. Компилаторът ще генерира диагностично съобщение.

1.6 Анонимни структури и обединения

C11 въвежда концепцията за анонимни структури и обединения. Структура или обединение, декларирани без име, се нарича анонимна структура, респективно анонимно обединение. Те не могат да бъдат декларирани самостоятелно, а само в декларацията на нормална структура или обединение. Членовете на анонимната структура или обединение се разглеждат като членове на съдържащата ги структура или обединение. Това правило се прилага рекурсивно, ако съдържащата ги структура или обединение също е анонимна.

Пример:

```
struct v {
      union { // анонимно обединение
            struct { int i, j; }; // анонимна структура
            struct { long k, l; } w; //!!!Внимание, това не е анонимна структура
      };
      int m;
} v1;

v1.i = 2;   // Правилно
v1.k = 3;   // Неправилно: вътрешната структура не е анонимна
v1.w.k = 5; // Правилно
```

1.7 Многопоточност

1.7.1 Управление на многопоточността в C11

В тази точка ще Ви запозная с основните средства (типове, константи и функции), предоставени от C11 за създаване на многопоточни програми.

Типове

thrd_t

Тип, който съдържа име на поток.

mtx_t

Тип, който съдържа име на мютекс.

thrd_start_t

Тип, представляващ указател към функция, която връща тип int и има параметър void-указател, т.е. . **int (*)(void*)**.

По същество потокът изпълнява функция, която се извиква при неговото създаване. В C11 тази функция трябва да изглежда така:

int име-функция-на-поток(void* параметър);

Константи

mtx_plain

Стойност, използвана за създаване на мютекс.

thrd_success

Стойност, връщана от функции, която показва, че заявената операция е успешно изпълнена.

thrd_error

Стойност, връщана от функции, която показва, че заявената операция не е изпълнена поради възникване на грешка.

thrd_nomem

Стойност, връщана от функции, която показва, че заявената операция не е изпълнена, поради липса на достатъчно памет.

Функции

```
int mtx_init(mtx_t *mtx, int type);
int mtx_lock(mtx_t *mtx);
int mtx_unlock(mtx_t *mtx);
int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);
_Noreturn void thrd_exit(int res);
int thrd_join(thrd_t thr, int *res);
```

Функцията **mtx_init()** се използва за създаване на мютекс. Ако операцията е успешна, в променливата, сочена от mtx, се записва уникална стойност, с която мютексът се идентифицира. Параметърът type определя свойствата на мютекса. Засега използвайте стойността mtx_plain. При успех mtx_init() връща thrd_succeed, в противен случай резултатът е thrd_error.

Функцията **mtx_lock()** заключва мютекса, сочен от mtx. Потокът, от който се извиква тази функция, влиза в блокирано състояние, ако мютексът е бил вече заключен от друг поток. При успех mtx_lock() връща thrd_succeed, в противен случай резултатът е thrd_error.

Функцията **mtx_unlock()** отключва мютекса, сочен от mtx. Мютексът трябва преди това да е бил заключен от извикващия поток. При успех mtx_unlock() връща thrd_succeed, в противен случай резултатът е thrd_error.

Функцията **thrd_create()** създава и стартира нов поток. Параметърът func представлява функцията, която се извиква при стартиране на потока, а обектът сочен от arg се използва като аргумент на func, т.е. . func(arg). Ако създаването на потока е успешно, променливата, сочена от thr, се инициализира с уникална стойност, която идентифицира потока. При успех thrd_create() връща thrd_succeed, при липса на достатъчно памет – thrd_nomem, и thrd_error, ако операцията не може да бъде изпълнена.

Функцията **thrd_exit()** завършва изпълнението на потока, от който се извиква и записва резултата от изпълнението в параметъра res.

Функцията **thrd_join()** блокира потока, който я извиква, докато потокът, сочен от thr завърши изпълнението си. Ако параметърът res не е нулев указател, в него се съхранява резултата от изпълнението на потока, сочен от thr. При успех thrd_join() връща thrd_succeed, в противен случай резултатът е thrd_error.

1.7.2 Примери

Създаване на поток

Създаването на поток изисква следните стъпки. Първо трябва да дефинирате променлива тип thrd_t, която ще се инициализира с уникално име на потока. След това трябва да си дефинирате функция, която ще се извиква при стартирането на потока. Ако искате да подадете някакви данни на тази функция, ще трябва също да ги дефинирате. Например може да създадете структура, която съдържа необходимите данни, да дефинирате и инициализирате структурна-променлива, чийто адрес ще бъде подаден при създаването на потока. Ако не е необходимо да подавате данни, просто използвайте NULL за аргумент.

Пример 1-3:

```
#include <stdio.h>
#include <threads.h>
int do_something(void* p);

int main(void)
{
	thrd_t  myThread;//Променлива на потока

	//Стартиране на потока
	switch(thrd_create(&myThread,do_something, NULL))
	{
	case thrd_success:
		printf("Thread started.\n");
		break;

	case thrd_nomem:
		printf("Failed to allocate thread memory.\n");
		//Умишлено няма break или друг оператор за преход
	case thrd_error:
		printf("Failed to start thread.\n");
		thrd_exit(1); // Излизане от главния поток с код за грешка
	}

	//Главният поток изчаква завършването на потока myThread
	thrd_join(myThread, NULL);

	return 0;
}

//Функция, която се извиква при стартиране на потока
int do_something(void* p)
{
	printf("Now the thread myThread is running.\n");
	thrd_exit(0); // Завършване на потока успешно
}
```



Функцията thrd_create() се използва в управляващия израз на switch. Ако стартирането на потока myThread е успешно (резултатът е thrd_success), на екрана се отпечатва съобщението **Thread started.** и главният поток също продължава изпълнението на кода си. Забележете, че ако thrd_create() върне като резултат thrd_nomem, на екрана ще се отпечата съобщението **Failed to allocate thread memory.** Операторът break или извикване на thrd_exit() във втория case умишлено е изпуснат. Затова програмата ще продължи с изпълнение на операторите на case thrd_error, ще отпечата съобщението **Failed to start thread.**, след което главният поток ще завърши изпълнението си, връщайки на операционната система резултат 1, с което показва, че е възникнал някакъв проблем.

Завършване на поток

Излизането от поток става с функцията thrd_exit(). Тя може да бъде извикана от всяка точка на потока. Аргументът, който се подава на функцията, представлява резултат от изпълнението на потока и може да се използва за каквото поискате например като статус на потока. Типично стойност 0 означава, че потокът е завършил изпълнението си успешно. Други стойности могат да се използват да укажат дали е имало някакви проблеми по време на изпълнението на потока. Вие може да извиквате thrd_exit() от всеки поток, за да излезете от него, дори и от главния поток. Разгледайте отново Пример 31-3.

Присъединяване на един поток към друг

Ако един поток трябва да изчака друг поток да завърши изпълнението си преди да продължи, е необходимо да извика функцията thrd_join(). Потокът, който извиква thrd_join(), се блокира. Типично главният поток изчаква изпълнението на потока или потоците, създадени от него преди да завърши изпълнението си. Ако пропуснете да направите това, главният поток може да завърши изпълнението си преди "дъщерните" потоци, прекратявайки тяхното изпълнение преждевременно. Следващият пример демонстрира синхронизацията между два потока с thrd_join(). Потокът read_data чете данни от клавиатурата, а потокът process_data ги обработва. Коректното изпълнение изисква process_data да изчака read_data преди да обработи данните.

Пример 1-4:

```
#include <stdio.h>
#include <threads.h>

int read_data(void *pdata);
int process_data(void * pdata);
```

```
typedef struct MyData
{
  thrd_t id;
  int a;
  int b;
}MyData;

int main(void)
{
	thrd_t process_id;
	MyData mydata;

	//Стартиране на потока read_data
	switch(thrd_create(&mydata.id, read_data, &mydata))
	{
	case thrd_nomem:
		printf("Failed to allocate read_data thread memory.\n");
	case thrd_error:
		printf("Failed to start read_data thread.\n");
		thrd_exit(1); // Завършване на потока със стойност за грешка 1
	}

	//Стартиране на потока process_data
	switch(thrd_create(&process_id, process_data, &mydata))
	{
	case thrd_nomem:
		printf("Failed to allocate process_data thread memory.\n");
	case thrd_error:
		printf("Failed to start process_data thread.\n");
		thrd_exit(1); // Завършване на потока с грешка
	}

	//Изчакване на потока proccess_data
	if(thrd_join(process_id, NULL) == thrd_error)
	{
		printf("process_data thread join by main thread failed.\n");
		thrd_exit(1); // Завършване на потока със стойност за грешка 1
	}

	return 0;
}

//Поток read_data
int read_data(void *pdata)
{
	MyData* pd = (MyData*)pdata;
	printf("Enter two integers: ");
	scanf("%d%d", &pd->a, &pd->b);

	thrd_exit(0);	// Завършване на потока успешно
}

// Поток process_data
int process_data(void * pdata)
{
	MyData* pd = (MyData*)pdata;
	int result = 0;

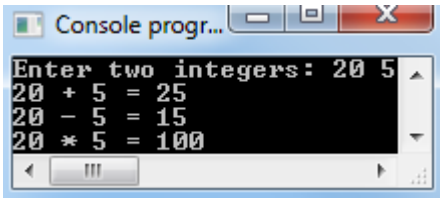
	//Изчакване на потока read_data
	if(thrd_join(pd->id, &result) == thrd_error)
	{
		printf("read_data thread join by process_data failed.\n");
		thrd_exit(-1);	// Завършване на потока със стойност за грешка -1
```

```
	}

	if(-1 == result) // Проверка на статуса на потока read_data
	{
		printf("thread read_data failed.\n");
		thrd_exit(-2);   // Завършване на потока със стойност за грешка -2
	}

	printf("%d + %d = %d\n", pd->a, pd->b, pd->a + pd->b);
	printf("%d - %d = %d\n", pd->a, pd->b, pd->a - pd->b);
	printf("%d * %d = %d\n", pd->a, pd->b, pd->a * pd->b);

	thrd_exit(0);      // Завършване на потока успешно
}
```



Използване на мютекс

Разгледайте следния пример, който използва два потока, които записват данни в една и съща структурна променлива. Единият поток записва стойности 10, 20 и 30, а другият 40, 50, 60. Главният поток отпечатва данните, които последният поток е записал в променливата.

Пример 1-5:

```
#include <stdio.h>
#include <threads.h>

mtx_t mutex;

int whrite_thread1(void*);
int whrite_thread2(void*);

typedef struct
{
	int a;
	int b;
	int c;
}myType;

int main(void)
{
	thrd_t thread1, thread2;
	myType myData;

	mtx_init(&mutex, mtx_plain);

	thrd_create( &thread1, &whrite_thread1, &myData);
	thrd_create( &thread2, &whrite_thread2, &myData);

	thrd_join( thread1, NULL);
	thrd_join( thread2, NULL);
```

```
	printf("myData.a = %d\n", myData.a);
	printf("myData.b = %d\n", myData.b);
	printf("myData.c = %d\n", myData.c);

	return 0;
}

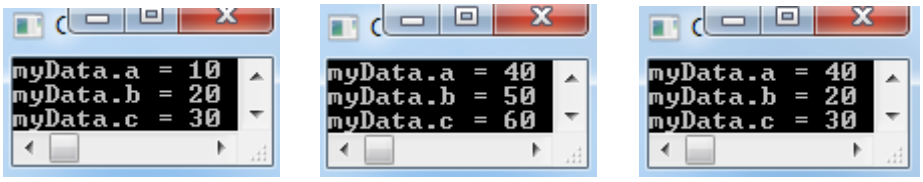
int whrite_thread1(void* pData)
{
	myType *p = (myType*)pData;

	p->a = 10;
	for(int i = 0; i < 100000; i++); // Закъснение, което просто да забави
thread1
	p->b = 20;
	p->c = 30;
	thrd_exit(0);
}

int whrite_thread2(void* pData)
{
	myType *p = (myType*)pData;

	p->a = 40;
	p->b = 50;
	p->c = 60;

	thrd_exit(0);
}
```



Ако стартирате програмата многократно, ще получите най-различни резултати. Например 10, 20, 30 (което означава, че потока thread1 последно е записвал данни в променливата) или 40, 50, 60 (което означава, че потока thread2 последно е записвал данни в променливата), или комбинация от двете групи числа, като 40, 20, 30 (което означава, че двата потока са записвали по едно и също време в променливата). Така както е написана програмата и двата потока имат равни права на достъп до структурната променлива myData и могат по едно и също време да я записват. Следващият пример е коригирана версия на предходния. Сега всеки поток използва мютекс, за да се защити от вмешателство от страна на другия поток.

Пример 1-6:

```
#include <stdio.h>
#include <threads.h>

mtx_t mutex;

int whrite_thread1(void*);
int whrite_thread2(void*);
```

```
typedef struct {
	int a;
	int b;
	int c;
}myType;

int main(void)
{
	thrd_t thread1, thread2;
	myType myData;
	mtx_init(&mutex, mtx_plain);
	thrd_create( &thread1, &whrite_thread1, &myData);
	thrd_create( &thread2, &whrite_thread2, &myData);

	thrd_join( thread1, NULL);
	thrd_join( thread2, NULL);

	printf("myData.a = %d\n", myData.a);
	printf("myData.b = %d\n", myData.b);
	printf("myData.c = %d\n", myData.c);

	return 0;
}

int whrite_thread1(void* pData)
{
	myType *p = (myType*)pData;

	mtx_lock(&mutex);
	p->a = 10;
	for(int i = 0; i < 100000; i++);
	p->b = 20;
	p->c = 30;
	mtx_unlock(&mutex);

	thrd_exit(0);
}

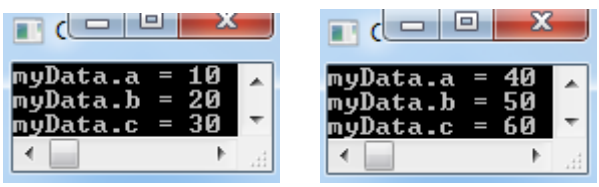
int whrite_thread2(void* pData)
{
	myType *p = (myType*)pData;

	mtx_lock(&mutex);
	p->a = 40;
	p->b = 50;
	p->c = 60;
	чmtx_unlock(&mutex);

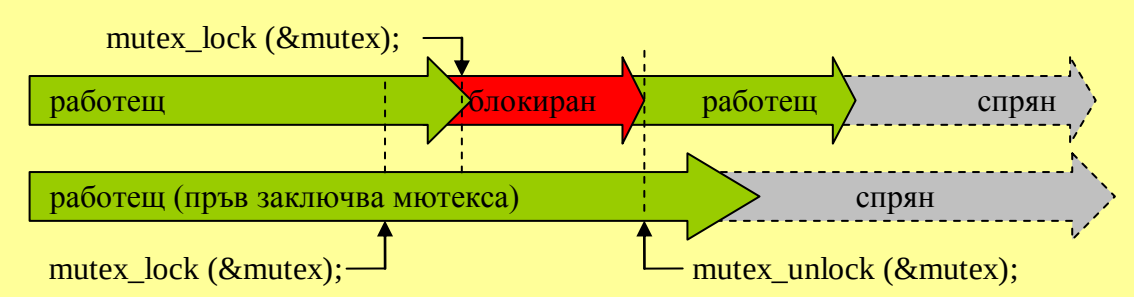
	thrd_exit(0);
}
```

Критична секция

Критична секция



Потокът, който пръв заключи мютекса придобива правото да записва единствен в променливата. Фиг.100 илюстрира Пример 1-6.



Фиг. 100 Илюстрация към Пример 1-6

Част IV: Стандартна Библиотека





Въведение в стандартната библиотека

1.1 Общи сведения

Езикът C предоставя голям набор от стандартни функции, наречени библиотечни, които можете да използвате наготово. Всяка библиотечна функция е декларирана в хедър, чието съдържание се прави налично в програмата чрез #include предпроцесорната директива.

Забележка[1]: Хедърът не е задължително файл. На компилаторите е позволено да предефинират вътрешно съдържанието на хедърите. Въпреки това на практика C стандартните хедъри се съхраняват във файлове.

Освен декларации на стандартни библиотечни функции, хедър-файловете съдържат декларации на макроси и типове данни, дефинирани в самия стандарт.

По отношение на функционалността си библиотечните функции могат да се разделят на следните категории:

- Функции за вход-изход
- Функции за обработка на низове и символи
- Функции за преобразуване
- Математически функции
- Функции за управление на динамична памет
- Управляващи функции
- Функции за управление на многопоточност
- Други функции

ANSI C стандартът включва следните хедъри:

№	хедър-файл	кратко описание
1	<assert.h>	Дефинира макроса assert() за диагностични цели.
2	<complex.h> C99	Осигурява поддръжка за комплексни числа
3	<ctype.h>	Функции за обработка на символи
4	<errno.h>	Съобщения за грешки в библиотечни функции
5	<fenv.h> C99	Осигурява поддръжка на числа с плаваща запетая
6	<float.h>	Дефинира имплементационно-зависими граници за типове

		с плаваща запетая
7	**<inttypes.h>** C99	Форматни преобразувания на целочислени типове
8	**<iso646.h>** C95	Макроси-синоними на С логическите и битовите оператори
9	**<limits .h>**	Дефинира имплементационно-зависими граници за целочислени типове и други неща
10	**<locale.h>**	Дефинира средства за подръжка на локализация
11	**<math.h>**	Функции за математически операции
12	**<setjmp.h>**	Дефинира средства за заобикаляне на нормалното извикване и връщане от функция
13	**<signal.h>**	Дефинира средства за обработка на сигнали
14	**<stdalign.h>** C11	Дефинира макроси за управление на подравняването
15	**<stdarg.h>**	Поддръжка на функции с променлив брой аргументи
16	**<stdatomic.h>** C11	Поддръжка на атомични типове
17	**<stdbool.h>** C99	Дефинира макроси за поддръжка на тип _Bool
18	**<stddef.h>**	Съдържа стандартни декларации на макроси и типове
19	**<stdint.h>** C99	Дефинира целочислени типове с фиксирана дължина
20	**<stdio.h>**	Функции за вход-изход
21	**<stdlib.h>**	Функции за обща употреба
22	**<stdnoreturn.h>** C11	Дефинира макрос noreturn
23	**<string.h>**	Функции за обработка на низове
24	**<tgmath.h>** C99	Дефинира поддръжка на математика с обобщени типове
25	**<thread.h>** C11	Функции за поддръжка на многопоточност
26	**<time.h>**	Функции за обработка на време и дата
27	**<uchar.h>** C11	Функции за поддръжка на Unicode символи
28	**<wchar.h>** C95	Функции за обработка на широки и многобайтови низове
29	**<wctype.h>** C95	Функции за обработка на широки символи

Табл. 53 Стандартни хедъри в C89

Горните индекси C95, C99 и C11 показват хедърите добавени в C95, C99 и C11 съответно.

Забележка: В момента е описана напълно само библиотеката по стандарта C89. Нововъведенията въведени в C99 и C11 ще бъдат добавени в следващи издания на книгата.



Описание на стандартната библиотека

1.1 Описание на стандартните хедъри

1.2 <assert.h>

1.2.1 Описание

Хедърът **<assert.h>** се използва за диагностични цели. Той дефинира само една макро-функция assert()

1.2.2 Декларации на типове

-

1.2.3 Декларации на макроси

-

1.2.4 Декларации на функции

```
void assert (int expression);
```

Забележка: Макро-функцията assert е представена като обикновена функция за по лесно възприемане.

1.2.5 Описание на функциите (макро-функциите)

1.2.5.1 assert()

```
#include <assert.h>
void assert (int expression);
```

Макросът assert() се използва за диагностика на целочислен израз. Ако *expression* е 0 (FALSE) се изпраща информация за грешка към стандартния поток за грешка **stderr** и се извиква функцията abort() , която прекратява изпълнението на програмата. Съобщението за грешка, което се извежда не е стандартизирано, но е от рода на:

Assertion failed: expression , file filename , line line-number

Ако *expression* е различен от 0 (TRUE) функцията не прави нищо и програмата продължава изпълнението си.
Макросът assert() типично се използва само в развойната фаза. В производствената фаза той обикновено се маха. За целта не е задължително да се изтрие от файла. Достатъчно е само пред включването на хедъра <assert.h> да се дефинира макроса NDEBUG. Когато този макрос е дефиниран, функционалността на макроса assert() е изключена и той не

прави нищо.

Пример 1-1:

```
/* #define NDEBUG */ /* Изключва функционалността на assert() */
#include <stdio.h>
#include <assert.h>

void foo(int* ptr)
{
	/* Диагностичен код */
	assert(NULL != ptr);

	/* Следва останалия код */
	/*.....................*/
}

int main(void)
{
	int* p = NULL;

	foo(p);

	return 0;
 }
```



1.3 <complex.h>

1.3.1 Описание

Хедърът <complex.h>[1] дефинира макроси и декларира функции, които поддържат комплексна аритметика.

Забележка[1]: В C11 ако е дефиниран макроса __STDC_NO_COMPLEX__, имплементацията не е необходимо да осигурява хедъра <complex.h>.

1.3.2 Декларации на типове

-

1.3.3 Декларации на макроси

complex

Разширява се до **_Complex**.

_Complex_I

Разширява се до константен израз от тип **const float _Complex** със стойността на имагинерната единица (число i, така че $i^2 = -1$).

imaginary[1]

Разширява се до **_Imaginary**.

Imaginary_I[1]

Разширява се до константен израз от тип **const float _Imaginery** със стойността на имагинерната единица.

Забележка: Този макрос е дефиниран само ако имплементацията поддържа имагинерни типове.

I

Разширява се до **_Imaginary_I** или **_Complex_I**. Ако **_Imaginary_I** не е дефиниран, **I** се разширява до **_Complex_I**.

1.3.4 Декларации на функции

```
double complex      cacos (double complex z);
float complex       cacosf(float complex z);
long double complex cacosl(long double complex z);
double complex      casin (double complex z);
float complex       casinf(float complex z);
long double complex casinl(long double complex z);
double complex      catan (double complex z);
float complex       catanf(float complex z);
long double complex catanl(long double complex z);
double complex      ccos  (double complex z);
float complex       ccosf (float complex z);
long double complex ccosl (long double complex z);
double complex      csin  (double complex z);
float complex       csinf (float complex z);
long double complex csinl (long double complex z);
double complex      ctan  (double complex z);
float complex       ctanf (float complex z);
long double complex ctanl (long double complex z);
double complex      cacosh(double complex z);
float complex       cacoshf(float complex z);
long double complex cacoshl(long double complex z);
double complex      casinh(double complex z);
float complex       casinhf(float complex z);
long double complex casinhl(long double complex z);
double complex      catanh(double complex z);
float complex       catanhf(float complex z);
long double complex catanhl(long double complex z);
double complex      ccosh(double complex z);
float complex       ccoshf(float complex z);
long double complex ccoshl(long double complex z);
double complex      csinh(double complex z);
float complex       csinhf(float complex z);
```

```
long double complex csinhl(long double complex z);
double complex      ctanh(double complex z);
float complex       ctanhf(float complex z);
long double complex ctanhl(long double complex z);
double complex      cexp(double complex z);
float complex       cexpf(float complex z);
long double complex cexpl(long double complex z);
double complex      clog(double complex z);
float complex       clogf(float complex z);
long double complex clogl(long double complex z);
double              cabs(double complex z);
float               cabsf(float complex z);
long double         cabsl(long double complex z);
double complex      cpow(double complex x, double complex y);
float complex       cpowf(float complex x, float complex y);
long double         complex cpowl(long double complex x, long double complex y);
double complex      csqrt(double complex z);
float complex       csqrtf(float complex z);
long double complex csqrtl(long double complex z);
double              carg(double complex z);
float               cargf(float complex z);
long double         cargl(long double complex z);
double              cimag(double complex z);
float               cimagf(float complex z);
long double         cimagl(long double complex z);
double              complex conj(double complex z);
float complex       conjf(float complex z);
long double complex conjl(long double complex z);
double complex      cproj(double complex z);
float complex       cprojf(float complex z);
long double complex cprojl(long double complex z);
double              creal(double complex z);
float               crealf(float complex z);
long double         creall(long double complex z);
```

1.3.5 Описание на функциите (макро-функциите)

1.3.5.1 cacos, cacosf, cacosl

1.3.5.2 casin, casinf, casinl

1.3.5.3 catan, catanf, catanl

1.3.5.4 ccos, ccosf, ccosl

1.3.5.5 csin, csinf, csinl

1.3.5.6 ctan, ctanf, ctanl

1.3.5.7 cacosh, ccoshf, cacoshl

1.3.5.8 casinh, casinhf, casinhl

1.3.5.9 catanh, catanhf. catnahl

1.3.5.10 ccosh, ccoshf, ccoshl

1.3.5.11 csinh, csinhf, csinhl

1.3.5.12 ctanh, ctanhf, ctanhl

1.3.5.13 cexp, cexpf, cexpl

1.3.5.14 clog, clogf, clogl

1.3.5.15 cabs, cabsf, cabsl

1.3.5.16 cpow, cpowf, cpowl

1.3.5.17 csqrt, csqrtf, csqrtl

1.3.5.18 carg, cargf, cargl

1.3.5.19 cimag, cimagf, cimagl

1.3.5.20 conj, conjf, conjl

1.3.5.21 cproj, cprojf, cprojl

1.3.5.22 creal, crealf, creall

1.4 <ctype.h>

1.4.1 Описание

Хедърът **<ctype.h>** се използва за тестване и преобразуване на символи.

1.4.2 Декларации на типове

-

1.4.3 Декларации на макроси

-

1.4.4 Декларации на функции

```
int isalnum (int character);
int isalpha (int character);
int isdigit (int character);
int iscntrl (int character);
int isgraph (int character);
int islower (int character);
int isupper (int character);
int isprint (int character);
int ispunct (int character);
int isspace (int character);
int isxdigit(int character);
int tolower (int character);
```

```
int toupper (int character);
```

1.4.5 Описание на функциите (макро-функциите)

1.4.5.1 isalnum()

```
#include <ctype.h>
int isalnum(int character);
```

Функцията isalnum() връща стойност различна от 0 (TRUE) ако ***character*** е буква от a-z или A-Z, или цифра от 0-9. В противен случай връща 0 (FALSE).

Пример 1-2:

```
#include<ctype.h>
#include<stdio.h>

int main(void)
{
	char string[4] = "a1#";
	int  i;

	for(i = 0; i < (sizeof(string) / sizeof(char) - 1) ; i++)
	{
		if (isalnum(string[i]))
		{
			printf("string[%d] is letter or digit\n",i);
		}
		else
		{
			printf("string[%d] isn't letter nor digit\n",i);
		}
	}

	return 0;
}
```

1.4.5.2 isalpha()

```
#include <ctype.h>
int isalpha(int character);
```

Функцията isalpha() връща стойност различна от 0 (TRUE) ако ***character*** е буква от a-z или A-Z. В противен случай връща 0 (FALSE).

Пример 1-3:

```
#include<ctype.h>
#include<stdio.h>

int main(void)
```

```
{
	char string[4] = "a1#";
	int  i;

	for(i = 0; i < (sizeof(string) / sizeof(char) - 1) ; i++)
	{
		if (isalpha(string[i]))
		{
			printf("string[%d] is a letter\n",i);
		}
		else
		{
			printf("string[%d] isn't a letter\n",i);
		}
	}

	return 0;
}
```

1.4.5.3 isdigit()

```
#include <ctype.h>
int isdigit(int character);
```

Функцията isdigit() връща стойност различна от 0 (TRUE) ако ***character*** е цифра от 0-9. В противен случай връща 0 (FALSE).

Пример 1-4:

```
#include<ctype.h>
#include<stdio.h>

int main(void)
{
	char string[4] = "a1#";
	int  i;

	for(i = 0; i < (sizeof(string) / sizeof(char) - 1) ; i++)
	{
		if (isdigit(string[i]))
		{
			printf("string[%d] is a digit\n",i);
		}
		else
		{
			printf("string[%d] isn't a digit\n",i);
		}
	}

	return 0;
}
```

1.4.5.4 iscntrl()

```
#include <ctype.h>
int iscntrl(int character);
```

Функцията iscntrl() връща стойност различна от 0 (TRUE) ако ***character*** е управляващ символ (0 - 0x1F и 0x7F в ASCII кодовата таблица, Табл.2). В противен случай връща 0 (TRUE).

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Табл. 54 ASCII кодова таблица

Пример 1-5:

```
#include<ctype.h>
#include<stdio.h>

int main(void)
{
	char string[4] = "\t1#";
	int  i;

	for(i = 0; i < (sizeof(string) / sizeof(char) - 1) ; i++)
	{
		if (iscntrl(string[i]))
		{
			printf("string[%d] is a control symbol\n",i);
		}
		else
		{
			printf("string[%d] isn't a control symbol\n",i);
		}
	}
```

```
	return 0;
}
```

1.4.5.5 isgraph()

```
#include <ctype.h>
int isgraph(int character);
```

Функцията isgraph() връща стойност различна от 0 (TRUE) ако ***character*** е графичен символ (0x21 - 0x7E в ASCII кодовата таблица). В противен случай връща 0 (FALSE).

Пример 1-6:

```
#include<ctype.h>
#include<stdio.h>

int main(void)
{
	char string[4] = "\t1#";
	int  i;

	for(i = 0; i < (sizeof(string) / sizeof(char) - 1) ; i++)
	{
		if (isgraph(string[i]))
		{
			printf("string[%d] is a graphic symbol\n",i);
		}
		else
		{
			printf("string[%d] isn't a praphic symbol\n",i);
		}
	}

	return 0;
}
```

1.4.5.6 islower()

```
#include <ctype.h>
int islower(int character);
```

Функцията islower() връща стойност различна от 0 (TRUE) ако ***character*** е символ от a-z. В противен случай връща 0 (FALSE).

Пример 1-7:

```
#include<ctype.h>
#include<stdio.h>

int main(void)
{
```

```
	char string[4] = "aB1";
	int  i;

	for(i = 0; i < (sizeof(string) / sizeof(char) - 1) ; i++)
	{
		if (islower(string[i]))
		{
			printf("string[%d] is a small letter\n", i);
		}
		else
		{
			printf("string[%d] isn't a small letter\n", i);
		}
	}

	return 0;
}
```

1.4.5.7 isupper()

```
#include <ctype.h>
int isupper(int character);
```

Функцията isupper() връща стойност различна от 0 (TRUE) ако ***character*** е символ от A-Z. В противен случай връща 0 (FALSE).

Пример 1-8:

```
#include<ctype.h>
#include<stdio.h>

int main(void)
{
	char string[4] = "aB1";
	int  i;

	for(i = 0; i < (sizeof(string) / sizeof(char) - 1) ; i++)
	{
		if (isupper(string[i]))
		{
			printf("string[%d] is a capital letter\n", i);
		}
		else
		{
			printf("string[%d] isn't a capital letter\n", i);
		}
	}

	return 0;
}
```

1.4.5.8 isprint()

```
#include <ctype.h>
int isprint(int character);
```

Функцията isprint() връща стойност различна от 0 (TRUE) ако ***character*** е символ, който може да се отпечата на екрана (0x20 - 0x7E в ASCII кодовата таблица, Табл.2). В противен случай връща 0 (FALSE).

Пример 1-9:

```
#include<ctype.h>
#include<stdio.h>

int main(void)
{
	char string[4] = "\tB1";
	int  i;

	for(i = 0; i < (sizeof(string) / sizeof(char) - 1) ; i++)
	{
		if (isprint(string[i]))
		{
			printf("string[%d] is a printable symbol\n", i);
		}
		else
		{
			printf("string[%d] isn't a printable simbol\n", i);
		}
	}

	return 0;
}
```

1.4.5.9 ispunct()

```
#include <ctype.h>
int ispunct(int character);
```

Функцията ispunct() връща стойност различна от 0 (TRUE) ако ***character*** е пунктуационен символ (всеки символ, който може да се отпечата на екрана без интервал, a-z, A-Z и 0-9). В противен случай връща 0 (FALSE).

Пример 1-10:

```
#include<ctype.h>
#include<stdio.h>

int main(void)
{
	char string[4] = "#B ";
	int  i;

	for(i = 0; i < (sizeof(string) / sizeof(char) - 1) ; i++)
	{
```

```
            if (ispunct(string[i]))
            {
                  printf("string[%d] is a printable symbol\n", i);
            }
            else
            {
                  printf("string[%d] isn't a printable simbol\n", i);
            }
      }

      return 0;
}
```

1.4.5.10 isspace()

```
#include <ctype.h>
int isspace(int character);
```

Функцията isspace() връща стойност различна от 0 (TRUE) ако ***character*** е white-символ (интервал ‘ ‘, хор. табулация ‘\t’, вер. табулация ‘\v’, връщане на курсора ‘\r’, нов ред ‘\n’, нова страница ‘\f’). В противен случай връща 0 (FALSE).

Пример 1-11:

```
#include<ctype.h>
#include<stdio.h>

int main(void)
{
      char string[4] = "\vB ";
      int  i;

      for(i = 0; i < (sizeof(string) / sizeof(char) - 1) ; i++)
      {
            if (isspace(string[i]))
            {
                  printf("string[%d] is a white symbol\n", i);
            }
            else
            {
                  printf("string[%d] isn't a white simbol\n", i);
            }
      }

      return 0;
}
```

1.4.5.11 isxdigit()

```
#include <ctype.h>
int isxdigit(int character);
```

Функцията isxdigit() връща стойност различна от 0 (TRUE) ако ***character*** е цифра от шестнадесетичната бройна система (0-9, a-f и A-F). В противен случай връща 0 (FALSE).

Пример 1-12:

```
#include<ctype.h>
#include<stdio.h>

int main(void)
{
	char string[4] = "1FR";
	int  i;

	for(i = 0; i < (sizeof(string) / sizeof(char) - 1) ; i++)
	{
		if (isxdigit(string[i]))
		{
			printf("string[%d] is a hexidecimal digit\n", i);
		}
		else
		{
			printf("string[%d] isn't a hexidecimal digit\n", i);
		}
	}

	return 0;
}
```

1.4.5.12 tolower()

```
#include <ctype.h>
int tolower(int character );
```

Функцията tolower() връща подадения аргумент в долен регистър ако е буква, в противен случай връща аргументта без да го изменя.

Пример 1-13:

```
#include<ctype.h>
#include<stdio.h>
#include<string.h>

int main(void)
{
	int  loop;
	char string[]="THIS IS A TEST";

	for(loop=0;loop<strlen(string);loop++){
		string[loop]=tolower(string[loop]);
	}

	printf("%s\n",string);

	return 0;
}
```

1.4.5.13 toupper()

```
#include <ctype.h>
int toupper(int character );
```

Функцията tolower() връща подадения аргумент в горен регистър ако е буква, в противен случай връща аргументта без да го изменя.

Пример 1-14:

```
#include<ctype.h>
#include<stdio.h>
#include<string.h>

int main(void)
{
	int  loop;
	char string[]="THIS IS A TEST";

	for(loop=0;loop<strlen(string);loop++)
	{
		string[loop]=toupper(string[loop]);
	}

	printf("%s\n",string);

	return 0;
}
```

1.5 <errno.h>

1.5.1 Описание

Хедърът **<errno.h>** дефинира няколко макроса, представляващи грешки, които могат да възникнат в определени стандартни библиотечни функции. Стойностите на тези макроси се записват в променлива **errno**[1] от тип int, при възникване на условие за грешка в съответната функция.

Забележка[1]: Съгласно C89, идентификатора **errno** не е задължително да е глобална променлива. Той може да се дефинира като макрос, който се разширява до модифицируема l-стойност от тип int. Например:

```
extern int* _errno();
#define errno  (* _errno())
```

Стандартът дефинира, че **errno** се нулира при стартиране на програмата, но никога не се нулира от библиотечна функция. Препоръчва се при използване на **errno** за проверка на грешка при извикване на библиотечна функция, **errno** да се нулира преди извикването на функцията и след това

да се провери стойността преди следващото извикване на друга библиотечна функция, т.е:

```
errno = 0;
library_function1(argument); /* Извикване на библиотечна функция */
if (ERROR_CODE == errno)
{
	/* Обработка на грешката */
}
```

където ERROR_CODE е някой от макросите за грешки (вижте 1.5.3 Декларации на макроси).

Имплементациите са свободни да дефинират свои макроси за грешки. Изискването е те да започват с буква E и цифра или с E и друга главна буква.

1.5.2 Декларации на типове

-

1.5.3 Декларации на макроси

C89 дефинира само следните макроси за грешки:

- EDOM - този код за грешка се записва в **errno** когато аргументът, с който се извиква функцията е извън дефинирания диапазон (**domain error**). Пример за възникване на такава грешка е извикването на функцията sqrt() с отрицателен аргумент.

Пример 1-15:

```
#include <stdio.h>
#include <math.h>
#include <errno.h>

int main(void)
{
	double r;

	errno = 0;
	r = sqrt(-2); /* sqrt() изисква аргумент > 0 */
	if (EDOM == errno)
	{
		printf("domain error occured in sqrt()\n");
	}

	return 0;
}
```

- ERANGE - този код за грешка се записва в **errno** когато резултата, връщан от функцията, не може да бъде представен от декларирания връщан тип (**range error**). Пример за възникване на такава грешка е

извикването на функцията pow() с твърде големи аргументи.

Пример 1-16:

```
#include <stdio.h>
#include <math.h>
#include <errno.h>

int main(void)
{
	double r;

	errno = 0;
	r = pow(20000000000, 40000000000);
	if (ERANGE == errno)
	{
		printf("range error occured in sqrt()\n");
	}

	return 0;
}
```

- EILSEQ (добавен от ISO/IEC 9899:1990/Amd 1:1995) - този код за грешка се записва в **errno** при възникване на грешка във функции за обработка на многобайтови символи (**illegal sequence**).

1.5.4 Декларации на функции

-

1.5.5 Описание на функциите (макро-функциите)

-

1.6 <fenv.h>

1.6.1 Описание

1.6.2 Декларации на типове

1.6.3 Декларации на макроси

1.6.4 Декларации на функции

1.6.5 Описание на функциите (макро-функциите)

1.7 <float.h>

1.7.1 Описание

Хедърът **<float.h>** дефинира макроси, които описват диапазона от стойности, прецизността и други свойства на типовете float, double и long double.

Числата с плаваща запетая се представят в т.нар. нормализирана форма:

±1 *0.мантиса *E ±експонента

1.7.2 Декларации на типове

-

1.7.3 Декларации на макроси

Следващата таблица описва минималните изисквания на C стантарта към числата с плаваща запетая.

Макрос	Стойност	Описание
FLT_ROUND	-1 - неопределен 0 - закръгляване към нула 1 - закръгляване към най-близката представима стойност 2 - закръгляване към +∞ (към следващото по-голямо число) 3 - закръгляване към -∞ (към следващото по-малко число)	Дефинира метода на закръгляване на числата с плаваща запетая.
FLT_RADIX	2	Дефинира базата (бройната система) на представяне на експонентата.
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG		Дефинира броя на цифрите в мантисата (дробната част).
FLT_DIG DBL_DIG LDBL_DIG	6 10 10	Максималния брой десетични цифри, които могат да бъдат представени без промяна след закръгляване.
FLT_MIN_EXP DBL_MIN_EXP LDBL_MIN_EXP		Минималната стойност на експонентата в база FLT_RADIX.
FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	-37 -37 -37	Минималната отрицателна стойност на експонентата в база 10.
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP		Максималната стойност на експонентата в база FLT_RADIX.
FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	+37 +37 +37	Минималната стойност на експонентата в база 10.
FLT_MAX DBL_MAX LDBL_MAX	1E+37 1E+37 1E+37	Максималното крайно число с плаваща запетая.
FLT_EPSILON DBL_EPSILON LDBL_EPSILON	1E-5 1E-9 1E-9	Положителната разлика между 1 и най-малкот представимо число по-голямо от 1.

FLT_MIN DBL_MIN LDBL_MIN	1E-37 1E-37 1E-37	Минималното крайно число с плаваща запетая.

Табл. 55 Макроси, описващи свойствата на числата с плаваща запетая

1.7.4 Декларации на функции

-

1.7.5 Описание на функциите (макро-функциите)

-

1.8 <inttypes.h>

1.8.1 Описание

1.8.2 Декларации на типове

1.8.3 Декларации на макроси

1.8.4 Декларации на функции

1.8.5 Описание на функциите (макро-функциите)

1.9 <iso646.h>

1.9.1 Описание

Хедърът <iso646.h> декларира макроси, които представляват синоними на C логическите и битовите оператори .

1.9.2 Декларации на типове

-

1.9.3 Декларации на макроси

```
#define     and         &&
#define     and_eq      &=
#define     bitand      &
#define     bitor       |
#define     compl       ~
#define     not         !
#define     not_eq      !=
#define     or          ||
#define     or_eq       |=
```

```
#define     xor         ^
#define     xor_eq      ^=
```

1.9.4 Декларации на функции

-

1.9.5 Оисание на функциите (макро-функциите)

-

1.10 <limits.h>

1.10.1 Описание

Хедърът **<limits.h>** дефинира макроси, които описват диапазона от стойности на всички целочислени типове.

1.10.2 Декларации на типове

-

1.10.3 Декларации на макроси

Следващата таблица описва минималните изисквания на C стантарта към целочислените типове.

Макрос	Стойност	Описание
CHAR_BIT	8	Брой на битовете в един байт.
SCHAR_MIN	-127	Минимална стойност на signed char.
SCHAR_MAX	+127	Максимална стойност на signed char.
UCHAR_MAX	255	Максимална стойност на unsigned char.
CHAR_MIN CHAR_MAX	SCHAR_MIN / 0 SCHAR_MAX/ UCHAR_MAX	Минимална и максимална стойност на тип char.
MB_LEN_MAX	1	Максимален брой байтове в многобайтов символ.
SHRT_MIN	-32767	Минимална стойност на short.
SHRT_MAX	+32767	Максимална стойност на short.
USHRT_MAX	65535	Максимална стойност на usigned short.
INT_MIN	-32767	Минимална стойност на int.
INT_MAX	+32767	Максимална стойност на int

UINT_MAX	65535	Максимална стойност на usigned int.
LONG_MIN	-2147483647	Минимална стойност на long.
LONG_MAX	+2147483647	Максимална стойност на long.
ULONG_MAX	4294967295	Максимална стойност на unsigned long.

Табл. 56 Минимални стойности на целочислените типове

1.10.4 Декларации на функции

-

1.10.5 Описание на функциите (макро-функциите)

-

1.11 <locale.h>

1.11.1 Описание

Хедърът **<locale.h>** подпомага създаването на програми на C, които са в състояние да се адаптират към местните културни конвенции. Например програмите могат да използват специфични набори от знаци или формати за валутна информация валидни в определни региони.

Хедърът <locale.h> декларира две функции **setlocale()** и **localeconv()**, структурен тип **lconv**, макрос **NULL** за нулева указателна константа, и макроси, чиито имена започват с **LC_** за категориите информация в един регион.

Функцията setlocale() ви позволява да запитвате или да задавате текущия регион. Информацията в региона е разделена на категории, които можете да запитвате и настройвате индивидуално. Следните целочислени макроси са определени за обозначаване на тези категории информация:

LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME

Тези макроси се разширяват до целочислени константни изрази с различни стойности. Имплементациите са свободни да дефинират допълнителни

макроси, започващи с LC_ следвани от главна буква.

Следващата таблица илюстрира правилата, които могат да се използват от четири държави за форматиране на парични величини.

Държава	Положителен формат	Отрицателен формат	Международен формат
Италия	L.1.234	-L.1.234	ITL.1.234
Холандия	F 1.234,56	F -1.234,56	NGL 1.234,56
Норвегия	kr1.234,56	kr1.234,56-	NOK 1.234,56
Швейцария	SFrs.1,234.56	SFrs.1,234.56C	CHF 1,234.56

За тези четири държави съответните стойности за паричните членове на структурата **lconv** са:

Членове на lconv	Италия	Холандия	Норвегия	Швейцария
int_curr_symbol	“ITL.”	“NLG ”	“NOK ”	“CHF ”
currency_symbol	“L.”	“F”	“kr”	“SFrs.”
mon_decimal_point	“”	“ ” ,	“ ” ,	“ ” .
mon_thousend_sep	“ ” .	“ ” .	“ ” .	“ ” ,
mon_grouping	“\3”	“\3”	“\3”	“\3”
positive_sign	“”	“”	“”	“”
negative_sign	“-”	“-”	“-”	“C”
int_frac_digits	0	2	2	2
frac_digits	0	2	2	2
p_cs_precedes	1	1	1	1
p_sep_by_space	0	1	0	0
n_cs_precedes	1	1	1	1
n_sep_by_space	0	1	0	0
p_sign_posn	1	1	1	1
n_sign_posn	1	4	2	2

1.11.2 Декларации на типове

```
struct lconv
{
	char *decimal_point;	/* "." */
	char *thousands_sep;	/* ""  */
	char *grouping;		/* ""  */
	char *int_curr_symbol;	/* ""  */
	char *currency_symbol;	/* ""  */
```

```
        char *mon_decimal_point;  /* ""  */
        char *mon_thousands_sep;  /* ""  */
        char *mon_grouping;       /* ""  */
        char *positive_sign;      /* ""  */
        char *negative_sign;      /* ""  */
        char int_frac_digits;     /* CHAR_MAX  */
        char frac_digits;         /* CHAR_MAX  */
        char p_cs_precedes;       /* CHAR_MAX  */
        char p_sep_by_space;      /* CHAR_MAX  */
        char n_cs_precedes;       /* CHAR_MAX  */
        char n_sep_by_space;      /* CHAR_MAX  */
        char p_sign_posn;         /* CHAR_MAX  */
        char n_sign_posn;         /* CHAR_MAX  */
};
```

Описание на членовете на структурата lconv

decimal_point

Символ за десетична точка, използван за непарични стойности. В подразбиращата се локализация този указател сочи към “.”.

thousands_sep

Символ за разделител на хилядните, използван за непарични стойности.

grouping

Указател към char-масив, чийто елементи указват размера на всяка група цифри в непарични единици. Всеки символ представлява цяло число, което обозначава броят на цифрите в текущата група. Числовият символ в първият елемент на масива, указва броят на цифрите в най-дясната група и т.н. Стойност ‘0’ означава, че предходната стойност ще се използва за останалите групи. Например ако масивът се състои от символите {‘3’, ‘0’}, това означава, че всички цифри са групирани по три.

int_curr_symbol

Низ от символи, използвани за международна валута. Първите три символи са специфицирани от ISO 4217:1987, а четвъртият е символа, който отделя символа за валута от паричните единици.

currency_symbol

Указател към международен валутен символ, използван за местна валута.

mon_decimal_point

Символ за десетична точка, използван в парични стойности.

mon_thousands_sep

Символ за разделител на хилядните, използван за парични стойности.

mon_grouping

Указател към char-масив, чийто елементи указват размера на всяка група цифри в парични единици. Всеки символ представлява цяло число, което обозначава броят на цифрите в текущата група. Числовият символ в първият елемент на масива, указва броят на цифрите в най-дясната група и т.н. Стойност ‘0’означава, че предходната стойност ще се използва за останалите групи.Например ако масивът се състои от символите {‘3’, ‘0’}, това означава, че всички цифри са групирани по три.

positive_sign

Знака използван за указване на положителни парични стойности.

negative_sign

Знака използван за указване на отрицателни парични стойности.

int_frac_digits

Указва броят цифрите след десетичната точка, които да бъдат показани в международни парични стойности.

frac_digits

Указва броят цифрите след десетичната точка, които да бъдат показани в парични стойности.

p_cs_precedes

Ако е 1, currency_symbol се появява пред положителна парична стойност. Ако е 0 currency_symbol се появява след положителна парична стойност.

p_sep_by_space

Ако е 1, currency_symbol е разделен с интервал от положителна парична стойност. Ако е 0, currency_symbol не е разделен с интервал от положителна парична стойност.

n_cs_precedes

Ако е 1, currency_symbol се появява пред отрицателна парична стойност. Ако е 0 currency_symbol се появява след отрицателна парична стойност.

n_sep_by_space

Ако е 1, currency_symbol е разделен с интервал от отрицателна парична стойност. Ако е 0, currency_symbol не е разделен с интервал от отрицателна парична стойност.

p_sign_posn
n_sign_posn

Указват позицията на положителния и отрицателния знак съответно. Следните стойности са валидни:

0 - Числото и currency_symbol са затворени заедно в скоби.
1 - Знаковият низ се поставя пред числото и currency_symbol.

2 - Знаковият низ се поставя след числото и currency_symbol.
3 - Знаковият низ се поставя непосредствено пред currency_symbol.
4 - Знаковият низ се поставя непосредствено след currency_symbol.

1.11.3 Декларации на макроси

NULL
Дефинира константен нулев указател.

Следващите макроси се използват като първи аргумент на функцията setlocale():

LC_ALL
Всички категории.

LC_COLLATE
Влияе на поведението на strcoll() и strxfrm() функциите.

LC_CTYPE
Влия на поведението на всички функции за обработка на символи (и многобайтови символи), с изключение на isdigit() и isxdigit().

LC_MONETARY
Влияе на паричната информация осигурявана от localeconv() функцията.

LC_NUMERIC
Влияе на форматиращата информацията за непарични стойности осигурявана от localeconv() функцията.

LC_TIME
Влияе на поведението на strftime() функцията.

1.11.4 Декларации на функции

```
char*         setlocale  (int category , const char *locale );
struct lconv* localeconv (void);
```

1.11.5 Описание на функциите (макро-функциите)

1.11.5.1 setlocale

```
#include <locale.h>
char*  setlocale  (int category , const char *locale );
```

Тази функция ви позволява да адаптирате програмата към местните условия на даден регион като час и дата, символи за десетична точка и валута, и други конвенции. Функцията връща указател към низ, който идентифицира новият регион, или текущия регион, ако се подаде нулев указател като втори аргумент.

Конвенциите на един регион са разделени на категории. Тези категории са описани в 1.6.3 Декларации на макроси и се използват като първи аргумент на функцията.

Вторият параметър **locale** е указател към низ, който указва региона. C89 дефинира две стандартни стойности за този параметър:

- “C” – Специфицира минималната среда за компилиране на C програми.
- “” – Специфицира имплементационно-зависима местна среда за компилиране на C програми.

Имплементациите могат да дефинират свои низове като аргумент на locale.

При стартиране на програмата регионалните настройките съответстват на тези, зададени с setlocale (LC_ALL, "C");

Функцията setlocale() връща указател към низ, описващ новите регионални настройки. Ако зададените настройки немогат да се изпълнят, функцията връща NULL.

Пример 1-17:

```
#include <locale.h>
#include <stdio.h>

int main(void)
{
  printf("The current locale is %s\n", setlocale(LC_ALL, NULL));

  printf("The new locale is %s\n", setlocale(LC_ALL, ""));

  return 0;
}
```

1.11.5.2 localeconv

```
#include <locale.h>
struct lconv* localeconv (void);
```

Тази функция връща указател към структура **lconv**, която съдържа информация за регионалните настройки, свързани с начина по който числата се форматират. Членовете на структурата и тяхното значение са описани в 1.6.2 Декларации на типове. Те не трябва да се променят ръчно от програмата.

Членовете на структурата тип char* са указатели към низ, всеки от който (освен **decimal_point**) може да сочи към празен низ "", което означава, че стойността не е налична в текушия регион или е с нулева дължина. Членовете тип char са неотрицателни числа, всяко от които може да бъде **CHAR_MAX**, което означава, че стойността не е налична в текущия регион.

Пример 1-18:

```
#include<locale.h>
#include<stdio.h>

int main(void)
{
  struct lconv locale_structure;
  struct lconv *locale_ptr=&locale_structure;

  locale_ptr=localeconv();

  printf("Decimal point:                    %s\n",locale_ptr->decimal_point);
  printf("Thousands Separator:              %s\n",locale_ptr->thousands_sep);
  printf("Grouping:                         %s\n",locale_ptr->grouping);
  printf("International Currency Symbol:    %s\n",locale_ptr->int_curr_symbol);
  printf("Currency Symbol:                  %s\n",locale_ptr->currency_symbol);
  printf("Monetary Decimal Point:           %s\n",locale_ptr->mon_decimal_point);
  printf("Monetary Thousands Separator:     %s\n",locale_ptr->mon_thousands_sep);
  printf("Monetary Grouping:                %s\n",locale_ptr->mon_grouping);
  printf("Monetary Positive Sign:           %s\n",locale_ptr->positive_sign);
  printf("Monetary Negative Sign:           %s\n",locale_ptr->negative_sign);
  printf("Monetary Fract Decimal Digits:    %c\n",locale_ptr->int_frac_digits);
  printf("Monetary Decimal Digits:          %c\n",locale_ptr->frac_digits);
  printf("Monetary + Precedes:              %c\n",locale_ptr->p_cs_precedes);
  printf("Monetary + Space:                 %c\n",locale_ptr->p_sep_by_space);
  printf("Monetary - Precedes:              %c\n",locale_ptr->n_cs_precedes);
  printf("Monetary - Space:                 %c\n",locale_ptr->n_sep_by_space);
  printf("Monetary + Sign Posn:             %c\n",locale_ptr->p_sign_posn);
  printf("Monetary - Sign Posn:             %c\n",locale_ptr->n_sign_posn);

  return 0;
}
```

1.12 <math.h>

1.12.1 Описание

Хедърът **<math.h>** съдържа декларации на математически функции за обработка на числа с плаваща запетая, и един макрос. Всички функции

използват аргументи тип double и връщат double сойности.

Ако входен аргумент е извън диапазона от стойности, за който функцията е дефинирана, настъпва грешка, наречена **domain error**. При този тип грешка функциите връщат имплементационно-зависима стойност и стойността на макроса **EDOM** се съхранява в **errno**.

Ако резултатът от функцията не може да се представи от тип double, настъпва грешка, наречена **range error**. Ако стойността е по-голяма отколкото може да се побере в тип double, функцията връща стойността на макроса **HUGE_VAL**, и стойността на макроса **ERANGE** се съхранява в **errno**. Ако стойността е твърде малка да се представи от тип double, функцията връща 0. В този случай дали стойността на макроса **ERANGE** се записва в **errno** е имплементационно зависимо.

errno, **EDOM** и **ERANGE** са дефинирани в хедъра <errno.h>.

Във всички случаи когато е дефинирано, че няма ограничение на входния диапазон от стойности се подразбира, че стойността е ограничена от минималната и максималната стойност на тип double.

1.12.2 Декларации на типове

-

1.12.3 Деларации на макроси

HUGE_VAL

1.12.4 Декларации на функции

```
double acos(double  x );
double asin(double  x);
double atan(double  x );
double atan2(doubly y , double  x );
double cos(double x );
double cosh(double  x );
double sin(double x );
double sinh(double  x );
double tan(double x );
double tanh(double  x );
double exp(double x );
double frexp(double x , int *exponent );
double ldexp(double x , int exponent );
double log(double x );
double log10(double x );
double modf(double  x , double *integer );
double pow(double x , double  y );
```

```
double sqrt(double  x );
double ceil(double  x );
double fabs(double  x );
double floor(double x );
double fmod(double  x , double  y );
```

1.12.5 Описание на функциите (макро-функциите)

1.12.5.1 acos

```
#include <math.h>
double acos(double  x );
```

Изчислява аркус косинус на аргумента x. Стойността на x трябва да е в диапазона [-1 ÷ +1] в противен случай настъпва **domain error**. Резултатът е в диапазона [0 ÷ π] радиани.

Пример 1-19:

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double val = -1.0;

  do {
       printf("Arc cosine of %f is %f.\n", val, acos(val));
       val += 0.1;
  } while(val<=1.0);

  return 0;
}
```

1.12.5.2 asin

```
#include <math.h>
double asin(double  x);
```

Изчислява аркус синус на аргумента x. Стойността на x трябва да е в диапазона [-1 ÷ +1] в противен случай настъпва **domain error**. Резултатът е в диапазона [-π/2 ÷ +π/2] радиани .

Пример 1-20:

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double val = -1.0;
```

```
  do {
    printf("Arc sine of %f is %f.\n", val, asin(val));
    val += 0.1;
  } while(val<=1.0);

  return 0;
}
```

1.12.5.3 atan

```
#include <math.h>
double atan(double  x );
```

Изчислява аркус тангенса на аргумента x. Стойността на x няма ограничение. Резултатът е в диапазона [-π/2 ÷ +π/2] радиани .

Пример 17-21:

```
#include <math.h>
#include <stdio.h>

int main (void)
{
  double val = -1.0;
  do {
    printf("Arc tangent of %f is %f.\n", val, atan(val));
    val += 0.1;
  } while(val<=1.0);

  return 0;
}
```

1.12.5.4 atan2

```
#include <math.h>
double atan2(double y , double  x );
```

Изчислява аркус тангенса на y/x, използвайки знака на двата аргумена, за да определи квадранта на връщаната стойност. Стойността на x и y не може да е 0 в противен случай може да настъпи **domain error**. Резултатът е в диапазона [-π ÷ +π] радиани .

Пример 1-22:

```
#include <math.h>
#include <stdio.h>
int main(void)
{
  double val = -1.0;
```

```
  do {
    printf("Atan2 of %f is %f.\n", val, atan2(val,1.0));
    val += 0.1;
  } while(val<=1.0);

  return 0;
}
```

1.12.5.5 cos

```
#include <math.h>
double cos(double x );
```

Изчислява косунуса на аргумента x. Стойността на x няма ограничение. Резултатът е в диапазона [-1 ÷ +1] радиани .

Пример 1-23:

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double val = -1.0;

  do {
    printf("Cos of %f is %f.\n", val, cos(val));
    val += 0.1;
  } while(val<=1.0);

  return 0;
}
```

1.12.5.6 cosh

```
#include <math.h>
double cos(double x);
```

Изчислява косунус хиперболичен на аргумента x. Стойността на входния аргумент и на резултата не са ограничени, но ако аргументът x е твърде голям може да настъпи **range error**.

Пример 1-24:

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double val = -1.0;

  do {
```

```
    printf("Cosh of %f is %f.\n", val, cosh(val));
    val += 0.1;
  } while(val<=1.0);

  return 0;
}
```

1.12.5.7 sin

```
#include <math.h>
double sin(double x );
```

Изчислява синуса на аргумента x. Стойността на x няма ограничение. Резултатът е в диапазона [-1 ÷ +1] радиани .

Пример 1-25:

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double val = -1.0;

  do {
    printf("Sin of %f is %f.\n", val, sin(val));
    val += 0.1;
  } while(val<=1.0);

  return 0;
}
```

1.12.5.8 sinh

```
#include <math.h>
double sinh(double x);
```

Изчислява синус хиперболичен на аргумента x. Стойността на входния аргумент и на резултата не са ограничени, но ако аргументът x е твърде голям може да настъпи **range error**.

Пример 1-26:

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double val = -1.0;

  do {
      printf("Sinh of %f is %f.\n", val, sinh(val));
```

```
    val += 0.1;
  } while(val<=1.0);

  return 0;
}
```

1.12.5.9 tan

```
#include <math.h>
double tan(double x);
```

Изчислява тангенса на аргумента x. Стойността на x няма ограничение.

Пример 1-27:

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double val = -1.0;

  do {
    printf("Tan of %f is %f.\n", val, tan(val));
    val += 0.1;
  } while(val<=1.0);

  return 0;
}
```

1.12.5.10 tanh

```
#include <math.h>
double tanh(double x);
```

Изчислява тангенс хиперболичен на аргумента x. Стойността на x няма ограничение. Резултатът е в диапазона [-1 ÷ +1].

Пример 1-28:

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double val = -1.0;

  do {
    printf("Tanh of %f is %f.\n", val, tanh(val));
    val += 0.1;
  } while(val<=1.0);
```

```
  return 0;
}
```

1.12.5.11 exp

```
#include <math.h>
double exp(double x );
```

Изчислява стойността e^x, където стойността на **e** е ~2.718282. Стойността на входния аргумент и на резултата не са ограничени, но ако аргументът x е твърде голям може да настъпи **range error**.

Пример 1-29:

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  printf("Value of e to the first: %f.", exp(1.0));

  return 0;
}
```

1.12.5.12 frexp

```
#include <math.h>
double frexp(double x , int *exponent );
```

Разбива числото x на мантиса **m** в обхвата [0 и 0.5 ÷ 1] и целочислена експонента **e**, така че **x = m * 2^e**. Връщаният резултат е мантисата **m**, а експонентата **e** се съхранява в обекта, сочен от **exponent**. Ако x е 0, то **m** и **e** също са 0.

Пример 1-30:

```
#include <math.h>
#include <stdio.h>

int main(void)
{

    int e;
    double f;

    f = frexp(10.0, &e);
    printf("%f %d", f, e); /* m = 0.625, e = 4 */

    return 0;
}
```

1.12.5.13 ldexp

```
#include <math.h>
double ldexp(double x , int exponent );
```

Връща като резултат x*2^{exponent}. Няма ограничение на стойностите на x и exponent, но е възможно да настъпи **range error**.

Пример 1-30:

```
#include <math.h>
#include <stdio.h>

int main(void)
{
    int e;
    double f;

    f = frexp(10.0, &e);
    printf("%f %d", f, e); /* m = 0.625, e = 4 */

    return 0;
}
```

1.12.5.14 log

```
#include <math.h>
double log(double x);
```

Връща логаритъм натурален от x, т.е. ln(x). Няма ограничение на стойността на x, но ще настъпи **domain error** ако стойността на x е отрицателна. Ако стойността на x е 0, може да настъпи **range error**.

Пример 1-31:

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double val = 1.0;

  do {
    printf("%f %f\n", val, log(val));
    val++;
  } while (val<=10.0);

  return 0;
}
```

1.12.5.15 log10

```
#include <math.h>
double log10(double x);
```

Връща логаритъм десетичен от x, т.е. $\log_{10}(x)$. Няма ограничение на стойността на x, но ще настъпи **domain error** ако стойността на x е отрицателна. Ако стойността на x е 0, може да настъпи **range error**.

Пример 1-32:

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double val = 1.0;

  do {
    printf("%f %f\n", val, log10(val));
    val++;
  } while (val<=10.0);

  return 0;
}
```

1.12.5.16 modf

```
#include <math.h>
double modf(double x, double *integer);
```

Разбива числото x на цяла и дробна част, всяка от които същия знак като аргумента. Връщаният резултат е дробната част, а цялата част се съхранява в обекта, сочен от integer.

Пример 1-33:

```
#include <math.h>
#include <stdio.h>

int main(void)
{
    double i;
    double f;

    f = modf(10.123, &i);
    printf("%f %f\n",i , f);

    return 0;
}
```

1.12.5.17 pow

```
#include <math.h>
double pow(double x, double y);
```

Връща x повдигнато на степен y, т.е x^y. **Domain error** настъпва при следните условия:

- Ако x е отрицателно число, а y не е цяло число.
- Ако резултатът не може да бъде представен когато x е 0 и y е по-малко или равно на 0.

При твърде големи стойности на x и y е възможно настъпването на **range error**.

Пример 1-34:

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double x = 10.0, y = 0.0;

  do {
    printf("%f\n", pow(x, y));
    y++;
  } while(y<11.0);

  return 0;
}
```

1.12.5.18 sqrt

```
#include <math.h>
double sqrt(double x);
```

Връща корен квадратен от числото x, т.е √x. **Domain error** настъпва ако x е отрицателно число.

Пример 1-35:

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  printf("%f\n", sqrt(16.0));

  return 0;
}
```

1.12.5.19 ceil

```
#include <math.h>
double ceil(double x);
```

Връща най-малкото цяло число по-голямо или равно на x. Казано с други думи floor() закръглява числото нагоре.

Пример 1-36:

```
#include <math.h>
#include <stdio.h>

int main(void)
{

  printf("%f\n", ceil(16.5));

  return 0;
}
```

1.12.5.20 fabs

```
#include <math.h>
double fabs(double x);
```

Връща абсолютната стойност на x, т.е |x|.

Пример 1-37:

```
#include <math.h>
#include <stdio.h>

int main(void)
{

  printf("%f\n", fabs(4.5));
  printf("%f\n", fabs(-4.5));

  return 0;
}
```

1.12.5.21 floor

```
#include <math.h>
double floor(double x);
```

Връща най-голямото цяло число по-малко или равно на x. Казано с други думи floor() закръглява числото надолу.

Пример 1-38:

```
#include <math.h>
#include <stdio.h>

int main(void)
{

  printf("%f\n", floor(16.5));

  return 0;
}
```

1.12.5.22 fmod

```
#include <math.h>
double fmod(double x, double y);
```

Връща остатъка r на x разделено на y. Резултатът може да се представи още като **r = x – i*y**, където i е остатъка на x/y (при y != 0), закръглен до първото цяло число надолу и имащо знака на x. Ако y е 0, от имплементацията зависи дали ще настъпи **domain error** или функцията ще върне 0.

Пример 1-38:

```
#include <math.h>
#include <stdio.h>

int main(void)
{

  printf("%f\n", fmod(10.0,3.0));
  printf("%f\n", fmod(-10.0,3.0));
  printf("%f\n", fmod(10.0,-3.0));
  printf("%f\n", fmod(-10.0,-3.0));

  return 0;
}
```

1.13 <setjmp.h>

1.13.1 Описание

Хедърът <setjmp.h> съдържа средства, които позволяват заобикаляне на нормалното извикване и връщане от функция. В хедъра са декларирани макросът **setjmp**, една функция **longjmp()** и един тип **jmp_buf**. Макро-функцията setjmp() се използва за съхраняване на извикващия контекст в

буфер от тип jmp_buf. Типът jmp_buf е имплементационно-зависим тип масив.

1.13.2 Декларации на типове

```
typedef .... jmp_buf;
```

1.13.3 Декларации на макроси

```
int setjmp(jmp_buf env);
```

Забележка: Стандартът ISO/IEC 989:1990 не дефинира дали **setjmp** е макрос или идентификатор, деклариран с външно свързване (**external linkage**).

Макро-функцията съхранява съдържанието на стека в буфер jmp_buf, указващо мястото на извикаване на макроса. Тази инфомация се използва после от функцията longjmp(). При извикване на longjmp(), изпълнението на програмата се прехвърля в мястото на извикване на макро-функцията setjmp(), по такъв начин, че все едно програмата се връща от изпълнение на макро-функцията setjmp().

При нормално връщане setjmp() връща 0. Ако връщането е осъществено посредством извикване на longjmp(), setjmp() връща стойност, указана от функцията longjmp(). В този случай тази стойност е винаги различна от 0.

ISO/IEC 989:1990 изисква макро-функцията setjmp() да се извиква само в следните контексти:

- Управляващ израз на оператор за избор или оператор за цикъл.

Пример 17-39:

```
if (setjmp(ebuf))
{
      /* Диретно връщане от setjmp() */
      /*.............................*/
}
else
{
      /* Косвено връщане от setjmp(), чрез longjmp() */
      /*.............................................*/
}
```

- Един от операндите на оператор за отношение или равенство, с друг операнд, представляващ целочислена константа, като резултатния израз е управляващ израз на оператор за избор или оператор за цикъл.

Пример 1-40:

```
if (0 == setjmp(ebuf))
{
      /* Диретно връщане от setjmp() */
      /*.............................*/
}
else
{
      /* Косвено връщане от setjmp(), чрез longjmp() */
      /*.............................................*/
}
```

- Като операнд на унарния операнд !, като резултатния израз е е управляващ израз на оператор за избор или оператор за цикъл.

Пример 1-41:

```
if (!setjmp(ebuf))
{
      /* Косвено връщане от setjmp(), чрез longjmp() */
      /*.............................................*/

}
else
{
     /* Диретно връщане от setjmp() */
      /*.............................*/
}
```

- В ролята на операторен израз (може и преобразуван към void)

Пример 1-42:

```
status = setjmp(ebuf);
```

или

```
(void)setjmp(ebuf);
```

1.13.4 Декларации на функции

```
void longjmp(jmp_buf environment, int value );
```

1.13.5 Описание на функциите (макро-функциите)

1.13.5.1 longjmp

```
#include <setjmp.h>
void longjmp(jmp_buf environment, int value );
```

При извикване на longjmp() изпълнението на програмта се прехвърля в мястото на последното извикване на макро-функцията setjmp(). Това място е указано в параметъра environment. Параметърът value указва стойността, връщана косвено от setjmp(). Ако value е 0, setjmp() връща 1. Причината, е че 0-та указва директно връщане от setjmp(). Всички други стойности указват косвено връщане от setjmp(), т.е имало е извикване на longjmp().

Ако longjmp() се извика преди първото извикване на setjmp() (т.е. с невалиден буфер) или функцията, съдържаща извикването на setjmp() завърши преди да се извика longjmp(), поведението е недефинирано.

Когато setjmp() върне стойност различна от 0, програмистта може да допусне, че статичните променливи съдържат коректни стойности както по времето на извикване на longjmp(). За автоматичните променливи, локални за функцията, която съдържа извикването на setjmp(), се гранатира, че съдържат коректни стойности само ако са квалифицирани с квалификатора volatile или не са били променяни между извикванията на setjmp() и longjmp().

ISO/IEC 989:1990 изисква longjmp() да работи правилно в контекста на прекъсвания, сигнали (вижте 1.14 <signal.h>) и всяка свързана с тях функция. Ако обаче longjmp() се извика от вложен обработчик на сигнали, поведението е недефинирано.

Изпълнението на longjmp() завършва като програмата прави преход в мястото на последното извикване на setjmp(), и стойността на параметъра value се връща като резултат от setjmp().

Типичната употреба на longjmp() е връщане от дълбока вложеност на функции при възникване на грешка.

Пример 1-43:

```
#include <setjmp.h>
#include <stdio.h>

void foo1(void);
void foo2(void);
void foo3(void);

jmp_buf     ebuf;

int main(void)
```

```
{
	int status;

	status = setjmp(ebuf);

	if (0 == status)
	{
		printf("Start of the calling sequence\n");
		foo1();
	}
	else
	{
		switch(status)
		{
			case	1:
			{
				printf("Error condition in foo1()\n");
				break;
			}

			case	2:
			{
				printf("Error condition in foo2()\n");
				break;
			}

			case	3:
			{
				printf("Error condition in foo3()\n");
				break;
			}
		}
	}

	return 0;
}

void foo1(void)
{
	foo2();
}

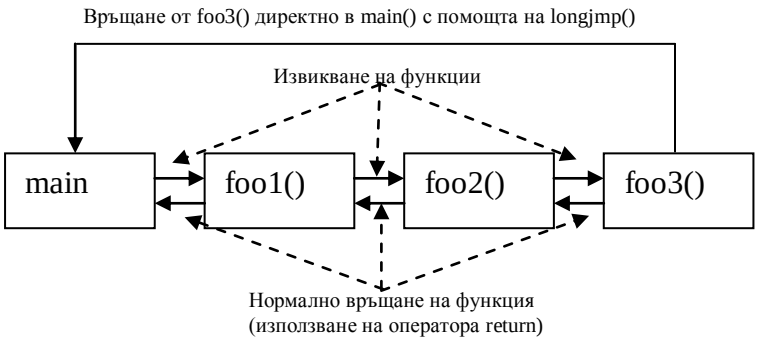

void foo2(void)
{
	foo3();
}

void foo3(void)
{
	/*...*/

	/* Настъпва някаква грешка */
	longjmp(ebuf,3);

	/*...*/
}
```

Фиг. 1 илюстрира нагледно механизма на нормално връщане от функциите foo1(), foo2(), foo3() и това с longjmp().



Фиг. 101 Използване на longjmp()

1.14 <signal.h>

1.14.1 Описание

Хедърът <signal.h> дефинира средства за обработка на сигнали. Сигналът е необичайно събитие, което настъпва по време на изпълнение на програмата. Сигналите могат да бъдат синхронни или асинхронни.

Синхронни сигнали са тези, които възникват от действия извършвани в самата програма, като деление на нула например или възникване на аритметични препълвания.

Асинхронните сигнали се предизвикват от външни за програмата събития. Много операционни системи например позволяват на потребителя да прекъсне или да спре работеща програма. Тези събития се третират като сигнали в C.

Всеки сигнал се идентифицира като положително число. ANSI/ISO: 9899_1990 стандартът дефинира следните сигнали:

- SIGABRT (Abort) – Този сигнал настъпва когато програмата завърши неуспешно както при извикване на функцията abort(). Най-добре е функцията-обработчик на този сигнал да завърши с return или exit().

- SIGFPE (Floating Point Exception) – Този сигнал се генерира от некоректни аритметични операции като деление на нула, препълване и т.н.

- SIGILL (Illegal instruction) – Този сигнал се генерира при изпълнение на невалидна машинна инструкция.

- SIGINT (Interrupt) – Този сигнал се генерира при натискането на специална комбинация от клавиши като Ctrl-C и други подобни. Този

сигнал може да се игнорира (вижте 2.5.9.2 signal).

- SIGSEGV (Segmentation Violation) – Този сигнал се генерира при невалиден достъп до паметта. Препоръчително е този сигнал да не се игнорира и функцията-обработчик да не се връща с return.

- SIGTERM (Termination) - Това е заявка прекратяване на изпълнението на програмата. Типично се изпраща от операционната система или друга програма.

Една имплементция не е нужно да генерира тези сигнали, освен като резултат от изричното извикване на функцията raise() (вижте 2.9.5.1 raise). Имплементациите могат да дефинират допънителни сигнали. Пълният набор от сигнали, техните свойства и обработка по подразбиране, зависи от имплементацията.

Всеки сигнал, който не се игнорира от програмата, трябва да се обработи. За целта се използва функция-обработчик. Това е специална функция, която се назначава на конкретен сигнал (или сигнали) с помощта на функцията signal() (вижте 2.5.9.2 signal), и която се извиква автоматично при възникването на сигнала. Ако един сигнал няма назначена функция-обработчик, се извиква подразбиращия се обработчик, чието поведение зависи от имплементацията. Синтактически функцията-обработчик е нормална C функция със следния прототип:

```
void име-функция(int signal);
```

Сигналът, причинил извикването на функцията-обработчик се предава като аргумент на параметъра **signal**. Действията, които могат да се извършат във функцията-обработчик са различни. Възможностите включват игнориране на сигнала, извършване на някакъв вид възстановяване от грешка, или спиране на изпълнението на програмата. Ако функцията-обработчик се връща от изпълнение, програмата възстановява изпълнението си от точката, в която сигналът е възникнал Изключение са следните случаи:

1. Ако сигналът е SIGABRT, програмата ще завърши (принудително) изпълнението си когато функцията-обработчик се върне от изпълнение.

2. Ако сигналът е SIGFPE, поведението е недефинирано ако функцията-обработчик се върне от изпълнение.

Хедърът <signal.h> дефинира още и тип **sig_atomic_t**. Това е целочислен тип достъпа, до който е неделима операция (операция, която не може да бъде прекъсната от асинхронни прекъсвания).

1.14.2 Декларации на типове

```
typedef ... sig_atomic_t
```

1.14.3 Декларации на макроси

SIG_DFL
SIG_ERR
SIG_IGN
SIGABRT
SIGFPE
SIGILL
SIGINT
SIGSEGV
SIGTERM

1.14.4 Декларации на функции

```
void (*signal(int sig , void (*func)(int)))(int);
int raise(int sig);
```

1.14.5 Описание на функциите (макро-функциите)

1.14.5.1 raise

```
#include <signal.h>
int raise(int sig);
```

Функцията raise() се използва за „ръчно” генериране на сигнал, указан с sig. При успех функцията връща нула, иначе връща ненулева стойност.

Ако сигнала, указан с sig, има назначена функция-обработчик (вижте 2.9.5.2 signal), той се извиква. Функцията raise() не връща резултат докато функция-обработчик не се върне. Ако функцията-обработчик не завършва изпълнението си (не се връща в извикващото я ниво с return), raise() връща 0 ако генерирането на сигнала е успешно, иначе връща ненулева стойност.

Ако сигнала, указан с sig, няма назначена функция-обработчик и не се игнорира, се извиква подразбиращия се обработчик, който прекратява изпълнението на програмата в повечето случаи.

1.14.5.2 signal

```
#include <signal.h>
void (*signal(int sig , void (*func)(int)))(int);
```

Функцията signal() задава функцията-обработчик на сигнала, указан с параметъра **sig.** Функцията-обработчик се задава с параметъра **func**. Нейния прототип трябва да е `void функция-обработчик(int);`. Освен валидно име на функция-обработчик, на праметъра func може да се подадат макросите **SIG_DFL** и **SIG_IGN**.

Ако се подаде **SIG_DFL**, тогава се използва подразбиращ се обработчик. Поведението на подразбиращия се обработчик зависи от имплементацията, но най-често резултатът е спиране на програмата.

Ако се подаде **SIG_IGN**, сигнала се игнорира.

Ако се подаде валидно име на функция-обработчик, тогава при генериране на сигнала първо се извиква еквивалента на `signal(sig, SIG_DFL)`[1]`;` , т.е сигнала отново се асоциира с подразбиращия се обработчик, или се извършва блокиране на сигнала по имплементационно-определен начин, и след това се извиква функцията-обработчик. Причината е да се избегне рекурсивното извикване на функцията-обработчик, ако по време на неговото изпълнение се генерира същия сигнал.

Забележка[1]: Ако сигнала е SIGILL дали SIG_DFL ще бъде зададен, зависи от имплементацията

Функцията-обработчик може да завърши изпълнението си с оператор return или с извикване на някоя от функциите abort(), exit() или longjmp().

Ако завърши с return и стойността на sig е SIGFPE или някоя имплеметационно-зависима стойност, отговаряща на изключение свързано с изчисления, поведението е недефинирано. Иначе програмата ще възобнови изпълнението си от мястото където е била прекъсната.

Ако се генерира асинхронен сигнал (това е сигнал, който не е резултат от функциите abort() и raise()), поведението е недефинирано ако функцията-обработчик извиква някоя стандартна библиотечна функция, различна от signal() (с първи аргумент номер на сигнала, причинил извикването на функцията-обработчик). Освен това ако такова извикване на функцията signal() върне като резултат SIG_ERR, стойността на errno е неопределена[1].

Забележка[1]: Ако някой сигнал се генерира от асинхронен обработчкик на сигнали,

поведението е недефинирано.

Поведението е недефинирано и ако функциите-обработчици на асинхронни сигнали използват обекти със статична продължителност на съхранение. Изключение са обекти със статична продължителност на съхранение тип volatile sig_atomic_t.

При стартиране на програмата еквивалента на

```
signal(sig, SIG_IGN);
```

може да бъде изпълнен за някои сигнали, избрани по имплементационно-дефиниран начин. Еквивалентът на

```
signal(sig, SIG_DFL);
```

се изпълнява за всички други сигнали, дефинирани от имплементацията.

При успешно изпълненеие, signal() връща указател към предишния обработчик на указания сигнал. Ако заявката не може да се изпълни резултата е SIG_ERR и положителна стойност се съхранява в errno.

Пример 1-44

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

void signal_handler(int);

int main(void)
{
  /*  Назначаване на функция-обработчик на сигнала SIGINT */
  if(signal(SIGINT, signal_handler) == SIG_ERR)
  {
    printf("An error occured while setting a signal handler.\n");

    exit(0);
  }

  /* Генериране на сигнала SIGINT с raise() */
  printf("Raising the interactive attention signal.\n");
  if(raise(SIGINT) != 0)
  {
    printf("Error raising the signal.\n");
    exit(0);
  }

  printf("Exiting.\n");

  return 0;
}

/* Функция-обработчик на сигнала SIGINT */
```

```
void signal_handler(int signal)
{
  printf("Interactive attention signal caught.\n");
}
```

1.15 <stdalign.h>

1.15.1 Описание

1.15.2 Декларации на типове

1.15.3 Декларации на макроси

1.15.4 Декларации на функции

1.15.5 Описание на функциите (макро-функциите)

1.16 <stdarg.h>

1.16.1 Описание

Хедърът <stdarg.h> декларира един тип и три макроса, които се използват за обхождане на списък с аргументи на функция, чийто брой и тип са неизвестни по време на компилиране. Такива функции се наричат още variadic- функции. Variadic-функциите се дефинират по следния начин:

Дефиниране на функция с променлив брой аргументи

```
тип-резултат име-функция(списък-фиксирани-параметри, ...)
{

}
```

Трите точки **...** след запетаята указват, че функцията приема променлив брой аргументи. Списъкът с фиксиран брой параметри трябва да съдържа поне един параметър.



Тъй като прототипа на една variadic-функция не съдържа типовете на променливия брой аргументи, те се подлагат на автоматично повишаване на типа при извикване на функцията.

За работа с variadic-функции се използват следните макроси дефинирани в хедъра <stdarg.h>:

- `void va_start( va_list argptr, lastparam );`

va_list - специален тип, който представлява указател към променлив брой аргументи.

```
typedef   ...   va_list
```

argptr - указател от тип **va_list**, към списъка с променлив брой аргументи.

lastparam - името на последния фиксиран параметър

Макросът va_start инициализира указателя **argptr** с позицията на най-десния параметър от списъка с фиксирани параметри. Този макрос се извиква винаги първи преди да се използват променливите аргументи.

- ```
  type va_arg( va_list argptr, type );
  ```

  argptr - указател от тип **va_list**, към списъка с променлив брой аргументи.

  type - тип на аргумента, чиято стойност се извлича.

  Макросът **va_arg** връща като резултат следващия аргумент.

- ```
  void va_end( va_list argptr );
  ```

  argptr - указател от тип **va_list**, към списъка с променлив брой аргументи.

  Макросът **va_end** деинициализаира указателя **argptr**. Този макрос трябва да се извиква веднъж за всяко извикване на **va_start**. Ако **va_end** се извика преди да се извика **va_start** или ако функцията завърши изпълнението си без да се извика **va_end**, поведението е недефинирано.

1.16.2 Декларации на типове

```
typedef     ...   va_list
```

1.16.3 Декларации на макроси (макро-функции)

```
void  va_start (va_list ap, last_arg );
type  va_arg   (va_list ap, type );
```

```
void  va_end   (va_list ap);
```

1.16.4 Декларации на функции

-

1.16.5 Описание на функциите (макро-функциите)

1.16.5.1 va_start

```
#include <stdarg.h>
void va_start( va_list argptr, lastparam );
```

Макросът **va_start** инициализира указателя **argptr** с позицията на най-десния параметър от списъка с фиксирани параметри. Параметърът **lastparam** е името на най-десния фиксиран параметър.Този макрос се извиква винаги първи преди да се използват променливите аргументи (с помощта на макроса **va_arg**).



Поведението е недефинирано ако параметърът lastparam е деклариран с клас-памет register и е тип масив или функция, или е тип, който не е съвместим с типа, който се получава след прилагането на автоматичното преобразуване на типа на аргументите.

1.16.5.2 va_arg

```
#include  <stdarg.h>
type  va_arg ( va_list argptr, type );
```

Макросът **va_arg** връща стойността на следващия аргумент (от списъка с променлив брой аргументи), а параметърът **type** указва неговия тип. Ако няма следващ аргумент, или ако типа, указан с **type** е несъвместим с действителния тип на следващия аргумент (след прилагане на автоматичното повишаване на типовете), поведението е недефинирано. Ако **va_arg** се извика преди макроса **va_start** поведението е недефинирано.

Първото извикване на **va_arg** след **va_start** извлича първия аргумент от списъка с променливи аргументи. Всяко последващо извикване извлича стойността на поредния аргумент.

1.16.5.3 va_end

```
#include  <stdarg.h>
void va_end(va_list argptr);
```

Макросът **va_end** се използва за коректно връщане на функцията в извикващото я ниво. Макросът **va_end** деинициализаира указателя **argptr**. Този макрос трябва да се извиква веднъж за всяко извикване на **va_start**. Ако **va_end** се извика преди да се извика **va_start** или ако функцията завърши изпълнението си без да се извика **va_end**, поведението е недефинирано.

Пример 1-45:

```
#include <stdio.h>
#include <stdarg.h>

void string_display(int str_num,...);

int main()
{
	string_display(2, "Hello", "I love C");
	string_display(3, "C", "is", "the best");

	return 0;
}

void string_display(int str_num,...)
{

  char* string_ptr[10];

  int  snum, i;

  /* Дефиниране на указател от тип va_list */
  va_list argptr;

  /* Ограничаване на броя на подадените низове */
  if (str_num > MAXSTRINGS){
	snum = MAXSTRINGS;
  }
  else{
	snum = str_num;
  }

  /* Инициализиране на указателя argptr */
  va_start( argptr, str_num );

  /* Извличане на входните аргументи */
  for ( i = 0; i < snum; ++i ){
	string_ptr [i] =  va_arg(argptr, char*);
  }

  /* Отпечатване на низовете*/
  for ( i = 0; i < snum; ++i ){
	puts(string_ptr [i]);
  }

  /* Деинициализиране на argptr */
```

```
  va_end( argptr );

  return;
}
```

1.17 <stdatomic.h>

1.17.1 Описание

1.17.2 Декларации на типове

1.17.3 Декларации на макроси

1.17.4 Декларации на функции

1.17.5 Описание на функциите (макро-функциите)

1.18 <stdbool.h>

1.18.1 Описание

1.18.2 Декларации на типове

1.18.3 Декларации на макроси

1.18.4 Декларации на функции

1.18.5 Описание на функциите (макро-функциите)

1.19 <stddef.h>

1.19.1 Описание

Хедърът <stddef.h> дефинира следните стандартни макроси и типове:

- `#define NULL ...`
- `#define offsetof (структурен-тип, име-структурен-член);`
- `typedef ... ptrdiff_t;`
- `typedef ... size_t;`
- `typedef ... wchar_t;`

1.19.2 Декларации на типове

- `typedef ... ptrdiff_t;`

Типът **ptrdiff_t** е имплементационно-дефиниран знаков целочислен тип(най-често long), който е тип на резултата от изваждане на два указателя.

Пример 1-46

```
#include <stdio.h>
#include <stddef.h>

int main(void)
{
	int arr[5] = {1,2,3,4,5};
	int *p1, *p2;
	ptrdiff_t diff;
                  /* | 1 | 2 | 3| 4 | 5 | */
	p1 = &arr[0];  /* --|              |   */
	p2 = &arr[4];  /*-----------------|   */

	diff = p2 - p1; /* брой елементи между двата указателя */

	printf("%d\n", diff);

	return 0;
}
```

- `typedef ... size_t;`

Типът **size_t** е имплементационно-дефиниран беззнаков целочислен тип(най-често unsigned long), който е тип на резултата на оператора sizeof.

Пример 1-47

```
#include <stdio.h>
#include <stddef.h>

int main(void)
{
	int arr[] = {1,2,3,4,5};
	size_t size;

	size = sizeof (arr); /* брой байтове, заемани от масива */

	printf("%d\n", size);

	return 0;
}
```

- `typedef ... wchar_t;`

Типът **wchar_t** е имплементационно-дефиниран знаков или беззнаков целочислен тип, който може да съхранява широки символи.

Повече за широките символи ще намерите при описанието на хедърите <wchar.h> и <wctype.h> и Приложение C.

1.19.3 Декларации на макроси

- `#define NULL ...`

Макросът NULL се използва за обозначаване на нулев константен указател. Указатели, които съдържат тази стойност се приема, че не сочат към нищо. Стойността на NULL е имплементационно-зависима. Повечето имплементации дефинират NULL като 0 или ((void*)0).

Макросът NULL е дефиниран за по-голямо удобство също и в хедърите <stdio.h>, <locale.h>, <stdlib.h>, <string.h>, <time.h> и <wchar.h>.

Пример 1-48

```
#include <stdio.h>

void foo (int *ptr)
{
	if (NULL == ptr)
	{
		printf ("The pointer is invalid\n");
	}
	else
	{
		printf ("The pointer is valid\n");
		*ptr = 10;
	}

	return;
}

int main(void)
{

	int *p = NULL; /* p не сочи към нищо */
	int a;

	foo (p);

	p = &a;		/* p сочи към обекта a */
	foo (p);

	/*...*/

	return 0;
}
```

- `#define offsetof (структурен-тип, име-структурен-член) ...`

Макро-функцията offsetof() връща стойност тип size_t, която представлява

отместването в байтове на члена, указан с име-структурен-член, принадлежащ на структурата, указана с структурен-тип. Ако членът е битово поле, резултатът е недефиниран.

Пример 1-49

```
#include <stdio.h>
#include <stddef.h>

typedef struct
{
	char ch;
	int  i;
	int  b0:3;
}MyStruct;

int main(void)
{
	size_t s;

	s = offsetof (MyStruct, ch);
	printf("s<ch> = %d\n", s);

	s = offsetof (MyStruct, i);
	printf("s<i> = %d\n", s);

	/* s = offsetof (MyStruct, b0); */ /* Грешка !!! */

	return 0;
}
```

1.19.4 Декларации на функции

-

1.19.5 Оисание на функциите (макро-функциите)

-

1.20 <stdint.h>

1.20.1 Описание

1.20.2 Декларации на типове

1.20.3 Декларации на макроси

1.20.4 Декларации на функции

1.20.5 Описание на функциите (макро-функциите)

1.21 <stdio.h>

1.21.1 Описание

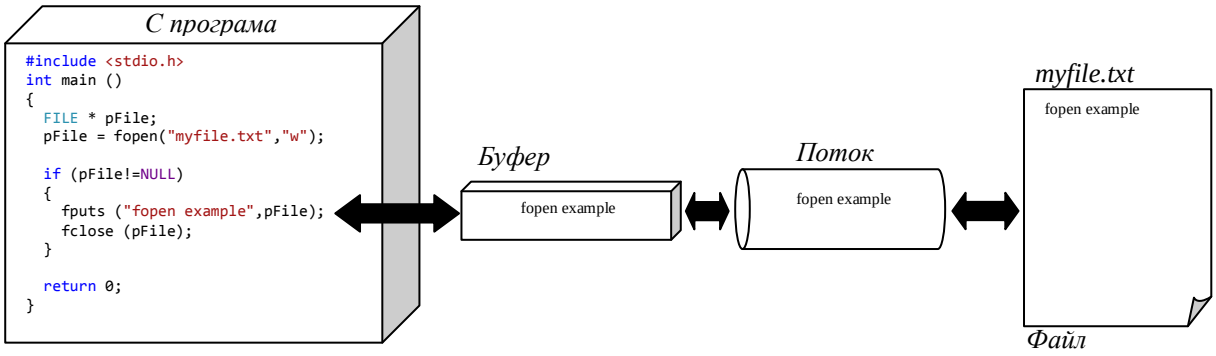
Хедърът **<stdio.h>** декларира три типа, няколко макроса и множество функции за извършване на входно-изходни операции, като извеждане на информация на екрана, въвеждане на данни от клавиатура, запис/четене на данни в/от изходни/входни устройства и т.н.

В контекста на C всяко физическо устройство, което може да обменя данни с програмата (клавиатура, монитор, принтер, хард диск и др.) се нарича файл. Различните файловете се различават значително във физическата си имплементация и начин на работа, и съответно достъпа до тях се осъществява по различен начин. За да скрие тази разлика и да осигури унифициран интерфейс за достъп, езикът C изгражда едно абстрактно ниво между програмата и файла, наречено поток. Посредством това абстрактно ниво всички файлове от гледна точка на програмистта са еднакви, т.е за четене и/или запис във различни файлове се използват еднакви библиотечни функции. Казаното до тук може да се обобщи по следния начин: при обмяна на данни между програмата и файл, се създава поток, който скрива детайлите свързани с действителния начин на запис/четене на данни във/от файла.

Един поток се асоциира (свързва) с файл посредством функцията fopen() и се освобождава посредством функцията fclose().

Потоците могат да бъдат **небуферирани**, **линейно буферирани**, и **напълно буферирани**. При небуфериран поток данните се записват/четат във/от файла директно. При линейно буфериран поток данните се записват/четат в/от файла на редове. Един ред (линия) се състои от последователност от символи, завършваща със символ за нов ред. При напълно буфериран поток данните се записват/четат във/от файла при запълване на буфера. Функциите setbuf() и setvbuf() могат да се използват за управление на буферирането на потоците.

Фиг.102 описва абстрактно връзката между C програма, поток и асоциирания с потока файл.



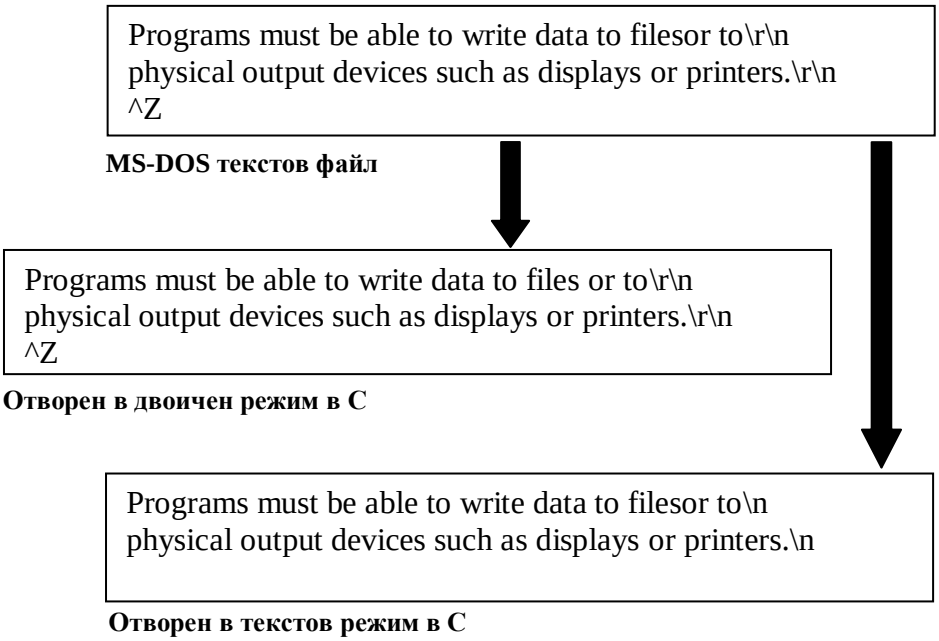
Фиг. 102 Връзка между С програма, потоци и файлове

Потоците се разделят на две категории: **текстови** и **байтови** потоци.

Текстовият поток се състои от редове. Всеки ред представлява нула или повече символи плюс символ за нов ред. Дали последния ред трябва да завършва със символ за нов ред зависи от имплеметацията. В зависимост от средата, в която се изпълнява програмата е възможно това, което програмата записва/чете да се различава от това, което в действителност е записано във файла. Например езикът С дефинира за край на ред символа ‘\n’, докато в MS-DOS символът за край на ред е комбинация от символите ‘\r’ и ‘\n’, а в MAC OS само ‘\r’.

Двоичният поток представлява последователност от символи. Всички символи се разглеждат просто като байтове данни без да се интерпретират по някакъв специален начин. Това означава, че не се извършват никакви преобразувания и ще има пълно съответствие между съдържанието на един двоичен файл и прочетеното/записаното от програмата.

Фиг.103 показва как един текстов файл в MS-DOS ще се интерпретира от С програма в зависимост от това дали файлът е отворен в двоичен или текстов режим.



Фиг. 103 Двоично и текстово представяне на файлове в C

При стартиране на една програма, автоматично се създават 3 стандартни потока:

- **stdin**

  Това е стандартен поток за вход. В повечето системи този поток е свързан с клавиатурата. Стандартния входен поток stdin може да се използва за аргумент на всички входно-изходни функции, които изискват входен поток за аргумент, например fgetc(), fgets() и т.н. Повечето операционни системи поддържат пренасочване на стандартния входен поток при стартиране на приложението. Например MS-DOS/Windows и UNIX поддържат следния синтаксис при стартиране на приложението от командния ред:

  C:\> myapplication<file.txt

  Когато приложението *myapplication* се стартира, всички операции за четене от стандартния входен поток ще четат данни не от клавиатурата, а от файла file.txt.

  Входния поток може да бъде пренасочен да използва друг източник на входнни данни също и в програмата с помощта на функцията freopen().

  По подразбиране входния поток stdin е напълно буфериран само ако е известно, че не сочи към интерактивно устройство. В противен случай буферирането е имплементационно-зависимо.

- **stdout**

  Това е стандартен поток за изход. В повечето системи този поток е свързан с монитор. Стандартния изходен поток stdout може да се използва за аргумент на всички входно-изходни функции, които изискват изходен поток за аргумент, например fputc(), fputs() и т.н. Повечето операционни системи поддържат пренасочване на стандартния изходен поток при стартиране на приложението. Например MS-DOS/Windows и UNIX поддържат следния синтаксис при стартиране на приложението от командния ред:

  C:\> myapplication>file.txt

  Когато приложението *myapplication* се стартира, всички операции за запис в стандартния изходен поток ще изпращат данни не към монитора, а към файла file.txt.

  Изходния поток може да бъде пренасочен да изпраща данни към друго устройство също и в програмата с помощта на функцията freopen().

  По подразбиране изходния поток stdout е напълно буфериран само ако е известно, че не сочи към интерактивно устройство. В противен случай буферирането е имплементационно-зависимо.

- **stderr**

  Това е стандартен поток за извеждане на грешки и предупреждения. В повечето системи този поток е свързан с монитор. Потокът stderr може да се използва за аргумент на всички входно-изходни функции, които изискват изходен поток за аргумент, например fputc(), fputs() и т.н. Повечето операционни системи поддържат пренасочване на потока stderr при стартиране на приложението. Например MS-DOS/Windows и UNIX поддържат следния синтаксис при стартиране на приложението от командния ред:

  C:\> myapplication 2> file.txt

  Потока stderr може да бъде пренасочен да изпраща съобщения към друго устройство също и в програмата с помощта на функцията freopen().

  При стартиране на програмата stderr никога не е напълно буфериран. Дали ще бъде линейно буфериран или небуфериран е имплементационно-зависимо.

Табл.57 показва в резюме всички входно-изходни функции, разделени по категории.

Кагегории	Функции	Описание
Достъп до файл	**fopen**	Отваря файл
	freopen	Асоциира съществуващ поток с друг файл
	fflush	Изчиства поток
	fclose	Затваря файла
	setbuf	Буферира поток
	setvbuf	Буферира поток
Директен вход-изход	**fread**	Чете блок от данни от файл
	fwrite	Записва блок от данни във файл
Неформатиран вход-изход	**fgetc**	Чете символ от поток
	getc	Чете символ от stdin
	fgets	Чете низ от поток
	fputc	Записва символ в поток
	putc	Записва символ в stdout
	fputs	Записва низ в поток
	getchar	Чете символ от stdin
	gets	Чете низ от stdin
	putchar	Записва символ в stdout
	puts	Записва низ в stdout
	ungetc	Връща символ обратно в поток
Форматиран вход-изход	**scanf**	Чете форматирани данни от stdin
	fscanf	Чете форматирани данни от поток
	sscanf	Чете форматирани данни от низ
	printf	Записва форматирани данни в stdout
	fprintf	Записва форматиран данни в поток
	sprintf	Записва форматиран данни в низ
	vprintf	Записва форматирани данни от списък с променлив брой аргументи в stdout
	vfprintf	Записва форматирани данни от списък с променлив брой аргументи в поток
	vsprintf	Записва форматирани данни от списък с променлив брой аргументи в низ
Позициониране във файл	**ftell**	Връща текущата позиция на индикатора в потока
	fgetpos	Съхранява текущата позиция на индикатора в потока
	fseek	Премества индикатора на позицията в зададено място във файла
	fsetpos	Премества индикатора на позицията в зададено място във файла
	rewind	Премества индикатора на позицията в началото на файла

Обработка на грешки	clearerr	Изчиства флаговете за грешки
	feof	Проверява флага за край на файл
	ferror	Проверява за флага за грешки
	perror	Отпечатва съобщение за грешка
Операции върху файлове	remove	Изтрива файл
	rename	Преименува файл
	tmpfile	Създава временен файл
	tmpnam	Създава уникално има на временен файл

Табл. 57 Входно-изходни функции

1.21.2 Декларации на типове

- `typedef ... size_t;`

Типът size_t е имплементационно-дефиниран беззнаков целочислен тип(най-често unsigned long), който е тип на резултата на оператора sizeof.

Този тип е дефиниран също и в <stddef.h>

- `typedef ... FILE`

FILE е тип на обект, който съдържа цялата необходима информация за управление на входно-изходен поток. Това включва указател към буфер на поток, индикатор на позицията във файл, флагове за грешки, указващи дали са възникнали грешки при четене/запис и флаг за край на файл, указващ дали е достигнат края на файл.

- `typedef ... fpos_t`

Типът fpos_t е типа на обекта, който се връща от функцията fgetpos(). Този обект съдържа цялата информация, необходима за специфицирането на всяка позиция в един файл.

1.21.3 Декларации на макроси

- **NULL**

  Макросът NULL се използва за обозначаване на нулев константен указател. Указатели, които съдържат тази стойност се приема, че не сочат към нищо. Стойността на NULL е имплементационно-зависима. Повечето имплементации дефинират NULL като 0 или ((void*)0).

Макросът NULL е дефиниран за по-голямо удобство също и в хедърите <stdio.h>, <locale.h>, <stdlib.h>, <string.h>, <time.h> и <wchar.h>.

- **_IOFBF**
- **_IOLBF**
- **_IONBF**

Тези макроси се разширяват до целочислени константни изрази с различни стойности, подходящи за използване като трети аргумент на функцията setvbuf();

- **BUFSIZ**

Цяло число, което представлява размера на буфера, използван от функцията setvbuf().

- **EOF**

Отрицателно цяло число, указващо че края на файла е достигнат.

- **FOPEN_MAX**

Цяло число, което указва минималния брой файлове, които системата гарантира, че могат да бъдат отворени едновременно.

- **FILENAME_MAX**

Цяло число, което указва най-голямата дължина на масив от символи, подходящ за съхранение на най-дългото възможно име на файл, който може да се отвори с функцията fopen(). Ако имплементацията не налага ограничение върху дължината на име на файл, препоръчително е тази стойност да се използва като максимална.

- **L_tmpnam**

Цяло число, което указва най-голямата дължина на масив от символи, подходящ за съхранение на най-дългото възможно временно име на файл, генерирано от функцията tmpname().

- **SEEK_CUR**
- **SEEK_END**
- **SEEK_SET**

Тези макроси се разширяват до целочислени константни изрази с различни стойности, подходящи за използване като трети аргумент на функцията fseek();

- **TMP_MAX**

Цяло число, указващо максималния брой на уникалните имена на файлове, които функцията tmpnam() може да генерира.

- **stderr**
- **stdin**
- **stdout**

Това са стнадартни потоци за вход-изход. Те представляват указатели към обекти тип FILE, които сочат към FILE обекти, асоциирани съответно със стандартен поток за грешка, стандартен поток за вход и стандартен поток за изход.

1.21.4 Декларации на функции

```
void   clearer  (FILE *stream);
int    fclose   (FILE *stream);
int    feof     (FILE *stream);
int    ferror   (FILE *stream);
int    fflush   (FILE *stream);
int    fgetpos  (FILE *stream, fpos_t *pos);
FILE*  fopen    (const char *filename, const char *mode);
size_t fread    (void *ptr, size_t size, size_t nmemb, FILE *stream);
FILE*  freopen  (const char *filename, const char *mode, FILE *stream);
int    fseek    (FILE *stream, long int offset, int whence);
int    fsetpos  (FILE *stream, const fpos_t *pos);
long   ftell    (FILE *stream);
size_t fwrite   (const void *ptr, size_tsize, size_tnmemb, FILE *stream);
int    remove   (const char *filename);
int    rename   (const char *old_filename, const char *new_filename);
void   rewind   (FILE *stream);
void   setbuf   (FILE *stream, char *buffer);
int    setvbuf  (FILE *stream, char *buffer, int mode, size_t size);
FILE*  tmpfile  (void);
char*  tmpnam   (char *str);
int    fprintf  (FILE *stream, const char *format, ...);
int    printf   (const char *format, ...);
int    sprintf  (char *str, const char *format, ...);
int    vfprintf (FILE *stream, const char *format, va_list arg);
int    vprintf  (const char *format, va_listarg);
int    vsprintf (char *str, const char *format, va_list arg);
int    fscanf   (FILE *stream, const char *format, ...);
int    scanf    (const char *format, ...);
int    sscanf   (const char *str, const char *format, ...);
int    fgetc    (FILE *stream);
char*  fgets    (char *str, int n, FILE *stream);
int    fputc    (int c, FILE *stream);
int    fputs    (const char *str, FILE *stream);
int    getc     (FILE *stream);
int    getchar  (void);
char*  gets     (char *str);
int    putc     (int char, FILE *stream);
int    putchar  (int c);
int    puts     (const char *str);
```

```
int    ungetc   (int char, FILE *stream);
void   perror   (const char *str);
```

1.21.5 Описание на функциите (макро-функциите)

1.21.5.1 clearerr

```
#include <stdio.h>
void   clearer  (FILE *stream);
```

Функцията clearerr() изчиства флага за край на файл и флага за грешка, свързани с потока указан с stream.

При успешно отваряне на файл с fopen() всички флагове за грешки на потока се нулират. При работа с файлове грешки могат да възникнат по различни причини. Функцията perror() може да се използва за получаване на информация за грешката.

Пример 1-46:

```
#include <stdio.h>

int main ()
{
  FILE * pFile;

  pFile = fopen("myfile.txt","r");
  if (pFile==NULL) perror ("Error opening file");
  else {
    fputc ('x',pFile);
    if (ferror (pFile)) {
      printf ("Error Writing to myfile.txt\n");
      clearerr (pFile);
    }

    fgetc (pFile);
    if (!ferror (pFile)){
      printf ("No errors reading myfile.txt\n");
       }
    fclose (pFile);
  }

  return 0;
}
```

1.21.5.2 fclose

```
#include <stdio.h>
int   fclose  (FILE *stream);
```

Функцията fclose() затваря файла, асоцииран с потока указан с stream. Ако файлът е отворен за запис, съдържанието на буфера за записва във файла преди той да бъде затворен. А ако е отворен за четене, свързания с него буфер се изчиства.

При успешно затваряне на файла fclose() връща 0. При възникване на грешка по време на затваряне връща EOF.

Пример 1-47:

```
#include <stdio.h>
int main ()
{
  FILE * pFile;
  pFile = fopen ("myfile.txt","w");
  fprintf (pFile, "fclose example");
  if(EOF == fclose (pFile))
  {
        perror ("flcose() error!");
  }

  return 0;
}
```

1.21.5.3 feof

```
#include <stdio.h>
int   feof  (FILE *stream);
```

Функцията feof() проверява флага за край-на-файл на потока, указан с stream.

Функцията feof() връща ненулева стойност ако потокът е достигнал края на файла, в противен случай връща 0.

Пример 1-48:

```
#include <stdio.h>

int main ()
{
  FILE * pFile;

  long n = 0;
  pFile = fopen ("myfile.txt","rb");
  if (pFile==NULL) perror ("Error opening file");
  else
  {
    while (!feof(pFile)) {
      fgetc (pFile);
      n++;
    }
```

```
    fclose (pFile);
    printf ("Total number of bytes: %d\n", n-1);
  }

  return 0;
}
```

1.21.5.4 ferror

```
#include <stdio.h>
int   ferror  (FILE *stream);
```

Функцията ferror() проверява флагът за грешка на потока, указан с stream. При всяка операция за четене или запис от/във файла, свързан с потока, флага за грешка се вдига ако възникне грешка. Ако операцията завърши без грешка, флага се нулира. Подробна информация за възникналата грешка може да се получи чрез функцията perror().

Функцията ferror() връща ненулева стойност ако флага за грешка е вдигнат, в противен случай връща 0.

Пример 1-49:

```
#include <stdio.h>

int main ()
{
  FILE * pFile;

  pFile=fopen("myfile.txt","r");
  if (pFile==NULL) perror ("Error opening file");
  else {
    fputc ('x',pFile);
    if (ferror (pFile)){
      printf ("Error Writing to myfile.txt\n");
       }
    fclose (pFile);
  }

  return 0;
}
```

1.21.5.5 fflush

```
#include <stdio.h>
int   fflush  (FILE *stream);
```

Функцията fflush() изчиства на буфера на потока, указан с stream. Това означава, че ако потокът е отворен за запис (или запис-четене, като последната операция е била запис), съдържанието на буфера се записва във

файла, асоцииран с потока. В другите режими поведението е недефинирано. Ако за аргумент на stream се използва нулев указател, буферите на всички отворени потоци се изчистват по описания по-горе начин.

Функцията fflush() връща 0 при успешно изпълнение или EOF при възникване на грешка по време на запис.

Пример 1-50:

```
#include <stdio.h>

char mybuffer[80];

int main()
{
   FILE * pFile;

   pFile = fopen ("example.txt","r+");

   if (pFile == NULL) {
       perror ("Error opening file");
   }
   else {
     fputs ("test",pFile);
     fflush (pFile);
     rewind (pFile);
     fgets (mybuffer,80,pFile);
     puts (mybuffer);
     fputs (" fflush",pFile);
     fclose (pFile);
     return 0;
  }
}
```

1.21.5.6 fgetpos

```
#include <stdio.h>
int   fgetpos  (FILE *stream, fpos_t *pos);
```

Функцията fgetpos() съхранява стойността на индикатора на текущата позиция във файла, асоцииран с потока stream, в обекта, сочен от указателя pos.

Функцията fgetpos() връща 0 при успешно изпълнение. При възникване на грешка връща ненулева стойност и записва имплементационно-зависима положителна стойност в променливата errno.

Пример 1-51:

```
#include <stdio.h>
```

```c
int main(void)
{
  FILE *fp;
  fpos_t mark;
  int i;
  char read[100] = {0};

  fp = fopen ("myfile.txt", "r");

  if (NULL == fp){
      perror ("Cannot open myfile.txt!");
  }
  else {

      fgetpos(fp, &mark);
      printf("mark = %d\n", mark);

      fread (read, 1, 5, fp);

      fgetpos(fp, &mark);
      printf("mark = %d\n", mark);

      for (i = 0; read[i] != '\0'; i++){
            printf("read[%d] = %c\n", i, read[i]);
      }

  }

  fclose(fp);
  return 0;
}
```

1.21.5.7 fopen

```c
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

Функцията fopen() отваря поток и го асоциира с файла, указан от filename. Режимът на достъп до файла се указва с mode. Параметърът mode може да приема един от следните аргументи:

- `"r"`[1]
  Отваряне на текстов файл в режим на четене.

- `"w"`[2]
  Отваряне на текстов файл в режим на запис.

- `"a"`[3]
  Отваряне на текстов файл в режим на добавяне.

- `"rb"`[1]
  Отваряне на двоичен файл в режим на четене.

- `"wb"`[2]

Отваряне на двоичен файл в режим на запис.

- `"ab"`[3]

Отваряне на двоичен файл в режим на добавяне.

- `"r+"`[1]

Отваряне на текстов файл в режим на четене и запис.

- `"w+"`[2]

Отваряне на текстов файл в режим на четене и запис.

- `"a+"`[3]

Отваряне на текстов файл в режим на четене и добавяне.

- `"r+b" или "rb+"`[1]

Отваряне на двоичен файл в режим на четене и запис.

- `"w+b" или "wb+"`[2]

Отваряне на двоичен файл в режим на четене и запис.

- `"a+b" или "ab+"`[3]

Отваряне на двоичен файл в режим на четене и добавяне.

Забележка[1]: Отварянето на файл (текстов или двоичен) в режим на четене (всеки режим, който съдържа символ r) пропада, ако файлът не съществува или не е достъпен аз четене.

Забележка[2]: При отваряне на файл (текстов или двоичен) в режим на запис (всеки режим, който съдържа символ w) , ако файл с посоченото име не съществува, се създава нов празен файл. Ако вече съществува файл със същото име, неговото съдържание се изтрива.

Забележка[3]: При отваряне на файл (текстов или двоичен) в режим на добавяне (всеки режим, който съдържа символ a) , ако файл с посоченото име не съществува, се създава нов празен файл. Ако вече съществува файл със същото име неговото съдържание се запазва и операциите за запис добавят нова информация в края на файла.

При отваряне един поток е напълно буфериран само ако може да се определи, че той не сочи към интерактивно устройство. Флаговете за грешка и край-на-файл се нулират.

Функцията fopen() връща при успех указател към обект тип FILE, който управлява потока. При възникване на грешка връща нулев указател.

Пример 1-52:

```
#include <stdio.h>
int main ()
{
  FILE * pFile;

  pFile = fopen ("myfile.txt","w");

  if (pFile != NULL)
  {
    fputs ("fopen example",pFile);
    fclose (pFile);
  }

  return 0;
}
```

1.21.5.8 fread

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Функцията fread() чете nmemb на брой обекта[1], всеки с размер size (в байтове), от файл, асоцииран с потока stream, и ги съхранява в буфера, указан с ptr. Индикаторът на текущата позицията на файла се увеличава с броя на успешно прочетените байтове. Ако възникне грешка стойността на индикатора на текущата позиция е неопределена. Ако един обект се прочете частично стойността на индикатора на текущата позиция е неопределена.

Забележка[1]: Обект може да бъде както от базов тип (char, int, double и т.н.), така и от потребителски тип, например структура.

Функцията fread() връща броя на успешно прочетените елементи, който може да е по-малък от nmemb, ако възникне грешка при четенето или се достигне края на файла. За да се определи кое от двете е вярно може да се използват функциите feof() и ferror(). Ако size или nmemb е 0, fread() връща 0.

Пример 1-53:

```
#include <stdio.h>

int main(void)
{
  FILE *fp;
  int i;
```

```
  char str[100] = {0};

  fp = fopen ("myfile.txt", "r");

  if (NULL == fp){
       perror ("Cannot open myfile.txt!");
  }
  else {

       fread (str, 1, 5, fp);

       for (i = 0; str[i] != '\0'; i++){
             printf("str[%d] = %c\n", i, str[i]);
       }

   }

   fclose(fp);
   return 0;
}
```

1.21.5.9 freopen

```
#include <stdio.h>
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

Функцията freopen() затваря файла, асоцииран с потока stream и асоциира потока с файла, указан с filename. Режимът на достъп до новия файл се определя от параметъра mode (виж 3.12.5.7 fopen).

Функцията freopen() връща потока stream при успех и нулев указател при неуспех.

Пример 1-54:

```
#include <stdio.h>

int main ()
{

  if (NULL == freopen ("myfile.txt","w",stdout))
  {
       fprintf( stderr, "Unable to redirect stdout.\n" );
  }
  else
  {
       printf ("This sentence is redirected to a file.");

       fclose (stdout);
  }

  return 0;
}
```

1.21.5.10 fseek

```
#include <stdio.h>
int    fseek    (FILE *stream, long int offset, int whence);
```

Функцията fseek() премества индикатора на позицията на файла, асоцииран с потока stream, с offset на брой байта, относно позицията, указана с whence. Параметърът whence приема един от следните аргументи:

- SEEK_SET
  Указва, че индикаторът на позицията трябва да се премести с offset на брой байта от началото на файла.

- SEEK_CUR
  Указва, че индикаторът на позицията трябва да се премести с offset на брой байта от текущата позиция във файла.

- SEEK_END
  Указва, че индикаторът на позицията трябва да се премести с offset на брой байта от края на файла.

Ако потокът stream е асоцииран с текстов файл, стойността на параметъра whence трябва да бъде SEEK_SET, а стойността на параметъра offset трябва да бъде 0 или тази върната от по-ранно извикване на функцията ftell() върху същия поток.

Функцията fseek() връща 0 при успех, в противен случай връща ненулева стойност.

Пример 1-55:

```
#include <stdio.h>

int main ()
{
  FILE * pFile;

  pFile = fopen ( "example.txt" , "wb" );
  fputs ( "This is an apple." , pFile );
  fseek ( pFile , 9 , SEEK_SET );
  fputs ( " sam" , pFile );
  fclose ( pFile );

  return 0;
}
```

1.21.5.11 fsetpos

```
#include <stdio.h>
int    fsetpos  (FILE *stream, const fpos_t *pos);
```

Функцията fsetpos() премества индикатора на позицията на файла, асоцииран с потока stream, в позицията, указана в параметъра pos и която е получена преди това с помощта на функцията fgetpos(). При успешно изпълнение на функцията, флага за край-на-файл се нулира, и всяко предишно действие на функцията ungetc() върху същия поток се отменя.

Функцията fsetpos() връща 0 при успех, в противен случай връща ненулева стойност и съхранява имплементационно зависима положителна стойност в променливата errno.

Пример 1-56:

```
#include <stdio.h>

int main ()
{
  FILE * pFile;
  fpos_t position;

  pFile = fopen ("myfile.txt","w");
  fgetpos (pFile, &position);
  fputs ("That is a sample",pFile);
  fsetpos (pFile, &position);
  fputs ("This",pFile);
  fclose (pFile);

  return 0;
}
```

1.21.5.12 ftell

```
#include <stdio.h>
long   ftell    (FILE *stream);
```

Функцията ftell() връща стойността на индикатора на позицията на файла асоцииран с потока stream. За двоични потоци тази стойност е броя на байтовете от началото на файла. За текстови потоци, резултата от тази функция не е специфициран, но може да се използва от функцията fseek(), за да възтанови позицията на индикатора.

Функцията ftell() връща -1L при неуспех и съхранява имплементационно зависима положителна стойност в променливата errno.

Пример 1-57:

```c
#include <stdio.h>

int main ()
{
  FILE *pFile;
  long size;

  pFile = fopen ("myfile.txt","rb");
  if (pFile==NULL) perror ("Error opening file");
  else
  {
    fseek (pFile, 0, SEEK_END);
    size=ftell (pFile);
    fclose (pFile);
    printf ("Size of myfile.txt: %ld bytes.\n",size);
  }

  return 0;
}
```

1.21.5.13 fwrite

```c
#include <stdio.h>
size_t fwrite   (const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Функцията fwrite() записва nmemb на брой обекта[1], всеки с размер size (в байтове), сочени от буфера ptr, във файла, асоцииран с потока stream. Индикаторът на текущата позицията на файла се увеличава с броя на успешно записаните байтове. Ако възникне грешка стойността на индикатора на текущата позиция е неопределена.

Забележка[1]: Обект може да бъде както от базов тип (char, int, double и т.н.), така и от потребителски тип, например структура.

Функцията fwrite() връща броя на успешно записаните елементи, който може да е по-малък от nmemb, ако възникне грешка при записа.

Пример 1-58:

```c
#include <stdio.h>

int main(void)
{
  FILE *fp;
  char write[] = "abcdefg";

  fp = fopen ("myfile.txt", "w");

  if (NULL == fp){
      perror ("Cannot open myfile.txt!");
  }
  else {
```

```
        fwrite(write, 1, 5, fp);
    }

    fclose(fp);
    return 0;
}
```

1.21.5.14 remove

```
#include <stdio.h>
int    remove   (const char *filename);
```

Функцията remove() изтрива файла, указан с filename. Ако файлът е отворен, поведението е имплементационно-зависимо.

Функцията remove() връща 0 при успех, в противен случай връща ненулева стойност.

Пример 1-59:

```
#include <stdio.h>

int main ()
{
   if( remove( "myfile.txt" ) != 0 )
     perror( "Error deleting file" );
   else
     puts( "File successfully deleted" );
   return 0;
}
```

1.21.5.15 rename

```
#include <stdio.h>
int    rename   (const char *old_filename, const char *new_filename);
```

Функцията rename() преименува файла, указан с old_filename, с името, указано с new_filename. Ако вече съществува файл със същото име като това зададено с new_filename поведението е имплементационно-зависимо.
Функцията rename() връща 0 при успех. При грешка rename() връща ненулева стойност и името на файла остава непроменено .

Пример 1-60:

```
#include <stdio.h>

int main ()
{
   int result;
   char oldname[] ="oldname.txt";
```

```
  char newname[] ="newname.txt";

  result= rename( oldname , newname );

  if ( result == 0 )
    puts ( "File successfully renamed" );
  else
    perror( "Error renaming file" );

  return 0;
}
```

1.21.5.16 rewind

```
#include <stdio.h>
void   rewind   (FILE *stream);
```

Функцията rewind() премества индикатора на позицията на файла, асоцииран с потока stream, в началото на файла. Поведението е еквивалентно на извикване на функцията fseek() със следните аргументи

```
(void)seek(stream, 0L, SEEK_SET).
```

Допълнително флага за грешка на потока се нулира.

Пример 1-61:

```
#include <stdio.h>

int main ()
{
  int n;
  FILE * pFile;
  char buffer [27];

  pFile = fopen ("myfile.txt","w+");
  for ( n='A' ; n<='Z' ; n++){
      fputc ( n, pFile);
  }
  rewind (pFile);
  fread (buffer,1,26,pFile);
  fclose (pFile);
  buffer[26]='\0';
  puts (buffer);

  return 0;
}
```

1.21.5.17 setbuf

```
#include <stdio.h>
void   setbuf   (FILE *stream, char *buffer);
```

Функцията setbuf() определя как потока stream се буферира. Тази функция трябва да се извика след като потока е бил асоцииран с отворен файл и преди всякави други операции с него. Праметърът buffer, сочи към буфера, който ще се използва от потока. За размер на буфера (в байтове) трябва да се използва макроса BUFSIZ. Ако buffer е нулев указател потокът е небуфериран.

Пример 1-62:

```
#include <stdio.h>

int main ()
{
  char buffer[BUFSIZ];
  FILE *pFile1, *pFile2;

  pFile1=fopen ("myfile.txt","w");
  pFile2=fopen ("myfile2.txt","a");

  setbuf ( pFile1 , buffer );
  fputs ("This is sent to a buffered stream",pFile1);
  fflush (pFile1);

  setbuf ( pFile2 , NULL );
  fputs ("This is sent to an unbuffered stream",pFile2);

  fclose (pFile1);
  fclose (pFile2);

  return 0;
}
```

1.21.5.18 setvbuf

```
#include <stdio.h>
int    setvbuf  (FILE *stream, char *buffer, int mode, size_t size);
```

Функцията setvbuf() определя как потока stream се буферира. Тази функция трябва да се извика след като потока е бил асоцииран с отворен файл и преди всякави други операции с него. Параметърът mode определя режима на буфериране и може да приеме следните аргументи:

- _IOFBF

  Потокът е напълно буфериран. При операция запис, данните първо се записват в буфера. След като буфера се напълни неговото съдържание се записва във файла и буфера се изчиства. При операция четене буфера се запълва изцяло преди данните да се подадат към програмата.

- _IOLBF

Потокът е линейно буфериран. При операция запис данните първо се записват в буфера. Съдържанието на буфера се записва във файла и буфера се изчиства при следните условия:

1. При поява на символ за нов ред .

2. При запълване на буфера.

При четене от файла буферът се пълни до появата на символ за нов ред преди данните да се подадат към програмата.

- _IONBF

Потокът е небуфериран. Операциите запис/четене в/от файла се осъществяват директно. В този случай параметрите buffer и size се игнорират.

Ако buffer е нулев указател, setvbuf() използва функцията malloc(), за да задели памет за буфера с размер size.

Функцията setvbuf() връща 0 при успех. Всяка друга стойност е индикация за грешка или невалиден аргумент на параметъра mode.

Пример 1-63:

```
#include <stdio.h>

int main ()
{
  FILE *pFile;

  pFile=fopen ("myfile.txt","w");

  setvbuf ( pFile , NULL , _IOFBF , 1024 );

  /* Следват операции с файла */
  /*.........................*/

  fclose (pFile);

  return 0;
}
```

1.21.5.19 tmpfile

```
#include <stdio.h>
FILE*  tmpfile  (void);
```

Функцията tmpfile() създава временен файл с режим на достъп wb+ (виж 3.12.5.7 fopen). Файла автоматично се изтрива при неговото затваряне или при завършване на програмата. Ако програмата завърши аварийно, дали файла ще бъде изтрит зависи от имплементацията.

Функцията tmpfile() при успех връща указател към асоциирания с файла поток. Ако файлът не може да бъде създаден tmpfile() връща нулев указател.

Пример 1-64:

```
#include <stdio.h>

int main ()
{
  char buffer [256];
  FILE * pFile;
  pFile = tmpfile ();

  if (NULL == pFile){
        perror("Error Creating a temporary file!");
  }
  else {

  }

  fputs ("Test tmpfile()",pFile);
  rewind(pFile);

  fgets(buffer,254,pFile);
  puts(buffer);

  fclose (pFile);

  return 0;
}
```

1.21.5.20 tmpnam

```
#include <stdio.h>
char*  tmpnam   (char *str);
```

Функцията tmpnam() генерира уникални имена подходящи за използване за имена на временни файлове (файлове създадени ръчно, а не с tmpnam()). Функцията генерира различни имена при всяко свое извикване до TMP_MAX пъти. Ако се извика над този брой, поведението е имплементационно-зависимо.

Ако str е нулев указател, генерираното име се съхранява в статичен буфер, дефиниран в самата функция и се връща указател към него. Ако str е валиден буфер, генерианото име се записва в него и tmpnam() връща

аргумента като резултат. Размерът на подадения буфер трябва да е поне L_tmpnam символа.

Пример 1-65:

```
#include <stdio.h>

int main ()
{
  char buffer [L_tmpnam];
  char * pointer;

  tmpnam (buffer);
  printf ("Tempname #1: %s\n",buffer);

  pointer = tmpnam (NULL);
  printf ("Tempname #2: %s\n",pointer);

  return 0;
}
```

1.21.5.21 fprintf

```
#include <stdio.h>
int    fprintf  (FILE *stream, const char *format, ...);
```

Функцията fprintf() изпраща форматиращ низ, указан с format, към изходния поток stream. Допълнително функцията приема променлив брой аргументи, които се изпращат към потока на мястото на т.нар форматни спецификатори, които са част от форматиращия низ. Форматните спецификартори указват начина, по който съответните аргументи се преобразуват в низове преди да се изпратят към потока. Всички символи от форматиращия низ, които не са част от форматен спецификатор или не са escape-последователности (например \n) се изпращат към потока без промяна. На всеки форматен спецификатор трябва да съответства аргумент от списъка с променлив брой аргументи. На първия форматен спецификатор (от ляво на дясно във форматиращия низ) отговаря първия аргумент от списъка с променлив брой аргументи и т.н. Изключение е само форматния спецификатор %%, който не изисква аргумент. Ако броят на аргументите е по-малък от броя на форматните спецификатори, изискващи аргумент, поведението е недефинирано. Ако броят на аргументите е по-голям от броя на форматните спецификатори, изискващи аргумент, излишните аргументи се изчисляват, но се игнорират.

Един форматен спецификатор има следния формат:

%[флагове][ширина][.прецизност][модификатор]спецификатор

Както се вижда, един форматен спецификатор започва със знак **%,**

последван от четири полета по избор (квадратните скоби указват, че тези полета не са задължителни) и задължително поле **спецификатор**. Значението на всяко от полетата е както следва:

спецификатор

Това поле определя типа на аргумента, който се преобразува в низов еквивалент. Табл.6 описва всички допустими изходни спецификатори.

Спецификатор	Тип на аргумент	Описание
d, i	signed int	Аргумента се преобразува в знаков десетичен низов еквивалент от вида **[-]ddd---d**, където d е десетична цифра 0÷9. Например -10 се преобразува в низа "-10".
o, u, x (или X)	unsigned int	Аргумента се преобразува в безнаков осмичен, десетичен или шестнадесетичен низов еквивалент от вида **dd---d**, където d е съответно осмична цифра 0÷7, десетична цифра 0÷9 или шестнадесетична цифра 0÷9 и a÷f (или A÷F) Например 10 се преобразува в низа "$12_{(8)}$", "$10_{(10)}$", "$a_{(16)}$" или "$A_{(16)}$".
f	double	Аргумента се преобразува в десетичен низов еквивалент от вида **[-]dd---d.dd---d**, където d е десетична цифра 0÷9. Например -12.543 се преобразува в низа "-12,543000"[1]. **Забележка[1]**: Броят на цифрите след десетичната точка се определя от полето **прецизност**. По подразбиране се използват 6 цифри.
e (или E)	double	Аргумента се преобразува в десетичен низов еквивалент от вида **[-]d.dd---de±dd** или **[-]d.dd---dE±dd**, където d е десетична цифра 0÷9. Например -12.543 се преобразува в низа "-1.254300e+001"[1]. **Забележка[1]**: Броят на цифрите след десетичната точка се определя от полето **прецизност**. По подразбиране се използват 6 цифри.
g (или G)	double	Аргумента се преобразува в десетичен низов еквивалент както спецификаора f или e(или E), т.е. **[-]dd---d.dd---d** или **[-]d.dd---de±dd** (или **[-]d.dd---dE±dd)**. Ако експонентата е по-малка от -4 или равна или по-голяма от зададената прецизност се използва спецификатор e (или E),

		иначе се използва спецификатор f. Допълващите нули се премахват. Десетичната точка се отпечатва само ако има ненулева десетична цифра. Например -1.254e-6 се преобразува в низа “-1.254e-006”, а -1.254e-3 се преобразува в низа “-0.001254”.
c	char	Аргумента се преобразува в еквивалентен символ тип unsigned char. Например 100 се преобразува в ‘a’ при използване на ASCII кодовата таблица.
s	char*	Този спецификатор указва, че съответния аргумент е низ. Нулевия символ не се изпраща в потока.
p	тип*	Аргумента е указател, чиято стойност се преобразува в последователност от символи. Например ако указателя ptr съдържа адрес 5DC8, той ще се преобразува в низа ”5DC8”.
n	int*	Аргумента е указател към цяло число, в което се записват броя на символите записани до тук в потока stream от това извикване на fprint(). Няма преобразуване на аргумента.
%	-	Символът % се изпраща в потока. Този спецификатор не изисква аргумент.

Табл. 58 Изходни спецификатори

модификатор

Това поле доопределя типа на аргумента, който се преобразува в низов еквивален. Табл.7 описва всички допустими модификатори и спецификаторите към които могат да се прилагат.

Модификатор	Спецификатор	Описание
h	d, i, o, u, x (или X)	Указва че аргумента е тип short int или unsigned short int.
h	n	Указва, че аргумента е указател към short int.
l	d, i, o, u, x (или X)	Указва че аргумента е от тип long int или unsigend long int
l	c	Указва, че аргумента е тип wchar_t (добавено от **ISO/IEC 9899:1990/Amd 1:1995**)
l	s	Указва, че аргумента е тип wchar_t* (добавено от **ISO/IEC 9899:1990/Amd 1:1995**)
l	n	Указва, че аргумента е указател към long int.
L	e (или E), f, g (или G)	Указва, че аргумента е от тип long double.

Табл. 59 Модификатори

прецизност

Определя прецизността на преобразуване на аргументта. За спецификаторите d, i, o, u, x (или X) определя минималния брой цифри, които да се преобразуват в низов еквивалент. За спецификаторите f, e (или E) определя максималния брой цифри след десетичната точка, които да се преобразуват в низов еквивалент. За спецификаторите g (или G) определя максималния брой значещи цифри, които да се преобразуват в низов еквивалент. За спецификатора s определя максималния брой символи, които да се изпратят към потока.

Това поле започва с точка (.), последвана от десетично число или *. Ако се използва *, прецизността се определя от следващия аргумент (който трябва да е тип int) от списъка с променлив брой аргументи.

Прецизност	Спецификатор	Описание
(празно)	d, i, o, u, x (или X)	Минимум 1 цифра се преобразува. Ако аргумента съдържа повече от 1 цифра всички те ще се преобразуват.
	f, e (или E)	Максимум 6 цифри след десетичната точка се преобразуват. Ако аргумента има повече от 6 цифри след десетичната точка, само 6 от тях ще се преобразуват.
	g (или G)	Всички значещи цифри се преобразуват.
. или .0	d, i, o, u, x (или X)	Използва подразбиращата се прецизност. Ако стойноста на аргумента е нула той не се преобразува.
	f, e (или E)	Символите след десетичната точка се игнорират.
	g (или G)	Приема се прецизност 1.
.N	d, i, o, u, x (или X)	Поне N на брой цифри се преобразуват. Ако N е по-голямо от броя на цифрите в аргумента празните позиции се уплътняват с нули.
	f, e (или E)	N на брой цифри след десетичната точка ще се преобразуват. Ако N е по-голямо от броя на цифрите след десетичната точка, ще бъдат добавени нули, така че броят на десетичните цифри и добавените след тях нули да е равна на N.
	g (или G)	N на брой значещи цифри ще бъдат преобразувани.

Табл. 60 Прецизност

ширина

Широчината на полето се задава с десетично цяло число или * и определя минималния брой позиции, в които аргумента да се преобразува. Ако

аргументът съдържа по-голям брой цифри от зададената ширина, тя се игнорира. Ако аргументът съдържа по-малък брой цифри от зададената ширина, празните позиции се уплътняват с интервали. Ако широчината на полето е зададена с *, то следващия аргумент (който трябва да е тип int), от списъка с променлив брой аргументи, определя широчината на полето.

флагове

Един или повече флагове (във произволен ред) със следното значение:

Флагове	Спецификатор	Описание
-	всички	Ляво подравняване (по подразбиране се използва дясно подравняване). Ако се използва в комбинация с флаг 0, флаг 0 се игнорира.
+	всички	Положителните числа се предхождат от знак +. Ако се използва в комбинация с флаг *интервал*, флаг *интервал* се игнорира.
интервал	всички	Добавя интервал пред положителните числа. Ако се използва в комбинация с флаг +, флаг *интервал* се игнорира.
#	o	Добавя водеща 0 пред резултата.
	x (или X)	Добавя 0x или 0X пред резултата.
	E, e, f, g, G	Десетичната точка винаги се добавя в резултата.
	g (или G)	Допълващите нули в резултата не се премахват.
0	d, i, o, u, x, X, e, E, f, g, и G	Водещи нули се използват да уплътнят ширината на полето вместо интервали. Ако се използва в комбинация с флаг - , флага 0 се игнорира.
	d, i, o, u, x, X	Ако е зададена прецизност, флага 0 се игнорира.

Табл. 61 Флагове

Функцията fprintf() връща при успех броят на символите записани в потока. При възникване на грешка fprintf() връща отрицателна стойност.

Пример 1-66:

```
#include <stdio.h>

int main ()
{
  FILE * pFile;

  pFile=fopen ("myfile.txt","w");
  if (pFile==NULL){
    perror ("The following error occurred");
  }
  else {
    fprintf(pFile,"C89 standard is created in %d", 1989);
    fclose (pFile);
  }
```

```
  return 0;
}
```

1.21.5.22 printf

```
#include <stdio.h>
int    printf   (const char *format, ...);
```

Функцията printf() е еквивалентна на fprintf() с първи аргумент stdout, т.е. форматирания низ по подразбиране се изпраща към стандартния изходен поток stdout.

Пример 1-67:

```
#include <stdio.h>

int main ()
{
  printf("C89 standard is created in %d\n", 1989);

  return 0;
}
```

1.21.5.23 sprintf

```
#include <stdio.h>
int    sprintf  (char *str, const char *format, ...);
```

Функцията sprintf() е еквивалентна на fprintf() с тази разлика, че форматирания низ се записва не в поток, а в символен масив указан с str. В края на записаните символи автоматично се добавя нулев символ. Нулевият символ не е част от върнатата стойност при успех.

Пример 1-68:

```
#include <stdio.h>

int main ()
{
  char str[100];

  sprintf(str,"C89 standard is created in %d\n", 1989);
  printf("%s", str);

  return 0;
}
```

1.21.5.24 vfprintf

```
#include <stdarg.h>
#include <stdio.h>
int    vfprintf (FILE *stream, const char *format, va_list arg);
```

Функцията vfprintf() е аналогична на fprintf() с тази разлика, че последния аргумент е от тип va_list (т.е. указател към списък с аргументи) и трябва да се инициализира с макро-функцията va_start() преди да се подаде на функцията. След връщане от vfprint() програмата трябва да извика макро-функцията va_end(). Функцията vfprintf() извиква макро-функцията va_arg().

Макро-функциите va_start( ), va_arg( ), va_end( ) и типа va_list са декларирани в хедъра <stdarg.h>.

Пример 1-69:

```
#include <stdio.h>
#include <stdarg.h>

void WriteFormatted (FILE * stream, const char * format, ...)
{
  va_list args;
  va_start (args, format);
  vfprintf (stream, format, args);
  va_end (args);
}

int main ()
{
   FILE * pFile;

   pFile = fopen ("myfile.txt","w");

   WriteFormatted (pFile,"Call with %d variable argument.\n",1);
   WriteFormatted (pFile,"Call with %d variable %s.\n",2,"arguments");

   fclose (pFile);

   return 0;
}
```

1.21.5.25 vprintf

```
#include <stdarg.h>
#include <stdio.h>
int    vprintf  (const char *format, va_list arg);
```

Функцията vprintf() е аналогична на printf() с тази разлика, че последния аргумент е от тип va_list (т.е. указател към списък с аргументи) и трябва да се инициализира с макро-функцията va_start() преди да се подаде на

функцията. След връщане от vprint() програмата трябва да извика макро-функцията va_end(). Функцията vprintf() извиква макро-функцията va_arg().

Макро-функциите va_start( ), va_arg( ), va_end( ) и типа va_list са декларирани в хедъра <stdarg.h>.

Пример 1-70:

```
#include <stdio.h>
#include <stdarg.h>

void WriteFormatted ( const char * format, ... )
{
  va_list args;
  va_start (args, format);
  vprintf (format, args);
  va_end (args);
}

int main ()
{
   WriteFormatted ("Call with %d variable argument.\n",1);
   WriteFormatted ("Call with %d variable %s.\n",2,"arguments");

   return 0;
}
```

1.21.5.26 vsprintf

```
#include <stdarg.h>
#include <stdio.h>
int    vsprintf (char *str, const char *format, va_list arg);
```

Функцията vsprintf() е аналогична на sprintf() с тази разлика, че последния аргумент е от тип va_list (т.е. указател към списък с аргументи) и трябва да се инициализира с макро-функцията va_start() преди да се подаде на функцията. След връщане от vsprint() програмата трябва да извика макро-функцията va_end(). Функцията vsprintf() извиква макро-функцията va_arg().

Макро-функциите va_start( ), va_arg( ), va_end( ) и типа va_list са декларирани в хедъра <stdarg.h>.

Пример 1-71:

```
#include <stdio.h>
#include <stdarg.h>

void WriteFormatted ( char *buffer, const char * format, ... )
{
```

```
  va_list args;
  va_start (args, format);
  vsprintf (buffer, format, args);
  va_end (args);
}

int main ()
{
   char str[100];

   WriteFormatted (str, "Call with %d variable argument.\n",1);
   printf("%s", str);
   WriteFormatted (str, "Call with %d variable %s.\n",2,"arguments");
   printf("%s", str);

   return 0;
}
```

1.21.5.27 fscanf

```
#include <stdio.h>
int    fscanf   (FILE *stream, const char *format, ...);
```

Функцията fscanf() е противоположна на fprintf(). Тя чете последователности от символи от входния поток stream, преобразува ги под управлението на форматиращ низ, указан с format, и съхранява резултатите в променливи, сочени от указатели, подадени като аргументи при извикване на функцията. Вида на преобразуването се указва с помощта на форматни спецификатори, които са част от форматиращия низ. На всеки форматен спецификатор трябва да съответства аргумент-указател. Изключение е само форматния спецификатор %%, който не изисква аргумент. Ако броят на форматните спецификатори, изискващи аргумент, е по-голям от броя на аргументите-указатели, поведението е недефинирано. Ако броят на форматните спецификатори, изискващи аргумент, е по-малък от броя на аргументите-указатели, аргументите-указатели се изчисляват, но се игнорират. Типът на аргументите-указатели трябва да отговаря на типа, указан във форматните спецификатори, в противен случай резултатът е недефиниран.

Форматиращия низ се състои от нула или повече директиви, които управляват четенето и преобразуването на символите от потока. Ако изпълнението на една директива пропадне, функцията fscanf() завършва изпълнението си. Причината за това може да е едно от следните две неща:

- липса на символи в потока (**input failure**)
- несъответсвие на символите в потока (**matching failure**)

Директивите се разделят на три категории:

1. ***White-символи*** (интервал ‘ ‘, хор. табулация ‘\t’, вер. табулация ‘\v’, връщане на курсора ‘\r’, нов ред ‘\n’, нова страница ‘\f’). Един white-символ във форматиращия низ води до прочитане и изхвърляне на всички поседователни white-символи от входния поток до първия символ, който е различен от white-символ. Обърнете внимание, че няколко последователни white-символи във форматиращия низ имат същия ефект като един white-символ.

2. ***Форматни спецификатори***. Форматният спецификатор е шаблон, съгласно който функцията fscanf() се опитва да подбере последователност от символи при четене от входния поток, да ги преобразува в съответния формат и да ги съхрани в съответния аргумент.

   Един форматен спецификатор има следния формат:

   %[флаг][ширина][модификатор]спецификатор

   Както се вижда, един форматен спецификатор започва със знак **%,** последван от три полета по избор (квадратните скоби указват, че тези полета не са задължителни) и задължително поле **спецификатор**. Значението на всяко от полетата е описано малко по-надолу.

   Форматния спецификатор се изпълнява по следния начин:

   - Входните white-символи[1] се прескачат, освен за форматните спецификатори, включващи спецификаторите **[, c** или **n** (описанието на спецификаторите е дадено малко по-долу).

     Забележка[1]: White-символите не се взимат под внимание, когато форматния спецификатор включва и полето ширина.

   - Чете се входна последователност от потока[1]. Входна последователност се дефинира като най-дългата последователност от символи, която отговаря на форматния спецификатор. Ако дължината на входната последователност е 0, изпълнението на входната последователност пропада (поради matching failure)

     Забележка[1]: Ако форматния спецификатор включва спецификатора **n** символи от потока не се четат.

- Входната последователност се преобразува[1] до типа, указан в форматния спецификатор. Резултатът от преобразуването се записва[2] в съответния аргумент. Ако типът на аргумента не отговаря на типа, изискван от форматния спецификатор, или резултатът от преобразуването не може да се побере в аргумента, поведението е недефинирано.

Забележка[1]: Ако форматният спецификатор включва спецификатора **%** или **n**, преобразуване не се извършва.

Забележка[2]: Ако форматният спецификатор включва флага *, резултатът от преобразуването не се съхранява в аргумент и се игнорира.

---

3. ***Други символи***. Всеки символ, различен от white-символи и не е част от форматен спецификатор, се сравнява със следващия символ за четене от потока. Ако са еднакви, нищо не се случва, просто прочетените от потока символи се изхвърлят. Ако обаче се различават, четенето от входния поток спира (поради matching failure) и функцията приключва изпълнението си. Входния символ, довел до това си остава в потока за последваща операция четене.

Значението на всяко от полетата във форматния спецификатор е както следва:

спецификатор

Спецификаторът определя типа на аргумента и съответно как една входна последователност от символи трябва да се преобразува. Например %d указва, че входната последователност трябва да се преобразува в цяло десетично число със знак и да се съхрани в целочислен обект, сочен от аргумента-указател. Табл.62 описва всички входни спецификатори.

Спецификатор	Тип на аргумент	Описание
d	signed int*	Входната последователност се преобразува в цяло десетично число със знак. Тя трябва да се състои само от символите ‘+’, ‘-‘, ‘0’÷’9’ и е същата като очакваната от функцията strtol() със стойност 10 за параметъра base.
i	signed int*	Бройната система, в която ще се преобразува входната последователност се определя от първите два символа. Ако първият символ е ‘1’÷’9’, то бройната система е десетична. Ако първия символ е ‘0’, а втория е ‘1’÷’7’, то бройната система е осмична. Ако първия символ е ‘0’, а втория е ‘x’ или ‘X’, то

		бройната система е шестнадесетична. Входната последователност трябва да е същата като очакваната от функцията strtol() със стойност 0 за параметъра base.
u	unsigned int*	Входната последователност се преобразува в цяло десетично число без знак. Тя трябва да се състои само от символите ‘+’, ‘-‘, ‘0’÷’9’ и трябва да е същата като очакваната от функцията strtoul() със стойност 10 за параметъра base.
o	unsigned int*	Входната последователност се преобразува в осмично число. Тя трябва да се състои само от символите ‘0’÷’7’ и трябва да е същата като очакваната от функцията strtoul() със стойност 8 за параметъра base.
x (или X)	unsigned int*	Входната последователност се преобразува в шестнадесетично число. Входната последователност трябва да се състои само от символите ‘0’÷’9’ и ‘a’÷’f’ (или ‘A’÷’F’) и трябва да е същата като очакваната от функцията strtoul() със стойност 16 за параметъра base.
e (или E), f, g (или G)	float*	Входната последователност се преобразува в знаково число с плаваща запетая. Входната последователност трябва да се състои само от символите ‘+’, ‘-‘ , ‘0’÷’9’, ‘.’, ‘e’, ’E’ и трябва да е същата като очакваната от функцията strtod().
s	char*	Входната последователност трябва да се състои от символи, различни от white-символи (‘␣‘, ‘\t’, ‘\v’, ‘\n’, ‘\r’, ‘\f’). Съответния аргумент трябва да бъде достатъчно голям да побере цялата последователност от символи плюс нулев символ, който се добавя автоматично.
[…]	char*	Входната последователност се състои само от символите, указани в квадратните скоби. Някои имплементации позволяват използването на символ ‘-‘ за задаване на диапазон от символи, например a-z, 0-9 и т.н.
[^…]	char*	Входната последователност се състои само от символи, различни от указаните в квадратните скоби. Някои имплементации позволяват използването на символ ‘-‘ за задаване на диапазон от символи, например a-z, 0-9 и т.н.
c	char*	Входната последователност може да е всеки символ, включително и white-символи. Съответния аргумент трябва да е масив от символи. Нулев символ не се добавя. Ако ширината не е специфицирана се приема за 1.
p	void**	Входната последователност трябва да е такава

		каквато е получена при %p на fscanf().
n	int*	Указва, че броя на прочетените до момента символи трябва да се съхрани в съответния аргумент. Символи от входния поток не се четат.
%	-	Входната последователност трябва да се състои само от символа'%'. Това е единствения спецификатор, който не изисква аргумент.

Табл. 62 Входни спецификатори

модификатор

Това поле доопределя типа на аргумента, който се преобразува в низов еквивален. Табл.63 описва всички допустими модификатори и спецификаторите към които могат да се прилагат.

Модификатор	Спецификатор	Тип на аргумента
h	d, i, o, u, x (или X)	short int* или unsigned short *
h	n	short int*
l	d, i, o, u, x (или X)	long int* или unsigned long *
l	n	long int*
l	e (или E), f, g (или G)	double*
L	e (или E), f, g (или G)	long double*

Табл. 63 Модификатори

ширина

Това поле се задава с цяло число и определя броя на символите от входната последователност, които ще се преобразуват съгласно форматния спецификатор. Ако този брой е по-голям от броя на символите в символната последователност, то подбраната символна последователност се преобразува и резултата се съхранява. Ако е по-малък, от входната последователност се избират само указания брой символи, а останалите символи остават в потока за последващо четене.

флаг

Това поле може да съдъжа единственио знака *. Ако този символ следва непосредствено след знака %, то fscanf() подбира и преобразува входна последователност от символи така както е специфицирано от форматния спецификатор, но не съхранява резултата. Ефектът е, че дадена входна последователност просто се прескача.

Функцията fscanf() връща броя на успешно съхранените в аргументите

символни последователности, който може да е по малък от броя на аргументите или дори 0 (поради matching failure). Функцията fscanf() връща EOF ако е възникнала грешка при четенето (input failure) или е достигнат края на файла, преди първото преобразуване.

Пример 1-72:

```
#include <stdio.h>

int main ()
{
  char str [80];
  float f;
  FILE * pFile;

  pFile = fopen ("myfile.txt","w+");
  fprintf (pFile, "%f %s", 3.1416, "PI");
  rewind (pFile);
  fscanf (pFile, "%f", &f);
  fscanf (pFile, "%s", str);
  fclose (pFile);
  printf ("I have read: %f and %s \n",f,str);

  return 0;
}
```

1.21.5.28 scanf

```
#include <stdio.h>
int    scanf    (const char *format, ...);
```

Функцията scanf() е същата като fscanf() с първи аргумент stdin, т.е входния поток е стандартния входен поток stdin.

Пример 1-73:

```
#include <stdio.h>

int main ()
{
  char str [80];
  int i;

  printf ("Enter your family name: ");
  scanf ("%s",str);
  printf ("Enter your age: ");
  scanf ("%d",&i);
  printf ("Mr. %s , %d years old.\n",str,i);
  printf ("Enter a hexadecimal number: ");
  scanf ("%x",&i);
  printf ("You have entered %#x (%d).\n",i,i);

  return 0;
}
```

1.21.5.29 sscanf

```
#include <stdio.h>
int    sscanf   (const char *str, const char *format, ...);
```

Функцията sscanf() е същата като fscanf() с тази разлика, че първия аргумент не е поток, а низ. Достигането на края на низа е еквивалентно на достигане на край-на-файл при fscanf().

Пример 1-74:

```
#include <stdio.h>

int main ()
{
  char sentence []="Rudolph is 12 years old";
  char str [20];
  int i;

  sscanf (sentence,"%s %*s %d",str,&i);
  printf ("%s -> %d\n",str,i);

  return 0;
}
```

1.21.5.30 fgetc

```
#include <stdio.h>
int    fgetc    (FILE *stream);
```

Функцията fgetc() чете поредния символ (unsigned char) от потока stream и премества индикатора на позицията към следващия символ.

Функцията fgetc() връща символа при успех. Ако е достигнат края на файла, fgetc() връща EOF и вдига флага за край-на-файла. Ако възникне грешка fgetc() връща EOF и вдига флага за грешка-на-потока.

Пример 1-75:

```
#include <stdio.h>
int main ()
{
  FILE * pFile;
  int c;
  int n = 0;

  pFile=fopen ("myfile.txt","r");
  if (pFile==NULL) perror ("Error opening file");
  else
  {
    do {
      c = fgetc (pFile);
```

```
      if (c == '$') n++;
    } while (c != EOF);

    fclose (pFile);
    printf ("The file contains %d dollar sign characters ($).\n",n);
  }

  return 0;
}
```

1.21.5.31 fgets

```
#include <stdio.h>
char*  fgets    (char *str, int n, FILE *stream);
```

Функцията fgets() чете символи от потока stream и ги съхранява в буфера str. Четенето и записването на символи в буфера спира при появата на едно от следните условия:

- При прочитане на n-1 на брой символа.
- При прочитане на символ за нов ред преди броят на прочетените символи да е достигнал n-1. Символа за нов ред също се копира в буфера
- При достигане края на файла.

Нулевия символ '\0' автоматично се добавя в края на съхранените в буфера символи.

Функцията fgets() връща str при успех. Ако файлът е празен или възникне грешка при четенето fgets() връща нулев указател.

Пример 1-76:

```
#include <stdio.h>

int main()
{
   FILE * pFile;
   char mystring [100];

   pFile = fopen ("myfile.txt" , "r");
   if (pFile == NULL) perror ("Error opening file");
   else {

     if ( fgets (mystring , 100 , pFile) != NULL )
       puts (mystring);

     fclose (pFile);
   }
```

```
   return 0;
}
```

1.21.5.32 fputc

```
#include <stdio.h>
int    fputc    (int c, FILE *stream);
```

Функцията fputc() записва символа c (преобразуван в тип unsigned char) във файла, асоцииран с потока stream, в текущата позиция ,указвана от индикатора на позицията и премества индикатора към следващата позиция.

Функцията fputc() връща записания символ при успех. При възникване на грешка, fputc() връща EOF и вдига индикатора за грешка-на-потока stream.

Пример 1-77:

```
#include <stdio.h>

int main ()
{
  FILE * pFile;
  char c;

  pFile = fopen ("alphabet.txt","w");
  if (pFile!=NULL) {

    for (c = 'A' ; c <= 'Z' ; c++)
      fputc ( c , pFile );

    fclose (pFile);
  }

  return 0;
}
```

1.21.5.33 fputs

```
#include <stdio.h>
int    fputs    (const char *str, FILE *stream);
```

Функцията fputs() записва низа, указван с str във файла, асоцииран с потока stream. Нулевия символ ‘\0’ не се записва.

Функцията fputs() връща неотрицателна стойност при успех. При възникване на грешка при записа fputs() връща EOF.

Пример 1-78:

```c
#include <stdio.h>

int main ()
{
   FILE * pFile;
   char sentence [256];

   printf ("Enter sentence to append: ");
   fgets (sentence,255,stdin);
   pFile = fopen ("mylog.txt","a");
   fputs (sentence,pFile);
   fclose (pFile);

   return 0;
}
```

1.21.5.34 getc

```c
#include <stdio.h>
int    getc     (FILE *stream);
```

Функцията getc() чете поредния символ (unsigned char) от потока stream и премества индикатора на позицията към следващия символ. Тя е еквивалентна на fgetc() с изключение на това, че е имплементирана като макрос.

Функцията getc() връща символа при успех. Ако е достигнат края на файла, getc() връща EOF и вдига флага за край-на-файла. Ако възникне грешка getc() връща EOF и вдига флага за грешка-на-потока.

Пример 1-79:

```c
#include <stdio.h>

int main ()
{
  FILE * pFile;
  int c;
  int n = 0;

  pFile=fopen ("myfile.txt","r");
  if (pFile==NULL) perror ("Error opening file");
  else
  {
    do {
      c = getc (pFile);
      if (c == '$') n++;
    } while (c != EOF);
    fclose (pFile);
    printf ("File contains %d$.\n",n);
  }

  return 0;
}
```

1.21.5.35 getchar

```
#include <stdio.h>
int    getchar  (void);
```

Функцията getchar() чете поредния символ (unsigned char) от стандартния входен поток stdin и премества индикатора на позицията към следващия символ. Тя е еквивалентна на getc() с аргумент stdin.

Функцията getchar () връща символа при успех. Ако е достигнат края на файла, getchar () връща EOF и вдига флага за край-на-файла. Ако възникне грешка getchar () връща EOF и вдига флага за грешка-на-потока.

Пример 1-80:

```
#include <stdio.h>

int main ()
{
  char c;

  puts ("Enter text. Include a dot ('.') in a sentence to exit:");
  do {
    c=getchar();
    putchar (c);
  } while (c != '.');

  return 0;
}
```

1.21.5.36 gets

```
#include <stdio.h>
char*  gets     (char *str);
```

Функцията gets() чете символи от стандартния входен поток stdin и ги съхранява в буфера str. Четенето и записването на символи в буфера спира при появата на едно от следните условия:

- При прочитане на символ за нов ред. Символа за нов ред не се копира в буфера .
- При достигане края на файла.

Нулевия символ ‘\0’ автоматично се добавя в края на съхранените в буфера символи.

Функцията gets () връща str при успех. Ако файлът е празен или възникне грешка при четенето gets () връща нулев указател.

Пример 1-81:

```
#include <stdio.h>

int main()
{
  char string [256];

  printf ("Insert your full address: ");
  gets (string);
  printf ("Your address is: %s\n",string);

  return 0;
}
```

1.21.5.37 putc

```
#include <stdio.h>
int    putc     (int char, FILE *stream);
```

Функцията putc() записва символа c (преобразуван в тип unsigned char) във файла, асоцииран с потока stream, в текущата позиция ,указвана от индикатора на позицията и премества индикатора към следващата позиция. Тя е еквивалентна на fputc() с изключение на това, че е имплементирана като макрос.

Функцията putc() връща записания символ при успех. При възникване на грешка, putc() връща EOF и вдига индикатора за грешка-на-потока stream.

Пример 1-82:

```
#include <stdio.h>

int main ()
{
  FILE * pFile;
  char c;

  pFile=fopen("alphabet.txt","w");

  for (c = 'A' ; c <= 'Z' ; c++)
  {
    putc (c , pFile);
  }

  fclose (pFile);

  return 0;
}
```

1.21.5.38 putchar

```
#include <stdio.h>
int    putchar  (int c);
```

Функцията putchar() записва символа c (преобразуван в тип unsigned char) във стандартния изходен поток stdout, в текущата позиция, указвана от индикатора на позицията и премества индикатора към следващата позиция. Тя е еквивалентна на fputc() с втори аргумент stdout.

Функцията putchar () връща записания символ при успех. При възникване на грешка, putchar () връща EOF и вдига индикатора за грешка-на-потока stdout.

Пример 1-83:

```
#include <stdio.h>

int main ()
{
  char c;
  for (c = 'A' ; c <= 'Z' ; c++)
        putchar (c);

  return 0;
}
```

1.21.5.39 puts

```
#include <stdio.h>
int    puts     (const char *str);
```

Функцията puts() записва низа str в стандартния изходен поток stdout и автоматично добавя символ за нов ред в изхода. Нулевия символ '\0' не се записва.

Функцията puts() връща неотрицателна стойност при успех. При възникване на грешка при записа puts() връща EOF.

Пример 1-84:

```
#include <stdio.h>

int main ()
{
  char string [] = "Hello world!";
  puts (string);

  return 0;
}
```

1.21.5.40 ungetc

```
#include <stdio.h>
int    ungetc   (int c , FILE *stream);
```

Функцията ungetc() “напъхва” символа c (преобразуван в unsigned char) обратно в буфера на потока, указан с stream, т.е тази функция извършва обратното на това което прави getc(). Функцията ungetc() може да се извика няколко пъти последователно, но не се гарантира, че всички извиквани ще завършат успешно, ако няма междинни операции за четене от потока. При последващо четене от потока ще бъдат прочетени първо обратно напъханите символи с ungetc() в ред обратен на тяхното връщане в потока, т.е. последния напъхан символ с ungetc() ще бъде прочетен първи.

Функцията ungetc() не влияе на файла, асоцииран с потока, т.е напъханите обратно в потока символи няма да се запишат във файла.

При успешно изпълнение на функциите fseek(), fsetpos() или rewind() всички символи напъхани в потока с ungetc() се изтриват.

Ако за аргумент на c се използва макросът EOF опрацията пропада и входния поток остава непроменен.

Успешно извикване на ungetc() нулира флага за края-на-файл на потока. Стойността на индикатора на позицията след четене или изтриване на всички обрано напъхани символи ще бъде същата както преди символите да бъдат напъхани в потока. За текстов поток, стойността на индикатора на позицията след успешно извикване на ungetc() не е специфицирана докато всички обратно напъхани символи не се прочетат или изтрият. За двоичен поток, индикатора на позицията се намалява след всяко успешно извикване на ungetc(). Ако неговата стойност е била нула преди извикването, тя е неопределена след успешно извикването на ungetc().

Функцията ungetc() връща обратно напъхания символ при успех, в противен случай връща EOF.

Пример 1-85:

```
#include <stdio.h>

int main ()
{
  FILE * pFile;
  int c;
  char buffer [256];

  pFile = fopen ("myfile.txt","rt");
  if (pFile==NULL) perror ("Error opening file");
  else {
       while (!feof (pFile)) {
```

```
            c=getc (pFile);
            if (c!=EOF) {
                  if (c == '#') ungetc ('@',pFile);
                  else ungetc (c,pFile);
                  fgets (buffer,255,pFile);
                  fputs (buffer,stdout);
            }
      }
  }

  return 0;
}
```

1.21.5.41 perror

```
#include <stdio.h>
void   perror   (const char *str);
```

Функцията perror() отпечатва съобщение в стандартния поток за грешка stderr в следния формат: първо се отпечатва потребителско съобщение указано с низа str, последвано от двоеточие и интервал, след което се отпечатва подходящо съобщение за грешка, последвано от символ за нов ред. Съобщението за грешка е същото като това генерирано от функцията strerror() с аргумент errno и зависи от имплементацията.

Пример 1-86:

```
#include <stdio.h>

int main ()
{
  FILE * pFile;

  pFile=fopen ("unexist.abc","rb");
  if (pFile==NULL)
    perror ("The following error occurred");
  else
    fclose (pFile);

  return 0;
}
```

1.22 <stdlib.h>

1.22.1 Описание

Хедърът <stdlib.h> дефининира стандартни функции, които могат да се разделят на следните категории по отношение на функционалността си:

- Функции за преобразуване на низове

- Функции за генериране на случайни числа
- Функции за управление на динамична памет
- Функции за управление на изпълняващата среда
- Функции за търсене и сортиране на масиви
- Функции за целочислена аритметика
- Функции за работа с многобайтови символи

Допълнително в този хедър са декларирани няколко типа и макроси.

Табл.64 показва в резюме всички стандартни функции, разделени по категории.

Кагегории	Функции	Описание
Преобразуване на низове	atof	Преобразуване на низ в double
	atoi	Преобразуване на низ в int
	atol	Преобразуване на низ в long
	strtod	Преобразуване на низ в double
	strtol	Преобразуване на низ в long
	strtoul	Преобразуване на низ в unsigned long
Генериране на случайни числа	rand	Генериране на произволно цяло число
	srand	Инициализиране на генератора на произволни числа
Управление на динамична памет	calloc	Заделяне на динамична памет
	free	Освобождаване на динамична памет
	malloc	Заделяне на динамична памет
	realloc	Изменение на вече заделена динамична памет
Управление на изпълняващата среда	abort	Преждевременно спиране на програмата
	atexit	Конфигуриране на функции за изпълнение при завършване на програмата.
	exit	Нормално спиране на програмата
	getenv	Получаване на стойността на проемнлива на средата
	system	Изпълнение на системни команди
Търсене и сортиране на масиви	bsearch	Двоично търсене в масив
	qsort	Сортиране на масив
Целочислена аритметика	abs	Изчисляване на абсолютна стойност на int обект

	div	Целочислено делене на int обекти
	labs	Изчисляване на абсолютна стойност на long обект
	ldiv	Целочислено делене на long обекти
Многобайтови символи	**mblen**	Определяне на дължина на многобайтов символ
	mbstowcs	Преобразуване на многобайтов низ в широк низ
	mbtowc	Преобразуване на многобайтов символ в широк символ
	wcstombs	Преобразуване на широк низ в многобайтов низ
	wctomb	Преобразуване на широк символ в многобайтов символ

Табл. 64 Стандартни функции

1.22.2 Декларации на типове

- ```
  typedef ... size_t;
  ```

  Типът size_t е описан в <stddef.h>

- ```
  typedef ... wchar_t;
  ```

  Типът wchar_t е описан в <stddef.h>

- ```
  typedef struct {
    int quot;
    int rem;
  } div_t;
  ```

  Типът div_t се използва за съхраняване на резултата от функцията div().

- ```
  typedef struct {
    int quot;
    int rem;
  } ldiv_t;
  ```

  Типът ldiv_t се използва за съхраняване на резултата от функцията ldiv().

1.22.3 Декларации на макроси

- NULL

  Макросът NULL е описан в <stdlib.h>

- EXIT_FAILURE

  Този макрос се разширява до целочислен израз, който може да се използва като аргумент на функцията exit() и указва на изпълняващата среда, че програмата е завършила с проблем.

- EXIT_SUCCESS

  Този макрос се разширява до целочислен израз, който може да се използва като аргумент на функцията exit() и указва на изпълняващата среда, че програмата е завършила безпроблемно.

- RAND_MAX

  Този макрос се разширява до целочислен израз, който представалява максималната стойност, която се връща от функцията rand().

- MB_CUR_MAX

  Този макрос се разширява до положителен целочислен израз, чиято стойност е максималния брой байтове в един многобайтов символ от символната таблица, използвана в текущите местни конвенции за категория LC_TYPE, и чиято стойност е по-малка от MB_LEN_MAX. Типът MB_LEN_MAX е дефиниран в <limits.h> . Категорията LC_TYPE, която е свързана с местните културни конвенции, е описана в <locale.h>.

1.22.4 Декларации на функции

-

1.22.5 Оисание на функциите (макро-функциите)

1.22.5.1 atof

```
#include <stdlib.h>
double atof (const char *str);
```

Функцията atof() преобразува символен низ, представляващ числова стойност тип double, в числовия му еквивалент. Например низа “-123.456”

ще се преобразува в числото -123.456. Низът може да представлява и реално число, изразено в научна нотация, например “-1.23456e2” (или “-1.23456E+2”). Всички водещи white-символи се игнорират. Ако в низа се срещне непозволен символ, той и всички символи след него се игнорират и функцията преобразува в числов еквивалент само тази част от низа, която се намира вляво от непозволения символ. Например низа “ \n\t\v -12.34av$%” ще се преобразува в числото -12.34.

С изключение на поведението си при грешка atof() е еквивалентна на

```
strtod (str, (char**)NULL).
```

При успех функцията atof() връща преобразуваното число. Ако низа не може да се преобразува в число, atof() връща 0.

Пример 1-87:

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
  double val;

  val = atof("-123.456");
  printf ("val = %f\n",val);

  val = atof("-1.23456e2");
  printf ("val = %f\n",val);

  val = atof(" \n\t\v-12.34av$%");
  printf ("val = %f\n",val);

  return 0;
}
```

1.22.5.2 atoi

```
#include <stdlib.h>
int atoi (const char *str);
```

Функцията atoi() преобразува символен низ, представляващ десетична числова стойност тип int, в числовия му еквивалент. Например низа “-123” ще се преобразува в числото -123. Всички водещи white-символи се игнорират. Ако в низа се срещне непозволен символ, той и всички символи след него се игнорират и функцията преобразува в числов еквивалент само тази част от низа, която се намира вляво от непозволения символ. Например низа “ \n\v\t-123av$%” ще се преобразува в числото -123.

С изключение на поведението при грешка atof() е еквивалентна на

```
(int)strtol (str, (char**)NULL, 10).
```

При успех функцията atoi() връща преобразуваното число. Ако низа не може да се преобразува в число, atoi() връща 0.

Пример 1-88:

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
  int val;

  val = atoi("-123");
  printf ("val = %d\n",val);

  val = atoi(" \n\t\v-123av$%");
  printf ("val = %d\n",val);

  return 0;

}
```

1.22.5.3 atol

```
#include <stdlib.h>
int atol (const char *str);
```

Функцията atol() преобразува символен низ, представляващ десетична числова стойност тип long, в числовия му еквивалент. Например низа "-123" ще се преобразува в числото -123. Всички водещи white-символи се игнорират. Ако в низа се срещне непозволен символ, той и всички символи след него се игнорират и функцията преобразува в числов еквивалент само тази част от низа, която се намира вляво от непозволения символ. Например низа " \n\v\t-123av$%" ще се преобразува в числото -123.

С изключение на поведението при грешка atol() е еквивалентна на

```
strtol (str, (char**)NULL, 10).
```

При успех функцията atol() връща преобразуваното число. Ако низа не може да се преобразува в число, atol() връща 0.

Пример 1-89:

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
```

```
  int val;

  val = atol("-123");
  printf ("val = %ld\n",val);

  val = atol(" \n\t\v-123av$%");
  printf ("val = %ld\n",val);

  return 0;
}
```

1.22.5.4 strtod

```
#include <stdlib.h>
double strtod(const char *str, char **endptr );
```

Функцията strtod() преобразува низа **str**, представляващ числова стойност тип double, в числовия му еквивалент. Например низа “-123.456” ще се преобразува в числото -123.456. Низът може да представлява и реално число, изразено в научна нотация, например “-1.23456e2” (или “-1.23456E+2”). Всички водещи white-символи се игнорират. Ако в низа се срещне непозволен символ, той и всички символи след него се игнорират и функцията преобразува в числов еквивалент само тази част от низа, която се намира вляво от непозволения символ. Например низа “ \n\t\v -12.34av$%” ще се преобразува в числото -12.34. Адресът на първия срещнат непозволен символ в низа се записва в указателя сочен от параметъра **endptr**, при условие че **endptr** не е нулев указател.

При успех функцията strtod() връща преобразуваното число. Ако низа не може да се преобразува в число, strtod() връща 0. Ако низа представлява число, но излиза извън диапазона от стойности на тип double, strtod() връща ±HUGE_VAL (в зависимост от знака на стойността) и макроса ERANGE се записва в errno. Ако стойността е твърде малка да се представи от тип double, strtod() връща 0 и макроса ERANGE се записва в errno.

Пример 1-90:

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
  double val;
  char *p;

  val = strtod("-123.456", NULL);
  printf ("val = %f\n",val);

  val = strtod(" \n\t\v-123.456av$%", &p);
  printf ("val = %f\n",val);
```

```
  printf ("p -> %s\n",p);

  return 0;
}
```

1.22.5.5 strtol

```
#include <stdlib.h>
long int strtol(const char * str, char **endptr , int base );
```

Функцията strtol() преобразува символния низ **str**, представляващ числова стойност тип long, в числовия му еквивалент. Например низа “-123” ще се преобразува в числото -123. Всички водещи white-символи се игнорират. Ако в низа се срещне непозволен символ, той и всички символи след него се игнорират и функцията преобразува в числов еквивалент само тази част от низа, която се намира вляво от непозволения символ. Например низа “ \n\v\t-123av$%” ще се преобразува в числото -123. Адресът на първия срещнат непозволен символ в низа се записва в указателя сочен от параметъра **endptr**, при условие че **endptr** не е нулев указател.

Параметърът **base** определя бройната система на интерпретиране на числовия низ. Ако base е 0 бройната система се определя от първите два символа на низа. Ако първият символ е цифра в диапазона 1 ÷ 9, бройната система е десетична. Ако първият символ е 0, а вторият е цифра в диапазона 1 ÷ 7, бройната система е осмична. Ако първият символ е 0, а вторият е x (или X), бройната система е шестнадесетична. Ако **base** е от 2 до 36, очакваната последователност от символи в низа е букви и цифри, представляваща цяло число в бройна система, специфицирна от **base**, евентулно предшествана от знак плюс или минус и не включваща целочислен суфикс. На буквите от a (или A) до z (или Z) се приписват стойности от 10 до 35. Само букви, чиято приписана стойност е по-малка от тази на **base** са позволени. Ако стойността на **base** е 16, последователността 0x или 0X може да предшестват последователността от символи, която се преобразува в числов еквивалент.

При успех функцията strtol() връща преобразуваното число. Ако низа не може да се преобразува в число, strtol() връща 0. Ако числовата стойност на низа излиза извън диапазона от стойности на тип long, strtol() връща LONG_MIN или LONG_MAX в зависимост от знака на стойността и макроса ERANGE се записва в errno.

Пример 1-91:

```
#include <stdio.h>
#include <stdlib.h>

int main ()
```

```
{
  long val;
  char *p;

  val = strtol("-123", NULL, 10);
  printf ("val<10> = %ld\n",val);

  val = strtol(" \n\t\v-123av$%", &p, 10);
  printf ("val<10> = %ld\n",val);
  printf ("p -> %s\n",p);

  val = strtol("0x123", NULL, 16);
  printf ("val<16> = %ld\n",val);

  val = strtol("abcd", &p, 12);
  printf ("val<12> = %ld\n",val);
  printf ("p -> %s\n",p);

  return 0;
}
```

1.22.5.6 strtoul

```
#include <stdlib.h>
unsigned long int strtoul(const char * str, char **endptr , int base );
```

Функцията strtoul() преобразува символния низ **str**, представляващ числова стойност тип unsigned long, в числовия му еквивалент. Например низа “123” ще се преобразува в числото 123. Всички водещи white-символи се игнорират. Ако в низа се срещне непозволен символ, той и всички символи след него се игнорират и функцията преобразува в числов еквивалент само тази част от низа, която се намира вляво от непозволения символ. Например низа “ \n\v\t123av$%” ще се преобразува в числото 123. Адресът на първия срещнат непозволен символ в низа се записва в указателя сочен от параметъра **endptr**, при условие че **endptr** не е нулев указател.

Параметърът **base** определя бройната система на интерпретиране на числовия низ. Ако **base** е 0 бройната система се определя от първите два символа на низа. Ако първият символ е цифра в диапазона 1 ÷ 9, бройната система е десетична. Ако първият символ е 0, а вторият е цифра в диапазона 1 ÷ 7, бройната система е осмична. Ако първият символ е 0, а вторият е x (или X), бройната система е шестнадесетична. Ако base е от 2 до 36, очакваната последователност от символи в низа е букви и цифри, представляваща цяло число в бройна система, специфицирна от **base**, евентулно предшествана от знак плюс или минус и не включваща целочислен суфикс. На буквите от a (или A) до z (или Z) се приписват стойности от 10 до 35. Само букви, чиято приписана стойност е по-малка от тази на **base** са позволени. Ако стойността на **base** е 16, последователността 0x или 0X може да предшестват последователността от

символи, която се преобразува в числов еквивалент.

При успех функцията strtoul() връща преобразуваното число. Ако низа не може да се преобразува в число, strtoul() връща 0. Ако числовата стойност на низа излиза извън диапазона от стойности на тип usigned long, strtoul() връща ULONG_MAX и макроса ERANGE се записва в errno.

Пример 1-92:

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
  long val;
  char *p;

  val = strtoul("123", NULL, 10);
  printf ("val<10> = %ul\n\n",val);

  val = strtol(" \n\t\v123av$%", &p, 10);
  printf ("val<10> = %ul\n",val);
  printf ("p -> %s\n\n",p);

  val = strtoul("-1", NULL, 10);
  printf ("val<10> = %ul\n\n",val);

  val = strtol("0x123", NULL, 16);
  printf ("val<16> = %ul\n\n",val);

  val = strtol("abcd", &p, 12);
  printf ("val<12> = %ul\n",val);
  printf ("p -> %s\n\n",p);

  printf("%ul", ULONG_MAX);

  return 0;
}
```

1.22.5.7 calloc

```
#include <stdlib.h>
void *calloc(size_t nmemb , size_t size );
```

Функцията calloc() заделя динамично памет за масив от **nmemb** на брой обекта, всеки с размер **size** в байтове. Общия брой заделена памет е **nmemb*size** байта. Всички битове се инициализират с 0.

Преди програмата да завърши изпълнението си, динамично заделената памет трябва да се освободи с free(), в противен случай тази памет остава заета и не може да се използва за други цели. Ефекта от неосвободена памет се нарича “загуба на памет” (“memory leakage”).

При успех calloc() връща указател към първия байт от динамично заделената памет. При неуспех calloc() връща нулев указател.

Пример 1-93:

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
  int * pi;

  pi = (int*)calloc (5, sizeof(int));

  if (NULL != pi)
  {
        int i;

        for (i = 0; i < 5; i++)
        {
              printf("pi[%d] = %d\n", i, pi[i]);
        }

        putchar('\n');
        free(pi);
  }

  return 0;
}
```

1.22.5.8 free

```
#include <stdlib.h>
void free(void *ptr);
```

Функцията free() освобождава динамично заделената памет (с помощта на calloc(), malloc(), realloc()) към която сочи указателя ptr. Ако ptr е нулев указател, никакви действия не се извършват. Ако ptr не сочи към динамично заделена памет с alloc(), malloc() или realloc() или тази памет е вече била освободена от предишно извикване на free() или realloc(), поведението е недефинирано.

Пример 1-94:

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
  double* pd;

  pd = (double*)malloc(sizeof(double));

  *pd = 3.14;
```

```
  printf("pd -> %f\n", *pd);

  free(pd);

  return 0;
}
```

1.22.5.9 malloc

```
#include <stdlib.h>
void * malloc ( size_t  size );
```

Функцията malloc() заделя памет, чийто размер е указан с **size** в байтове. Заделената памет е неинициализирана.



Преди програмата да завърши изпълнението си, динамично заделената памет трябва да се освободи с free(), в противен случай тази памет остава заета и не може да се използва за други цели. Ефекта от неосвободена памет се нарича “загуба на памет” (“memory leakage”).

При успех malloc() връща указател към първия байт от динамично заделената памет. При неуспех malloc() връща нулев указател.

Пример 1-95:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct
{
	int x;
	int y;
}TCoord;


int main ()
{
  TCoord* p;

  p = (TCoord*)malloc(sizeof(TCoord));

  if (NULL != p){
    p->x = 10;
    p->y = 20;
    printf("p -> x = %d\n", p -> x);
    printf("p -> y = %d\n", p -> y);
    free(p);
  }

  return 0;
}
```

1.22.5.10 realloc

```
#include <stdlib.h>
void *realloc(void *ptr, size_t  size);
```

Функцията realloc() променя размера на вече динамично заделен блок памет, сочен от **ptr**. Новия размер е указан с **size**. Старото съдържание в паметта се запазва. Ако **size** е по-голям от размера на предишния блок, то новодобавените байтове имат неопределена стойност. Ако **size** е по-малък от размера на предишния блок, то излишните байтове в края на блока се махат. Ако realloc() не може да изпълни заявката по някаква причина, то обекта, сочен от **ptr** остава непроменен. Ако **ptr** е нулев указател, функцията realloc() се държи като malloc(). Ако **ptr** е валиден указател към динамично заделена памет, а **size** е 0, то realloc() се държи като free(), т.е освобождава паметта, сочена от **ptr**. Ако **ptr** сочи към памет, която не е заделена с malloc(), calloc() или realloc(), или паметта вече е била освободена от предишно извикване на free() или realloc(), поведението е недефинирано.

Функцията realloc() връща при успех указател към новозаделения блок памет. При неуспех или ако size е 0 realloc() връща нулев указател.

Пример 1-96:

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
  unsigned char* p_old;
  unsigned char* p_new;

  p_old = (unsigned char*)malloc(sizeof(unsigned char) * 5);

  if (NULL != p_old){

    int i;

      for (i = 0 ; i < 5 ; i++ )
      {
            p_old[i] = i;
      }

      for (i = 0 ; i < 5 ; i++ )
      {
            printf("p_old[%d] = %d\n", i, p_old[i]);
      }
      printf("\n\n");

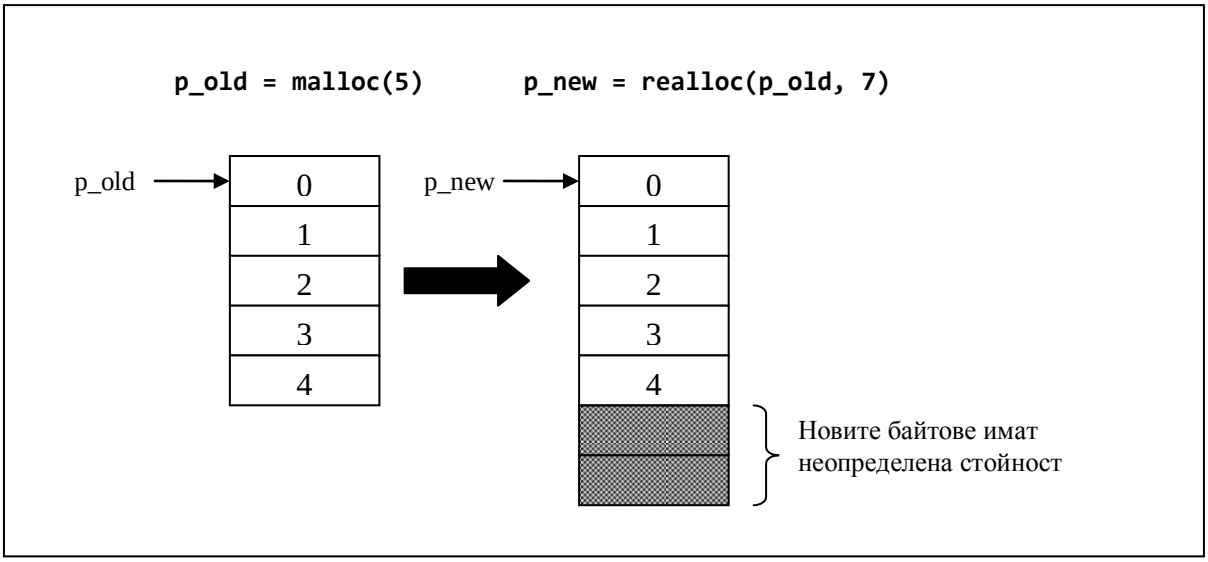
      p_new = (unsigned char*)realloc(p_old, 7);

    if (NULL != p_new){
            for (i = 0 ; i < 7 ; i++ )
            {
                  printf("p_new[%d] = %d\n", i, p_new[i]);
```

```
            }
        }

      free(p_new);
    }

    return 0;
}
```



Фиг. 104 Преоразмеряване на динамична памет с realloc()

1.22.5.11 abort

```
#include <stdlib.h>
void abort(void);
```

Функцията abort() води до принудително прекратяване изпълнението на програмата и връщане на управлението на изпълняващата среда (обикновено операционна система).

Функцията abort() генерира сигнал SIGABRT. Ако този сигнал не се прихваща или функцията обработчик на този сигнал върне резултат, програмата завършва изпълнението си и имплентационно-зависима стойност, указваща неуспех, се връща на изпълняващата среда.

Сигналът SIGABRT е описан в <signal.h>.

Дали отворени потоци ще бъдат затворени при извикване на abort() е имплементационно-зависимо.

Пример 1-97:

```
#include <stdio.h>
#include <signal.h>
```

```c
#include <stdlib.h>

/* Функция-обработчик на SIGABRT */
void sigabrt_handler(int sig)
{
      printf("SIGABRT raised by abort().\n");
}

int main ()
{

  signal(SIGABRT, sigabrt_handler); /* Задаване на обработчик на SIGABRT */

  /* Някакъв код...*/

  abort();

  printf("This will never executes.");

  return 0;
}
```

1.22.5.12 atexit

```c
#include <stdlib.h>
int atexit(void (*func)(void));
```

Функцията atexit() се използва за регистриране на функция, указана с func, която да се извика при нормално завършване на програмата. C89 дефинира, че поне 32 функции могат да бъдат регистрирани по този начин. Извикването на функциите при завършване на програмата става в ред обратен на регистрирането им, т.е. последната регистрирана функция ще се извика първа. Една функция може да бъде регистрирана повече от един път пъти.

При успех atexit() връща 0, в противен случай връща ненулева стойност.

Пример 1-98:

```c
#include <stdio.h>
#include <stdlib.h>

void fnExit1 (void)
{
  puts ("Exit function 1.");
}

void fnExit2 (void)
{
  puts ("Exit function 2.");
}

int main ()
{
  atexit (fnExit1);
```

```
  atexit (fnExit2);
  puts ("Main function.");

  return 0;
}
```

1.22.5.13 exit

```
#include <stdlib.h>
void exit (int status);
```

Функцията exit() извършва нормално завършване на програмата. Всички функции, регистрирани с atexit() се извикват в обратен на регистрирането им ред. Буферите на всички отворени потоци се изчистват, всички отворени потоци се затварят и всички временни файлове, създадени с tmpfile(), се изтриват, и най-накрая управлението се връща на изпълняващата среда (най-често операционната система). Ако стойността на status е 0 (или EXIT_SUCCESS) това е индикация към изпълняващата среда, че програмата е завършила изпълнението си успешно. Ако стойността на status е EXIT_FAILURE това е индикация към изпълняващата среда, че програмата е завършила изпълнението си неуспешно. Параметъра status може да приема и други имплементационно-дефинирани стойности.

Пример 1-99:

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
  FILE * pFile;

  pFile = fopen ("myfile.txt","r");

  if (pFile==NULL)
  {
    printf ("Error opening file");
    exit (EXIT_FAILURE);
  }
  else
  {
    /* файлови операции... */
  }

  return 0;
}
```

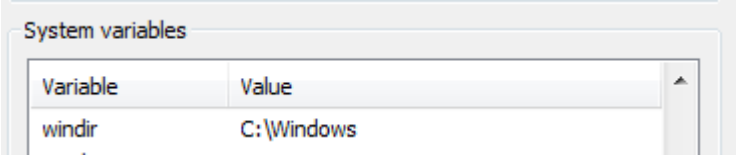
1.22.5.14 getenv

```
#include <stdlib.h>
char *getenv(const char *name);
```

Функцията getenv() претърсва списъка с променливите на изпълняващата среда (**environment variables**) за името указано с низа name. Този списък зависи от имплементацията. C89 също така не дефинира функции за достъп до тези променливи.

Функцията getenv() връща при съвпадение указател към низ, представляващ стойността на променливата на средата, указана с name. Този низ не трябва да се промена от програмата. Ако списъка с променливи на средата не съдържа променлива с име, указано name, getenv() връща нулев указател.

Пример 1-100:




```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
  char *p_var;

  p_var = getenv("windir");

  if (NULL != p_var){
       printf("%s\n", p_var);
  }

  return 0;
}
```

1.22.5.15 system

```
#include <stdlib.h>
int system(const char *string);
```

Функцията system() подава команда, указана с низа string на изпълняващата среда за изпълнение от команден процесор. Като аргумент на string може да се използва нулев указател, за да се провери дали изпълняващата среда поддържа команден процесор.

Ако sting е нулев указател функцията system() връща ненулева стойност ако изпълняващата среда има команден процесор, в противен случай връща 0.

Ако string е ненулев указател, функцията system() връща имплементационно-зависима стойност. Най-често тази стойност специфицира статуса от изпълнението на командата.

Пример 1-101:

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
  int SystemStatus;

  SystemStatus = system(NULL); /* Запитване за наличие на команден процесор */

  if (0 != SystemStatus)
  {
        printf ("The host environment supports a command processor.\n");

        system("dir"); /* Валидна команда в Windows */
  }
  else
  {
        printf ("The host environment doesn't support a command processor.\n");
  }


  return 0;
}
```

1.22.5.16 bsearch

```
#include <stdlib.h>
void *bsearch(const  void *key,
              const  void *base,
              size_t       nmemb,
              size_t       size,
              int (*compar)(const void *, const void *));
```

Функцията bsearch() извършва двоично търсене в сортиран по нарастващ порядък масив. Параметрите на функцията имат следното значение:

- key
  Указва търсената стойност.

- base
  Указва началото на масива.

- nmemb
  Указва броят на елементите на масива.

- size
  Указва размера в байтове на всеки елемент на масива.

- compar
  Указва функцията, която сравнява търсената стойност с елементите на масива. Тя се извиква автоматично от bsearch(), но функционалността и се попълва от потребителя и трябва да спазва следните правила.

  1. Функцията трябва да има следната дефиниция

     ```
     int име-функция(const void * pkey, const void *pelem)
     {
          /*...*/
          return result;
     },
     ```

     където на параметъра pkey се подава key, а на pelem елемента на масива, който се сравнява с key.

  2. Връщаната стойност result трябва да бъде

     0 – ако търсената стойност е равна на сравняваната
     <0 – ако търсената стойност е по-малка от сравняваната
     >0 – ако търсената стойност е по-голяма от сравняваната

Функцията bsearch() при съвпадение връща указател към търсения елемент в масива. Ако търсената стойност не е намерена, bsearch връща нулев указател. Ако масива съдържа повече от една стойности, които съвпадат с търсента, коя от тях ще бъде върната е неспецифицирано.

Пример 1-102:

```
#include <stdio.h>
#include <stdlib.h>

int compareints (const void * a, const void * b)
{
  return ( *(int*)a - *(int*)b );
}

int values[] = { 10, 20, 25, 40, 90, 100 };

int main ()
{
  int * pItem;
  int key = 40;
  pItem = (int*) bsearch (&key, values, 6, sizeof (int), compareints);
```

```
  if (pItem!=NULL)
    printf ("%d is in the array.\n",*pItem);
  else
    printf ("%d is not in the array.\n",key);

  return 0;
}
```

1.22.5.17 qsort

```
#include <stdlib.h>
void qsort(void   *base,
           size_t  nmemb,
           size_t  size,
           int (*compar)(const void *, const void*));
```

Функцията qsort() извършва сортиране на масив по нарастващ порядък. Параметрите на функцията имат следното значение:

- base
  Указва началото на масива.

- nmemb
  Указва броят на елементите на масива.

- size
  Указва размера в байтове на всеки елемент на масива.

- compar
  Указва функцията, която сравнява елементите на масива. Тя се извиква автоматично от qsort(), но функционалността и се попълва от потребителя и трябва да спазва следните правила.

  1. Функцията трябва да има следната дефиниция

     ```
     int име-функция(const void * pelem1, const void *pelem2)
     {
          /*...*/
          return result;
     },
     ```

     където на параметъра pelem1 елемент на масива, а на pelem2 друг елемента на масива.

  2. Връщаната стойност result трябва да бъде

     0 – ако двата елемента са равни
     <0 – ако pelem1 е по-малък pelem2
     >0 – ако pelem1 е по-голям от pelem2

Ако масивът съдържа няколко елемента с еднакви стойности, техният ред в сортирания масив е неспецифициран.

Пример 1-103:

```
#include <stdio.h>
#include <stdlib.h>

int values[] = { 40, 10, 100, 90, 20, 25 };

int compare (const void * a, const void * b)
{
  return ( *(int*)a - *(int*)b );
}

int main ()
{
  int n;

  qsort (values, 6, sizeof(int), compare);
  for (n=0; n<6; n++)
     printf ("%d ",values[n]);

  return 0;
}
```

Пример 1-104:

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>

int main(void)
{
      char string_array[10][50]={"John",
                                 "Jane",
                                 "Mary",
                                 "Rogery",
                                 "Dave",
                                 "Paul",
                                 "Beavis",
                                 "Astro",
                                 "George",
                                 "Elroy"};
      int i;

      qsort(string_array,10,50,strcmp);

      for(i = 0; i < 10 ; i++){
            printf ("%s\n", string_array[i]);
      }

      return 0;
}
```

1.22.5.18 abs

```
#include <stdlib.h>
int abs(int x);
```

Функцията abs() връща абсолютната стойност на числото x. Ако резултатът не може да се представи[1] от тип int, поведението е недефинирано.

$$|x| = \begin{cases} x, \text{ако } x > 0 \\ -x, \text{ако } x < 0 \end{cases}$$

Забележка[1]: Ако отрицателните числа се представят в код с допълненени до 2 (**2's complement**), абсолютната стойност на най-отрицателното число, което може да се представи от тип int (INT_MIN), излиза извън диапазона от стойности на тип int.

Пример 1-105:

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
  printf ("n=%d\n",abs(23));
  printf ("m=%d\n",abs(-11));
  printf ("m=%d\n",abs(INT_MIN));/* !!! Недефиниран резултат*/

  return 0;
}
```

1.22.5.19 div

```
#include <stdlib.h>
div_t div(int numer, int denom);
```

Функцията div() изчислява частното и остатъка от делението на numer на denom. Ако резултатът от делението не е цяло число, дробната част се изхвърля.

Функцията div() връща структурна-променлива от тип div_t, която съдържа два члена: quot и rem. Членът quot съдържа частното на резултата, а rem остатъка и следното условие е в сила:

numer = quot*denom + rem

Ако резултатът не може да се представи, поведението е недефинирано.

Пример 1-106:

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
  div_t val;

  val = div(10,2);
  printf ("quot<+10/+2> = %+d  rem<+10%%+2> = %+d\n", val.quot, val.rem);

  val = div(-10,2);
  printf ("quot<-10/+2> = %+d  rem<-10%%+2> = %+d\n", val.quot, val.rem);

  val = div(10,-2);
  printf ("quot<+10/-2> = %+d  rem<+10%%-2> = %+d\n", val.quot, val.rem);

  val = div(-10,-2);
  printf ("quot<-10/-2> = %+d  rem<-10%%-2> = %+d\n", val.quot, val.rem);

  val = div(10,3);
  printf ("quot<+10/+3> = %+d  rem<+10%%+3> = %+d\n", val.quot, val.rem);

  val = div(-10,3);
  printf ("quot<-10/+3> = %+d  rem<-10%%+3> = %+d\n", val.quot, val.rem);

  val = div(10,-3);
  printf ("quot<+10/-3> = %+d  rem<+10%%-3> = %+d\n", val.quot, val.rem);

  val = div(-10,-3);
  printf ("quot<-10/-3> = %+d  rem<-10%%-3> = %+d\n", val.quot, val.rem);

  return 0;
}
```

1.22.5.20 labs

```
#include <stdlib.h>
long int labs(long x);
```

Функцията labs() е същата като abs() с тази разлика, че типът на параметъра и резултата е long.

1.22.5.21 ldiv

```
#include <stdlib.h>
ldiv_t ldiv(long numer, long denom);
```

Функцията ldiv() е същата като div() с тази разлика, че аргументите са от тип long.

1.22.5.22 rand

```
#include <stdlib.h>
int rand(void);
```

Функцията rand() връща псевдо-случайно число в диапазона 0 ÷ RAND_MAX.

Пример 1-107:

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{

  printf ("random = %d\n", rand());
  printf ("random = %d\n", rand());
  printf ("random = %d\n", rand());

  return 0;
}
```

1.22.5.23 srand

```
#include <stdlib.h>
void srand(unsigned int seed);
```

Функцията srand() захранва генератора на произволни числа, използван от функцията rand(). Използване на една и съща стойност като аргумент на seed принуждава rand() да връща една и съща последователност от псевдо-случайни числа при последователно извикване на rand(). Ако rand() се извика преди srand(), ефекта е все едно srand() е била извикана с аргумент 1.

Пример 1-108:

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{

  printf ("%d ", rand());
  printf ("%d ", rand());
  printf ("%d\n\n", rand());
```

```
  srand(1);
  printf ("%d ", rand());
  printf ("%d ", rand());
  printf ("%d\n\n", rand());

  srand(10);
  printf ("%d ", rand());
  printf ("%d ", rand());
  printf ("%d\n\n", rand());

  srand(10);
  printf ("%d ", rand());
  printf ("%d ", rand());
  printf ("%d\n\n", rand());

  return 0;
}
```

1.22.5.24 mblen

```
#include <stdlib.h>
int mblen (const char *str, size_t n);
```

Функцията mblen() определя броя на байтовете на многобайтовия символ[1], сочен от str. Параметърът n определя колко байта да бъдат взети под внимание от функцията.

Забележка[1]: Повече за многобайтовите символи и методите на кодиране може да намерите в Приложение.

Ако str е нулев указател, функцията mblen() връща ненулева стойност, ако типът на кодиране на многобайтовите символи е **state-dependent**[2], в противен случай връща 0. Ако str не е нулев указател, функцията mblen() връща 0 ако str сочи към нулев символ, или връща броя на байтовете, които изграждат многобайтовия символ (ако следващите n или по-малко байта формират валиден символ), или -1 ако следващите n или по-малко байта, не формира валиден многобайтов символ.

Забележка[2]: Поведението на многобайтовите функции зависи от категорията LC_TYPE на избрания C регион. За кодирания тип state-dependent, всяка функция се поставя в нейното начално състояние, ако на указателя към многобайтовия символ (или низ от многобайтови символи) str се подаде нулев указател. При последващи извиквания на функцията с str, различен от нулев указател, вътрешното състояние на функцията се променя когато е необходимо.

Освен, че вътрешното състояние на функцията mbtowc() не се променя, функцията mblen() е еквивалентна на

mbtowc((wchar_t*)0, str, n);

Пример 1-109:

```
#include <stdio.h>
#include <stdlib.h>

void printbuffer (const char* pt, size_t max)
{
  int length;
  wchar_t dest;

  mblen (NULL, 0);         /* нулиране на  mblen */
  mbtowc (NULL, NULL, 0);  /* нулиране на mbtowc */

  while (max>0) {
    length = mblen (pt, max);
    if (length<1) {
      break;
    }
    mbtowc(&dest,pt,length);
    printf ("[%lc]",dest);
    pt  += length;
    max -= length;
  }
}

int main()
{
  const char str [] = "test string";

  printbuffer (str,sizeof(str));

  return 0;
}
```

Пример 1-110:

```
#include <stdio.h>
#include <stdlib.h>

int  mbcheck ( const char  *s)
{
      int n = -1;

      for( mblen( NULL,  0) ;  ;  s  +=  n ) {
            if( (n = mblen( s,  MB_CUR_MAX))  <=  0) {
                  break;
            }
      }

      return n;
}

int main()
{
      int i;

      i = mbcheck("Test multibyte string.");

      if (0 == i) {
```

```
		printf ("The string is valid multibyte string.\n");
	}
	else {
		printf ("The string contains invalid multibyte characters.\n");
	}

	return 0;
}
```

1.22.5.25 mbstowcs

```
#include <stdlib.h>
size_t mbstowcs (schar_t *pwcs, const char *str, size_t n);
```

Функцията mbstowc() преобразува многобайтов низ в широкосимволен низ[1]. Параметърът pwcs е указател към буфера където ще се съхраняват широките символи, параметърът str е указател към многобайтовия низ, а параметърът n указава максималния брой на широките символи, които да бъде записани в буфера pwcs. Преобразуването продължава докато се срещне нулев символ, невалиден многобайтов символ или броят на преобразуваните символи достигне n. Ако обектите, сочени от pwcs и str се застъпват, поведението е недефинирано.

Забележка[1]: Повече за многобайтовите и широки символи и методите на кодиране може да намерите в Приложение.

Функцията mbstowcs() добавя автоматично нулев символ към широкосимволния низ само ако броят на преобразуваните символи е по-малък от n, в противен случай нулев символ не се добавя.

Функцията mbstowcs() при успех връща[2] броя на символите съхранени в pwcs (без нулевия символ). Ако при преобразуването се срещне невалиден многобайтов символ функцията mbstowcs() връща -1.

Забележка[2]: Поведението на многобайтовите функции зависи от категорията LC_TYPE на избрания C регион. За кодирания тип state-dependent, всяка функция се поставя в нейното начално състояние, ако на указателя към многобайтовия символ (или низ от многобайтови символи) str се подаде нулев указател. При последващи извиквания на функцията с str, различен от нулев указател, вътрешното състояние на функцията се променя когато е необходимо. При извикване с str като нулев указател, функцията връща ненулева стойност ако кодирането е state-dependent, в противен случай връща 0.

Пример 1-111:

```
#include <stdio.h>
```

```c
#include <stdlib.h>

int main()
{
  wchar_t wstr[10];
  int n;

  mbstowcs(wstr, NULL, 0); /* нулиране на mbstowcs */

  n = mbstowcs (wstr, "This is test multibyte string", 5);

  if (-1 != n)
  {
        if (5 == n) {
            wstr[5] = L'\0'; /* добавяне на нулев символ */
        }

        printf("%ls\n", wstr);
  }

  return 0;
}
```

1.22.5.26 mbtowc

```c
#include <stdlib.h>
int mbtowc (whcar_t *pwc, const char *str, size_t n);
```

Функцията mbtowc() преобразува многобайтовия символ[1], сочен от str в широк символ и го съхранява в буфера, сочнен от pwc. Параметърът n определя максималния брой байтове, сочени от str, които да бъдат проверени за валиден многобайтов символ.

Забележка[1]: Повече за многобайтовите символи и методите на кодиране може да намерите в Приложение.

Ако str е нулев указател, функцията mbtowc() връща ненулева стойност, ако типът на кодиране на многобайтовите символи е **state-dependent**[2], в противен случай връща 0. Ако str не е нулев указател, функцията mbtowc() връща 0 ако str сочи към нулев символ, или връща броя на байтовете, които изграждат многобайтовия символ (ако следващите n или по-малко байта формират валиден многобайтов символ), или -1 ако следващите n или по-малко байта не формират валиден многобайтов символ.

Забележка[2]: Поведението на многобайтовите функции зависи от категорията LC_TYPE на избрания C регион. За кодирания тип state-dependent, всяка функция се поставя в нейното начално състояние, ако на указателя към многобайтовия символ (или низ от многобайтови символи) str се подаде нулев указател. При последващи извиквания на

функцията с str, различен от нулев указател, вътрешното състояние на функцията се променя когато е необходимо.

Пример 1-112:

```
#include <stdio.h>
#include <locale.h>
#include <stdlib.h>

int main()
{
	int i = 0, n = 0;
	wchar_t wc;
	char mbstring[256] = "This is originally a multibyte string.\n";

	printf( "The current locale is %s.\n", setlocale(LC_CTYPE, "" ));

	while ( (n = mbtowc( &wc, &mbstring[i], MB_CUR_MAX )) != 0 )
	{
		if ( n == -1 )
		{
		   fputs( "Encoding error in multibyte string", stderr );
		   break;
		}

		printf( "%lc", (wint_t)wc );
		i += n;
	}

	return 0;
}
```

1.22.5.27 wcstombs

```
#include <stdlib.h>
size_t wcstombs (char *str, const wchar_t *pwcs, size_t n);
```

Функцията wcstombs() преобразува един или повече широки символи[1] от низа pwcs в многобайтови символи, започвайки от начално състояние (**initial shift state**), и съхранява резултата в буфера str. Параметърът n указава максималния брой байтове, които могат да бъдат записани в буфера str. Преобразуването спира когато се срещне нулев символ, достигната е границата от n байта или ако записа на поредния преобразуван многобайтов символ би превишил n (в този случай нито един байт от символа не се записва в буфера str). Ако обектите, сочени от pwcs и str се застъпват, поведението е недефинирано.

Забележка[1]: Повече за многобайтовите и широки символи и методите на кодиране може да намерите в Приложение C и D.

Функцията wcstombs() добавя автоматично нулев символ към многобайтовия низ само ако броят на записаните в str байтове е по-малък от n, в противен случай нулев символ не се добавя.

Функцията wcstombs() при успех връща[2] броя на записаните в str байтове (без нулевия символ). Ако се срещне широк символ, който не отговаря на валиден многобайтов символ, функцията wcstombs() връща -1.

Забележка[2]: Поведението на многобайтовите функции зависи от категорията LC_TYPE на избрания C регион. За кодирания тип state-dependent, всяка функция се поставя в нейното начално състояние, ако на указателя към многобайтовия символ (или низ от многобайтови символи) str се подаде нулев указател. При последващи извиквания на функцията с str, различен от нулев указател, вътрешното състояние на функцията се променя когато е необходимо. При извикване с str като нулев указател, функцията връща ненулева стойност ако кодирането е state-dependent, в противен случай връща 0.

Пример 1-113:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  const wchar_t str[] = L"wcstombs example";
  char buffer[100];
  int ret;

  printf ("wchar_t string: %ls \n",str);

  ret = wcstombs( buffer, str, sizeof(buffer) );

  if (-1 != ret)
  {
        printf ("multibyte string: %s \n",buffer);
  }

  return 0;
}
```

1.22.5.28 wctomb

```
#include <stdlib.h>
int wctomb (char *str, wchar_t wchar);
```

Функцията wctomb() преобразува широкия символ wchar в многобайтов[1] и съхранява резултата в буфера (включително всяка промяна в състоянието), сочен от str. Най-много MB_CUR_MAX байта ще бъдат съхранени. Ако стойността на wchar е 0, функцията wctomb() остава в нейното първоначално състояние (**initial shift state**).

Забележка[1]: Повече за многобайтовите и широките символи и методите на кодиране може да намерите в Приложение.

Ако str е нулев указател, функцията wctomb() връща ненулева стойност, ако типът на кодиране на многобайтовите символи е **state-dependent**[2], в противен случай връща 0. Ако str не е нулев указател, функцията wctomb() връща броя на байтовете, на многобайтовия символ, който отговаря на wchar. Ако стойността на wchar не съответства на валиден многобайтов символ, функцията wctomb() връща -1.

Забележка[2]: Поведението на многобайтовите функции зависи от категорията LC_TYPE на избрания C регион. За кодирания тип state-dependent, всяка функция се поставя в нейното начално състояние, ако на указателя към многобайтовия символ (или низ от многобайтови символи) str се подаде нулев указател. При последващи извиквания на функцията с str, различен от нулев указател, вътрешното състояние на функцията се променя когато е необходимо.

Пример 1-114:

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

int main()
{
  char mbbuffer[MB_LEN_MAX] = { '\0' };
  wchar_t wcs[ ] =  L"Wir stehen auf den Füßen von Riesen";
  int n = 0, i = 0;

  printf( "The current locale is %s.\n", setlocale( LC_CTYPE, "" ));
  printf( "The locale's multibyte encoding is %s.\n",
          ( wctomb( NULL, L'\0' ) ? "state-dependent" : "not state-dependent" ));

  do {
    n += wctomb( mbbuffer, wcs[i] );
  } while ( wcs[i++] != L'\0' );

  printf( "The wide string \"%ls\" requires a buffer of %u bytes.\n", wcs, n);

  return 0;
}
```

1.23 <stdnoreturn.h>

1.23.1 Описание

1.23.2 Декларации на типове

1.23.3 Декларации на макроси

1.23.4 Декларации на функции

1.23.5 Описание на функциите (макро-функциите)

1.24 <string.h>

1.24.1 Описание

Хедърът <string.h> функции за обработка на низове и байтови масиви. Табл.65 показва в резюме всички тези функции, разделени по категории.

Кагегории	Функции	Описание
Копиране	memcpy	Копиране на блок памет
	memmove	Преместване на блок памет
	strcpy	Коиране на низ
	strncpy	Копиране на символи от низ
Свързване на низове	strcat	Свързване на низове
	strncat	Добавяне на сивмоли към низ
Сравняване	memcmp	Сравнявнане на блокове памет
	strcmp	Сравняване на низове
	strcoll	Сравняване на низове, чрез използване на местните културни конвенции
	strncmp	Сравняване на символи на два низа
	strxfrm	Преобразуване на низ чрез използване на местните културни конвенции
Търсене в низ	memchr	Търсене на байт в блок памет
	strchr	Търсене на символ в низ
	strcspn	Търсене на символи в низ
	strpbrk	Търсене на символи в низ
	strrchr	Търсене на последното появяване на символ в низ
	strspn	Търсене на символ в низ
	strstr	Търсене на подниз
	strtok	Разделяне на низ на лексеми
Други	memset	Инициализиране на блок

		памет
	strerror	Получаване на указател към съобщение за грешка
	strlen	Определяне дължина на низ

Табл. 65 Функции за обработка на низове и байтови масиви

1.24.2 Декларации на типове

- **size_t**

  Типът size_t е описан в <stddef.h>

1.24.3 Декларации на макроси

- **NULL**

  Макросът NULL е описан в <stddef.h>

1.24.4 Декларации на функции

```
void   *memcpy   ( void * destination, const void * source, size_t num );
void   *memmove  ( void * destination, const void * source, size_t num );
char   *strcpy   ( char * destination, const char * source );
char   *strncpy  ( char * destination, const char * source, size_t num );
char   *strcat   ( char * destination, const char * source );
char   *strncat  ( char * destination, char * source, size_t num );
int     memcmp   ( const void * ptr1, const void * ptr2, size_t num );
int     strcmp   ( const char * str1, const char * str2 );
int     strcoll  ( const char * str1, const char * str2 );
int     strncmp  ( const char * str1, const char * str2, size_t num );
size_t  strxfrm  ( char * destination, const char * source, size_t num );
void   *memchr   (const void *str, int c, size_t n);
char   *strchr   (const char *str, int c);
size_t strcspn   ( const char * str1, const char * str2 );
char   *strpbrk  (const char *str1, const char *str2);
char   *strrchr  (const char *str, int c);
size_t  strspn   (const char *str1, const char *str2);
char   *strstr   (const char *str1, const char *str2);
char   *strtok   (char *str1, const char *str2);
void   *memset   (void *str, int c, size_t n);
char   *strerror (int errnum);
size_t  strlen   (const char *str);
```

1.24.5 Оисание на функциите (макро-функциите)

1.24.5.1 memchr

```
#include <string.h>
void *memchr (const void *str, int c, size_t n);
```

Функцията memchr() търси за първата поява на символа c (преобразуван в unsigned char) в първите n на брой символи (всеки интерпретиран като unsigned char) в низа str.

Фукцията memchr() при успех връща указател към намерения символ, в противен случай връща нулев указател.

Пример 1-115:

```
#include <stdio.h>
#include <string.h>

int main ()
{
  char * pch;
  char str[] = "Example string";

  pch = (char*) memchr (str, 'p', strlen(str));

  if (pch!=NULL) {
    printf ("'p' found at position %d.\n", pch-str);
  }
  else {
    printf ("'p' not found.\n");
  }

  return 0;
}
```

1.24.5.2 memcmp

```
#include <string.h>
int memcmp (const void * str1 , const void *str2 , size_t n);
```

Функцията memcmp() сравнява първите n байта на блоковете памет, сочени от str1 и str2.

Функцията memcmp() връща 0 ако първите n байта са равни, > 0 ако str1 е по-голям от str2 (байта i (i = 0 ÷ n-1) в str1 има по-голяма числова стойност от байта i в str2), < 0 ако str1 е по-малък от str2 (байта i в str1 има по-малка числова стойност от байта i в str2).

Пример 1-116:

```c
#include <stdio.h>
#include <string.h>

typedef unsigned char uint8;

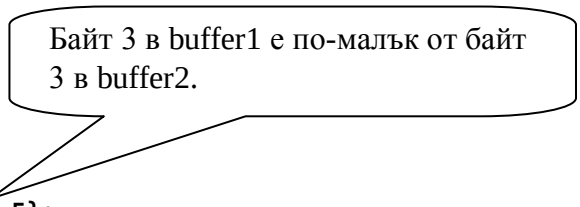
int main ()
{
  uint8 buffer1[] = {1,2,3,4,5};
  uint8 buffer2[] = {1,2,3,6,7};

  int n;

  n = memcmp ( buffer1, buffer2, sizeof(buffer1));

  if (n>0) {
        printf ("buffer1 > buffer2\n");
  }
  else if (n<0) {
        printf ("buffer1 < buffer2\n");
  }
  else {
        printf ("buffer1 == buffer2\n");
  }

  return 0;
}
```



1.24.5.3 memcpy

```c
#include <string.h>
void *memcpy (void *str1 , const void *str2 , size_t n);
```

Функцията memcpy() копира n байта от обекта, сочен от str2, в обекта, сочент от str1. Ако обектите се застъпват, поведението е недефинирано. str1 трябва да е достатъчно голям[1], за да побере n байта на str2.

Забележка[1]: За да не се получи препълване при копирането, размера в байтове на обекта, сочен от str1, трябва да е поне n байта, т.е sizeof(str1) >= n.

Функцията memcpy() връща str1.

Пример 1-117:

```c
#include <stdio.h>
#include <string.h>

typedef unsigned char uint8;

int main ()
{
  uint8 buffer1[] = {1,2,3,4,5};
  uint8 buffer2[10] = {0}; /* Нулиране на buffer2 */
```

```
  int n,i;

  /* Копиране на buffer1 в buffer2 */
  memcpy ( buffer2, buffer1, sizeof(buffer1));

  /* Отпечатване на копираните в buffer2 стойности */
  for (i = 0; i < sizeof(buffer1) ; i++) {
        printf ("buffer2[%d] = %d\n", i, buffer2[i]);
  }

  return 0;
}
```

1.24.5.4 memmove

```
#include <string.h>
void *memmove (void *str1 , const void *str2 , size_t n);
```

Функцията memmove() копира n байта от обекта, сочен от str2, в обекта, сочент от str1. Байтовете първо се копират във временен буфер, след което се копират в обекта, сочен от str1. Тази особеност позволява обектите да се застъпват в паметта.

Функцията memmove() връща str1.

Пример 1-118:

```
#include <stdio.h>
#include <string.h>

int main ()
{
  char str[] = "memmove can be very useful.....";
  memmove (str+20,str+15,11);
  puts (str);

  return 0;
}
```

1.24.5.5 memset

```
#include <string.h>
void *memset (void *str, int c, size_t n);
```

Функцията memset() копира стойността на параметъра c (преобразувана до unsigned char) в първите n на брой байта на блока памет, сочен от str.

Функцията memset() връща str.

Пример 1-119:

```
#include <stdio.h>
#include <string.h>

int main ()
{

  unsigned char mem[5];
  unsigned char *pmem;
  int i;

  pmem = (unsigned char*)memset(mem, 10, 5);

  for (i = 0; i <  5 ; i++)
  {
        printf("mem[%d] = %d\n", i, pmem[i]);
  }

  return 0;
}
```

1.24.5.6 strcat

```
#include <string.h>
char *strcat (char *str1 , const char *str2 );
```

Функцията strcat() добавя копие на низа str2 към низа str1. Първият символ на str2 презаписва нулевия символ на str1. str1 трябва да е достатъчно голям[1], за да побере символите на str2, включително и нулевия символ. Ако низовете се застъпват, поведението е недефинирано.

Забележка[1]: Големината на буфера, в който се съхраняват символите на низа, сочен от str1 трябва да има големина поне (strlen(str1) + strlen(str2) + 1) байта, за да се гарантира, че ще може да побере всички символи на низа str2 плюс нулев символ.

Функцията strcat() връща str1.

Пример 1-120:

```
#include <stdio.h>
#include <string.h>

int main ()
{
  char str[80];

  strcpy (str,"these ");
  strcat (str,"strings ");
  strcat (str,"are ");
```

```
  strcat (str,"concatenated.");
  puts (str);

  return 0;
}
```

1.24.5.7 strncat

```
#include <string.h>
char *strncat(char *str1 , const char *str2 , size_t n);
```

Функция strncat() добавя n символа от низа str2 към низа str1. Първият символ на str2 презаписва нулевия символ на str1. Нулев символ автоматично се добавя към резултата. str1 трябва да е достатъчно голям[1], за да побере символите на str2, включително и нулевия символ. Ако низовете се застъпват, поведението е недефинирано.

Забележка[1]: Големината на буфера, в който се съхраняват символите на низа, сочен от str1 трябва да има големина поне (strlen(str1) + n + 1) байта, за да се гарантира, че ще може да побере n символи от низа str2 плюс нулев символ.

Функцията strncat() връща str1.

Пример 1-121:

```
#include <stdio.h>
#include <string.h>

int main ()
{
  char str1[20];
  char str2[20];

  strcpy (str1,"To be ");
  strcpy (str2,"or not to be");
  strncat (str1, str2, 6);
  puts (str1);

  return 0;
}
```

1.24.5.8 strchr

```
#include <string.h>
char *strchr(const char *str, int c);
```

Функцията strchr() търси за първата поява на символа c (преобразуван до

unsigned char) в низа str. Нулевият символ също се разглежда като част от низа.

Функцията strchr() при успех връща указател към намерения символ, в противен случай връща нулев указател.

Пример 1-122:

```
#include <stdio.h>
#include <string.h>

int main ()
{
  char str[] = "Programing language C.";
  char *pch;

  pch = strchr(str, 'l');

  if (NULL != pch)
  {
     printf ("The first occurrence of \'l\' found in position %d\n", pch - str);
  }

  return 0;
}
```

1.24.5.9 strrchr

```
#include <string.h>
char *strrchr (const char *str, int c);
```

Функцията strchr() търси за последната поява на символа c (преобразуван до unsigned char) в низа str. Нулевия символ също се разглежда като част от низа.

Функцията strchr() при успех връща указател към намерения символ, в противен случай връща нулев указател.

Пример 1-123:

```
#include <stdio.h>
#include <string.h>

int main ()
{
  char str[] = "Programing language C.";
  char *pch;

  pch = strrchr(str, 'g');

  if (NULL != pch)
  {
        printf ("The last occurrence of \'g\' found in position %d\n", pch - str);
```

```
  }

  return 0;
}
```

1.24.5.10 strcmp

```
#include <string.h>
int strcmp(const char * str1 , const char *str2 );
```

Функцията strcmp() сравнява низовете str1 и str2. Низовете се сравняват символ по символ като се взимат числовите им стойности. Сравняването спира ако се срещнат различни символи или нулев символ символ в един от двата низа.

Функцията strcmp() връща 0 ако низовете са еднакви, > 0 ако str1 е по-голям от str2 (символът в str1 има по-голяма числова стойност от символа в str2 или символа в str2 е нулев сивмол), < 0 ако str1 е по-малък от str2 (символът в str1 има по-малка числова стойност от символа в str2 или символа в str1 е нулев сивмол).

Пример 1-124:

```
#include <stdio.h>
#include <string.h>

int main ()
{

  printf ("%d\n", strcmp("abcd", "abcd")); /*   0 */
  printf ("%d\n", strcmp("bcde", "abcd")); /* > 0 */
  printf ("%d\n", strcmp("abcd", "bcde")); /* < 0 */


  return 0;
}
```

1.24.5.11 strncpm

```
#include <string.h>
int strncmp (const char *str1 , const char *str2 , size_t n);
```

Функцията strncmp() сравнява първите n символа на низовете str1 и str2. Символите се сравняват като се взимат числовите им стойности. Сравняването спира ако се срещнат различни символи, нулев символ символ в един от двата низа или след като се сравнят всичките n символа.

Функцията strncmp() връща 0 ако всички n символи са еднакви, > 0 ако i-я

(i = 0 ÷ n-1) символ в str1 има по-голяма числова стойност от i-я символ в str2 или i-я символ в str2 е нулев символ, < 0 ако i-я символ в str1 е по-малък от i-я символ в str2 или i-я символ в str1 е нулев символ.

Пример 1-125:

```
#include <stdio.h>
#include <string.h>

int main ()
{
  char str1[] = "abcdefhijkl";
  char str2[] = "abcdef";

  int n, i;

  i = 3;
  n = strncmp (str1, str2, i);

  if (0 == n) {
      printf ("str1<i> is equal than str2<i>\n");
  }
  else if (n > 0) {
    printf ("str1<i> is bigger than str2<i>\n");
  }
  else {
     printf ("str1<i> is lesser than str2<i>\n");
  }

  return 0;
}
```

1.24.5.12 strcoll

```
#include <string.h>
int strcoll (const char *str1 , const char *str2 );
```

Функцията strcoll() сравнява низовете str1 и str2 сивмол по символ подобно на strcmp(). Разликата е, че strcoll() може да използва категорията LC_COLLATE, за да зададе специфични правила, свързани с местните културни конвенции (виж <locale.h>), за сравняване на низовете.

Функцията strcoll() връща 0 ако низовете са еднакви, > 0 ако str1 е по-голям от str2 (символът в str1 има по-голяма числова стойност от символа в str2 или символа в str2 е нулев сивмол), < 0 ако str1 е по-малък от str2 (символът в str1 има по-малка числова стойност от символа в str2 или символа в str1 е нулев сивмол).

Пример 1-126:

```
#include <stdio.h>
#include <string.h>
```

```
int main ()
{

  printf ("%d\n", strcoll ("abcd", "abcd")); /*   0 */
  printf ("%d\n", strcoll ("bcde", "abcd")); /* > 0 */
  printf ("%d\n", strcoll ("abcd", "bcde")); /* < 0 */


  return 0;
}
```

1.24.5.13 strcpy

```
#include <string.h>
char *strcpy (char *str1 , const char *str2);
```

Функцията strcpy() копира низа str2 в низа str1 (включително и нулевия символ). Ако се получи застъпване, поведението е недефинирано. str1 трябва да е достатъчно голям[1], за да побере символите на str2, включително и нулевия символ.

Забележка[1]: Големината на буфера, в който се съхраняват символите на низа, сочен от str1, трябва да има големина поне (strlen(str2) + 1) байта, за да се гарантира, че ще може да побере всички символи на низа str2 плюс нулев символ.

Функцията strcpy() връща str1.

Пример 1-127:

```
#include <stdio.h>
#include <string.h>

int main ()
{

  char str[40];

  strcpy(str, "Programming langauge C.");
  printf("%s\n", str);

  return 0;
}
```

1.24.5.14 strncpy

```
#include <string.h>
char *strncpy (char *str1 , const char *str2 , size_t n);
```

Функцията strcpy() копира n символи от низа str2 в низа str1. Копирането спира ако са копирани n символи или се достигне нулевия символ на str2. Ако броя на символите в str2 е по-голям от n, то нулев символ не се добавя автоматично към резултатния низ. Ако броя на символите в str2 е по-малък от n, автоматично се добавят нулеви символи докато общия брой на копираните символи стане n. Ако низовете се застъпват, поведението е недефинирано. str1 трябва да е достатъчно голям[1], за да побере n на брой символи.

Забележка[1]: Големината на буфера, в който се съхраняват символите на низа, сочен от str1, трябва да има големина поне (n +1) байта, за да се гарантира, че ще може да побере n символа от str2 плюс нулев символ.

Функцията strncpy() връща str1.

Пример 1-128:

```
#include <stdio.h>
#include <string.h>

int main ()
{
  char str1[]= "To be or not to be";
  char str2[40];
  char str3[40];

  /* пълно копиране */
  strncpy ( str2, str1, sizeof(str2) );

  /* частично копиране (само 5 символа) */
  strncpy ( str3, str2, 5 );
  str3[5] = '\0';   /* ръчно добавяне на нулев символ */

  puts (str1);
  puts (str2);
  puts (str3);

  return 0;
}
```

1.24.5.15 strcspn

```
#include <string.h>
size_t strcspn (const char *str1 , const char *str2 );
```

Функцията strcspn() намира първия символ от низа str1, който съвпада с някой от символите в низа str2.

Функцията strcspn() връща индекса на намерения в str1 символ. Ако

низовете нямат общи символи, strcspn() връща броя на символите в str1.

Пример 1-129:

```
#include <stdio.h>
#include <string.h>

int main ()
{
  char str[]  = "abcd%&$67";
  char keys[] = "1234567890";
  int i;

  i = strcspn (str,keys);
  printf ("i = %d.\n",i);

  return 0;
}
```

1.24.5.16 strspn

```
#include <string.h>
size_t strspn (const char *str1 , const char *str2 );
```

Функцията strspn() намира първия символ от низа str1, който не съвпада с никой от символите в низа str2.

Функцията strspn() връща индекса на намерения в str1 символ. Ако всички символи на str1 се съдържат в str2, strspn() връща броя на символите в str1.

Пример 1-130:

```
#include <stdio.h>
#include <string.h>

int main ()
{
  char str[]  = "1234a";
  char keys[] = "1234567890";
  int i;

  i = strspn (str,keys);
  printf ("i = %d.\n",i);

  return 0;
}
```

1.24.5.17 strerror

```
#include <string.h>
char *strerror (int errnum );
```

Функцията strerror() връща указател към низ, представляващ съобщение за грешка. Номерът на съобщението се задава с errnum. Конкретните съобщения за грешка са имплементационно-зависими и не трябва да се променят от програмата.

Пример 1-131:

```
#include <stdio.h>
#include <string.h>

int main ()
{
  int i;

  for (i = 0 ; i < 10 ; i++)
  {
        printf("String msg No %d: %s\n", i, strerror(i));
  }

  return 0;
}
```

1.24.5.18 strlen

```
#include <string.h>
size_t strlen (const char *str);
```

Функцията strlen() връща броя на символите на низа str. Нулевият символ не се отчита.

Пример 1-132:

```
#include <stdio.h>
#include <string.h>

int main ()
{
  char str[] = "Programming language C.";

  printf("The length of str is %d characters.\n", strlen(str));

  return 0;
}
```

1.24.5.19 strpbrk

```
#include <string.h>
char *strpbrk (const char *str1 , const char *str2 );
```

Функцията strbrk() намира първия символ от низа str1, който съвпада с

някой от символите в низа str2.

Функцията strbrk() връща указател към намерения в str1 символ. Ако низовете нямат общи символи, strbrk() връща нулев указател.

Пример 1-133:

```
#include <stdio.h>
#include <string.h>

int main ()
{
  char str[]  = "abcd6";
  char keys[] = "1234567890";
  char *pch;

  pch = strpbrk (str,keys);

  if (NULL != pch) {
      printf ("The first matching symbol is = %c.\n",*pch);
  }
  else {
    printf ("No matching symbols.\n");
  }

  return 0;
}
```

Пример 1-134:

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char string[]="Programming language C!";
  char *string_ptr;

  /* Замени всички интервали с - */

  while((string_ptr = strpbrk(string," "))!=NULL) {
    *string_ptr='-';
  }

  printf("New string is \"%s\".\n",string);

  return 0;
}
```

1.24.5.20 strstr

```
#include <string.h>
char *strstr (const char *str1 , const char *str2 );
```

Функцията strstr() претърсва низа str1 за първата поява на низа str2 в него.

Нулевия символ на str2 не се взима под внимание.

Функцията strstr() връща указател към първия символ на намерения в str1 подниз, съвпадащ със низа str2. Ако няма съвпадение, strstr() връща нулев указател. Ако str2 е нулев низ, strstr() връща str1.

Пример 1-135:

```
#include <stdio.h>
#include <string.h>

int main ()
{
  char str[] = "Programming language Java";
  char *p_substr;

  p_substr = strstr (str, "Java");

  if (NULL != p_substr) {
       strncpy (p_substr, "C", strlen(p_substr));
       puts(str);
  }

  return 0;
}
```

1.24.5.21 strtok

```
#include <string.h>
char *strtok (char *str1 , const char *str2 );
```

Функцията strtok() разбива низа str1 на лексеми. Разделителите между лексемите се указват с низа str2. При всяко извикване strtok връща указател към лексема или нулев указател ако низа не съдържа лексеми. Процесът на разбиване на лексеми се извършва по следния начин:

1. При първото извикване, на функцията се подава низа, който ще се разбива на лексеми. В низа str2 се указват смволите, които да се използват за разделители.

2. При всяко последващо извикване, на функцията се подава NULL. Разделителите в str2 могат да се сменят за всяко извиквана на strok().

3. Точка 2 се повтаря докато strok() върне нулев указател.

Първото извикване на strtok() търси за първия символ в str1, който не е разделител. Ако няма такъв символ (str1 се състои само от разделители, указани в str2), това означава, че низа str1 не съдържа лексеми и strtok() връща нулев указател. Ако се намери такъв символ, той се маркира като

начало на лексема и функцията продължава да търси за следващия разделител. Ако няма друг разделител, целия низ представлява една лексема и strtok() връща указател към нея, а последващо извикване на strtok() ще върне нулев указател. Ако се намери следващ разделител, той се заменя с нулев символ, с което се маркира края на лексемата и strok() връща указател към началото и. Функцията strtok() съхранява вътрешно указател към следващия символ, който се намира след намерения разделител. При последващо извикване на strok() с нулев указател за първи параметър, процесът ще продължи от този символ. Разбиването на низа str1 на лексеми ще приключи, когато поредното извикване на strtok() с нулев указател върне нулев указател.

Пример 1-136:

```
#include  <string.h>

int main()
{
      static  char  str[]  =  "?a???b,,,#c";
      char  *t;

      /*  t сочи към лексема "a"  */
      t  =  strtok(str,  "?");

      /*  t сочи към лексема "??b"  */
      t  = strtok(NULL,  ",");

      /*  t сочи към лексема "c"  */
      t  = strtok(NULL,  "#,")  ;

      /*  t е нулев указател  */
      t  = strtok(NULL,  "?");

      return 0;
}
```

Пример 1-137:

```
#include <stdio.h>
#include <string.h>

int main ()
{
  char *p_token;
  char str[] = "The programming language C. C is wondeful language!";

  p_token = strtok(str, " .!");

  while(NULL != p_token) {
       printf ("%s\n", p_token);
       p_token = strtok(NULL, " .!");
  }

  return 0;
}
```

1.24.5.22 strxfrm

```
#include <string.h>
size_t strxfrm(char *str1 , const char *str2 , size_t n);
```

Функцията strxfrm() трансформира низа str2 и записва резултата в низа str1. Параметърът n определя максималния брой символи (включително нулевия символ), които функцията може да запише в str1. Ако str1 и str2 се застъпват, поведението е недефинирано.

Видът на трансформацията зависи от стойността на категорията LC_COLLATE и е такава, че ако функцията strcmp() се приложи към два трансформирани с strxfrm() низа, резултатът е 0, > 0 или < 0 и отговаря на резултата от strcoll(), приложена върху същите два първоначални низа преди да бъдат трансформирани с strxfrm(). Аргументът на параметъра str1 може да бъде нулев указател, ако параметърът n е 0.

Функцията strxfrm() връща дължината на трансформирания низ (нулевият символ не е включен). Ако върнатата стойност е по-гояма или равна на n, съдържанието на str1 е неопределено.

Пример 1-138:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
  size_t  len;
  char   *transformed;
  char    original[] = "original";

  /* Определяне дължината на original */
  len = strxfrm(NULL,  original, 0);

  /* Заделяне на памет за трансформирания низ */
  transformed = (char*) malloc(len +  1);

  /* Трансформиране на низа */
  strxfrm(transformed,  original, len);

  /* Отпечатване на първоначалния и трнансформирания низ */
  puts(original);
  puts(transformed);

  /* Освобождаване на паметта */
  free(transformed);

  return 0;
}
```

1.25 <tgmath.h>

1.25.1 Описание

1.25.2 Декларации на типове

1.25.3 Декларации на макроси

1.25.4 Декларации на функции

1.25.5 Описание на функциите (макро-функциите)

1.26 <thread.h>

1.26.1 Описание

1.26.2 Декларации на типове

1.26.3 Декларации на макроси

1.26.4 Декларации на функции

1.26.5 Описание на функциите (макро-функциите)

1.27 <time.h>

1.27.1 Описание

Хедърът <time.h> декларира макроси, типове и функции за обработка на дата и време.

1.27.2 Декларации на типове

- `typedef ... size_t`

  Типът size_t е описан в <stddef.h>

- `typedef ... clock_t`

  Аритметичен тип, способен да представи броя тактове на процесора.

- `typedef ... time_t`

  Аритметичен тип, способен да представя време.

- ```
  struct tm {
   int tm_sec;    /* секунди (0 to 61)                  */
   int tm_min;    /* минути (0 to 59)                   */
   int tm_hour;   /* час (0 to 23)                      */
   int tm_mday;   /* ден (1 to 31)                      */
   int tm_mon;    /* месец (0 to 11)                    */
   int tm_year;   /* година                             */
   int tm_wday;   /* дни след неделя (0 to 6, Неделя=0) */
   int tm_yday;   /* дни от януари 1 (0 to 365)         */
   int tm_isdst;  /* лятно часово време                 */
                    /* > 0 лятно часово време е активно     */
                    /*   0 лятно часово време не е активно  */
                    /* < 0 лятно часово време не е налично  */
  };
  ```

  Структурен тип, чийто членове се използват за съхранение на време и дата.

1.27.3 Декларации на макроси

- NULL
  Макроса NULL е описан в <stddef.h>

- CLOCK_PER_SEC

  Този макрос се разширява до целочислен израз, представляващ брой процесорни такта за секунда.

1.27.4 Декларации на функции

```
char      *asctime   (const struct tm *timeptr);
clock_t    clock     (void);
char      *ctime     (const time_t *timer);
double     difftime  (time_t  time1, time_t time2);
struct tm *gmtime    (const time_t *timer);
struct tm *localtime (const time_t *timer);
time_t     mktime    (struct tm *timeptr );
size_t     strftime  (char *str, size_t maxsize, const char *format,
                      const struct tm *timeptr);
time_t     time      (time_t *timer);
```

1.27.5 Оисание на функциите (макро-функциите)

1.27.5.1 time

```
#include <time.h>
time_t time (time_t *timer);
```
```

Фукцията time() изчислява текущото календарно време и го съхранява в обекта, сочен от timer, при условие, че timer не е нулев указател.

Фукцията time() при успех връща обект тип time_t1, в който се съхранява текущото календарно време. Ако календарното време не е налично, фукцията time() връща -1.

Забележка[1]: Въпреки че C89 не специфицира начина на интерпретиране на тип time_ t, повечето операционни системи дефинират, че time_t е целочислен тип, който може да съдържа броя секунди изтекли от полунощ на 1 януари 1970г.

Резултатът от time() може да се използва от функциите ctime(), gmtime() и localtime(), които го преобразуват в подходяща форма за представяне на дата и час.

Пример 1-139:

```
#include <stdio.h>
#include <time.h>

int main ()
{
  time_t timer;
  char  *time_str;
  struct tm *t;

  timer    = time(NULL);

  /* Използване на ctime()*/
  time_str = ctime(&timer);
  printf ("%s\n", time_str);

  /* Използване на gmtime()*/
  t = gmtime(&timer);
  printf ("%d\\", t->tm_mday);
  printf ("%d\\", t->tm_mon + 1);
  printf ("%ld ", t->tm_year + 1900);
  printf ("%d:", t->tm_hour);
  printf ("%d:", t->tm_min);
  printf ("%d\n\n", t->tm_sec);

  return 0;
}
```

1.27.5.2 clock

```
#include <time.h>
clock_t clock (void);
```

Функцията clock() измерва изтеклото процесорно време в тактове.

Началният момент, спрямо който се извършва измерването е имлементационно-зависим, но е свързан със стартирането на програмата.

Функцията clock() връща броя процесорни тактове изминали от началния момент. Тази стойност, разделена на макроса CLOCKS_PER_SEC дава броя изминали секунди от началния момент. Ако процесорното време не е налично, функцията clock() връща -1.

Пример 1-140:

```
#include <stdio.h>
#include <time.h>

int main(void)
{
  clock_t start;
  clock_t stop;
  double  time;
  int i;

  printf ("Wait...\n");

  start = clock();
  for (i = 0 ; i < 400000000 ; i++);
  stop = clock();

  time = (stop - start) / (double)CLOCKS_PER_SEC;

  printf ("The loop run for %f\n", time);

  return 0;
}
```

Пример 1-141:

```
#include <stdio.h>
#include <time.h>

typedef unsigned char uint8;

int main(void)
{
  clock_t start;
  clock_t temp;
  uint8   delay;

  delay = 1;
  start = clock();
  temp  = start;

  /* Цикъл за закъснение delay на брой секунди */
  while((temp/CLOCKS_PER_SEC-start/CLOCKS_PER_SEC) < delay) {

        temp = clock();
  }

  printf("Took %ld ticks to wait %d second(s).\n",temp-start, delay);

  return 0;
```

```
}
```

1.27.5.3 asctime

```
#include <time.h>
char *asctime (const struct tm *timeptr );
```

Функцията asctime() преобразува разложеното на компоненти време (в променлива от тип struct tm) в низ от вида `"Sun Sep 16 01:03:52 1973\n\0"`.

Функцията asctime() връща указател към низа.

Пример 1-142:

```
#include <stdio.h>
#include <time.h>

int main ()
{
  time_t rawtime;
  struct tm * timeinfo;

  time ( &rawtime );
  timeinfo = localtime ( &rawtime );
  printf ( "The current date/time is: %s", asctime (timeinfo) );

  return 0;
}
```

1.27.5.4 ctime

```
#include <time.h>
char *ctime (const time_t *timer);
```

Функцията ctime() преобразува календарното време, сочено от timer в местно време под формата на низ. Тя е еквивалентна на

```
asctime(locatime(timer));
```

Функцията ctime() връща указател към низа.

Пример 1-143:

```
#include <stdio.h>
#include <time.h>

int main (void)
{
  time_t rawtime;
```

```
  time ( &rawtime );
  printf ( "The current local time is: %s", ctime (&rawtime) );

  return 0;
}
```

1.27.5.5 difftime

```
#include <time.h>
double difftime (time_t  time1, time_t time0);
```

Функцията difftime() изчислява разликата между две календарни времена: time1 – time0.

Функцията difftime() връща разликата, изразена в секунди.

Пример 1-144:

```
#include <stdio.h>
#include <time.h>

typedef unsigned char uint8;

int main(void)
{
  time_t start;
  time_t temp;
  uint8   delay;
  double   sec;

  delay = 10;
  start = time(NULL);
  temp  = start;

  printf("Wait...\n");

  /* Цикъл за закъснение delay на брой секунди */
  while( (sec = difftime(temp, start)) < delay) {

        temp = time(NULL);
  }

  printf("Delay time %0.2f\n", sec);

  return 0;
}
```

1.27.5.6 gmtime

```
#include <time.h>
struct tm *gmtime (const time_t *timer);
```

Функцията gmtime() разлага календарното време, сочено от timer, на компоненти. Времето се представя като Координирано Универсално Време (**UTC**) известно още като Средно Време по Гринуич (**Greenwich Mean Time - GMT**).

При успех функцията gmtime() връща указател към статичен обект тип struct tm, в който се съхранява времето, разложено на компоненти. Ако UTC не е налично, gmtime() връща нулев указател.

Пример 1-145:

```
#include <stdio.h>
#include <time.h>

int main(void)
{
  time_t timer;
  struct tm *t;

  time (&timer);

  t = gmtime(&timer);

  printf ("%d\\", t->tm_mday);
  printf ("%d\\", t->tm_mon + 1);
  printf ("%ld ", t->tm_year + 1900);
  printf ("%d:", t->tm_hour);
  printf ("%d:", t->tm_min);
  printf ("%d\n\n", t->tm_sec);

  return 0;
}
```

1.27.5.7 localtime

```
#include <time.h>
struct tm *localtime (const time_t *timer);
```

Функцията localtime() разлага календарното време, сочено от timer, на компоненти. Времето се представя съгласно местната времева зона.

Пример 1-146:

```
#include <stdio.h>
#include <time.h>

int main(void)
{
  time_t timer;
  struct tm *t;

  time (&timer);
```

```
  t = localtime(&timer);

  printf ("%d\\", t->tm_mday);
  printf ("%d\\", t->tm_mon + 1);
  printf ("%ld ", t->tm_year + 1900);
  printf ("%d:", t->tm_hour);
  printf ("%d:", t->tm_min);
  printf ("%d\n\n", t->tm_sec);

  return 0;
}
```

1.27.5.8 mktime

```
#include <time.h>
time_t mktime (struct tm *timeptr );
```

Функцията mktime() преобразува обект от тип sruct tm, сочен от timeptr, в обект от тип time_t в местно време. Първоначалните стойности на членовете tm_wday и tm_yday на структурата се игнорират, а стойностите на останалите членове не са ограничени до указаните по-горе (виж 3.15.2 Декларация на типове). При успешно завършване, се задават подходящи стойности на членовете tm_wday и tm_yday, а на другите членове се задават стойности (в позволения диапазон), които да представят специфичното календарно време: крайната стойност на tm_mday не се задава докато не се определят стойностите на tm_mon и tm_year.

Функцията mktime() връща специфицираното календарно време под формата на обект тип time_t. Ако календарното време не може да бъде представено, функцията mktime() връща -1.

Пример 1-147:

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

/* Намира кой ден от седмицата е 01.01.2013 */

int main(void)
{
  struct tm time_struct;

  char *days[7] = {"Sunday", "Monday", "Tuesday", "Wednsday", "Thursday",
                   "Friday", "Saturday"};

  time_struct.tm_year  = 2013-1900;
  time_struct.tm_mon   = 0;
  time_struct.tm_mday  = 1;
  time_struct.tm_sec   = 0;
  time_struct.tm_min   = 0;
  time_struct.tm_hour  = 0;
  time_struct.tm_isdst = -1;
```

```
  if(mktime(&time_struct)==-1)
  {
    printf("Error getting time.\n");
    exit(0);
  }

  printf("January 1, 2013 is a %s.\n",days[time_struct.tm_wday]);

  return 0;
}
```

Пример 1-148:

```
#include <time.h>
#include <stdio.h>

int main(void)
{
  time_t t, tt;
  struct tm *pt;

  time(&t);
  pt = localtime(&t);
  tt = mktime(pt);

  printf("%s\n", ctime(&t));
  printf("%s\n", ctime(&tt));

  return 0;
}
```

1.27.5.9 strftime

```
#include <time.h>
size_t strftime (char           *str,
                 size_t          maxsize,
                 const char     *format ,
                 const struct tm *timeptr );
```

Функцията strftime() преобразува времето представено чрез обект от тип struct tm в низ под управлението на форматиращ низ. Параметърът timeptr, сочи към преобразувания обект. Форматиращия низ се указва с format. Резултата се съхранява в низа str. Параметърът maxsize указва максималния брой символи, които да бъдат записани в str (включително и нулевия символ).

Форматиращия низ се състои от обикновени символи и форматни спецификатори. Форматните спецификатори започват с % и имат специално значение. Табл.66 описва всички позволени спецификатори.

Форматен спецификатор	Описание	Пример
%a	абревиатура на ден от	Thu

	седмицата	
%A	пълно име на седмицата	Thursday
%b	абревиатура на месеца	Aug
%B	пълно име на месеца	August
%c	дата и време	Thu Aug 14:55:02 2013
%d	ден от месеца (0-31)	23
%H	час в 24 часов формат	14
%I	час в 12 часов формат	02
%j	ден от годината (001 - 366)	235
%m	месец (01-12)	08
%M	минути (00-59)	23
%p	AM или PM	PM
%S	секунди (00-61)	02
%U	номер на седмица с Неделя като първи ден от първата седмица (00-53)	21
%w	ден от седмицата (0-6), Неделя е 0	1
%W	номер на седмица с Понеделник като първи ден от първата седмица (00-53)	20
%x	дата	08/23/01
%X	време	14:55:02
%y	година (без указване на века) (00-99)	01
%Y	година (с указване на века)	2001
%Z	Времева зона или абревиатура	CDT
%%		%

Табл. 66 Форматни спецификатори на strftime()

Ако резултатния низ съдържа не повече от maxsize символи (включително и нулевия символ) функцията strftime() връща броя на символите съхранени в низа str (без нулевия символ). В противен случай, функцията strftime() връща 0 и съдържанието на низа str е неопределено.

Пример 1-149:

```
#include <stdio.h>
#include <time.h>

int main ()
{
  time_t rawtime;
```

```
  struct tm * timeinfo;
  char buffer [80];

  time ( &rawtime );
  timeinfo = localtime ( &rawtime );

  strftime (buffer,80,"Now it's %I:%M%p.",timeinfo);
  puts (buffer);

  return 0;
}
```

1.28 <uchar.h>

1.28.1 Описание

1.28.2 Декларации на типове

1.28.3 Декларации на макроси

1.28.4 Декларации на функции

1.28.5 Описание на функциите (макро-функциите)

1.29 <wchar.h>

1.29.1 Описание

Хедърите <stdio.h>, <stdlib.h>, <string.h> и <time.h> декларират функции за обработка на байтови низове: това са низове от символи тип char. Хедърът <wchar.h> декларира подобни функции, но за обработка на широки низове: това са низове от символи тип wchar_t. Имената на тези функции като цяло са същите като е добавена една допълнителна буква w, например printf() -> wprintf().

Освен това <wchar.h> декларира допълнителни функции за преобразуване на многобайтови символи в широки символи и обратно, в допълнение към тези декларирани в <stdlib.h>. Функциите декларирани в <wchar.h> могат да се разделят на следните категории.

- Вход-изход на широки символи
- Преобразуване на широки символи в числови еквиваленти
- Функции за обработка на широки низове
- Функции за форматиране на време и дата в широки низове

- Преоабразуване на многобайтови символи в широки и обратно

1.29.2 Декларации на типове

- `typedef ... size_t;`

  Типът size_t е описан в <stddef.h>.

- `typedef ... wchar_t;`

  Типът wchar_t е описан в <stddef.h>.

- `struct tm`

  Типът struct tm е описан в <time.h>.

- `typedef ... wint_t;`

  Това е целочислен тип достаттъчно голям да побере всички стойностти на тип wchar_t и стойността на макроса WEOF.

- `typedef ... mbstate_t;`

  Този тип съдържа информация за състоянието (**shift state**) при преобразуване между многобайтови и широки символи.

  Кодорането на многобайтови символи може да има различни състояния, които променят интерпретацията на следващите байтове.

  Всички валидни многобайтови последователности трябва да започват и свършват с едно и също състояние, наречено начално състояние (**initial shift state**). Ако mbstate_t е 0, това е индикация за начално състояние.

1.29.3 Декларации на макроси

- **NULL**

  Макроса NULL е описан в <stddef.h>

- **WCHAR_MAX**

Представлявава максималната стойност, която може да се представи от тип wchar_t.

- **WCHAR_MIN**

  Представлявава минималната стойност, която може да се представи от тип wchar_t.

- **WEOF**

  Константа от тип wint_t, чиято стойност която се различава от всички символи от разширената символна таблица. За разлика от EOF, стойността на WEOF може да е положителна. Различни функции могат да върнат като резултата WEOF, за да укажат, че е достигнат края на файла или е настъпила грешка.

1.29.4 Декларации на функции

Пълнен списък на всички декларирани в <wchar.h> функции може да намерите на следния адрес:

http://www.cplusplus.com/reference/cwchar/

1.29.5 Описание на функциите (макро-функциите)

Пълно описание на функциите може да намерите на следния адрес:

http://www.cplusplus.com/reference/cwchar/

1.30 <wctype.h>

1.30.1 Описание

Хедърът <wctype.h> декларира функции за обработка на широки символи, които са аналогични на функциите декларирани в <ctype.h> за работа с байтови символи, плюс няколко допълнителни функции.

1.30.2 Декларации на типове

0

- `typedef ... wint_t;`

  Типът wint_t е описан в <wchar.h>.

- `typedef ... wctrans_t;`

  Описание на този тип може да намерите на адрес:

  http://www.cplusplus.com/reference/cwctype/wctrans_t/

- `typedef ... wctype_t;`

  Описание на този тип може да намерите на адрес:

  http://www.cplusplus.com/reference/cwctype/wctype_t/

1.30.3 Декларации на макроси

1.30.4 Декларации на функции

Пълнен списък на всички декларирани в <wctype.h> функции може да намерите на следния адрес:

http://www.cplusplus.com/reference/cwchar/

1.30.5 Оисание на функциите (макро-функциите)

Пълно описание на функциите може да намерите на следния адрес:

http://www.cplusplus.com/reference/cwchar/