ANSI C

C89
C99
C11

Курс за начинаещи

Първо издание

Димо Петков

ANSI C
Курс за начинаещи

Първо издание

София, 2014 г.

Емил Кирилов, технически редактор

ISBN 978-954-8628-83-9
Предпечат: БГкнига

Кратко съдържание:

Съдържание:

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 (най-вече на английски език), описващи езика С. Всяка от тях си има своите предимства и недостатъци. Като начинаещи С-програмисти вероятно сте се сблъскали с въпросите "Откъде да започна?", "Коя книга е подходяща за мен?", "Колко книги трябва да прочета, за да науча езика С?". Създадох тази книга като отговор на тези въпроси, за да спестя на начинаещите С-програмисти цялото лутане в търсене на правилния път. Надявам се, че съм успял да постигна до голяма степен целта си. Може да изпращате Вашите въпроси и забележки на dimope@abv.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

В момента, в който започнах да чета черновата, разбрах, че става въпрос за наистина нещо много различно от техническата литература, с която съм се сблъсквал до момента. Мога да кажа, че това е четиво, което е много

подходящо за начинаещи, но от него могат да се учат и напредналите програмисти. Обикновено малко от книгите на пазара отделят внимание на това как да инсталираме средата за програмиране, как да я настроим и да създадем първия си проект. От скромния ми опит като преподавател съм забелязал, че хората, които искат да се занимават с програмиране или по-точно изучаването на някакъв език, срещат основно трудности не с написването на първата си програма от типа "Hello World!", а по-скоро с това да направят проект и да го стартират. Много от хората се отказват на този етап, колкото и нереално да звучи това.

По-напред в книгата се забелязва съчетанието на детайлност на поднесената информация и достъпния език. Тук проличава, че книгата е създадена от инженер, голям професионалист и практик, а не от човек подчинен на академичния начин на мислене. Примерите показват максимално точно проблеми, които се срещат не само в изучаването на езика C, но и в създаването на реални софтуерни продукти. Трябва да се обърне сериозно внимание на факта, че в края на книгата са засегнати и новите стандарти на езика C.

В заключение мога да кажа само едно: независимо дали читателят е начинаещ програмист, или човек с много опит, то определено си заслужава да отдели внимание както на тази книга, така и на останалите от поредицата.

Благодарности

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

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

Предговор

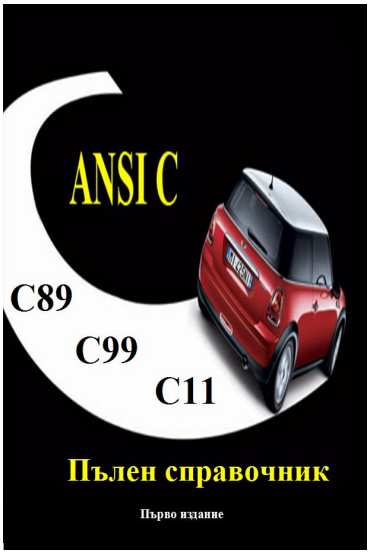
С е мощен процедурно-ориентиран език, създаден през 1972 година от Денис Ричи. Създаването на С предизвиква революция в програмирането, което го прави най-предпочитания език за програмиране следващите две-три десетилетия. С развитието на технологиите възникват и по-сложни за решаване проблеми. Това води 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 С. Въпреки че тези езици ограничават сферата на използване на С, 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 първата 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. Процесът включва три стъпки. Първата стъпка е да се запознаете с елементите на езика, което е и целта на настоящата книга. Като начало ще пишете и изпълнявате Вашите програми на персонален компютър.

Настоящата книга разглежда почти всички аспекти на езика С, но не влиза в дълбоки детайли, които биха затруднили начинаещия програмист. Предвидена е втора книга, която разглежда детайлно езика С и може да се използва за справочник дори от опитни програмисти.



Книгата е в процес на разработка.

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 С код, на базата на конкретен микроконтролер.

Книгата е в процес на разработка.

По мнението на много специалисти С е фундаментален език, който всеки програмист би трябвало да знае. Аз също споделям това мнение. С 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 С много ще Ви улесни, в случай че решите да изучавате C++, C# или Java.

За кого е предназначена тази книга

Тази книга 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.

Какво да очаквате от тази книга

Тъй като езикът C съдържа някои тънкости, чието обяснение би объркало всеки начинаещ, то настоящата книга е фокусирана върху базовите знания, свързани с езика C. Освен че ще придобиете добри познания за структурата на езика, ще разберете и как той функционира. За максимално улеснение на абсолютно начинаещите C-програмисти е добавена и предварителна информация, необходима за разбирането на описания материал. Прочитайки книгата, Вие ще имате всички необходими знания да разберете детайлите и тънкостите на езика C, описани в следващата книга **ANSI C. Пълен справочник**.

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 пет части:

Част I Ви запознава с минимума предварителни знания, които ще са Ви необходими при изучаването на езика C с помощта на тази книга. Тези знания включват представянето на информацията в един компютър, начина на представяне на числата в различни бройни системи (двоична, десетична, шестнайсетична) и основните компоненти на една компютърна система от гледна точка на един програмист.

Част II Ви запознава с елементите на езика C, като разглежда само стандарта C89(C90). Това е стандартът, който е добре установен и намира най-широко приложение в програмирането на микроконтролери.

Част III разглежда основните разлики и нововъведения, добавени в следващата версия на езика C: C99. Повечето компилатори[1], в по-малка или по-голяма степен, също поддържат този стандарт.

Забележка[1]: Компилаторът е специална програма, която преобразува програмния код в код, подходящ за изпълнение от компютъра.

Част IV прави преглед на 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 V разглежда стандартната библиотека на езика C.

Как да четете тази книга

Тъй като тази книга е предназначена основно за начинаещи, е необходимо да я прочетете цялата, без да прескачате глави. Част I е основополагаща, затова, ако нещата описани там са Ви непознати, прочетете я внимателно и се опитайте да извлечете максимума от нея. Четете всяка глава внимателно и изпълнявайте примерите. Опитайте се да отговорите на въпросите за самопроверка в края на главата. Сверете Вашите отговори с моите. Убедете се, че сте разбрали материала от текущата глава и продължете към следващата. Ако все пак се окаже, че не успявате да си изясните конкретно нещо, това не бива да Ви обезкуражава. Отбележете си го и продължете напред. На по-късен етап, когато сте натрупали повече знания, се върнете и разгледайте отново темата, която не Ви е ясна.

Опитал съм се да използвам максимално кратки и едновременно илюстративни примери. Въпреки че може да ги копирате в текстовия редактор, препоръчително е да ги въвеждате на ръка. Това ще Ви помогне да свикнете да въвеждате код и да осмисляте всеки ред от програмата по-добре. След като сте разбрали как работи даден пример, може да използвате лист и химикалка и да го напишете на листа, като отново мислено (може и на глас 🙂) разсъждавате върху всеки ред. Постепенно с натрупването на повече знания, сами ще си задавате въпроси какво би станало, ако дадена конструкция се напише по друг начин. Когато стигнете до тази фаза, не се чудете, просто променете примера и вижте какъв ще бъде резултатът. Затова не се страхувайте да експериментирате.

Успех!

Примерите в книгата

Примерите в книгата могат да бъдат изтеглени от:

www.progstarter.com

С стандарти

Езикът С е стандартизиран първо през 1989 г. от ANSI (American National Standards Institute) комитета. През 1990 г. 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 C89 или C90, и се означават съответно като **ANSI X3.159-1989** и **ISO/IEC 9899:1990**. 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 C95)

ISO/IEC 9899:1990/Cor 2:1996

През 1999 г. е приет нов стандарт **ISO/IEC 9899:1999**, който заменя предходния с всички корекции и добавки. Този стандарт е известен още като C99. Той също претърпява изменения през следващите няколко години както е показано долу:

ISO/IEC 9899:1999/Cor 1:2001

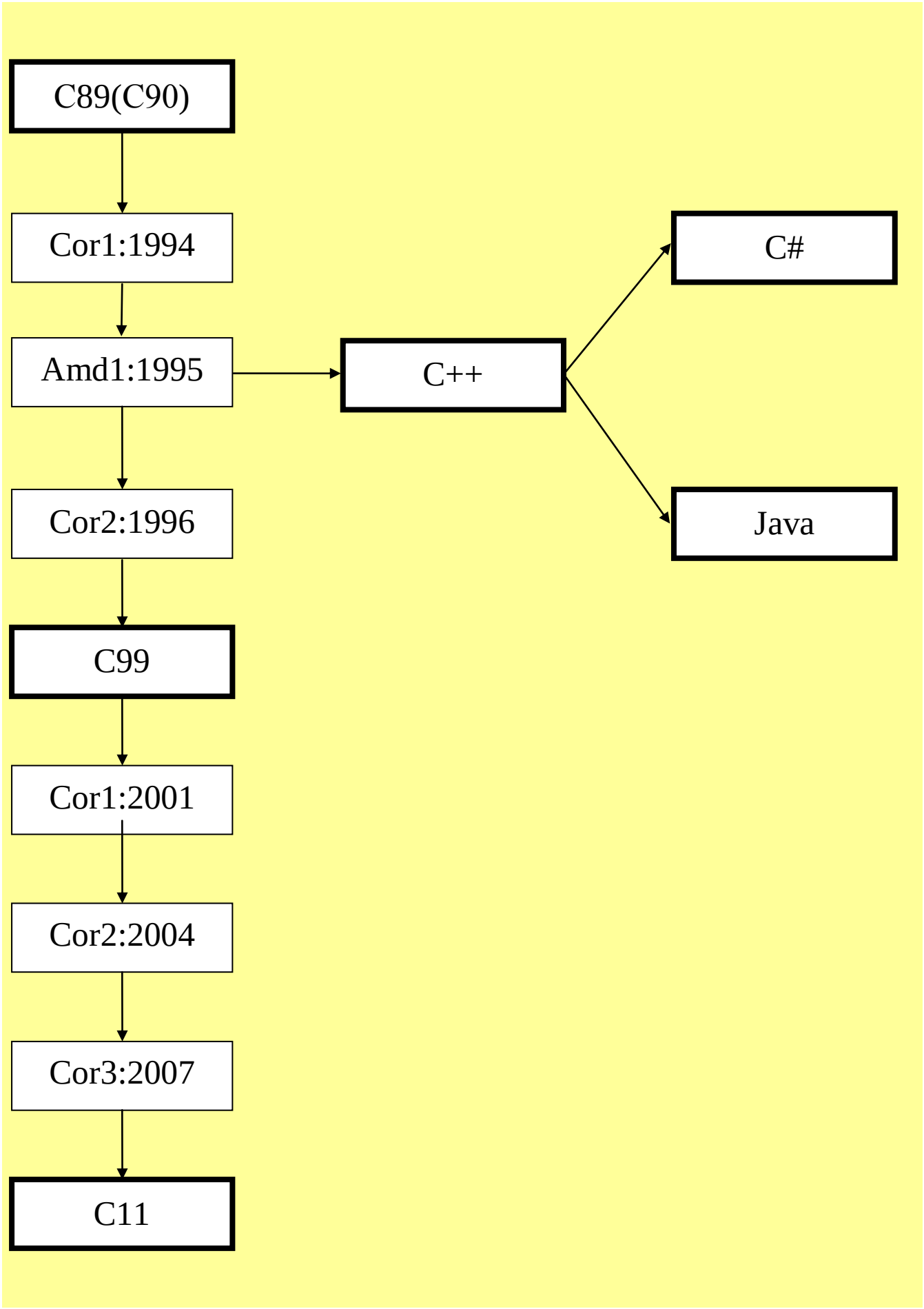
ISO/IEC 9899:1999/Cor 2:2004

ISO/IEC 9899:1999/Cor 3:2007

През 2011 г. е приет нов стандарт **ISO/IEC 9899:2011**, формално наричан C11.

Стандартът C95 (ISO/IEC 9899:1990/Amd 1:1995) е основа на езика C++, 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.

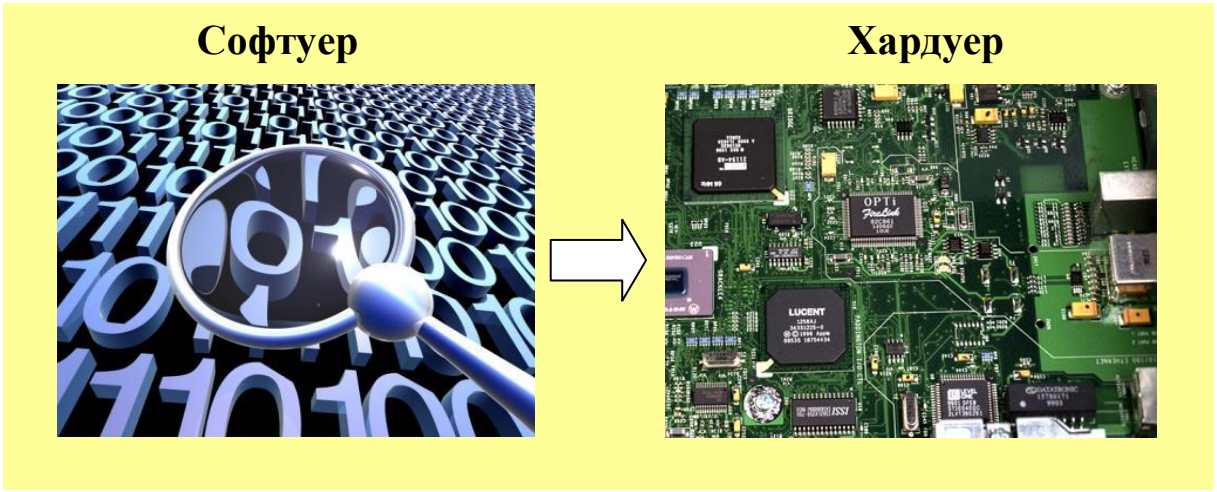


Фиг. 1 Стандартизиране на езика С

Част I: Компютри и програмиране

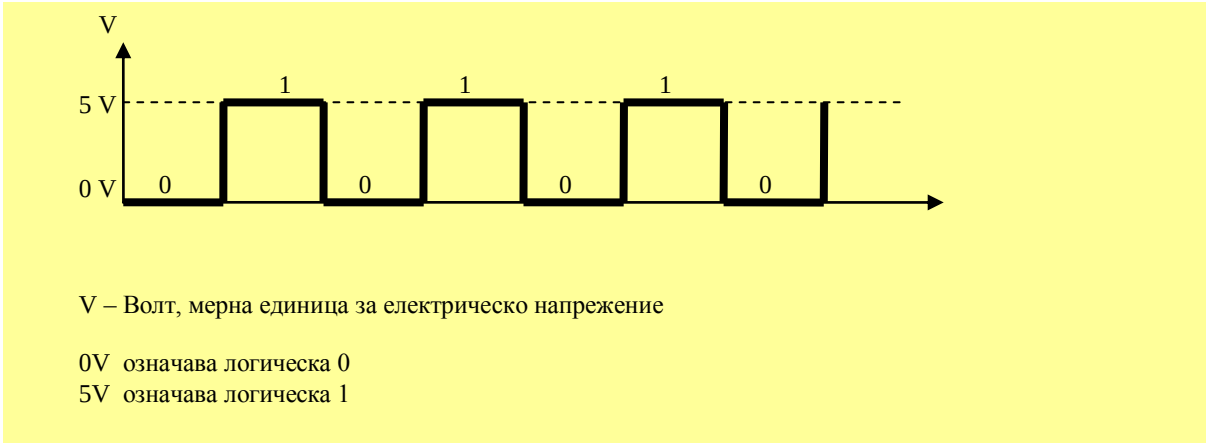
1 Въведение в компютрите

Най-общото определение за компютър е електронно устройство за въвеждане, обработка и извеждане на информация, състоящо се от два неразделни компонента: **хардуер** и **софтуер**. Хардуерът включва всички електронни елементи, от които е изграден компютърът. Софтуерът е програма, която "казва" на хардуера как да обработва информацията.



Фиг. 2 Главни компоненти на компютър

Съвременните компютри са цифрови. Независимо каква информация искаме да обработим, тя трябва да бъде подадена на компютъра в цифров (двоичен) вид. Цифровата информация е съвкупност от комбинации само от две стойности: нула (0) и единица (1). Причината информацията да се представя в цифров вид е, че тези две стойности се реализират от електронните компоненти много лесно само с помощта на два електрически сигнала: Например 0V и 5V (Фиг.3).



Фиг. 3 Представяне на стойностите 0 и 1 с помощта на електрически сигнали

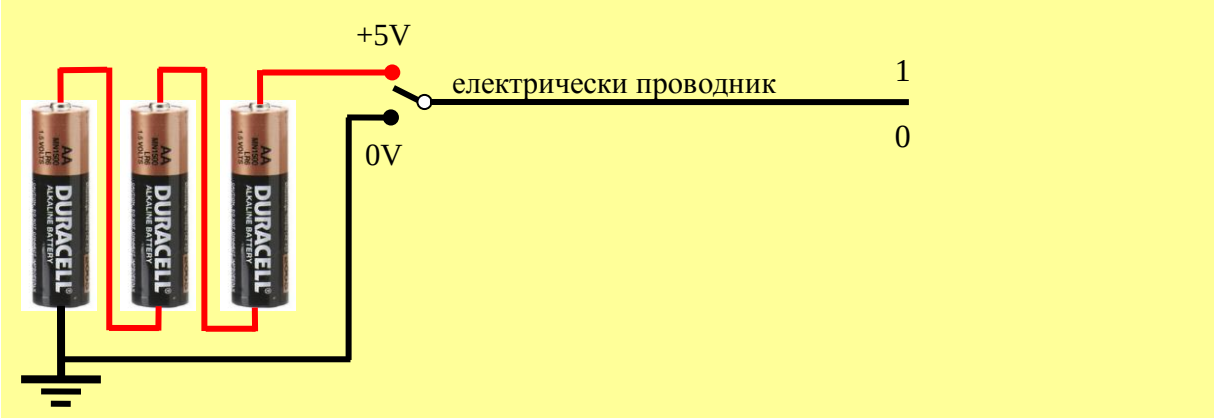
Тъй като стойностите 0 и 1 са абстрактни понятия, те се наричат още логическа 0 и логическа 1.

Най-малкото количество информация, която може да се предаде в цифров

вид е 0 или 1 и се нарича **бит**, т.е. . един бит информация може да приема само две стойности: 0 или 1.

Най-малкото количество информация, която може да се предаде в цифров вид, е 0 или 1 и се нарича бит.

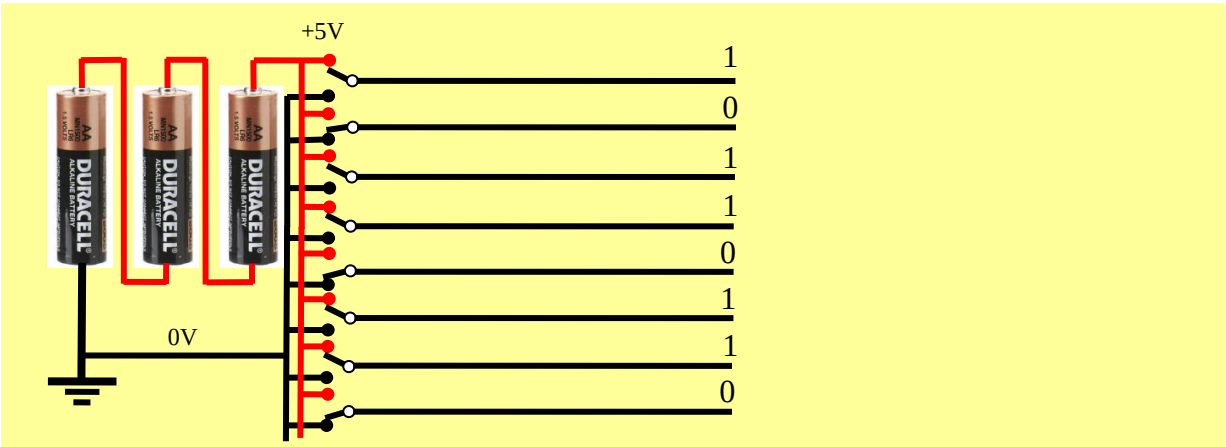
Фиг.4 показва как можем да предадем един бит информация по електрически проводник.



Фиг. 4 Предаване на бит по електрически проводник

Когато подадем на проводника напрежение 5V с помощта на ключето, това е еквивалентно предаване на лог.1 по него. Когато подадем на проводника 0V, това е еквивалентно предаване на лог.0.

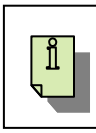
За да представим повече цифрова информация наведнъж, ще трябва да използваме няколко бита в паралел, респективно няколко проводника (Фиг.5).



Фиг. 5 Предаване на 8 бита информация по електрически проводници

Съвкупността от 8 бита се нарича байт. Байтът е фундаментална мерна единица за количество информация в цифровата електроника. По-долу са дадени мерни единици и означения, които ще срещате често.

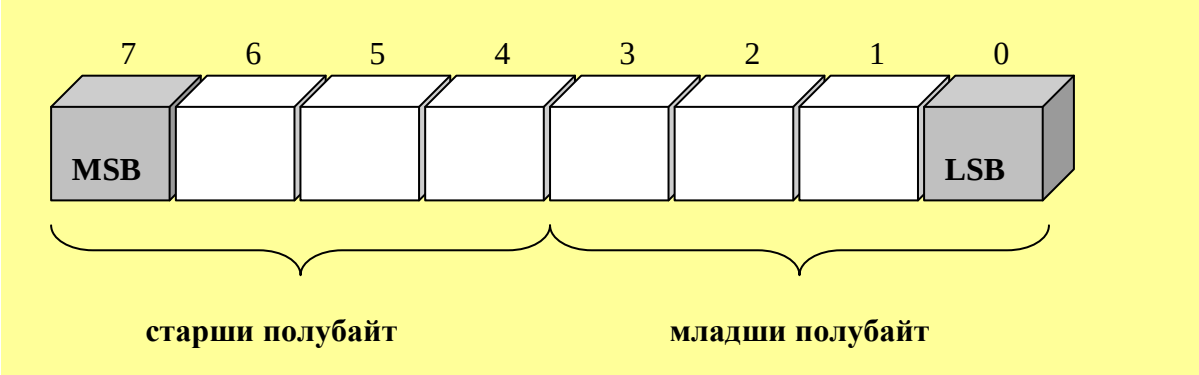
1 бит		[1bit]
1 байт	≈ 8 бита	[1B]
1 килобайт	≈ 1024 байта	[1kB]
1 мегабайт	≈ 1024 килобайта	[1MB]
1 гигабайт	≈ 1024 мегабайта	[1GB]
1 терабайт	≈ 1024 гигабайта	[1TB]



Съвкупност от 8 бита формира байт. Съвкупност от 4 бита формира полубайт.

Група проводници за предаване на битове в паралел формират шина. Броят на проводниците в шината определя колко битова е тя. Например шина с 8 проводника се нарича 8-битова шина. Вероятно сте чували термините 8-битов, 16-битов, 32-битов, а напоследък и 64-битов компютър. Това е точно големината на шината, по която данните (битовете) се предават.

Преди да продължим напред, ще Ви запозная с още няколко термина. На следващата фигура е показан един байт с всичките 8 бита в него.

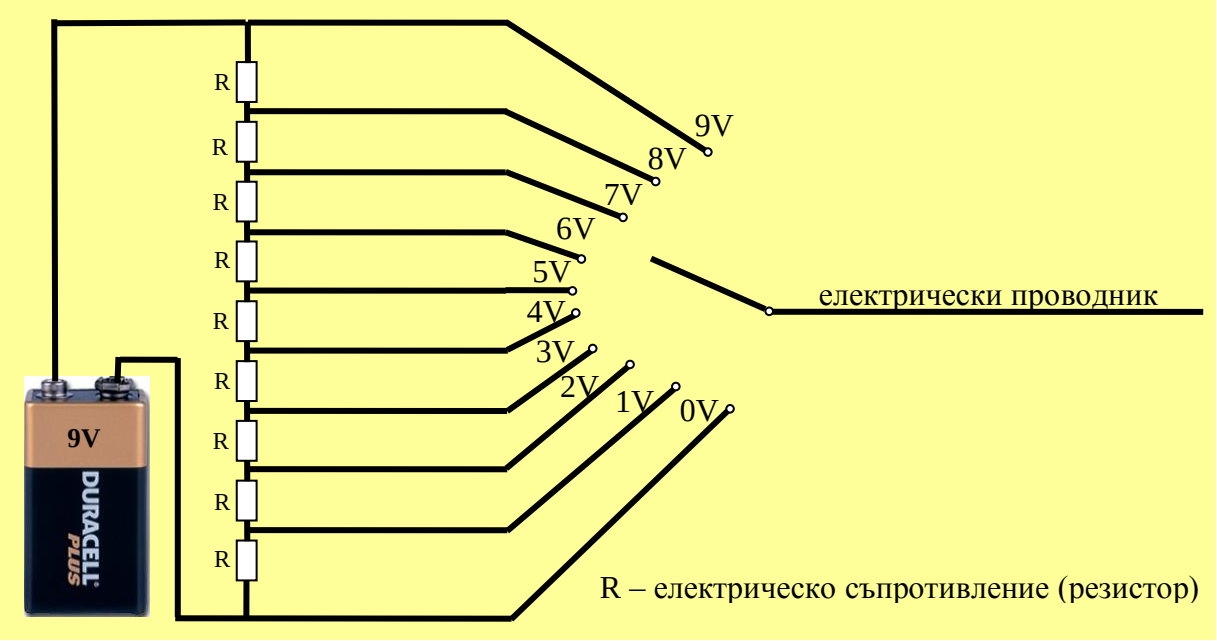


Фиг. 6 Битове в байт

Всеки бит си има номер. Най-десният бит има номер 0 и се нарича най-младши бит (**LSB –Least Significant Bit**) или само младши бит. Най-левият бит има номер 7 и се нарича най-старши бит (**MSB –Most Significant Bit**) или само старши бит. Тази номерация на битовете е много важна, тъй като информацията, която се предава в даден бит, се интерпретира по различен начин, ако се предава в друг. Например байт със стойност 00000001 означава различна информация от байт със стойност например 10000000. Съвкупността от 4 бита се нарича още полубайт. Младшите четири бита (0÷3) в байт се наричат още младши полубайт, а старшите четири бита (4÷7) – старши полубайт.

В следващата глава ще разберете как точно се интерпретира двоичната информация (двоичните числа), закодирана в един или повече бита.

За да разберете защо представянето на информацията с двоични числа е по-лесна и удобна за обработка от хардуера в сравнение с десетичните числа, с които сме свикнали да работим в ежедневието, нека да разгледаме схема аналогична на Фиг.4, която позволява да генерираме всички десетични цифри. За по-лесно ще използваме източник на напрежение 9V, с който можем да генерираме всичките 10 десетични цифри 0÷9 под формата на напрежения 0V÷9V.



Фиг. 7 Предаване на информация по електрически проводник в десетичен формат

В зависимост от позицията на ключа, по проводника могат да се предадат десет различни стойности под формата на десет различни напрежения.

Сега сравнете Фиг.7 с Фиг.4. Дори и да не разбирате от електротехника и електроника, би трябвало да може да оцените факта, че реализацията на Фиг.7 с помощта на електронни елементи би била доста по-сложна от тази на Фиг.4.

Въпроси за самопроверка

1. Кои са основните компоненти на един компютър и как си взаимодействат те?

2. Какъв е форматът на информацията, който съвременните компютри обработват?

3. От какви стойности се състои информацията, обработвана от съвременните компютри?

4. Кое е най-малкото количество информация в контекста на съвременните компютри?

5. Какво е шина в контекста на компютрите?

6. Какво е байт?

7. Ако имате информация от 10 байта, на колко бита се равнява тя?

8. Какво е MSB и LSB?

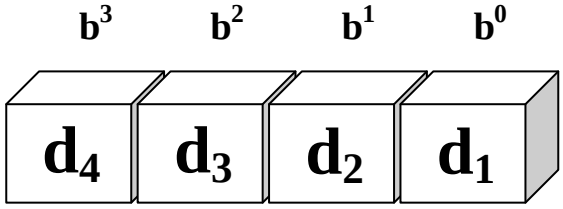
2 Бройни системи

2.1 Въведение

От предходната глава би трябвало вече да сте разбрали, че "езикът" на съвременните компютри се състои само от числа, които са комбинации от нули и единици. Тези числа се наричат двоични, тъй като се изграждат само с помощта на тези две цифри. Използването само на два символа (0 и 1) за представяне на числа се нарича **двоична бройна система**. Бройната система за представяне на числата в ежедневието на хората се нарича **десетична бройна система**. Както може би вече се досещате, тя се нарича така, защото използва 10 символа (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) за представяне на всякакви числа. Освен тези две бройни системи, в компютърната техника се използва широко още една бройна система – **шестнайсетична бройна система**. Тя използва 16 символа (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F).

Броят на символите, използвани от една бройна система, се нарича основа или база **b** (**radix** или **base**) на бройната система. Двоичната бройна система е с основа b = 2, десетичната – b = 10, и шестнайсетичната – b = 16.

Позицията на всяка цифра в едно число (важи и за трите бройни системи) има параметър, наричан тегло. Най-дясната цифра има тегло b^0, следващата цифра вляво от нея има тегло b^1 и т.н. Например, ако имаме едно четири цифрено число $d_4d_3d_2d_1$, отделните цифри имат тегло както е показано по-долу.



Изброените бройни системи се наричат още позиционни бройни системи, тъй като позицията, в която се намира даден символ определя неговата стойност. Стойността на даден символ в едно число може да се изчисли като се умножи символа по неговото тегло. Стойността на едно число може да бъде изразена в десетична бройна система чрез сумата на стойностите на отделните символи.

Пример:

$$d_4d_3d_2d_1 = d_1*b^0 + d_2*b^1 + d_3*b^2 + d_4*b^3$$

Различните бройни системи използват общи символи, което може да доведе до грешно тълкуване в коя бройна система е дадено число. Когато е възможна такава ситуация, след най-дясната цифра се поставя долен индекс, който указва в коя бройна система е числото.

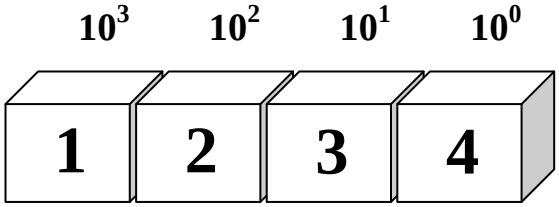
Пример:

1010_{10} - числото е в десетична бройна система;
1010_{2} - числото е в двоична бройна система;
1010_{16} - числото е в шестнайсетична бройна система.

2.2 Десетична бройна система

Десетичната бройна система използва 10 символа (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) за представяне на всякакви числа и има основа b = 10. Съответно тегловните коефициенти са 10^0, 10^1, 10^2 , 10^3 , 10^4 , 10^5 , 10^6...

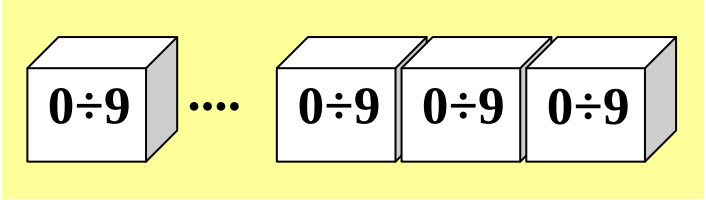
Пример:



Десетичното число 1234 може да се изрази и по следния начин:

$1234 = 4*10^0 + 3*10^1 + 2*10^2 + 1*10^3 = 4 + 30 + 200 + 1000 = 1234$

Въпреки че това е бройната система, която познаваме добре, ще разгледаме как става десетичното броене, тъй като това ще Ви помогне да разберете как се извършва броенето и в другите бройни системи. Всяка позиция в едно десетично число може да се променя от 0 до 9 (Фиг.8).



Фиг. 8 Стойности на десетичните позиции

Когато една позиция има стойност 9 и добавим 1 към нея, стойността в позицията се нулира, а следващата позиция се увеличава с единица. Табл.1 онагледява този процес.

0 + 1	90 + 1	990 + 1
1 + 1	91 + 1	991 + 1
2 + 1	92 + 1	992 + 1
3 + 1	93 + 1	993 + 1
4 + 1	94 + 1	994 + 1
5 + 1	95 + 1	995 + 1
6 + 1	96 + 1	996 + 1
7 + 1	97 + 1	997 + 1
8 + 1	98 + 1	998 + 1
9 + 1	99 + 1	999 + 1
10	100	1000
...	...	...

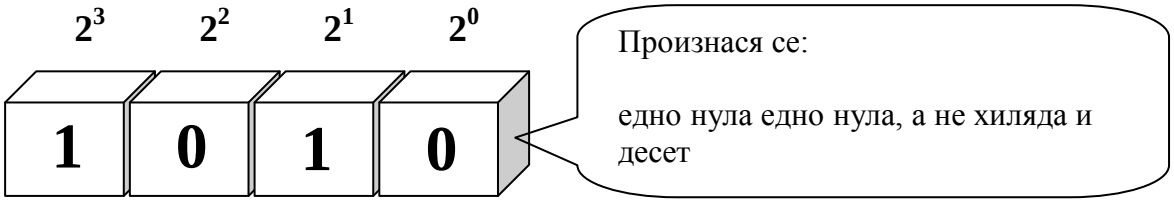
Табл. 1 Десетично броене

Максималният брой десетични числа, които могат да се формират от n-цифрено десетично число, е 10^n. Например с 3-цифрено десетично число могат да се представят $10^3 = 1000$ на брой числа (от 0 до 999).

2.3 Двоична бройна система

Двоичната бройна система използва два символа (0 и 1) за представяне на всякакви числа и има основа b = 2. Съответно тегловните коефициенти са 2^0, 2^1, 2^2 , 2^3 , 2^4 , 2^5 , 2^6...

Пример:



Двоичното число 1010 може да се изрази в десетична бройна система по следния начин:

$1010_2 = 0*2^0 + 1*2^1 + 0*2^2 + 1*2^3 = 0 + 2 + 0 + 8 = 10_{10}$

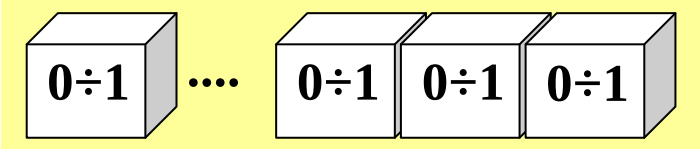
т.е. . двоичното число 1010 отговаря на десетичното число 10.

Табл.2 показва двоичните еквиваленти на десетичните цифри от 0 до 9.

десетична бройна система	двоична бройна система
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001

Табл. 2 Двоични еквиваленти на десетичните цифри от 0 до 9

Броенето в двоична бройна система е аналогично на десетичната. Всяка позиция в едно двоично число може да се променя от 0 до 1 (Фиг.9).



Фиг. 9 Стойности в двоичните позиции

Когато една позиция има стойност 1 и добавим 1 към нея, стойността в позицията се нулира, а следващата позиция се увеличава с единица. Табл.3 онагледява този процес.

0 + 1	1011 + 1
1 + 1	1100 + 1
10 + 1	1101 + 1
11 + 1	1110 + 1
100 + 1	1111 + 1
101 + 1	10000 + 1
110 + 1	10001 + 1
111 + 1	10010 + 1
1000 + 1	10011 + 1
1001 + 1	10100 + 1
1010 + 1	10101 + 1

Табл. 3 Двоично броене

Максималният брой двоични числа, които могат да се формират от n-цифрено двоично число, е 2^n. Например с 8-цифрено двоично число могат да се представят $2^8 = 256$ на брой двоични числа (от 0 до 255 десетично). Всяка двоична цифра носи 1 бит информация. Термините **n-цифрено двоично число** и **n-битово число** са еквивалентни.

Обърнете внимание, че едно десетично число може да бъде представено като двоично чрез различен брой битове. Например числото 4 може да се кодира с минимум 3 бита, т.е. . като 100 (вижте Табл.2). Същият ще е резултатът, ако числото се кодира с 4, 8, 16 или повече бита, т.е. .

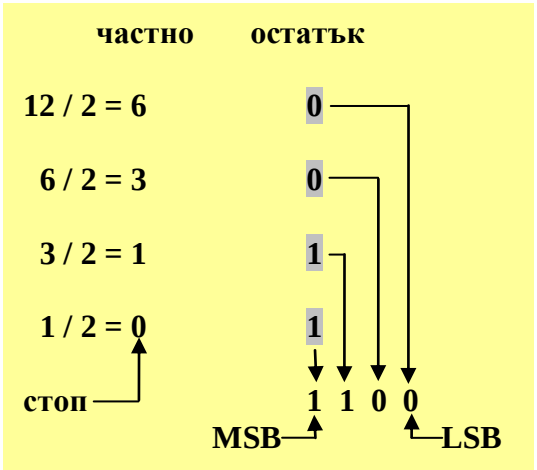
0100
00001000
00000000 00001000

Това лесно може да го проверите, ако се опитате да преобразувате всяко едно от горните двоични числа в техния десетичен еквивалент по познатия вече начин. Например:

$0100 = 0*2^0 + 0*2^1 + 1*2^2 + 0*2^3 = 0 + 0 + 4 + 0 = 4$

Всички битове вляво от последния ненулев бит в едно двоично число не променят неговата стойност.

Едно десетично число може да бъде преобразувано в двоично чрез последователно деление на 2. Например, за да преобразуваме десетичното число 12 в двоично, започваме с деление на 12 на 2. След това делим всяко получено частно на 2 докато частното стане 0. Остатъците, получени от всяко деление, формират двоичния еквивалент на десетичното число. Остатъкът, получен от първото деление, представлява младшия бит (LSB), а остатъкът, получен от последното деление, е старшият бит (MSB). Фиг.10 илюстрира този процес нагледно:



Фиг. 10 Преобразуване на десетично число в двоично

2.4 Шестнайсетична бройна система

Когато започнете да пишете С програми, ще се убедите, че използването на десетични числа в кода често е твърде неудобно. Тогава на помощ идва шестнайсетичната бройна система, която позволява големи числа да се запишат в по-компактен вид. Шестнайсетичната бройна система използва шестнайсет символа (0 ÷ 9 и A ÷ F или a ÷ f) за представяне на всякакви числа и има основа b = 16. Съответно тегловните коефициенти са 16^0, 16^1, 16^2 , 16^3 , 16^4 , 16^5 , 16^6...

Пример:

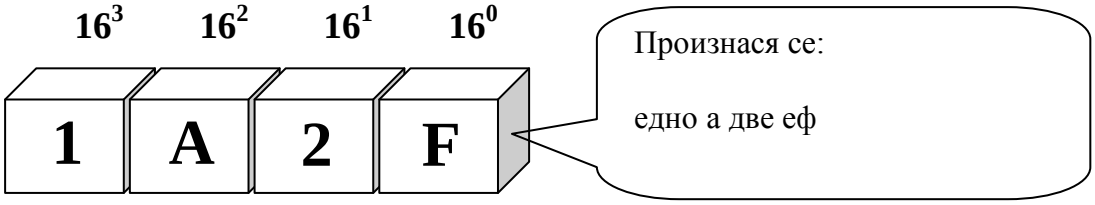


Табл.4 показва десетичните и двоичните еквиваленти на шестнайсетичните цифри 0 ÷ F.

шестнайсетична бройна система	десетична бройна система	двоична бройна система
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A (или a)	10	1010
B (или b)	11	1011
C (или c)	12	1100
D (или d)	13	1101
E (или e)	14	1110
F (или f)	15	1111

Табл. 4 Десетични и двоични еквиваленти на шестнайсетичните цифри от 0 ÷F

Шестнайсетичното число 1A2f може да се изрази в десетична бройна

система по следния начин:

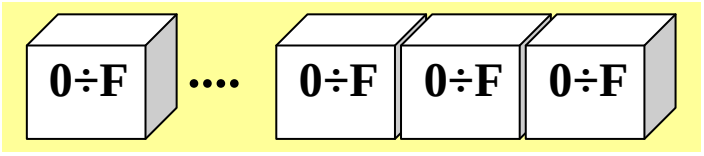
$1A2F = F*16^0 + 2*16^1 + A*16^2 + 1*16^3 = 15*16^0 + 2*16^1 + 10*16^2 + 1*16^3 = 6703_{10}$

т.е. . Шестнайсетичното число 1A2F е еквивалентно на десетичното число 6703.

Шестнайсетичните числа дават по-ясна представа за битовото представяне на числото, в сравнение с десетичните. Например, знаейки двоичните еквиваленти на шестнайсетичните цифри, лесно може да определите, че в числото 1A2F младшите четири бита са 1111, следващите четири бита са 0010, следващите четири бита са 1010 и старшите четири бита са 0001.

Шестнайсетичните числа често се записват с представката 0x или 0X. Например 1A2F е същото като **0x**1A2F или **0X**1A2F.

Броенето в шестнайсетична бройна система е аналогично на десетичната. Всяка позиция в едно шестнайсетично число може да се променя от 0 до F (или f) (Фиг.11).



Фиг. 11 Стойности в шестнайсетични позиции

Когато една позиция има стойност F и добавим 1 към нея, стойността в позицията се нулира, а следващата позиция се увеличава с единица. Табл.5 онагледява този процес.

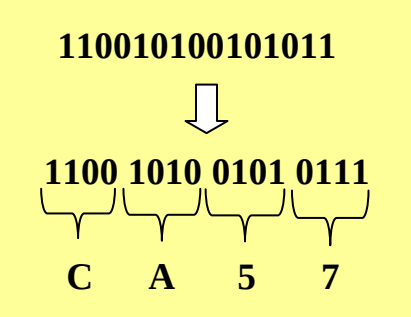
0 + 1	9 + 1	12 + 1	1B + 1
1 + 1	A + 1	13 + 1	1C + 1
2 + 1	B + 1	14 + 1	1D + 1
3 + 1	C + 1	15 + 1	1E + 1
4 + 1	D + 1	16 + 1	1F + 1
5 + 1	E + 1	17 + 1	20 + 1
6 + 1	F + 1	18 + 1	21 + 1
7 + 1	10 + 1	19 + 1	...
8 + 1	11 + 1	1A + 1	...

Табл. 5 Шестнайсетично броене

Използвайки информацията от Табл.4, лесно може да преобразувате едно

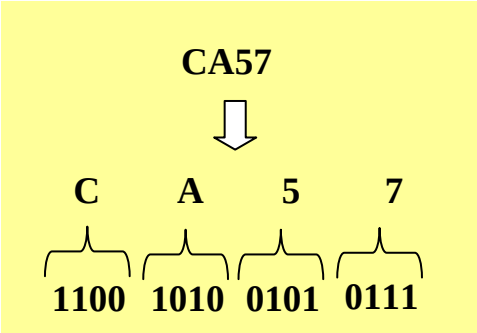
шестнайсетично число в двоично и обратно.

За да преобразувате едно двоично число в шестнайсетично, разделете битовете на двоичното число на групи по 4 бита, започвайки от младшия бит. След това преобразувайте всяка група в шестнайсетичния еквивалент. Фиг.12 илюстрира този процес нагледно.



Фиг. 12 Преобразуване на двоично число в шестнайсетично

За да преобразувате едно шестнайсетично число в двоично, преобразувайте всяка шестнайсетична цифра в 4-битово двоично число (Фиг.13).



Фиг. 13 Преобразуване на шестнайсетично число в двоично

2.5 Заключение

Ако се затруднявате да извършвате преобразувания от една бройна система в друга, може да използвате вградения в Windows калкулатор.

Преди да продължим напред нека да направим едно обобщение по отношение на употребата на трите бройни системи. Хардуерът на един компютър е реализиран така, че да използва двоична бройна система (вижте отново Фиг.3, 4 и 5). Двоичната система обаче в повечето случаи е неудобна за употреба от човек, особено при големи числа. Затова когато се пишат програми, се използват десетични и шестнайсетични числа, които в крайна сметка се преобразуват в двоични преди програмата да се зареди в компютъра за изпълнение.

Следващата таблица показва десетичните числа от 1 до 15 и техните двоични и шестнайсетични еквиваленти. Научете я наизуст.

десетична бройна система	двоична бройна система	шестнайсетична бройна система
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A (или a)
11	1011	B (или b)
12	1100	C (или c)
13	1101	D (или d)
14	1110	E (или e)
15	1111	F (или f)

Табл. 6 Сравнителна таблица на десетични, двоични и шестнайсетични числа

Въпроси за самопроверка

1. Как се наричат трите бройни системи, които се използват в компютърната техника?

2. Какво представлява основата (базата) на една бройна система?

3. Преобразувайте десетичното число 200 в двоична и шестнайсетична бройна система.

4. Преобразувайте двоичното число 10101111 в десетична и шестнайсетична бройна система.

5. Преобразувайте шестнайсетичното число FC59 в десетична и двоична бройна система.

3 Представяне на знакови цели числа в двоична бройна система

3.1 Въведение

В точка 2.3 разгледахме представяне само на беззнакови цели числа в двоична бройна система. Двоичната бройна система може да се използва и за представяне на знакови числа (отрицателни и положителни). Съществуват различни формати за представяне на знакови числа в двоична бройна система. По известните сред тях са прав код (**signed-bit magnitude**), обратен код (**1's complement**) и допълнителен код (**2's complement**).

3.2 Представяне на знакови цели числа в прав код

При правия код положителните и отрицателните числа се представят по същия начин както беззнаковите. Най-старшият бит се заделя за знаков бит. Стойност 0 в този бит указва, че числото е положително, а стойност 1 – отрицателно. Останалите битове се използват за кодиране на големината на числото. Недостатък на правия код е, че нулата има двойно представяне: +0 и -0.

Табл.7 показва представянето на 8-битови знакови числа в прав код.

+127	01111111
+126	01111110
+125	01111101
...	...
+2	00000010
+1	00000001
+0	00000000
-0	10000000
-1	10000001
-2	10000010
...	...
-125	11111101
-126	11111110
-127	11111111

Табл. 7 Представяне на 8-битови знакови числа в прав код

3.3 Представяне на знакови цели числа в обратен код

При обратния код положителните числа се представят по същия начин както беззнаковите. Отрицателните числа се представят чрез инвертиране на положителния еквивалент на числото (всички нули се заменят с 1, а единиците с 0). Например +9 се представя като 00001001, а -9 като 11110110. Както и при правия код, обратният код има недостатък, че нулата има двойно представяне: +0 и -0. Табл.8 показва представянето на 8-битови знакови числа в обратен код.

+127	01111111
+126	01111110
+125	01111101
...	...
+2	00000010
+1	00000001
+0	00000000
-0	11111111
-1	11111110
-2	11111101
...	...
-125	10000010
-126	10000001
-127	10000000

Табл. 8 Представяне на 8-битови знакови числа в обратен код

3.4 Представяне на знакови цели числа в допълнителен код

При допълнителния код положителните числа се представят по същия начин както беззнаковите. Отрицателните числа се представят като положителният еквивалент на числото се преобразува в обратен код и към резултата се добави 1. Например +9 се представя като 00001001, а -9 се представя като:

11110110 (обратен код на +9)
+
00000001

_____
11110111 (допълнителен код на -9)

Допълнителният код има следните предимства по отношение на правия и обратния код:

- нулата има само едно представяне (00000000)

- операцията изваждане може да се извърши с помощта на операцията събиране[1], т.е. . една и съща апаратна част на процесора може да се използва за извършване на операциите изваждане и събиране.

Забележка[1]: Ако не разбирате това, не се притеснявайте. Компютърът извършва това скрито от Вас. За експеримент разгледайте следния пример:

7 – 3 = 7 + (-3) = 4

0111 (двоично представяне на 7)
1101 (-3 в допълнителен код)
0100 (двоично представяне на 4)

```
1 1 1 1 ←  пренос
  0 1 1 1 (7)
+
  1 1 0 1 (-3 в допълнителен код)
 -----------
1 0 1 0 0 (4)
↖ преносът от най-старшите битове се игнорира
```

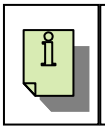
Както сами виждате, извършва се операция събиране, а резултатът е същият като изваждане на 7 – 3 (преносът, получен при сумирането на най-старшите битове, се изхвърля).

Табл.9 показва представянето на 8-битови знакови числа в допълнителен код.

+127	01111111
+126	01111110
+125	01111101
...	...
+2	00000010
+1	00000001
0	00000000
-1	11111111
-2	11111110
...	...
-125	10000011
-126	10000010
-127	10000001
-128	10000000

Табл. 9 Представяне на 8-битови знакови числа в допълнителен код.

Както се вижда от Табл.9, допълнителният код позволява в едно 8-битово число да се кодира пълния брой от 256 (2^8) възможни числа със знак (-128 ÷ +127).



Допълнителният код е най-предпочитаният код за представяне на отрицателни цели числа.

Въпроси за самопроверка

1. Представете следните числа в допълнителен код.

   0, -1, -120, -127

4 Представяне на десетични дробни числа в двоична бройна система

4.1 Въведение

Освен цели десетични числа, двоичната бройна система може да представя и дробни десетични числа. Двоичните дробни числа могат да се представят с фиксирана запетая[1] (**fixed-point**) и с плаваща запетая (**floating-point**).

Забележка[1]: Въпреки че използваме термина "запетая", за разделителен символ между цялата и дробната част в англоезичната литература се използва символа точка.

Езикът C използва плаваща запетая за представяне на дробни числа. Има различни стандарти с плаваща запетая. За да добиете представа какво се има предвид под "плаваща запетая", вземете десетичните числа **6.0247* 10^{23}**, **3 .7291* 10^{-27}**, **−1.0341* 10^{2}** и т.н. Ние казваме, че всички тези числа имат 5 значещи цифри на точност. Скалиращият фактор 10^{23}, 10^{-27} и т.н. определя действителната позиция на десетичната запетая по отношение на значещите цифри, т.е. . запетаята „плава“ в зависимост от скалиращия фактор.

Езикът C не дефинира конкретен стандарт, но най-широко използван е индустриалният стандарт **IEEE 754-1985** и неговият наследник **IEEE 754-2008** . Следващата точка разглежда как този стандарт представя значещите цифри и скалиращия фактор на едно десетично число в двоична бройна система, без да навлиза в големи детайли.

4.2 Стандарт IEEE 754 за представяне двоични числа с плаваща запетая

Стандартът IEEE 754 предоставя два формата:

- единична точност (**single precision**)
- двойна точност (**double precision**)

Двоичните дробни числа се представят в следната форма:

$\mathbf{V = (-1)^{S} *1.M*2^{E'-127}}$ - единична точност

$\mathbf{V = (-1)^{S} *1.M*2^{E'-1023}}$ - двойна точност

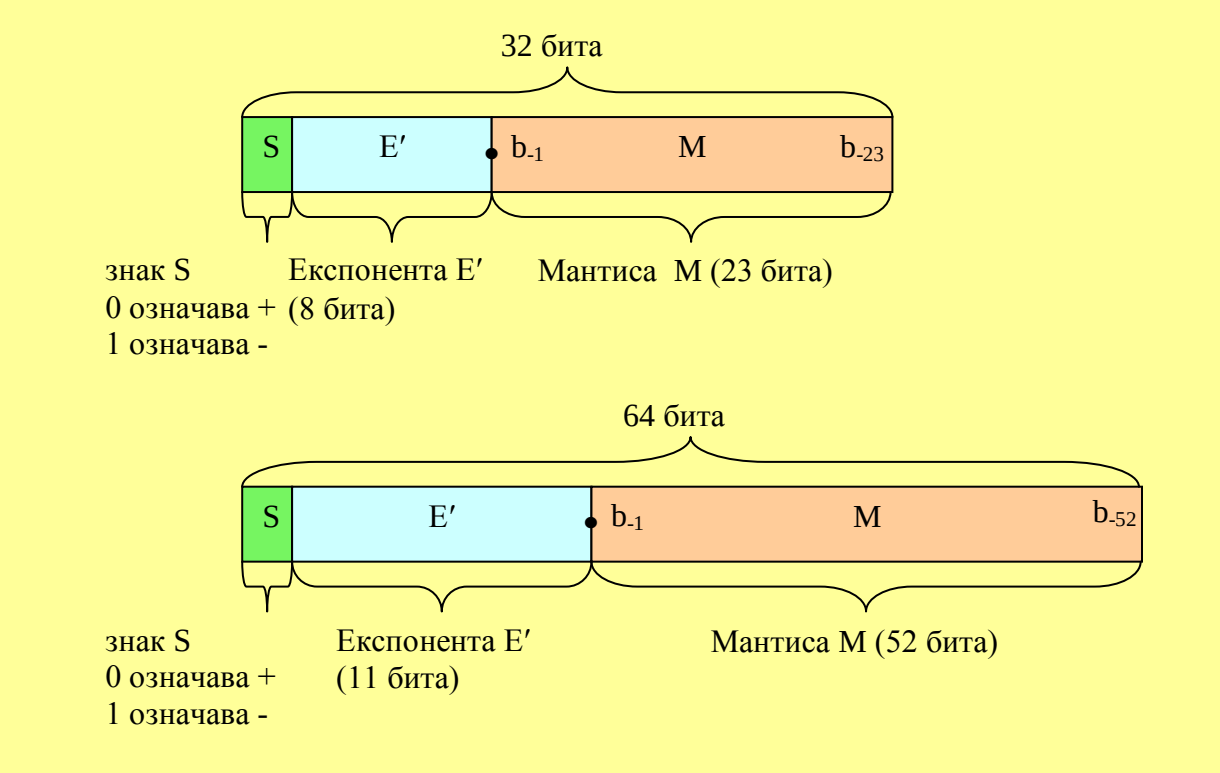
където

s – знак, определя дали числото е положително (s = 0) или отрицателно (s = 1).

M – мантиса, съдържа битове на дробната част на числото.

E = E′ -127 (1023) – експонента, която определя действителната позиция на двоичната запетая.

Фиг.14 илюстрира битовото представяне на числата с плаваща запетая с единична и двойна точност.



Фиг. 14 Формати с плаваща запетая в IEEE 754

Представянето на двоичните значещи цифри във формата 1.M = 1. b_{-1} b_{-2}... b_{-23} (или b_{-52}) се нарича нормализирана форма. Стойността на значещите цифри се изчислява като:

$$\mathbf{V(B) = 1 + b_{-1}*2^{-1} + b_{-2}*2^{-2} + , \ldots ,+ b_{-23}*2^{-23} (b_{-52}*2^{-52})}$$

Водещата единица и двоичната запетая след нея съществуват винаги и затова не участват в действителното битово представяне, но се подразбират. Скалиращият фактор 2^E, където E = E′ - 127 (единична точност) или E = E′ - 1023 (двойна точност), определя действителната позиция на двоичната запетая, а от там и действителната стойност на числото.

Следните разсъждения се отнасят за формата с единична точност, но могат да се приложат и за формата с двойна точност:

Експонентата E′ може да се изменя от 0 до 255. Стойностите 0 и 255 се използват за представяне на специални ситуации, като +∞, -∞ и други. Следователно за представяне на нормални стойности E′ се изменя от 1 до 254. Това означава, че действителната експонента E се изменя от -126 до +127. Скалиращият фактор има обхват от 2^{-126} до 2^{+127}, който съответства на $10^{-38} \div 10^{+38}$ в десетична бройна система.

Табл.10 илюстрира по-важните характеристики на форматите с единична и двойна точност.

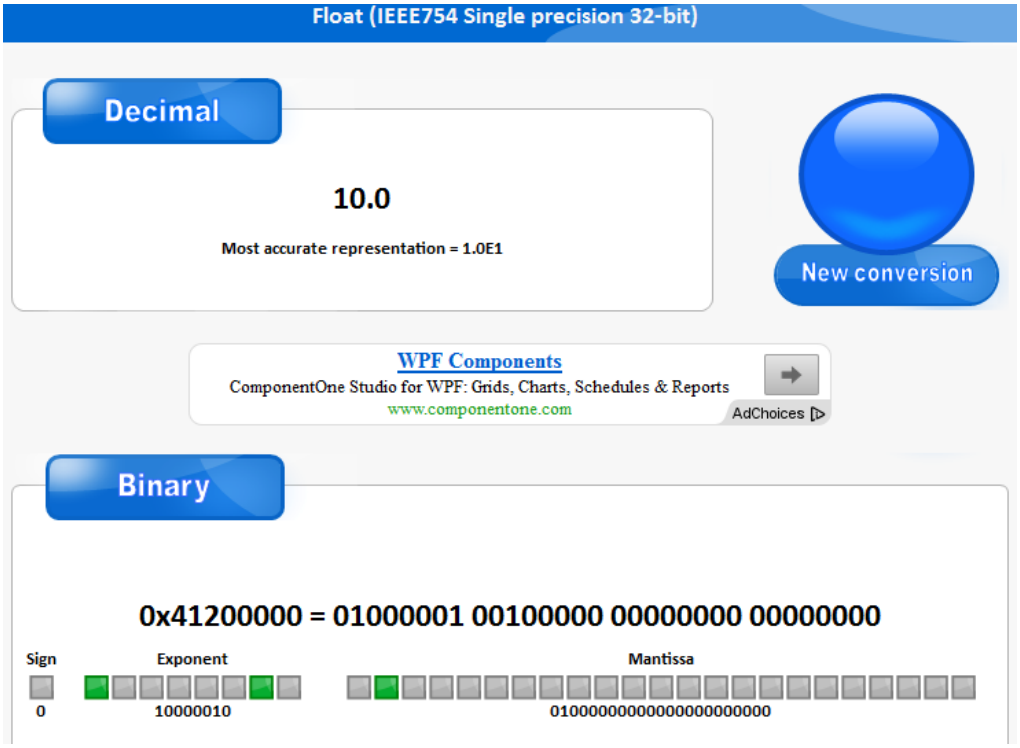
формат	широчина	обхват	точност
единична точност	32 бита	$\pm 1.18*10^{-38} \div \pm 3.4*10^{38}$	7 значещи десетични цифри
двойна точност	64 бита	$\pm 2.23*10^{-308} \div \pm 1.80*10^{308}$	15 значещи десетични цифри

Табл. 10 Основни характеристики на форматите в IEEE 754

Може да използвате онлайн конвертора от следния линк:

http://www.binaryconvert.com/convert_float.html?decimal=049048046048

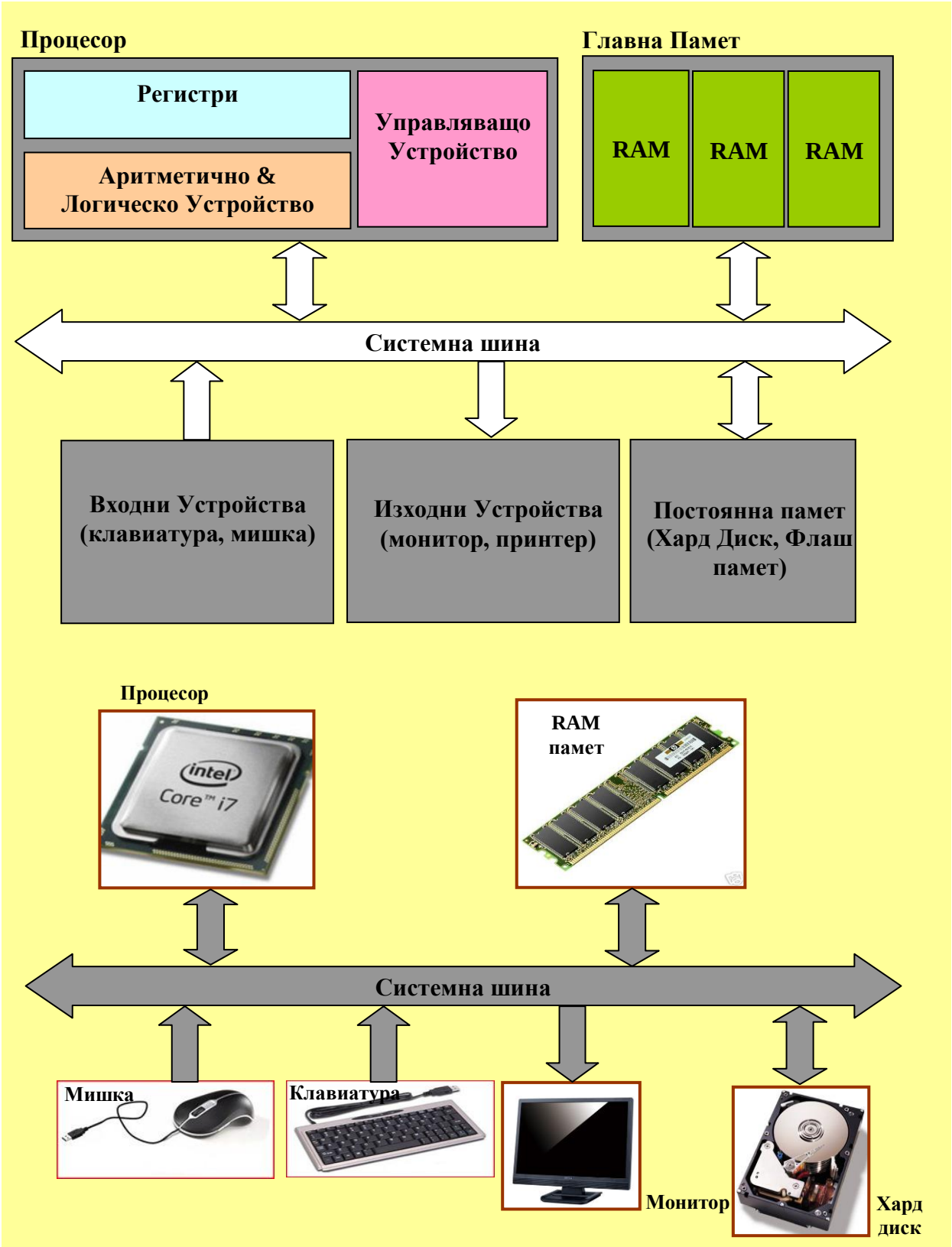
и да извършвате различни преобразувания (Фиг.15).



Фиг. 15 Онлайн конвертор

5 Архитектура на компютъра

Фиг.16 показва обобщена архитектура на компютърна система и физическата реализация на основните компоненти на персонален компютър.



Фиг. 16 Компютърна система

Компютърната система се състои от следните основни компоненти:

- Процесор (**CPU**);
- Главна памет (**RAM**);
- Постоянна памет (**ROM**);
- Входни устройства;
- Изходни устройства.

Всички тези компоненти се свързват помежду си посредством системна шина. Системната шина се състои от три подшини:

- Адресна шина

  Информацията, предавана по тази шина, се явява адрес. Този адрес определя източника или приемника на данните, предавани по шината за данни. Всички устройства, с които процесорът може да обменя информация, си имат адрес.

- Шина за данни

  Информацията, предавана по тази шина, се явява полезните данни, които компютърът ще обработва по някакъв начин. Типично шината за данни е широка 8, 16, 32 или 64 бита и определя разредеността на компютъра (дали компютърът е 8-, 16-, 32- или 64-битов). Количеството данни, които един процесор може да обработва наведнъж, се нарича още машинна дума или само дума.

- Управляваща шина

  Информацията, предавана по тази шина, има управляващ характер. Например дали процесорът ще чете данни от дадено устройство, или ще изпраща данни към него.

Централният процесор (**CPU – Central Processing Unit**) или просто процесор е "мозъкът" на една компютърна система. Той се състои от Аритметично & Логическо устройство (**АЛУ**), Регистри и Управляващо Устройство (**УУ**). Процесорът е отговорен за обработката на данни. Как ще се обработват данните, зависи от програмата, която му е зададена да изпълнява. От гледна точка на процесора, програмата е набор от инструкции (команди), които му указват какво да прави. Типичните инструкции, които един процесор изпълнява, са събиране и изваждане на числа, сравняване на числа, по-сложните процесори извършват умножение

и деление на числа и много други. УУ отговаря за извличането на инструкциите на програмата и декодирането им. АЛУ, както подсказва неговото име, извършва аритметични и логически операции (като събиране, изваждане, умножение, преместване наляво или надясно и др.) върху данните. Регистрите на процесора са специални клетки памет, до които достъпът е най-бърз. Част от тези регистри имат строго специализирано предназначение, а друга част могат да се използват за временно съхранение на данни и се наричат регистри с общо предназначение. Типично регистрите с общо предназначение са широки колкото шината за данни, т.е. . ако шината за данни е 32 бита, то и тези регистри са 32-битови. Данните, върху които АЛУ извършва някаква операция, се наричат най-общо операнди.

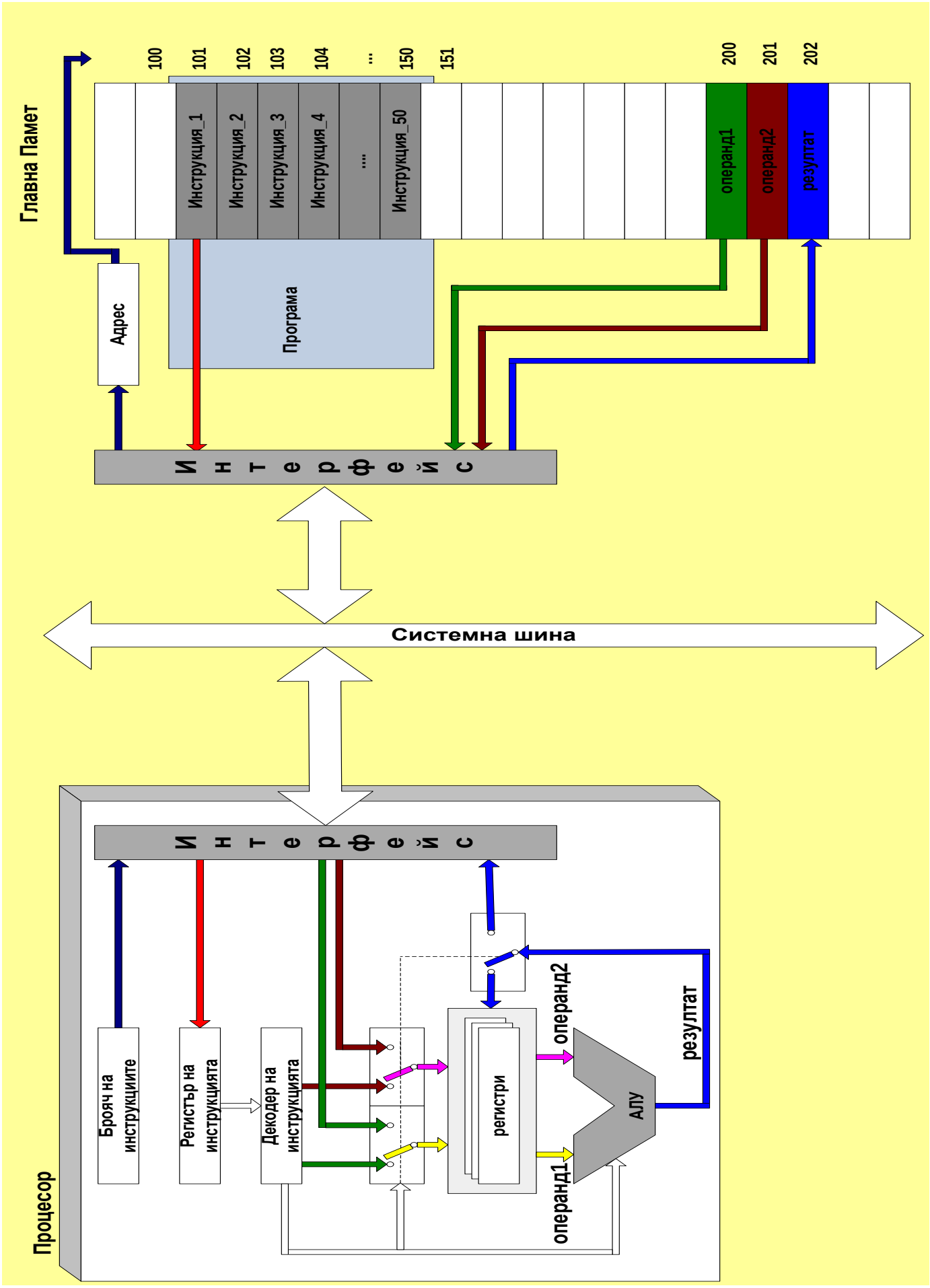
Главната памет (**RAM – Random Access Memory**), наричана още оперативна памет, се използва като "работна площ", в която се зарежда програмата при нейното стартиране, също и за временно съхранение на данни и резултати, използвани от програмата. Тази памет може да се разглежда като съставена от отделни клетки и всяка клетка си има числов адрес, чрез който може да се достига. Типично клетките са широки 1 байт и са организирани в машинни думи. Например в 32-битов компютър, машинната дума се състои от 4 клетки памет (4x8 = 32 бита). При изключване на захранването на компютъра цялата информация, намираща се в Главната Памет, се губи.

Входните Устройства (мишка, клавиатура и др.) се използват за въвеждане на данни в една програма по време на нейното изпълнение. Тези данни се запазват в паметта. По този начин процесорът може да извършва различни операции с тях.

Изходните Устройства (монитор, принтер и др.) се използват за извеждане на резултати от изпълнението на една програма или на данни от паметта.

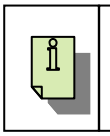
Постоянната Памет **(ROM – Read Only Memory**) се използва за съхранение на програмите. Съдържанието на тази памет не зависи от захранването на компютъра. При стартиране на една програма, тя се зарежда от Постоянната Памет в Главната Памет, откъдето започва изпълнението. Постоянната Памет може също да се разглежда и като Входно-Изходно устройство, тъй като освен за съхранение на програми, тази памет може да се използва за съхранение на данни, използвани от програмите, както и за съхранение на резултати, получени от изпълнението им.

Фиг.17 показва подробна блокова схема на процесора и взаимодействието между неговите компоненти. Обобщено работата на процесора може да се опише по следния начин:



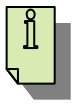
Фиг. 17 Компоненти на процесор

Когато стартирате една програма за изпълнение, в **Брояча на инструкциите** се зарежда адреса в паметта, където се намира първата инструкция на програмата. Инструкцията се чете от паметта и се копира в **Регистъра на Инструкциите**, а в **Брояча на инструкциите** автоматично се зарежда адреса на следващата за изпълнение инструкция. Извлечената инструкция се декодира от **Декодера на Инструкциите**, който в резултат на това "казва" на АЛУ каква операция трябва да извърши. Данните, върху които ще се извършва операцията, се зареждат в регистрите на процесора. В зависимост от инструкцията данните могат да се съдържат в самата инструкция или да се извлекат от Главната Памет. Също така в зависимост от инструкцията резултатът от операцията може да се съхрани отново в някой от регистрите на процесора или в Главната Памет. Накратко казано, изпълнението на една програма представлява извличане на инструкциите, от които тя се състои, тяхното декодиране и изпълнението им.



Изпълнението на една програма представлява извличане на инструкциите, от които тя се състои, тяхното декодиране и изпълнението им.

Инструкциите, от които един процесор "разбира", се наричат машинни инструкции. Една машинна инструкция е комбинация от единици (1) и нули (0). Различните процесори имат различна архитектура (например различен брой регистри) и различен набор от инструкции. Например Intel процесорите имат свой набор от инструкции, който се различава от набора инструкции на Motorola процесорите. Поредицата от инструкции в една програма се създава от програмисти. Директното използване на машинни инструкции от програмистите при създаване на програма, както сами може да се досетите, е много труден и бавен процес. За улеснение на програмистите наборът от машинни инструкции на един процесор се заменя с псевдо инструкции, които се състоят от по-лесни за разбиране и запомняне от човек символи. Псевдо инструкциите се наричат още асемблерни инструкции. Те се преобразуват в машинни инструкции чрез специална програма, наречена Асемблер.



Процесорът изпълнява машинни инструкции, които са просто комбинация от нули и единици. Писането на програми (поредица от инструкции) чрез директно използване на подобни инструкции е изключително труден и бавен процес. Затова всяка машинна инструкция се заменя в програмата с асемблерна инструкция, чийто синтаксис е по-лесен за запомняне и употреба от човек. Асемблерната програма се обработва от специална програма наречена Асемблер, която преобразува асемблерните инструкции в съответните машинни инструкции.

За да онагледим описаното по-горе, нека допуснем, че имаме един хипотетичен 16-битов процесор, който има 16 регистъра за обща употреба, примерно с имена REG0, REG1, … , REG15 (всеки от тях също е 16-битов). Фиг.18, 19 и 20 показват формата на няколко хипотетични машинни инструкции на този процесор и съответните им асемблерни инструкции.

Първо е дадено описанието на асемблерната инструкция, а след това на машинната инструкция, до която ще се преобразува от Асемблера. Асемблерните инструкции се състоят от абревиатури на английски думи, които подсказват какво прави тази инструкция. В скоби след името на инструкцията е дадено и пълното име на английски.

LDR (Load Register)

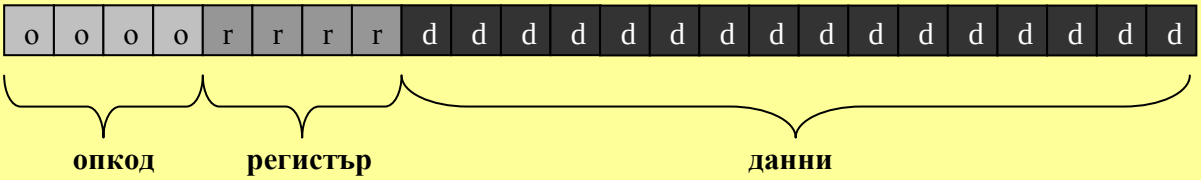
Синтаксис на асемблерната инструкция :

LDR Rn , 16-битова стойност

Описание на асемблерната инструкция:

16-битовата стойност, указана в инструкцията, се записва в регистър Rn (n = 0÷15).

Формат на машинната инструкция:



опкод (код на операцията):

Част от инструкцията, която определя вида на извършваната операция.

oooo	операция
0000	запис на стойност в регистър

регистър:

Определя кой от 16-те регистъра се използва в операцията.

rrrr	регистър
0000	REG0
0001	REG1
…	…
1111	REG15

данни:

Съдържа 16-битова стойност, която се зарежда в избрания регистър.

dddddddddddddddd
0000000000000000
0000000000000001
…
1111111111111111

Пример:

```
LDR     REG0 , 1A2C     ; Шестнайсетичното число 1A2C се записва в REG0
```

Фиг. 18 Описание на инструкция LDR

MOVR (Move Register in RAM)

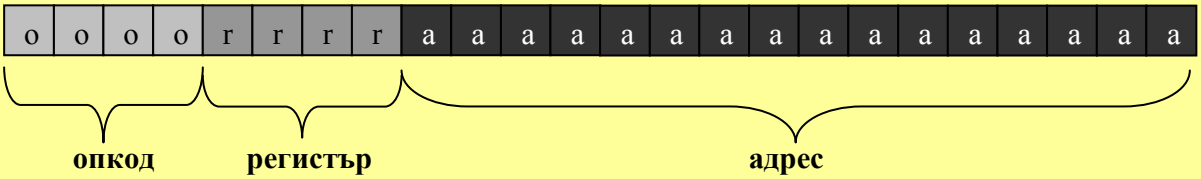
Синтаксис:

MOVR Rn, @адрес

Описание на инструкцията:

Стойността на регистър Rn (n = 0÷15) се записва на посочения адрес в RAM паметта.

Формат на инструкцията:



опкод (код на операцията):

Част от инструкцията, която определя вида на извършваната операция.

oooo	**операция**
0001	запис на регистър в RAM паметта

регистър:

Определя кой от 16-те регистъра се използва в операцията.

rrrr	**регистър**
0000	REG0
0001	REG1
…	…
1111	REG15

адрес:

Съдържа 16-битов адрес, на който се записва стойността на избрания регистър.

aaaaaaaaaaaaaaaa
0000000000000000
0000000000000001
…
1111111111111111

Пример:

MOVR REG0, @1A1A ; Стойността на REG0 се записва на адрес 1A1A в RAM

Фиг. 19 Описание на инструкция MOVR

ADDR (Add Registers)
SUBR (Subtract Registers)

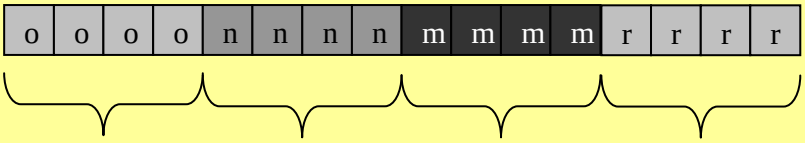
Синтаксис:

ADDR Rn, Rm, Rr
SUBR Rn, Rm, Rr

Описание на инструкцията:

Сумира/Изважда регистър Rm (m = 0 ÷15) с/от регисър Rn (n = 0 ÷15) и съхранява резултата в регистър Rr (r = 0 ÷15), т.е. Rr = Rn – Rm.

Формат на инструкцията:



опкод **регистър Rn** **регистър Rm** **регистър Rr**

опкод (код на операцията):

Част от инструкцията, която определя вида на извършваната операция

oooo	**операция**
0010	събиране на регистри
0011	изваждане на регистри

регистър Rn, Rm, Rr:

Определя кой от 16-те регистъра се използват в операцията.

nnnn/mmmm/rrrr	**регистър**
0000	REG0
0001	REG1
…	…
1111	REG15

Пример:

```
ADDR    REG0, REG1, REG2        ; REG2 = REG0 + REG1
SUBR    REG0, REG1, REG2        ; REG2 = REG0 - REG1
```

Фиг. 20 Описание на инструкция ADDR/SUBRR

Нека сега да напишем една съвсем кратка програма, която събира и изважда две числа и съхранява резултата някъде в паметта. Първо ще напишем програмата на машинен език (с машинни инструкции), а след това на асемблерен език (с асемблерни инструкции) и ще сравним двете програми. В лявата колона са разположени инструкциите, а вдясно е дадено описание, какво прави съответната инструкция.

машинни инструкции	описание
00000001000000000010100	Запиши 20_{10} в REG0
00000000000000000001010	Запиши 10_{10} в REG1
0010000000010010	REG2 = REG0 + REG1
00010010000110100011010	Запиши стойността на REG2 на адрес 1A1A
0011000000010010	REG2 = REG0 - REG1
00010010000110100011100	Запиши стойността на REG2 на адрес 1A1C

Фиг. 21 Примерна програма на машинен език

асемблерни инструкции	описание
LDR REG0, 20	Запиши 20_{10} в REG0
LDR REG1, 10	Запиши 10_{10} в REG1
ADD REG0, REG1, REG2	REG2 = REG0 + REG1
MOVR REG2, @1A1A	Запиши стойността на REG2 на адрес 1A1A
SUB REG0, REG1, REG2	REG2 = REG0 - REG1
MOVR REG2, @1A1C	Запиши стойността на REG2 на адрес 1A1C

Фиг. 22 Примерна програма на асемблерен език

От тези примери, макар и хипотетични, сами може да прецените, че асемблерните инструкции са по-лесни за запомняне и използване, отколкото съответните машинни инструкции. Това позволява програмистите да създават по-големи и по-сложни програми. За съжаление, няма универсален асемблерен език. Както споменах по-рано, всеки процесор има свой набор от асемблерни инструкции и съответно своя програма Асемблер. Това означава, че ако искате да прехвърлите програма, писана на асемблерен език за един процесор - върху друг, ще трябва да я пренапишете наново, като използвате асемблерните инструкции на втория процесор. Това е и един от големите недостатъци на асемблерния език. Освен това, писането на асемблерен език изисква много добро познаване на архитектурата на компютъра (регистри на процесора, как е организирана паметта и т.н.). Също така, въпреки че асемблерните инструкции са по-разбираеми в сравнение с машинните, научаването и помненето им също е голямо предизвикателство. Тези недостатъци дават тласък в търсенето на други методи за създаване на по-големи и по-сложни програми. Така се появяват езици от по-високо ниво какъвто е езикът C.

Езикът С предоставя едно абстрактно ниво на програмиране, което не зависи (или поне в по-голямата си част) от процесора, за който се пише програмата. Например операциите събиране и изваждане в езика С винаги изглеждат така:

операнд1 + операнд2
операнд1 - операнд2

независимо от използвания процесор.

Програма, наречена Компилатор, преобразува С кода в асемблерни инструкции за съответния процесор, които след това се преобразуват в машинни инструкции от Асемблера.

Например горната програма би могла да се напише на С по следния начин:

С инструкции	**описание**
unsigned char sum;	Дефиниране на променлива с име sum
unsigned char sub;	Дефиниране на променлива с име sub
sum = 20 + 10;	Сумирай 20 + 10 и съхрани резултата в променлива sum
sub = 20 – 10;	Извади 20 – 10 и съхрани резултата в променлива sub

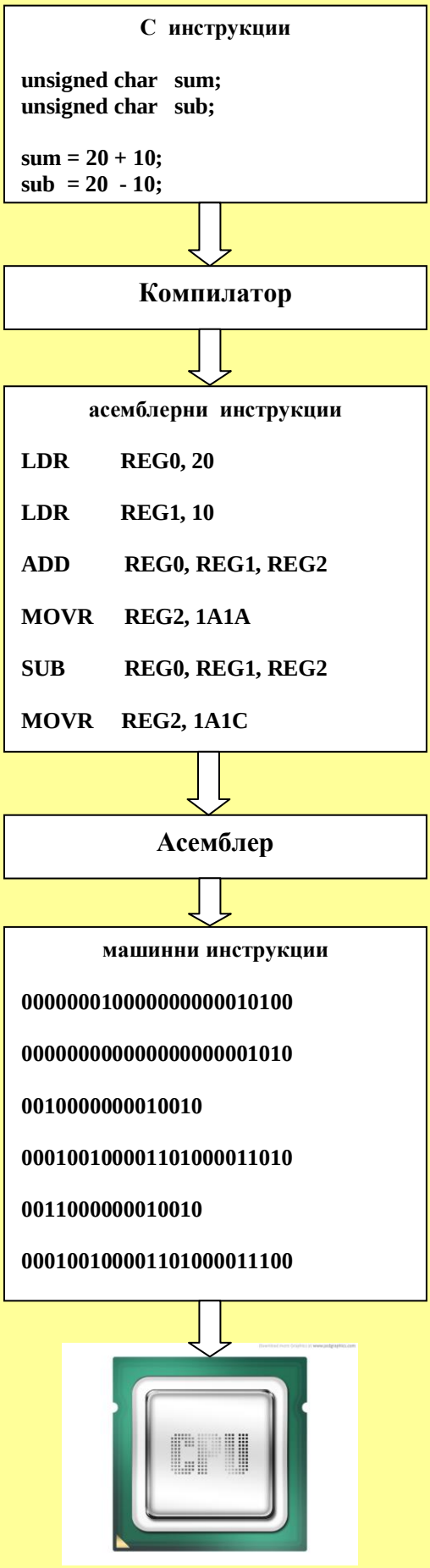
Фиг. 23 Примерна програма на С

Показаните С инструкции се интерпретират по следния начин:

unsigned char sum; -> Задели произволна 8 битова клетка от паметта и я наречи sum
unsigned char sub; -> Задели произволна 8 битова клетка от паметта и я наречи sub

sum = 20 + 10; -> Събери числата 10 и 20 и съхрани резултата в sum
sub = 20 - 10; -> Извади 10 от 20 и съхрани резултата в sub

Както виждате, когато пишете на С, не се интересувате от имена на регистри, нито от конкретни адреси в паметта. За всичко това се грижи компилаторът. Ако компилирате горната програма с компилатор за нашия хипотетичен процесор, тя ще се преобразува първо до програмата от Фиг.22. След това Асемблерът ще я преобразува до машинни инструкции, както е показано на Фиг.21, след което програмата е готова за изпълнение от процесора (Фиг.24).



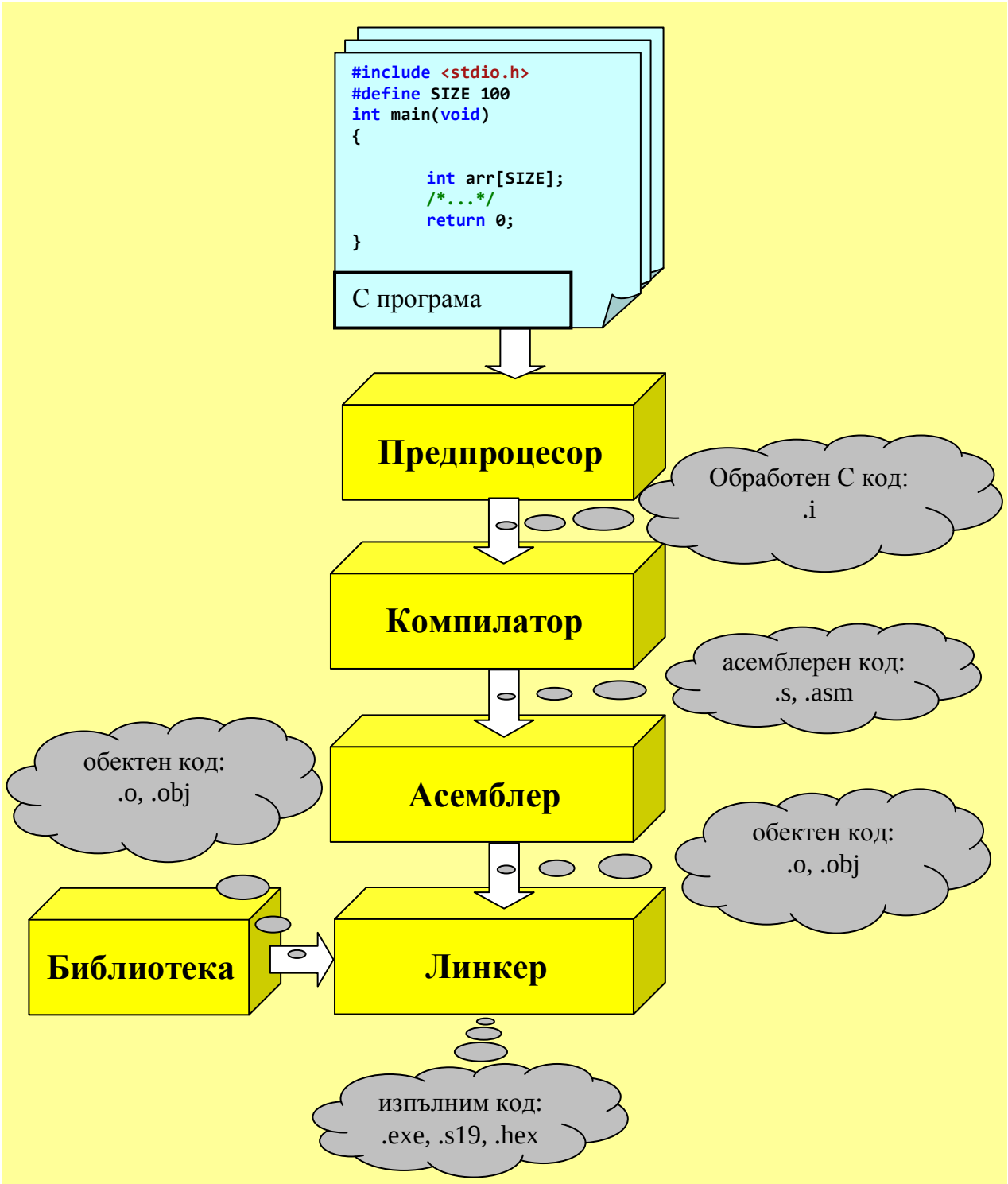
Фиг. 24 Компилиране на примерната С програма от Фиг.23

Важен момент, който трябва да разберете, е, че няма универсален С компилатор. Ако искате да компилирате С програма за процесор А, трябва да използвате компилатор, който преобразува С кода в асемблерни инструкции за процесор А. Ако искате да компилирате С програма за процесор Б, трябва да използвате компилатор, който преобразува С кода в асемблерни инструкции за процесор Б (Фиг.25).

Фиг. 25 Компилиране на С програма за различни процесори

6 Компилиране на С програма

Компилирането на една програма се нарича процесът на преобразуване на програмния код в код (машинни инструкции), подходящ за изпълнение от съответния компютър. Обобщеният процес на компилиране на една С програма изглежда по следния начин:



Фиг. 26 Създаване и компилиране на С програма

Една С програма се състои от един или няколко сорс-файла с разширение .c. Преди да се компилират, тези файлове се обработват от програма

наречена Предпроцесор. Обикновено обработените от Предпроцесора С файлове имат разширение **.i**. След това, всеки обработен файл се компилира. Процесът на компилация представлява преобразуване на обработения С код в асемблерен код. Обикновено обработените от Компилатора файлове имат разширение **.s** или **.asm**. Асемблерният код от своя страна се обработва от програма, наречена Асемблер, която го преобразува в т.нар. обектен код (машинни инструкции). Обикновено обработените от Асемблера файлове имат разширение **.o** или **.obj**. Линкерът свързва обектните файлове в един изпълним файл. Това е крайният файл от целия процес на компилация, който е готов за изпълнение от компютъра. На Линкера могат да се подават и външни обектни файлове (например получени от стандартни С библиотечни файлове). Изпълнимите файлове имат различни разширения в зависимост от това къде ще се изпълнява кода. Например изпълним файл предназначен да се изпълнява на персонален компютър има разширение **.exe**. Изпълними файлове, предназначени за изпълнение в микроконтролери, имат разширения като **.s19**, **.hex** и много други.

7 Интегрирани среди за програмиране

7.1 Общи сведения

Повечето производители на C компилатори осигуряват интегрирани среди за разработка на C програми (**IDE -Integrated Development Enviornment**). Тези среди съдържат текстов редактор, компилатор и всичко необходимо за написването и тестването на C програми. Наличието на всички необходими инструменти под формата на един софтуерен пакет значително опростява писането, компилирането, тестването и дебъгването на програмите, особено за начинаещи. В тази книга Ви предлагам две безплатни интегрирани среди: **Microsoft Visual Studio C++ Express** и **Pelles C for Windows**.

Средата **Microsoft Visual Studio C++ Express** поддържа само стандарта C89, затова ще я използваме само за примерите от Част II, където се разглежда стандарта C89(C90).

Средата **Pelles C for Windows** поддържа стандартите C99 и C11 и ще я използваме за примерите от Част III и IV.

Преди да започнете да пишете програми, в която и да е среда, е необходимо да създадете проект. Следващите точки описват как да създадете проект в гореспоменатите две среди.

7.2 Microsoft Visual Studio C++ 2010 Express

7.2.1 Създаване на проект

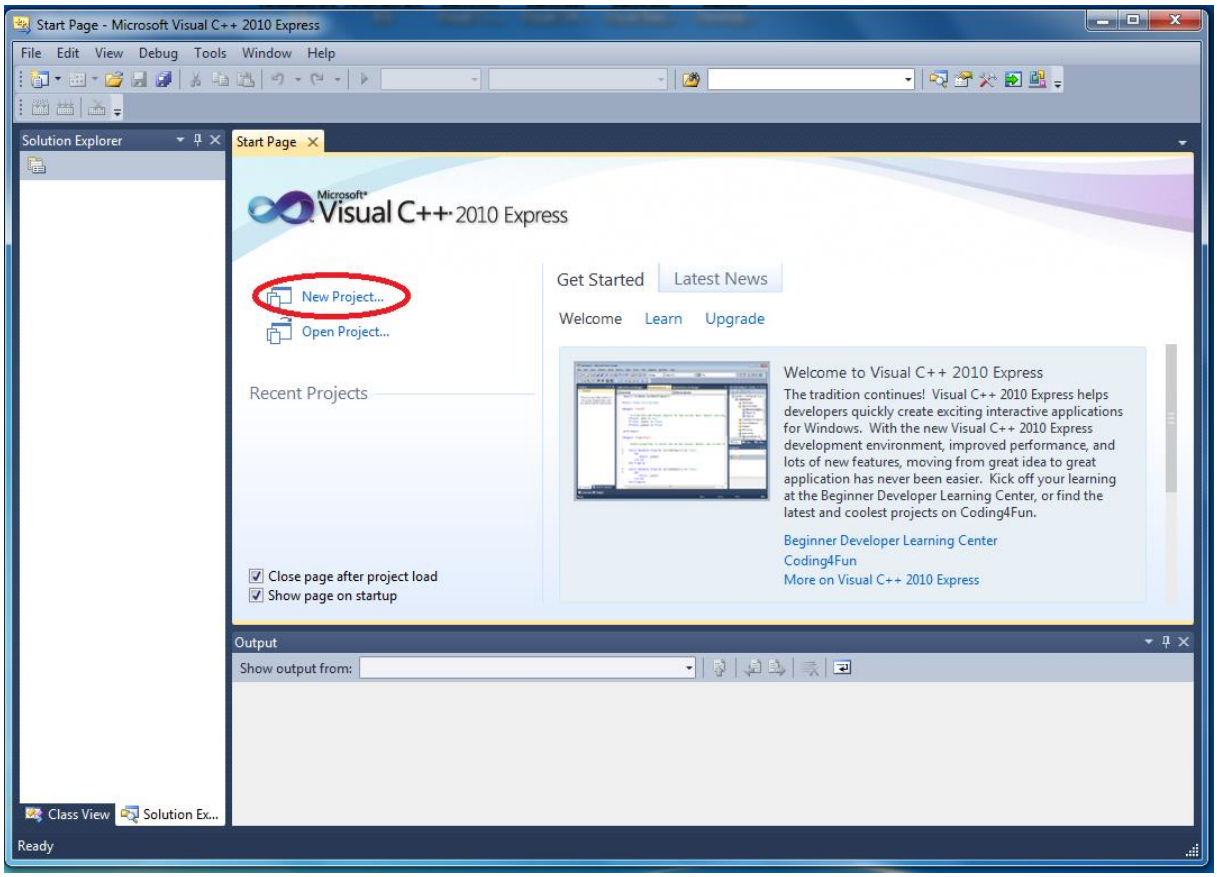
Средата **Microsoft Visual Studio C++ 2010 Express** (или по-нова) може да бъде свалена безплатно от сайта на Microsoft:

(http://www.microsoft.com/visualstudio/eng/downloads#d-2010 express).

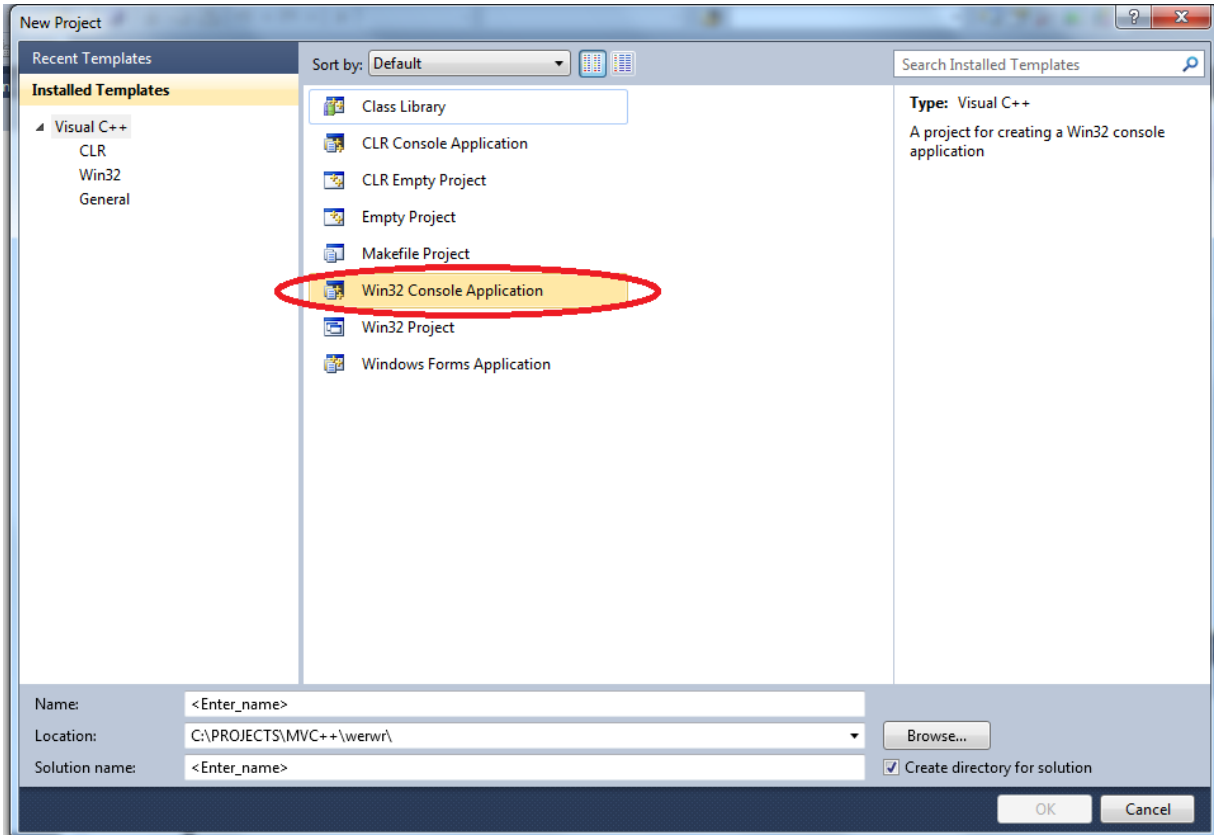
Стъпките по създаването на проект в средата на **Microsoft Visual Studio C++ 2010 Express** са следните.

1. Стартирайте Microsoft Visual C++ Express

2. Изберете New Project

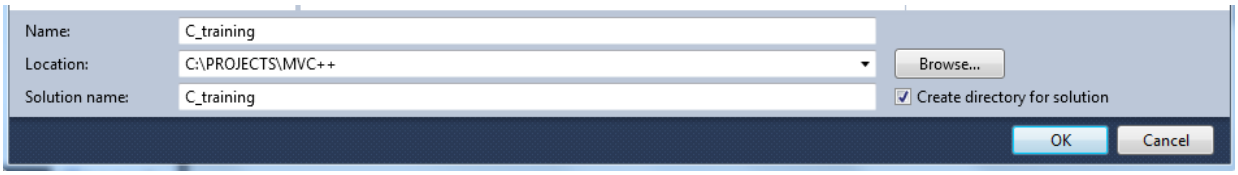


3. Изберете **Win32 Console Application**



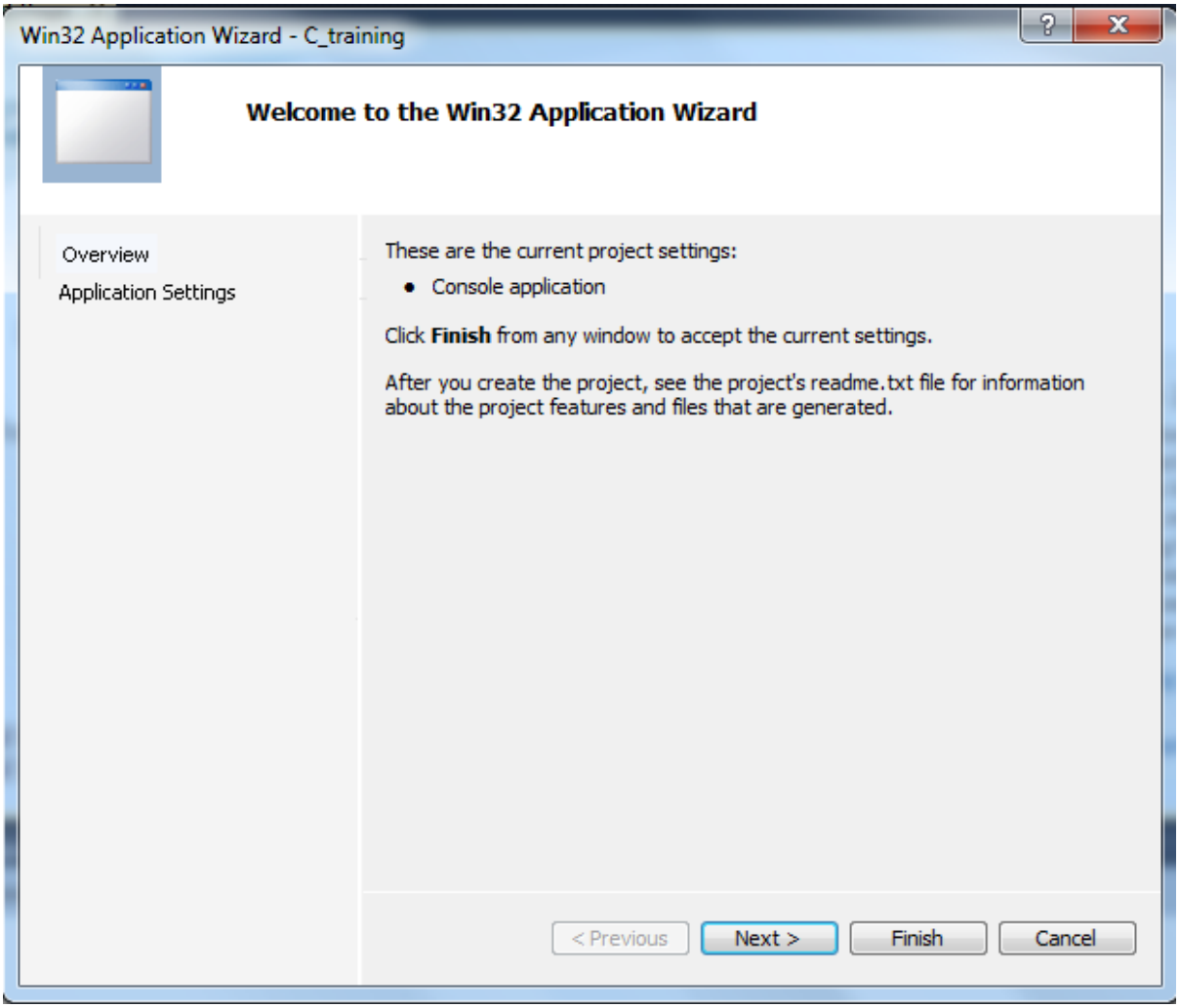
В полето Name: въведете име на проекта (използвайте името C_training). В полето Location: въведете пътя където ще се съхранява проекта (създайте

следната директория C:\PROJECTS\MVC++).

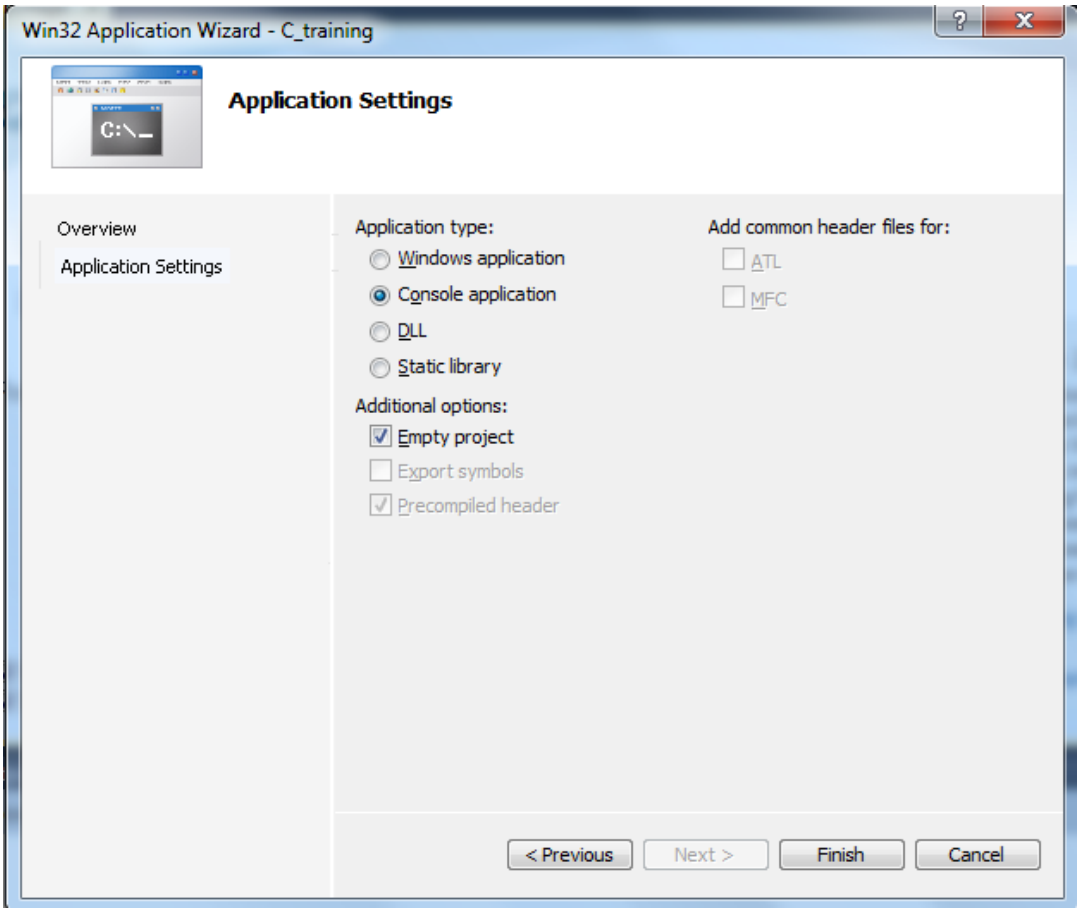


Натиснете бутона OK.

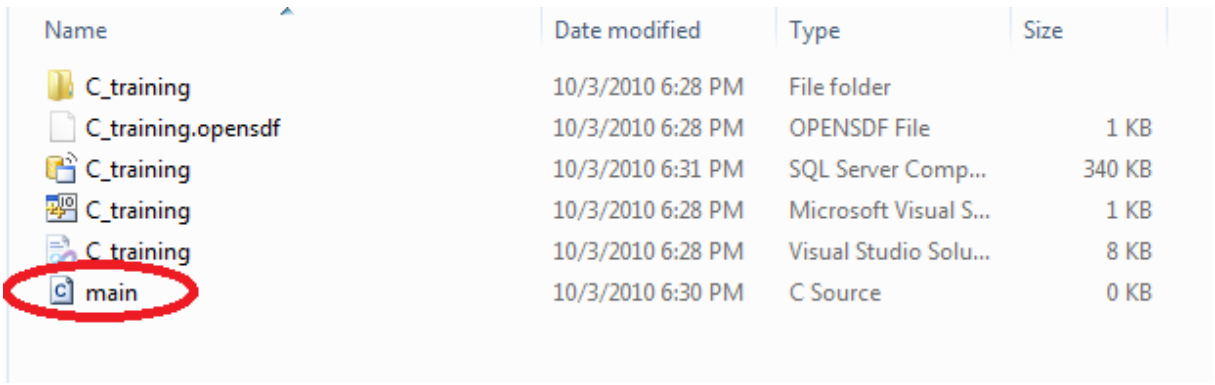
4. В прозореца **Win32 Application Wizard** натиснете бутона Next



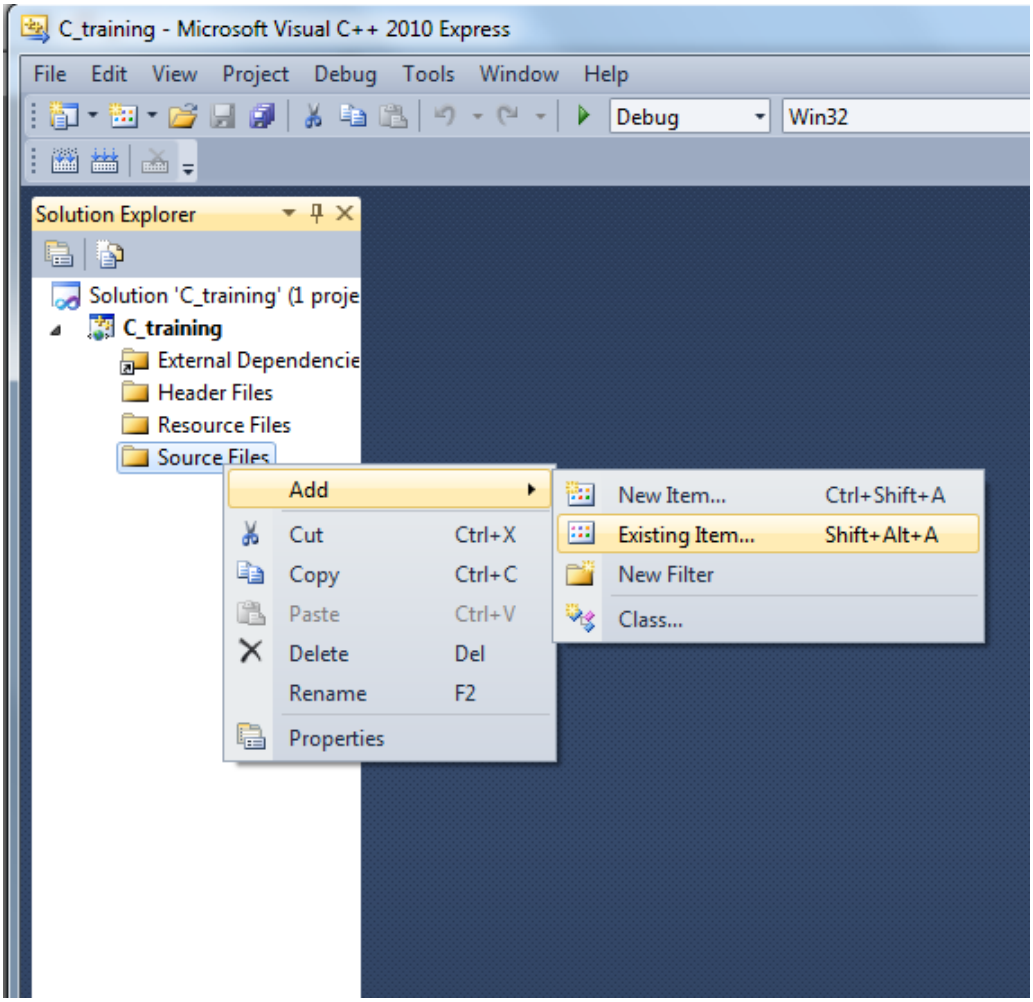
5. Изберете опциите, както е показано по-долу, и натиснете Finish



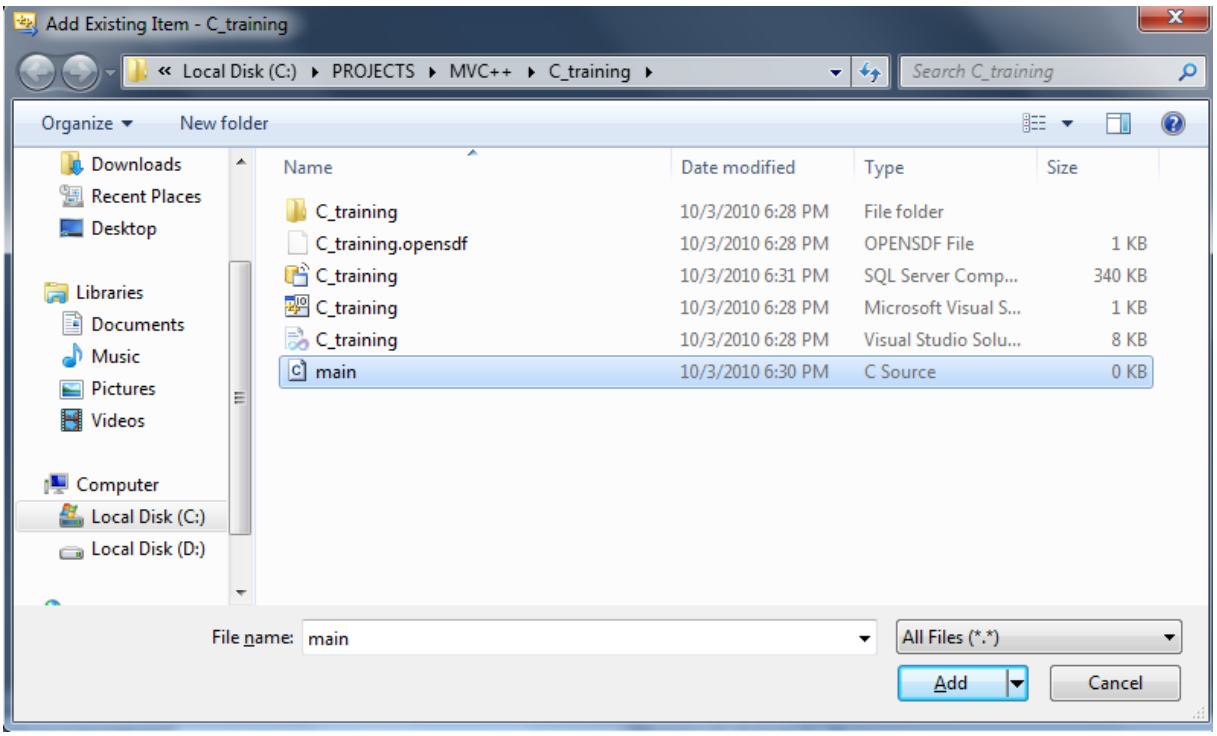
6. В директорията C:\PROJECTS\MVC++\C_training създайте файл с име main.c



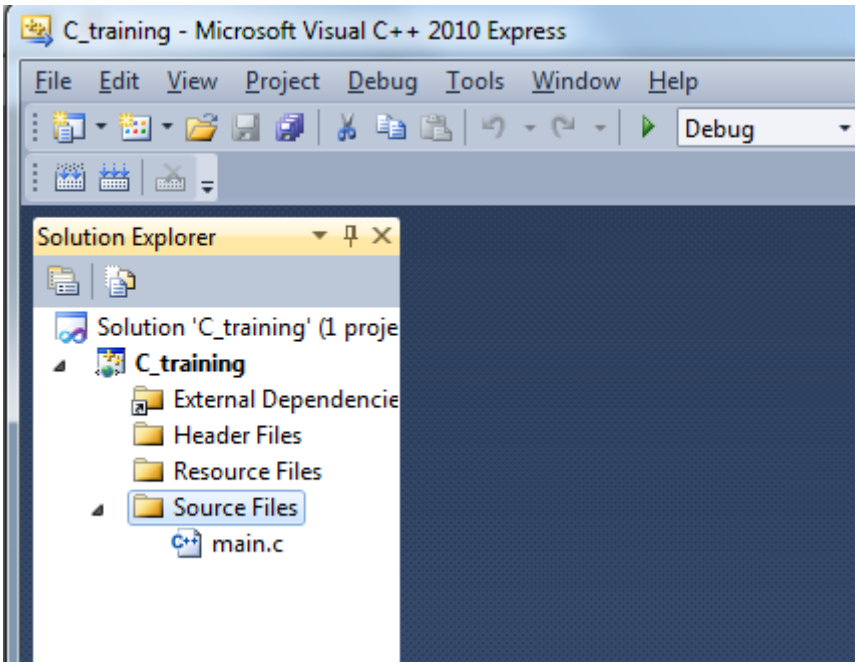
7. Добавете файла main.c в проекта като с десен бутон щракнете върху Source Files и изберете **Add\Exixting Item…**



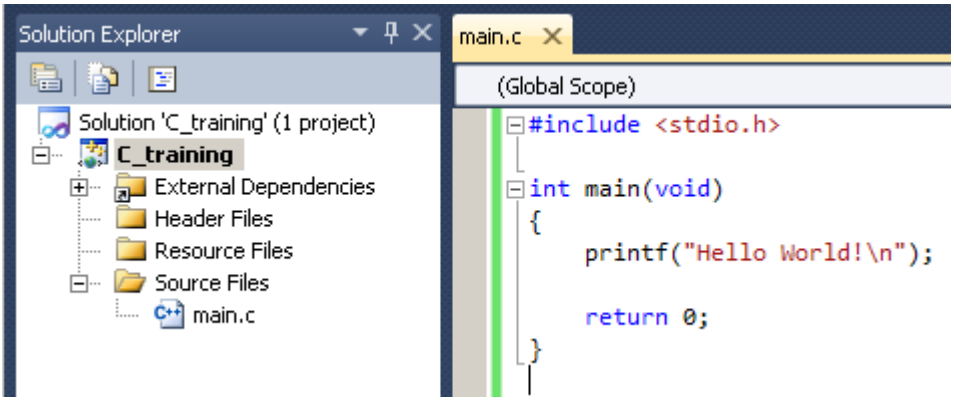
От прозореца Add Existing Item изберете файла main.c и натиснете бутона Add.



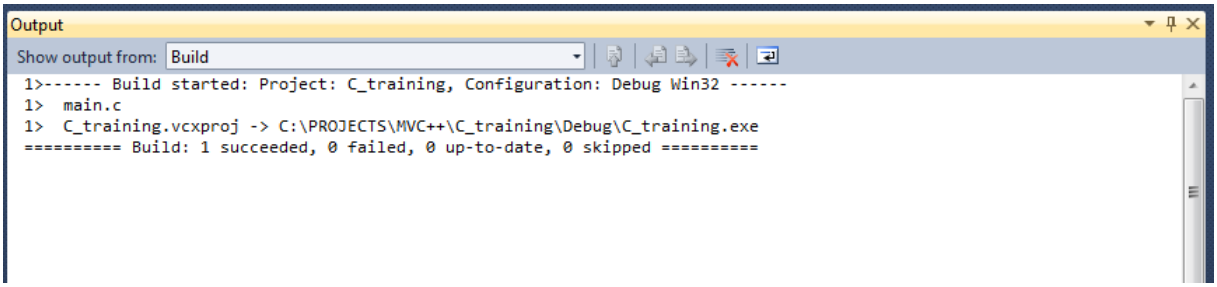
Файлът main.c е добавен в проекта и може да добавяте вашия код в него.



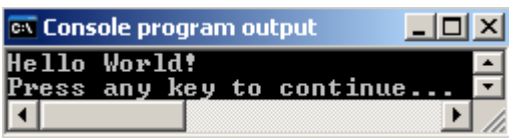
8. Отворете main.c с двукратно щракване върху него и въведете кода както е показано на снимката



9. Натиснете бутона F7, за да компилирате файла main.c. Ако сте въвели кода вярно, в прозореца Output трябва да видите следното:



10. Стартирайте програмата с **Ctrl+F5**. В конзолния прозорец трябва да видите съобщението "Hello World!"



11. Натиснете произволен клавиш от клавиатурата, за да затворите конзолния прозорец.

Ако сте изпълнили всички 11 точки, както е описано по-горе, значи вече имате създаден проект и сте готови да продължите напред.

7.2.2 Компилиране, изпълнение и дебъгване на кода

В тази глава накратко са показани някои от лентите с инструменти на развойната среда **Microsoft Visual Studio C++ 2010 Express**, които ще са ви необходими при изучаването на езика C с помощта на тази книга.

- Build

 - Компилира всички .c файлове, изграждащи програмата (F7)

 - Компилира само .c файловете, които са променени

- Debug

 - Стартира изпълнението на програмата (F5, Ctrl+F5)
Ctrl+F5 задържа конзолния прозорец отворен до натискане на клавиш.

 - Спира изпълнението на програмата (Shift+F5)

 - Изпълнява поредния оператор[1] на програмата. Влиза в тялото на функция[2] при изпълнение на оператор за извикване на функцията (F11).

Забележка[1]: Операторът е инструкция, която извършва някакво действие.

Забележка[2]: Функцията е парче програмен код, което съдържа оператори.

 - Изпълнява поредния оператор на програмата. Операторът за извикване на функция се изпълнява като единичен оператор, т.е. . операторите в тялото на функцията се изпълняват наведнъж (F10).

 - Изпълнява всички оператори на текущата функция наведнъж и

преминава към изпълнение на оператора, намиращ непосредствено след извикването на функцията (Shift+F11).

Ако не виждате лентата Debug, покажете я чрез поставяне на отметка в полето View -> Toolbars -> Debug

Използвайте тези две ленти, за да компилирате и изпълнявате кода стъпка по стъпка.

В случай на синтактична грешка компилаторът ще издаде съобщение за грешка. Погледнете примера по-долу.

```
(Global Scope)
 1  /* Пример 1.1 */
 2
 3  #include <stdio.h>
 4  #include <stdlib.h>
 5
 6  void myfunc(void);
 7
 8  int main(void)
 9  {
10      /* Извикай потребителската функцията myfunc */
11      myfunc() ←— Липсва точка и запетая
12
13      /* Извикай библиотечната функцията system  */
14      system("pause");
15      return 0;
16  }
17
18  /* Дефиниция на функцията myfunc */
19  void myfunc(void)
20  {
21      /*Отпечатай съобщение на дисплея */
22      printf("First C program\n");
23
24      return;
25  }
26
```

Когато се опитате да компилирате кода, компилаторът ще генерира следното съобщение за грешка:

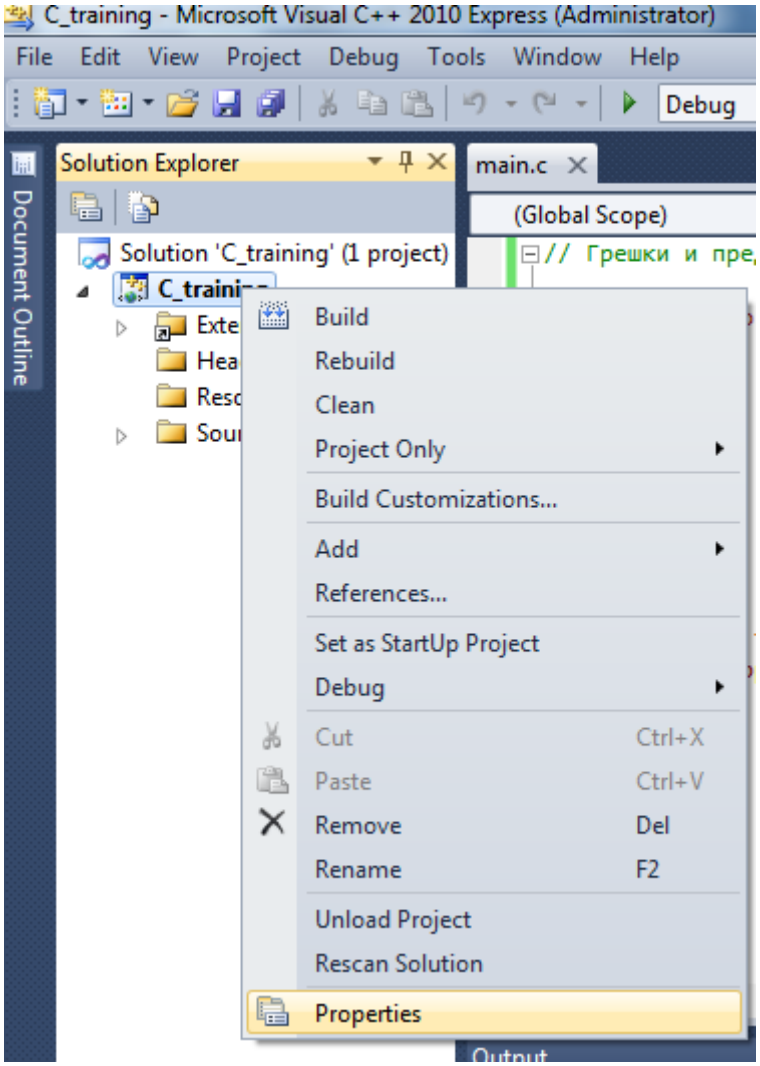
```
Output
Show output from: Build
1>------ Build started: Project: C_training, Configuration: Debug Win32 ------
1>  main.c
1>c:\projects\mvc++\c_training\main.c(14): error C2146: syntax error : missing ';' before identifier 'system'
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

Чрез двукратно щракване върху съобщението може да проверите на кой ред от програмата компилаторът открива грешка. Трябва да знаете, че не винаги грешката се намира на този ред. Ако не откривате грешка, прегледайте редовете преди този, на който компилаторът открива грешка.

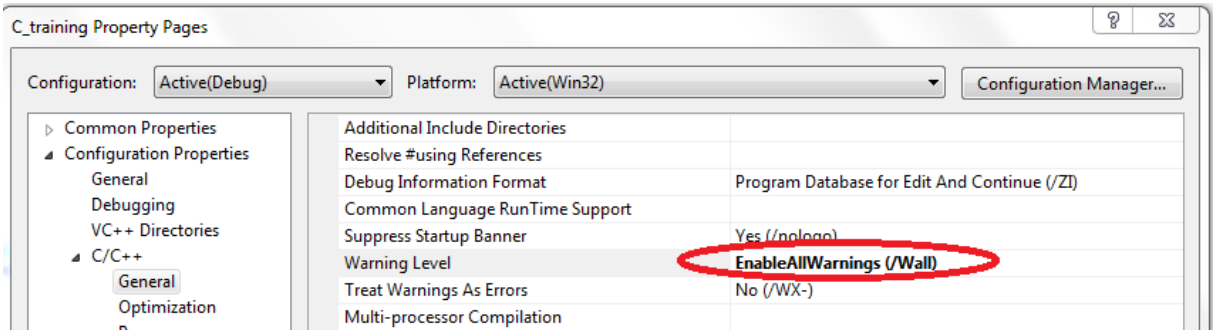
Както ще се убедите в нашия пример, компилаторът не открива точка и запетая преди извикването на функцията **system()**. Причина за това е, че на предходния ред липсва точка и запетая след извикването на функцията **myfunc()**.

Освен съобщения за грешка, компилаторът може да генерира и предупреждаващи съобщения. Тъй като езикът C не е строго типизиран език и дава голяма свобода на програмиста, той позволява конструкции, които могат да доведат до трудно откриваеми логически грешки. Ето защо, компилаторът ще ви предупреди да обърнете внимание на такъв тип конструкции.

Microsoft Visual Studio C++ 2010 Express (както и повечето други среди за разработка) позволява степента на предупреждение да се конфигурира. За да позволите на компилатора да генерира предупредителни съобщения от всякакъв вид, щракнете с десен бутон върху името на проекта и изберете Properties както е показано.



В полето C/C++ -> General -> Warning Level изберете опцията **EnableAllWarnings(/Wall)**.

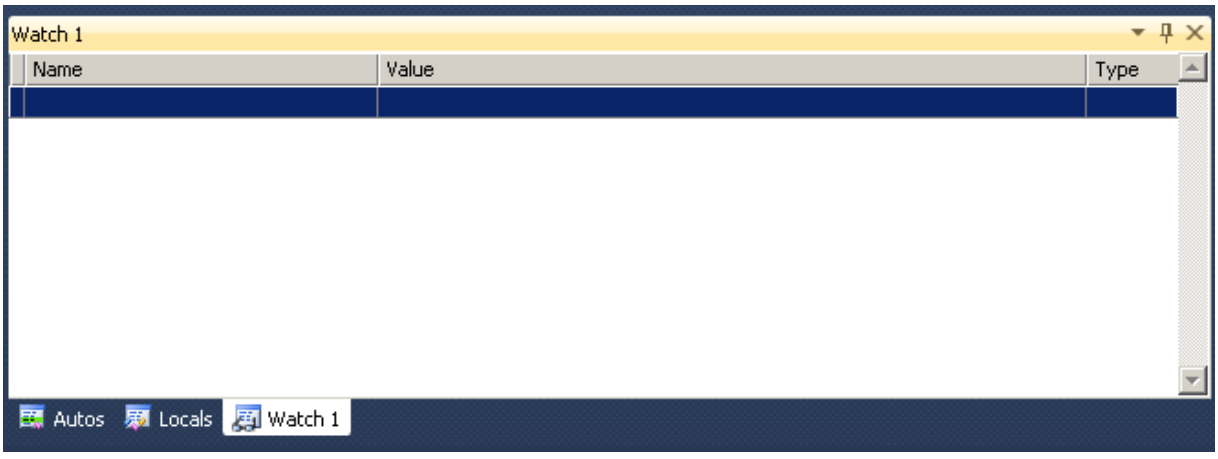


Друго полезно нещо, което всяка среда за програмиране предоставя, е прозорец, в който може да наблюдавате текущите стойности на променливите в кода. В следващите точки ще научите повече за променливите, засега просто запомнете, че **променлива**, това е мястото, в което съхранявате данни.

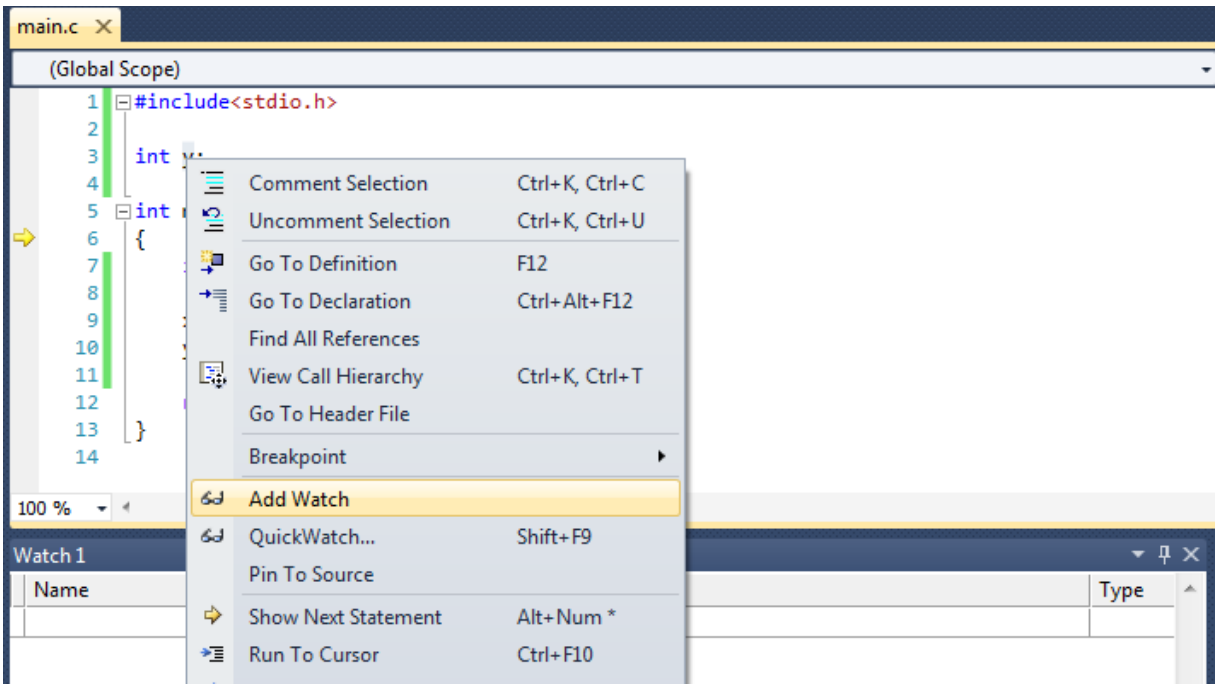
Например в следващата програма са дефинирани две променливи x и y.

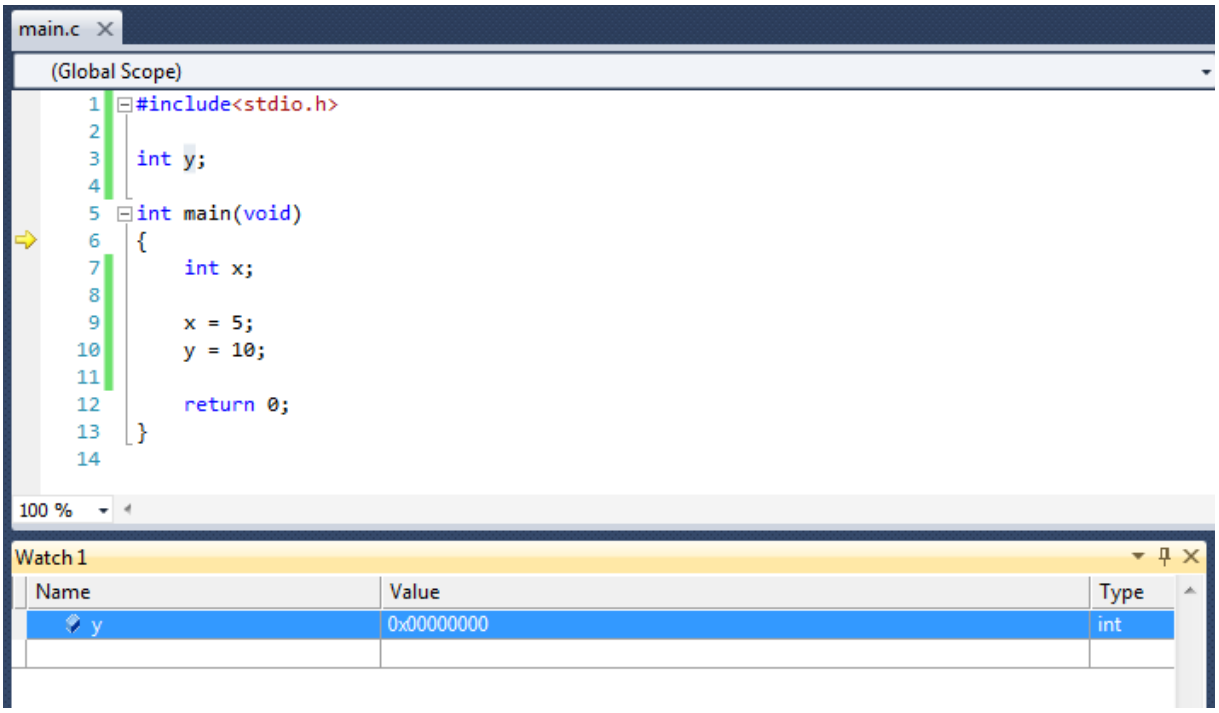
```
main.c
(Global Scope)
1  #include<stdio.h>
2
3  int y;
4
5  int main(void)
6  {
7      int x;
8
9      x = 5;
10     y = 10;
11
12     return 0;
13 }
14
```

Когато тръгнете да изпълнявате кода стъпка по стъпка (бутони F10 и F11), под прозореца, в който въвеждате сорс-кода, се добавя допълнителен прозорец с три етикета: Autos, Locals и Watch1.



Щраквайки с десен бутон върху името на променлива и избирайки **Add Watch** може да добавяте променливата в прозореца Watch1 и да наблюдавате как се променя нейната стойност в хода на изпълнение на програмата.





7.3 Pelles C for Windows

7.3.1 Създаване на проект

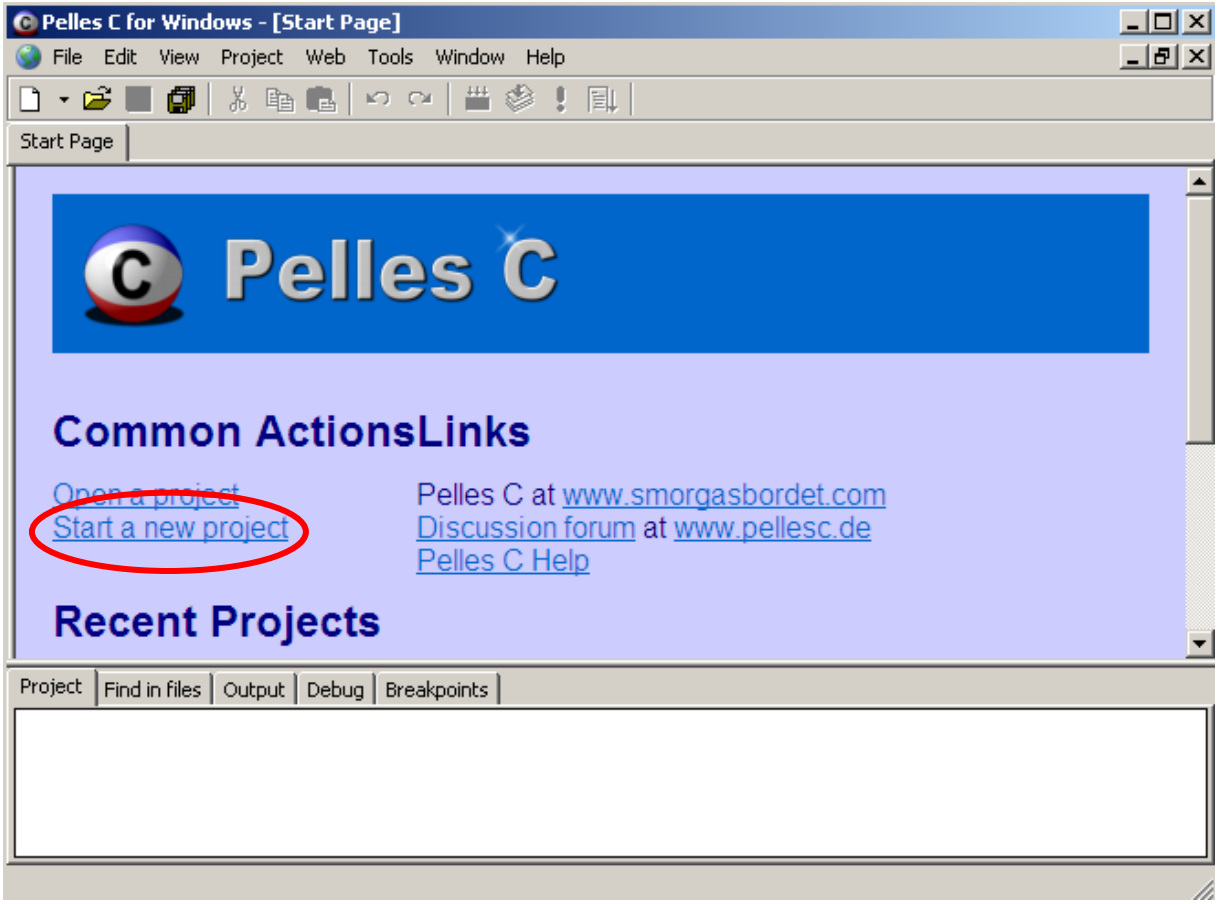
Средата **Pelles for Windows** може да бъде свалена безплатно от следния сайт:

(http://www.pellesc.de/index.php?page=download&lang=en).

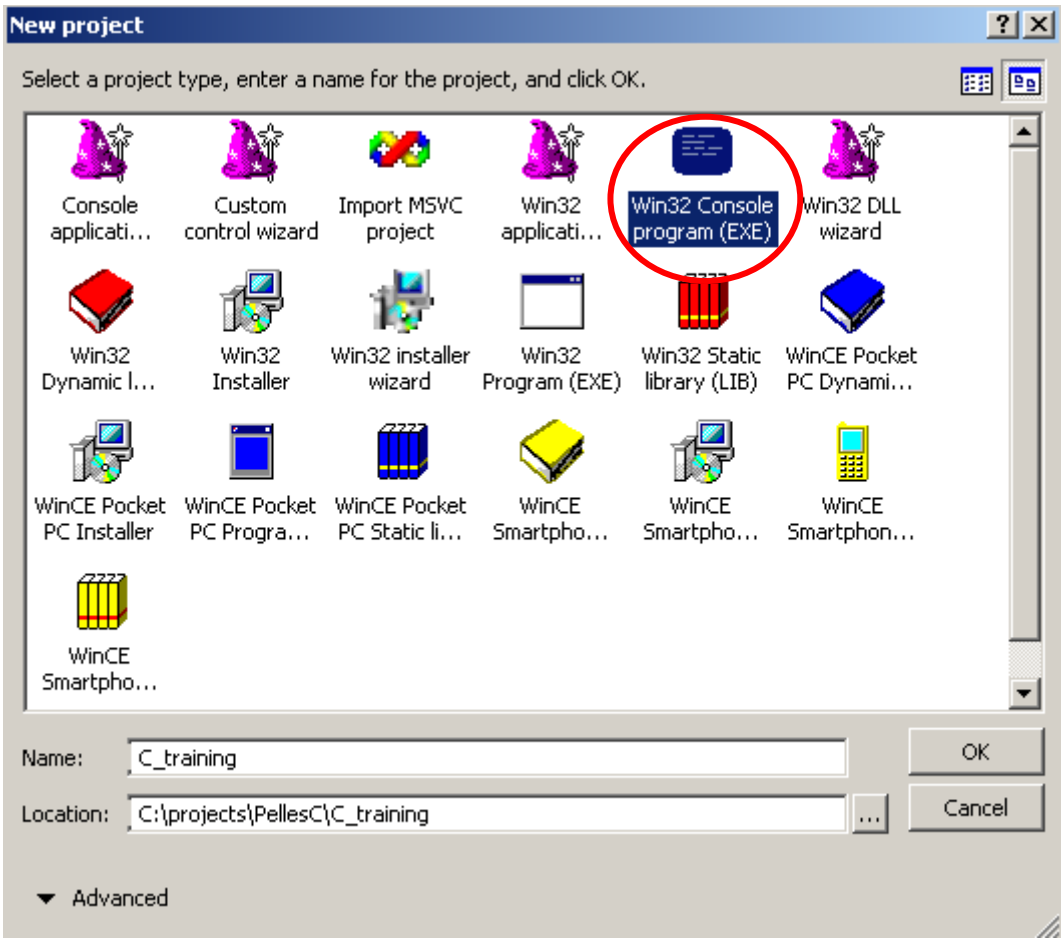
Стъпките по създаването на проект в средата на **Pelles for Windows** са следните:

1. Стартирайте Pelles C IDE

2. Изберете Start a new project



3. Изберете Win32 Console program (EXE)

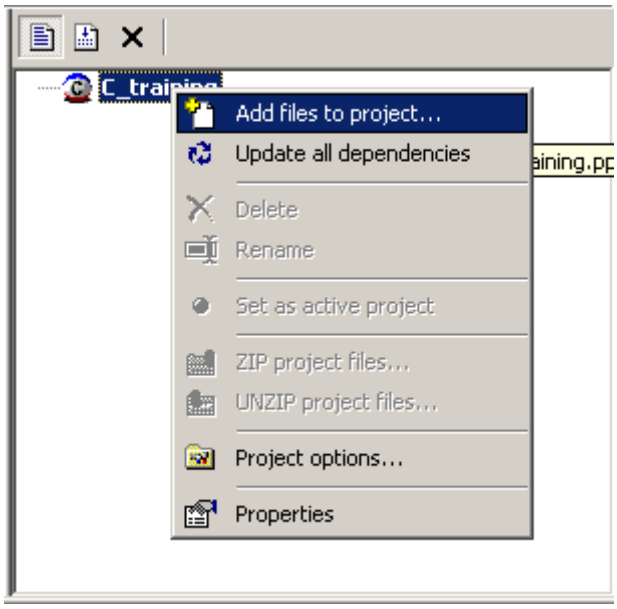


В полето Name: въведете име на проекта (използвайте името C_training). В полето Location: въведете пътя където ще се съхранява проектът (създайте следната директория C:\PROJECTS\PellesC).

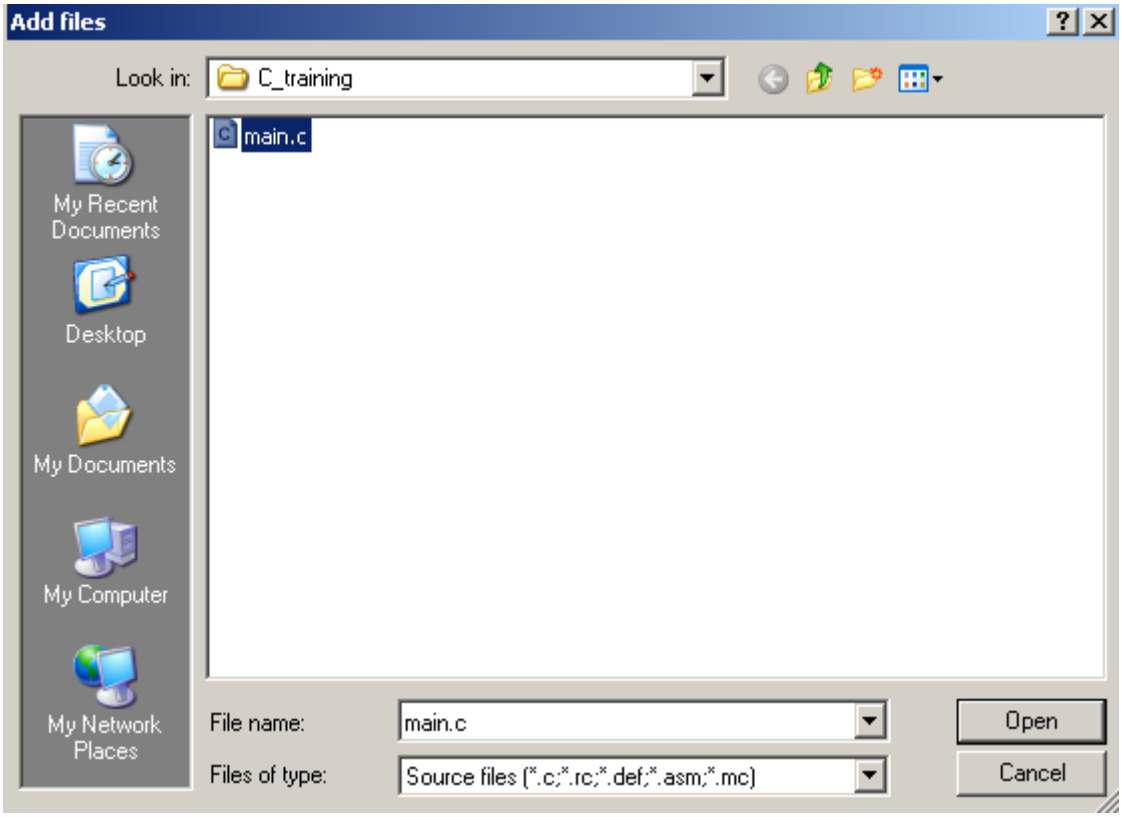
Натиснете бутона OK.

4. В директорията C:\PROJECTS\PellesC създайте файл с име main.c

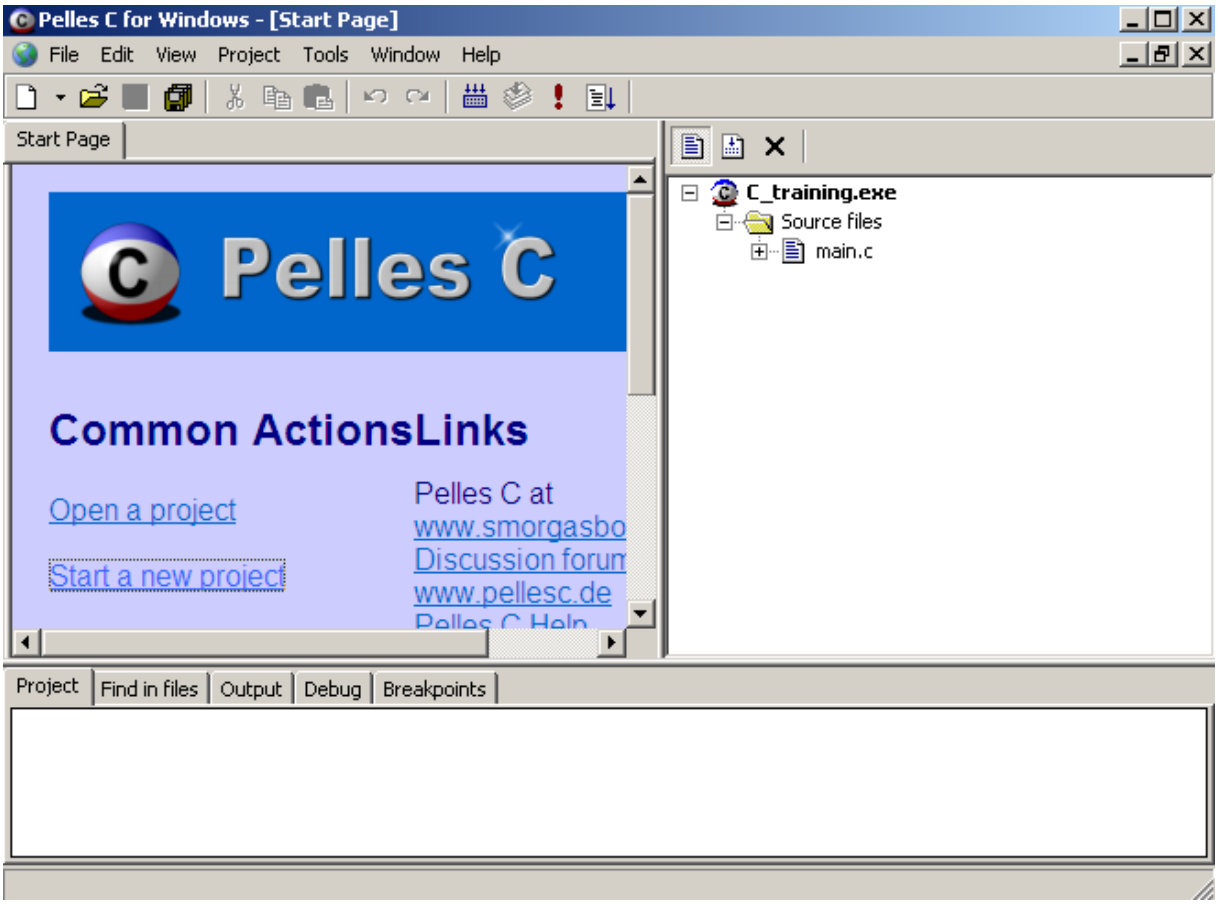
5. Добавете файла main.c в проекта като с десен бутон щракнете върху C_training и изберете **Add files to poject…**



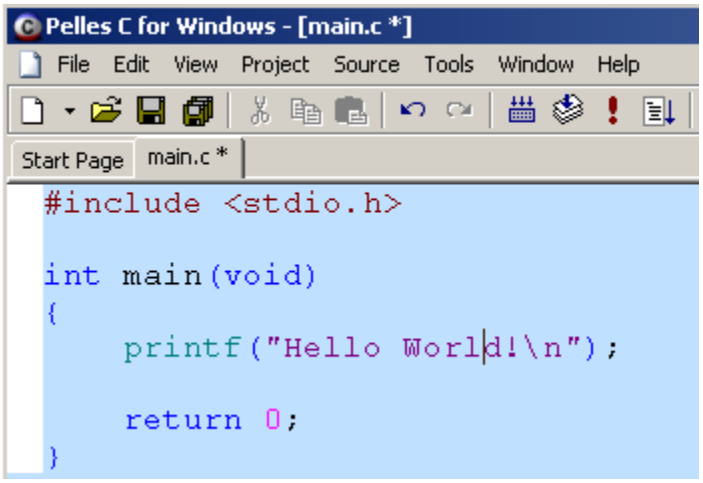
От прозореца Add files изберете файла main.c и натиснете Open.



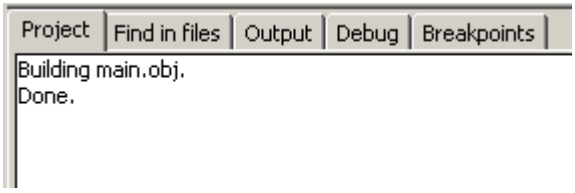
Файлът main.c е добавен в проекта и може да добавяте вашия код в него.



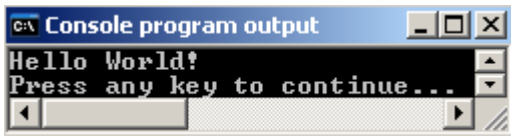
6. Отворете main.c с двукратно щракване върху него и въведете кода както е показано на снимката:



7. Натиснете  , за да компилирате файла main.c. Ако сте въвели кода вярно, в прозореца Project трябва да видите следното:



8. Стартирайте програмата с **Ctrl+F5**. В конзолния прозорец трябва да видите съобщението "Hello World!"



Ако сте изпълнили всички 8 точки, както е описано по-горе, значи вече имате създаден проект и сте готови да продължите напред.

7.3.2 Компилиране, изпълнение и дебъгване на кода

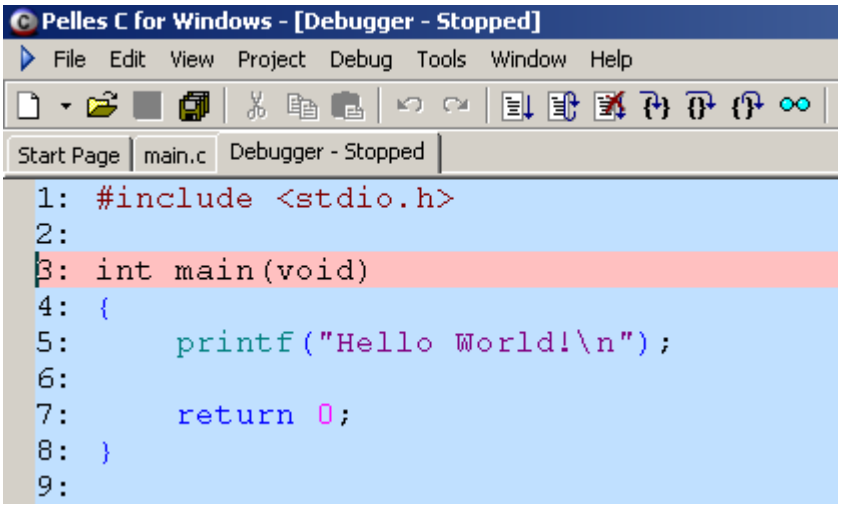
В тази глава накратко са показани някои от лентите с инструменти на развойната среда **Pelles C for Windows**, които ще са ви необходими при изучаването на езика C, с помощта на тази книга.

 - Компилира всички .c файлове, изграждащи програмата (Ctrl+B)

 - Компилира само текущо активния copc-файл

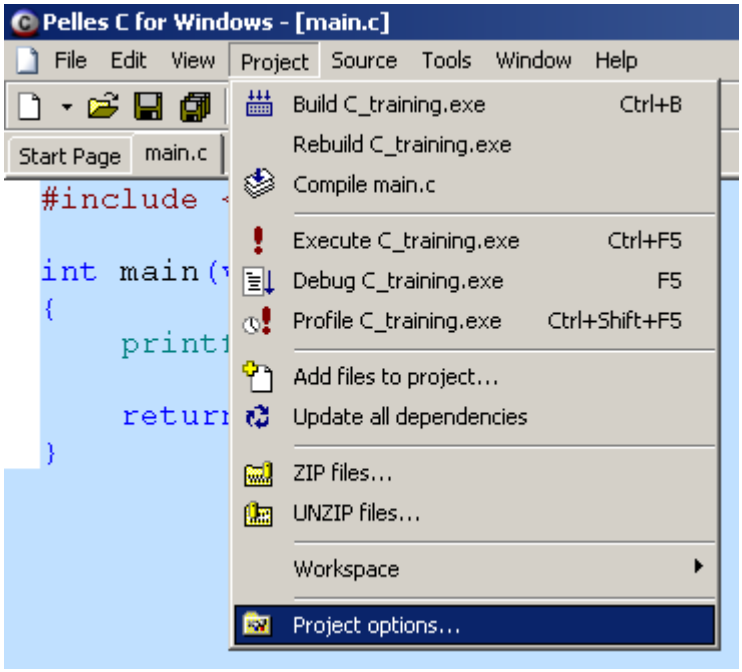
 - Стартира изпълнението на програмата (Ctr+F5)

 - Стартира програмата в режим на дебъгване[1] (F5)



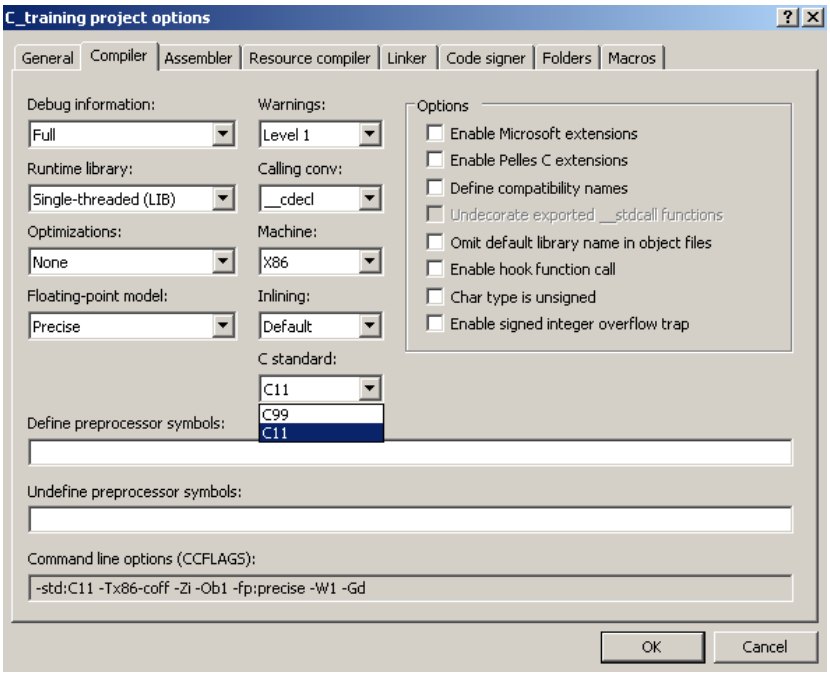
Забележка[1]: Преди да започнете да дебъгвате програмата е необходимо да конфигурирате следните настройки:

1. Изберете **Project/Project options…**



2. В прозорците Compiler и Assembler задайте Debug Information: **Full** и Optimizations:**None**, а в прозореца Linker задайте Debug Information: **CodeView format**.

В прозореца Compiler може също да конфигурирате нивото на предупреждения, както и версията на компилатора (C99 или C11), с която да компилирате кода.



В режим на дебъгване следните допълнителни бутони стават активни:

 - Стартира изпълнение на програмата (F5)

 - Рестартира програмата

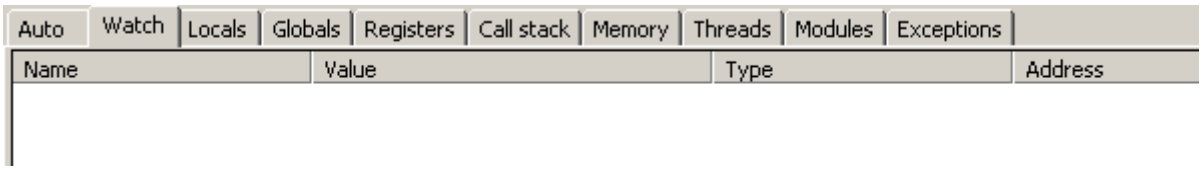
 - Излизане от дебъг режим (Shift+F5)

 - Изпълнява поредния оператор на програмата. Влиза в тялото на функция при изпълнение на оператор за извикване на функцията (F11)

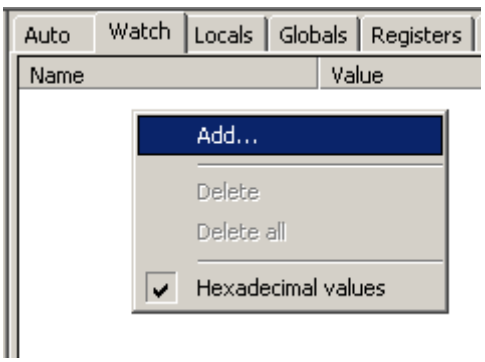
 - Изпълнява поредния оператор на програмата. Операторът за извикване на функция се изпълнява като единичен оператор, т.е. . операторите в тялото на функцията се изпълняват наведнъж (F10)

 - Изпълнява всички оператори на текущата функция наведнъж и преминава към изпълнение на оператора, намиращ непосредствено след извикването на функцията (Shift+F10)

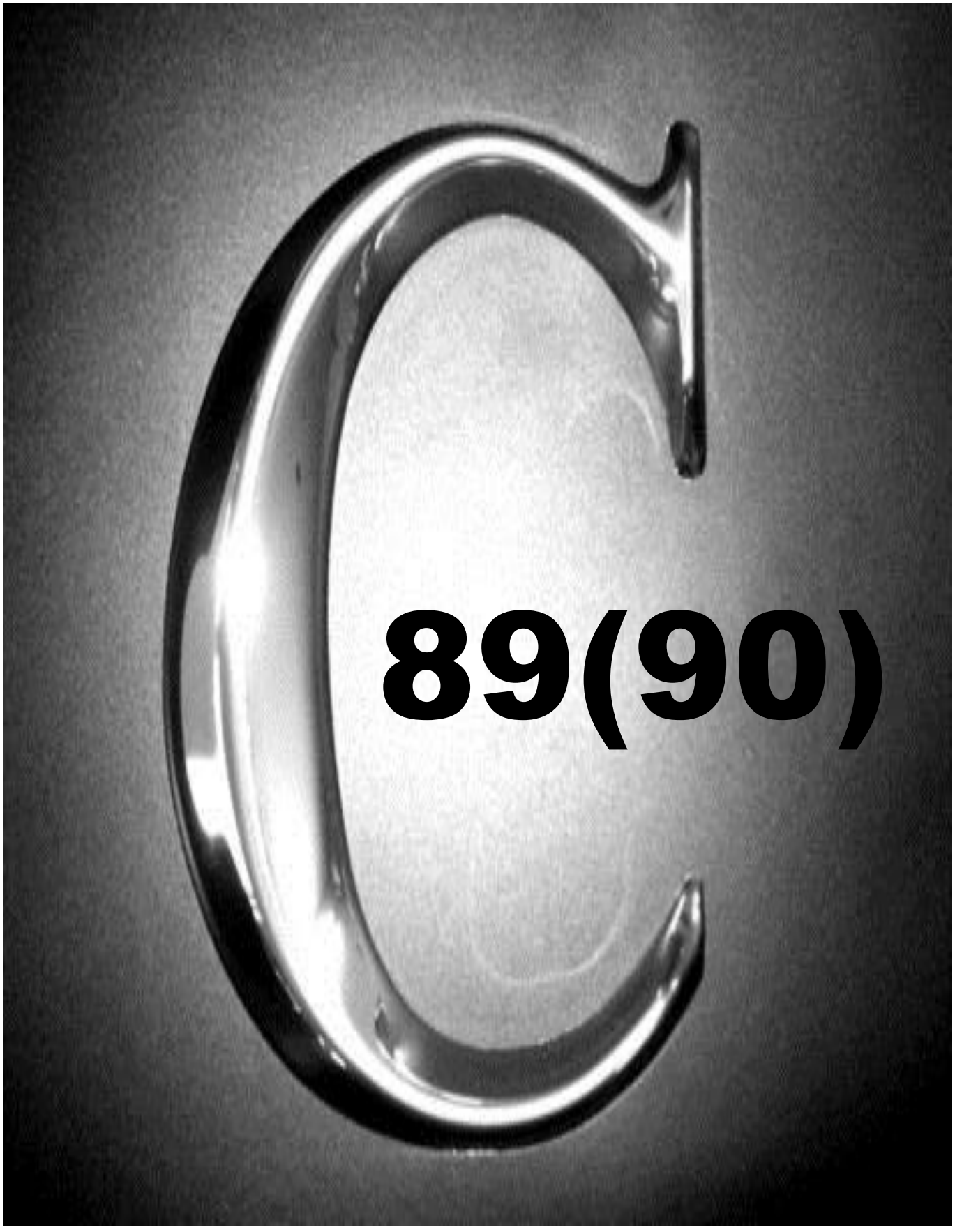
Когато програмата е в дебъг режим, следният прозорец е наличен.



Щраквайки с десен бутон в прозореца Watch, може да добавяте променливите, които сте създали в програмата и да наблюдавате как се променят техните стойности в хода на нейното изпълнение.



Част II: C89(C90)



8 Въведение в С програмирането

8.1 Конструкция на С програма

Най-общо една С програма може да се представи по следния начин:

```
/*********************************************************
 *  1. Включване на хедър-файлове                        *
 *********************************************************/
#inlcude  < хедър-файл1.h >
#inlcude  < хедър-файл2.h >
....
#inlcude  < хедър-файлN.h >

/*********************************************************
* 2. Описание на прототипите на потребителските функции  *
*********************************************************/
тип    име-функция1(списък-параметри);
тип    име-функция2(списък-параметри);
....
тип    име-функцияN(списък-параметри);

/*********************************************************
* 3. Дефиниране на глобални променливи                   *
*********************************************************/
тип    име-променлива1;
тип    име-променлива2;
....
тип    име-променливаN;

/*********************************************************
* 4. Начало на С програмата                              *
*    Изпълнението на С програмата започва тук!           *
*********************************************************/
int main(void)
{
      /* 4.1 Дефиниране на локални променливи  */
      тип    име-променлива1;
      тип    име-променлива2;
      ....
      тип    име-променливаN;

       /* 4.2 Потребителски код                */
       оператор1;
       оператор2;
        ....
       операторN;

        return 0;
}

/*********************************************************
*          5. Дефиниране на потребителските функции      *
*********************************************************/
тип   име-функция1(списък-параметри)
{
      /* 5.1 Дефиниране на локални променливи  */
      тип    име-променлива1;
      тип    име-променлива2;
      ....
      тип    име-променливаN;
```

```
        /* 5.2 Потребителски код                */
        оператор1;
        оператор2;
        ....
        операторN;
}

тип    име-функция2(списък-параметри)
{
        /* 5.1 Дефиниране на локални променливи  */
        тип    име-променлива1;
        тип    име-променлива2;
        ....
        тип    име-променливаN;

        /* 5.2 Потребителски код                */
        оператор1;
        оператор2;
        ....
        операторN;
}

....

тип    име-функцияN(списък-параметри)
{
        /* 5.1 Дефиниране на локални променливи  */
        тип    име-променлива1;
        тип    име-променлива2;
        ....
        тип    име-променливаN;

        /* 5.2 Потребителски код                */
        оператор1;
        оператор2;
        ....
        операторN;
}
```

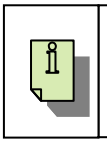
Основните части на една С програма са :

- **Хедър-файлове** – Това са файлове с разширение .h, които съдържат информация необходима за изпълнението на програмата, например прототипи на библиотечни функции. Съдържанието на тези файлове се включва с помощта на предпроцесорната директива **#include.**

- **Прототипи на функции** – Прототипът на една функция дава информация на компилатора за самата функция. Въпреки че те не са задължителни, всяка професионално написана програма използва прототипи.

- **Глобални променливи** – променливата е област от паметта, използвана за съхранение на данни. Когато една променлива е обявена извън всички функции, тя се нарича глобална. Глобалните променливи могат да се използват от всички функции.

- **Функция main()** – Основните градивни елементи на една С програма са функциите. Функцията в езика С представлява подпрограма, която се изпълнява при нейното извикване. Всяка една С програма започва изпълнението си от специална функция, наречена **main()**, която се извиква автоматично при стартиране на програмата. Тази функция е определена от стандарта С и задължително трябва да присъства във всяка една С програма. Изпълнението на програмата започва след отварящата фигурна скоба { и завършва след затварящата фигурна скоба } на **main()**.



Функцията main() задължително трябва да присъства във всяка С програма. Всяка С програма започва изпълнението си от функцията main().

- **Локални променливи** – когато една променлива е обявена вътре в една функция, тя се нарича локална. Локалните променливи могат да се използват само от функцията, в която са обявени. Те не са достъпни за други функции.

- **Функции** - Функциите се разделят на библиотечни и потребителски. Библиотечните функции се предоставят от производителя на компилатора и се използват наготово от потребителя. Библиотечните функции от своя страна се разделят на стандартни и нестандартни. Стандартни са тези, които са дефинирани от С стандарта. Към тях всеки производител на С компилатор може да добави свои библиотечни функции. Потребителските функции се създават от самия програмист. Подобно на функцията **main()**, изпълнението на кода на всяка функция започва и завършва с отваряща и затваряща фигурна скоба. Тялото (кодът) на една функция се изгражда с помощта на оператори. Функциите могат да приемат стойности и да връщат резултат.

- **Оператори** - Операторът е програмна конструкция, която извършва някакво действие. Всички оператори в С завършват с точка и запетая. Тя служи за разпознаване на край на оператор. След като един оператор се изпълни, програмата преминава към изпълнението на следващия оператор.

Операторите в С завършват с точка и запетая. Тя служи за разпознаване на край на оператор.

- **Коментари** – Коментарът представлява потребителско съобщение затворено между **/*** и ***/.** Всичко поставено между тези символи се игнорира от компилатора и служи единствено за пояснение на програмата. Например:

```
/* Това е коментар */
```

или

```
/* Това също
е коментар, но на няколко реда */
```

Въпреки че коментарите не са задължителни, те са много полезни. Всяка професионално написана програма използва коментари, за да документира кода и действията, които се извършват.

Най-кратко казано, една С програма се състои от функции, които от своя страна се състоят от оператори.

Нека сега да напишем първата С програма.

Пример 8-1:

```
/* Включване на библиотечни хедър-файлове */
#include <stdio.h>  /* Съдържа прототипа на функцията printf() */
#include <stdlib.h> /* Съдържа прототипа на функцията system() */

/* Включване на прототипи на потребителски функции */
void myfunc(void);

/* Входна точка на програмата */
int main(void)
{
	/* Извикай потребителската функцията myfunc */
	myfunc();

	/* Извикай библиотечната функцията system   */
	system("pause");

	return 0;
}

/* Дефиниция на функцията myfunc */
void myfunc(void)
{
	/*Отпечатай съобщение на дисплея */
	printf("First C program\n");

	return;
}
```

Обяснение на програмата ред по ред:

`/* Включване на библиотечни хедър-файлове */` - Както беше споменато, всичко затворено между`/*`и `*/` е коментар и се игнорира от

компилатора.

`#include <stdio.h>` - Включва библиотечния хедър-файл stdio.h в програмата. Този файл е необходим за правилната работа на стандартната библиотечна функция **printf()**.

`#include <stdlib.h>` - Включва библиотечния хедър-файл stdlib.h в програмата. Този файл е необходим за правилната работа на стандартната библиотечна функция **system()**.

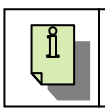
`/* Включване на прототипи на потребителски функции */` - Отново коментар, който подсказва какво следва след него.

`void myfunc(void);` - Прототип на потребителската функция myfunc. Думата void е една от ключовите думи в езика C. Както беше споменато, функциите могат да приемат и да връщат стойности. Първата дума void пред името myfunc означава, че функцията не връща стойност, а втората дума void в скобите указва на компилатора, че функцията не приема стойности. Понятието "ключова дума" означава, че една дума е дефинирана от стандарта C и има специално значение в програмата. Всички ключови думи в C са дадени в края на тази глава.

`int main(void)` – програмата започва с извикване на функцията **main()**. Ключовата дума int указва на компилатора, че main връща цяло число след завършване на работата си, а void указва, че **main()** не приема стойности.

`/* Извикай функцията myfunc */` - Отново коментар, който пояснява оператора след него

`myfunc();` - Тази конструкция извиква потребителската функцията myfunc. За да извикате функция, която не приема стойности, след името поставете празни скоби както е показано. Извикването на функция е оператор и затова завършва с точка и запетая. Когато една функция се извика, изпълнението на програмата продължава с изпълнението на операторите на тази функция.



Извикването на функция е оператор и затова завършва с точка и запетая.

`return 0;` - Операторът return е ключова дума, която предизвиква прекратяване на изпълнението на функцията, от която е извикан, в случая изпълнението на функцията main() се прекратява. Стойността 0 след return се връща от **main()** като резултат от нейното изпълнение. В случая **main()** се извиква от операционната система при стартиране на програмата и връща стойност 0 при завършването си. Стойност 0 указва на операционната система, че програмата е завършила изпълнението си

успешно.

`/* Дефиниция на функцията myfunc */` - Този коментар указва, че следва описанието на функцията myfunc. Описанието или дефиницията на една функция се състои от заглавие на функцията и тяло на функцията.

`void myfunc(void)` – Това е заглавие на функцията myfunc. Заглавието на една функция е същото като прототипа, само че без точка и запетая. То е последвано от тялото. Тялото на една функция съдържа нейните оператори и е затворено между { и }.

`/* Отпечатай съобщение на дисплея */` - Този коментар пояснява действието на оператора след него.

`printf("First C program\n");` - printf е една от стандартните библиотечни функции в езика C. Тя служи за извеждане на информация на стандартния изход, който по подразбиране се явява екранът. Тази функция е доста гъвкава, но като начало запомнете, че ако искате да изведете съобщение на екрана, поставете това съобщение, затворено в кавички, в скобите след printf. Ще забележите, че последователността от символи \ и n не се отпечатва на екрана. В C последователността **\n** се възприема като един символ, който води до преместване на курсора на нов ред. В глава 11 Извеждане на данни на екрана ще научите повече за това.

`/* Извикай библиотечната функцията system */` - Този коментар пояснява действието на оператора след него.

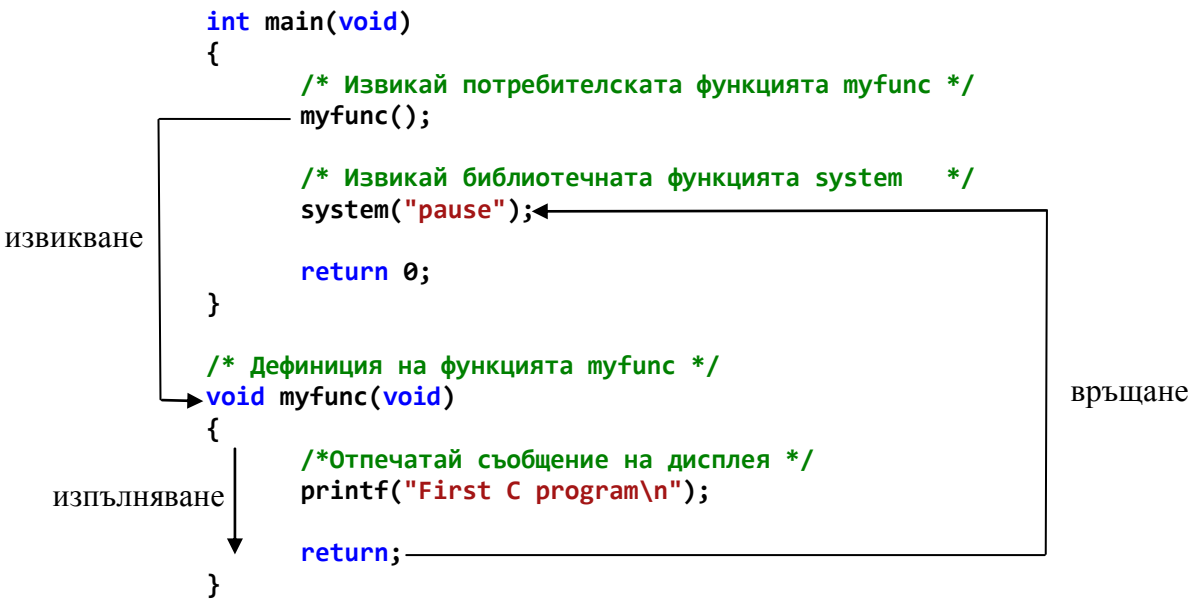
`system("pause");` - Извиква библиотечната функция **system()**. В случая на тази функция се предава като аргумент[1] низ[2] "pause", което предизвиква временно спиране на изпълнение на програмата до натискане на произволен клавиш. В противен случай, програмата ще се изпълни много бързо и ще се затвори, и няма да видите резултата.

Забележка[1]: Стойностите, които се подават на функциите при тяхното извикване, се наричат аргументи.

Забележка[2]: В C последователност от символи, затворени в двойни кавички, се нарича низ (**string**).

`return;` - Завършване на функцията myfunc. Тъй като myfunc не връща стойност, след return няма указана такава. След завършване на изпълнението на една функция, програмата продължава с изпълнението на оператора, намиращ се непосредствено след извикването на функцията. В нашия случай това е операторът `system("pause");`. Механизмът на извикване и завършване на една функция изглежда схематично по следния

начин.



В най-простия случай една С програма може да се състои само от главната функция **main()**.

Пример 8-2:

```
/* Включване на библиотечни хедър-файлове */
#include <stdio.h>
#include <stdlib.h>

/* Входна точка на програмата */
int main(void)
{
	/*Отпечатай съобщение на дисплея */
	printf("First C program\n");

	/* Извикай библиотечната функцията system	*/
	system("pause");

	return 0;
}
```

Както ще се убедите, резултатът от тази програма е същият като предходния. В повечето случаи Вашите програми ще включват и други функции, написани от Вас.

Друг важен момент, на който трябва да се обърне внимание, е подредбата на кода. Както се вижда от примерите, операторите се намират на някакъв отстъп навътре от отварящата фигурна скоба на функцията, също между отделните оператори има празни редове. Това се прави единствено за нагледност на кода, а не поради синтактични изисквания на езика, т.е. . следния код е синтактично верен, но не толкова нагледен.

```
int main(void)
{
```

```
/*Отпечатай съобщение на дисплея */
printf("First C program.");
return 0;
}
```

или

```
int main(void)
{
/*Отпечатай съобщение на дисплея */
printf("First C program.");return 0;
}
```

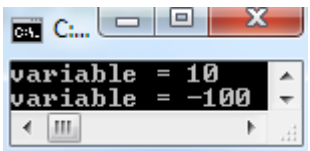
Във втория пример е показано, че на един ред могат да се поставят няколко оператора един след друг. Това е възможно, тъй като компилаторът разпознава край на оператор по точката и запетаята след него. Това позволява след първия оператор на същия ред да се постави втори и т.н.

8.2 Променливи и константи

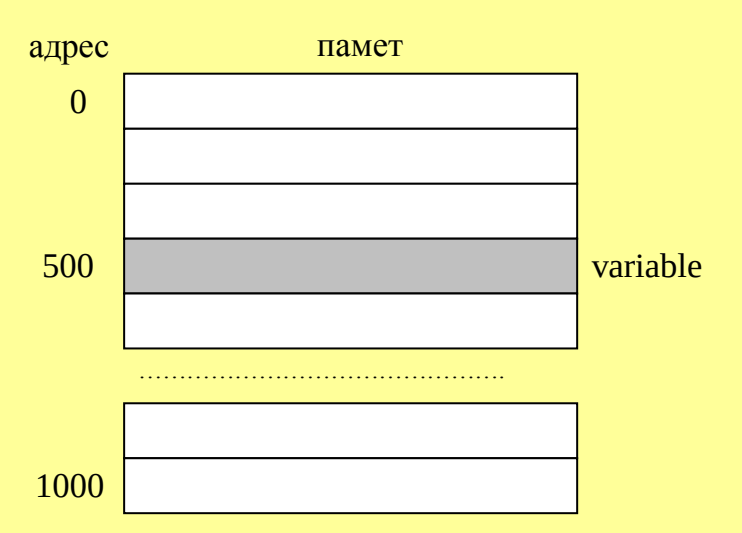
Предходните примери отпечатват кратко съобщение на екрана. Нека сега да напишем една проста програма, която използва някакви стойности. Стойностите се съхраняват в области от паметта на компютъра, наречени променливи. Самият термин "променлива" подсказва, че стойността, която се съхранява в нея, може да се променя.

Пример 8-3:

```
#include <stdio.h>

int main(void)
{
    int variable;  /* Заделяне на памет за променлива variable */

    variable = 10; /* На variable се присвоява стойност 10     */
    printf("variable = %d\n", variable); /* Отпечатай variable */
    variable = -100; /* На variable се присвоява нова стойност -100 */
    printf("variable = %d\n", variable); /* Отпечатай variable */

    return 0;
}
```



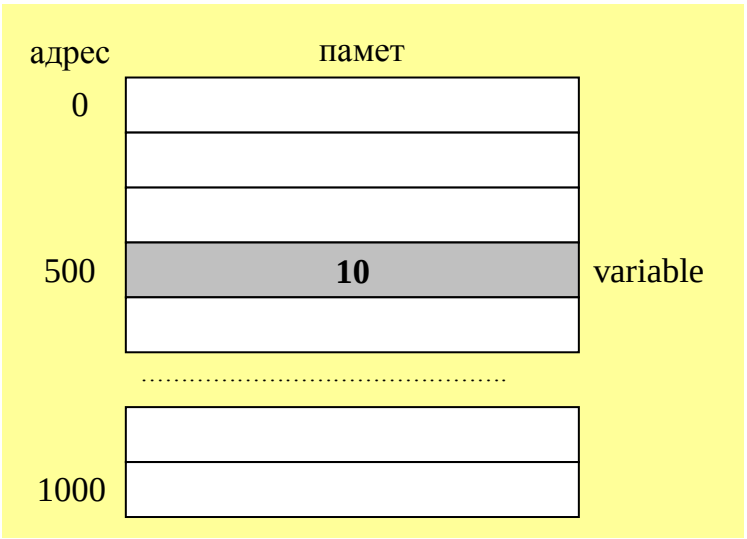
Нека разгледаме тази програма по-отблизо. Конструкцията **int variable;** се нарича декларация на променлива и принуждава компилатора да задели част от паметта за тази променлива и да асоциира адреса на тази памет с името variable (Фиг.27).



Фиг. 27 Заделяне на памет за променлива

Всяка променлива има тип, който указва какви стойности може да се съхраняват в нея. Променливата variable има тип int, който указва, че в променливата могат да се съхраняват само цели числа (положителни и отрицателни).

Конструкцията **variable = 10;** записва стойност 10 в variable (Фиг.28).

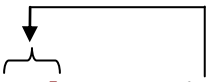


Фиг. 28 Присвояване на стойност на променлива

Числото 10 е десетична целочислена константа.

Конструкцията **printf("variable = %d\n", variable);** отпечатва стойността на променливата variable на екрана в десетичен формат. Тъй като още не сте запознати с т.нар. форматни спецификатори, нека да разгледаме тази конструкция малко по-подробно. Както споменах в началото, не всички символи, затворени в двойни кавички, се отпечатват от printf() на екрана така, както се виждат. Последователността **%d** се нарича форматен спецификатор и е указание, че стойността на целочислената променлива

variable, вдясно от запетаята, трябва да се отпечата на негово място, т.е. .



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

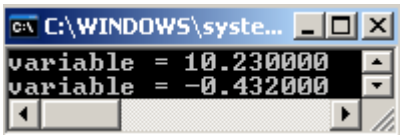
Форматните спецификатори са разгледани по-подробно в 11 Извеждане на данни на екран.

Останалите конструкции вече би трябвало да са ясни. На variable се присвоява друга стойност (-100), след което се отпечатва на екрана.

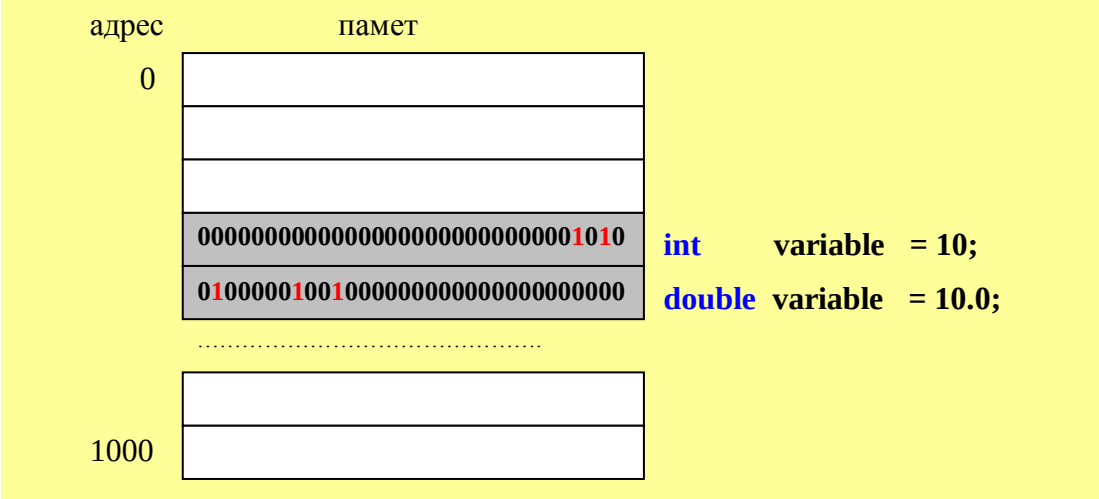
Сега ще Ви запозная с още един тип, който позволява в него да се съхраняват дробни числа.

Пример 8-4:

```
#include <stdio.h>

int main(void)
{
    double variable;      /* Заделяне на памет за променлива variable */

    variable = 10.23;     /* На variable се присвоява стойност 10.23  */
    printf("variable = %f\n", variable); /* Отпечатай variable        */

    variable = -0.432; /* На variable се присвоява нова стойност -0.432 */
    printf("variable = %f\n", variable); /* Отпечатай variable          */

    return 0;
}
```



Тази програма е аналогична на предходната, с тази разлика, че променливата variable е тип double, а не int. В такива променливи можем да съхраняваме дробни числа. С използва формат с плаваща запетая за съхранение на дробните числа. Това означава че десетичното цяло число 10 и десетичното дробно число 10.0 ще имат различно двоично представяне в паметта на компютъра. Ако приемем, че компютърът използва 32 бита за съхранение на цели числа тип int и стандарт IEEE 754 с единична точност за съхраняване на тип double, то следващата фигура показва нагледно вътрешното битово представяне на тези две стойности.

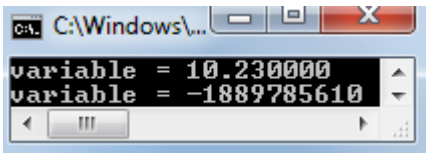


Фиг. 29 Вътрешно представяне на тип int и double

Забележете, че форматният спецификатор за отпечатване на стойности тип double е **%f**. Той указва на функцията printf(), че променливата variable е тип double и ще трябва да интерпретира вътрешното представяне съгласно използвания формат за кодиране на дробни числа. Ако използвате спецификатора **%d**, за да отпечатате число с плаваща запетая, ще бъдете изненадани от резултата.

Пример 8-5:

```
#include <stdio.h>

int main(void)
{
    double variable;      /* Заделяне на памет за променлива variable */

    variable = 10.23;     /* На variable се присвоява стойност 10.23  */
    printf("variable = %f\n", variable); /* Отпечатай variable        */
    printf("variable = %d\n", variable); /* !!!Отпечатай variable     */

    return 0;
}
```



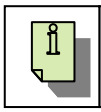
В Пример 8-5 второто извикване на printf() дава доста странен резултат. Причината е, че функцията printf() приема, че числото, което трябва да се отпечата, е тип int и интерпретира вътрешното му представяне съгласно използвания формат за кодиране на знакови цели числа (най-често допълнителен код).

Може да декларирате няколко променливи от даден тип наведнъж. Например конструкцията **int x, y;** декларира променливи x и y от тип int.

8.3 Запознаване с функциите

Функцията е парче код, което се изпълнява при извикването й. Вече познавате библиотечната функция printf(). В тази точка ще Ви запозная с писането на Ваши собствени функции. В Пример 8-1 Ви показах функция, която не приема стойност и не връща резултат. Сега ще Ви запозная с функции, които приемат стойности и връщат резултат.

Когато една функция приема стойност, това трябва да се укаже на компилатора. За целта в скобите след името на функцията указваме типа и името на променливата, която ще приеме тази стойност. Когато създаваме собствена функция, ние създаваме две неща. Първо, създаваме самата функция, което включва заглавие на функцията и нейното тяло. Това се нарича дефиниция на функция. Дефиницията на една функция се разполага извън всички други функции, т.е. . не може в тялото на една функция да дефинирате друга функция. Може да разположите дефиницията преди или след функцията main() или дори в отделен файл, но това ще стане ясно, когато официално разгледаме функциите в 22 Функции. Второто нещо, което създаваме, е декларация на функцията. Декларацията на функция се нарича още прототип. Прототипът изглежда като заглавието на функцията, но няма тяло и завършва с точка и запетая. Ролята на прототипа е да покаже на компилатора какви стойности приема функцията и какъв резултат връща или дали изобщо връща такъв. На базата на тази информация компилаторът следи дали функцията се извиква с правилния тип и брой стойности и дали типът на връщания от функцията резултат отговаря на указания в прототипа. Ако компилаторът "забележи" някаква нередност, той ще Ви уведоми за това като генерира съобщение за грешка, когато се опитате да компилирате кода. Прототипът трябва да се разположи преди извикването на функцията в кода. Обикновено това става в началото на сорс-файла. Стойността, която се подава на функция при нейното извикване, се нарича аргумент, а променливата, която я приема, се нарича параметър.



Стойността, която се подава на функция, се нарича аргумент, а променливата, която я приема, се нарича параметър.

Променливите, декларирани в една функция, се наричат локални. Те могат да се използват само в тялото на функцията и не са достъпни извън него. Обърнете внимание, че параметрите на функцията също се явяват локални променливи.

Нека сега да илюстрираме описаното до тук с един прост пример:

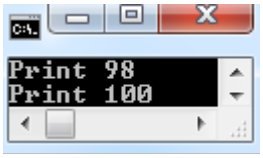
Пример 8-6:

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

```c
/* Прототип на функцията print_integer */
void print_integer(int x);

int main(void)
{
   int value; /* Деклариране на локална променлива value */

   value = 98; /* Присвояване на стойност на value */
   print_integer(value);/* Използване на value като аргумент на print_integer() */
   print_integer(100);  /* Използване на константата 100 като аргумент на
                           print_integer() */
   return 0;
}

/* Дефиниция на функцията print_integer */
void print_integer(int x)
{
      printf("Print %d\n", x);

      return;
}
```

Заглавие на функция

Тяло на функция



Фиг.30 показва какво се случва, когато подаваме аргумент на функция по време на извикването.

```c
        print_integer(value);
                              ||
void print_integer(int x)
{
    printf("Print %d\n",x);

    return;
}
```

```c
        print_integer(100);
                              ||
void print_integer(int x)
{
    printf("Print %d\n",x);

    return;
}
```

Фиг. 30 Подаване на стойност на функция

Аргументът може да бъде променлива, константа или както ще видим по-нататък израз, състоящ се от променливи и константи. Във всички случаи стойността на аргумента се присвоява на параметъра.

Всякакви промени на параметъра в тялото на функцията не оказват никакво влияние на аргумента. Функцията print_integer() не връща резултат. Това е указано с ключовата дума void пред името на функцията. Такива функции завършват изпълнението си с извикване на оператора return без никаква стойност след него. Ако операторът return се изпусне, затварящата фигурна скоба на функцията действа аналогично на return, т.е. . следният код е абсолютно допустим.

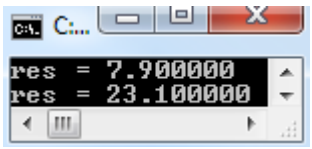
```c
void print_integer(int x)
```

```
{
      printf("Print %d\n", x);
}/* Затварящата скоба на void-функциите действа като оператор return */
```

Когато функция има два или повече параметри, те се разделят със запетая помежду си. При извикването, аргументите също се разделят със запетая един от друг. Следващият пример илюстрира функция, която приема два аргумента и връща резултат:

Пример 8-7:

```
#include <stdio.h>

/* Прототип на функцията add */
double add(double a, double b);

int main(void)
{
   double res;

   res = add(2.3, 5.6);
   printf("res = %f\n", res);
   res = add(10.2, 12.9);
   printf("res = %f\n", res);

   return 0;
}

/* Дефиниция на функцията add */
double add(double a, double b)
{
   double sum;

   sum = a + b;

   return sum; /* Стойността на локалната променлива sum се връща като резултат */
}
```

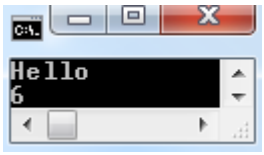


Функцията add() сумира стойностите, подадени на двата параметъра, съхранява резултата в локална променлива sum и връща стойността на sum като резултат. Стойността, връщана от една функция, може да се присвои на променлива, както е в нашия пример или просто да се игнорира. Например функцията printf() връща като резултат броя на изведените на екрана символи.

Пример 8-8:

```
#include <stdio.h>

int main(void)
{
```

```
        int DisplayedSymbols;

        DisplayedSymbols = printf("Hello\n"); /* Връщаният резултат се използва */
        printf("%d\n", DisplayedSymbols); /* Връщаният резултат не се използва */

        return 0;
}
```



Както забелязвате, резултатът от първото извикване на printf() се съхранява в променлива DisplayedSymbols, докато при второто извикване не се интересуваме от него и той просто се губи.

Стойността, която се връща с оператора return, може да бъде константа, променлива или както ще видим по-нататък израз, състоящ се от променливи и константи. Една функция може да съдържа повече от един оператор return.

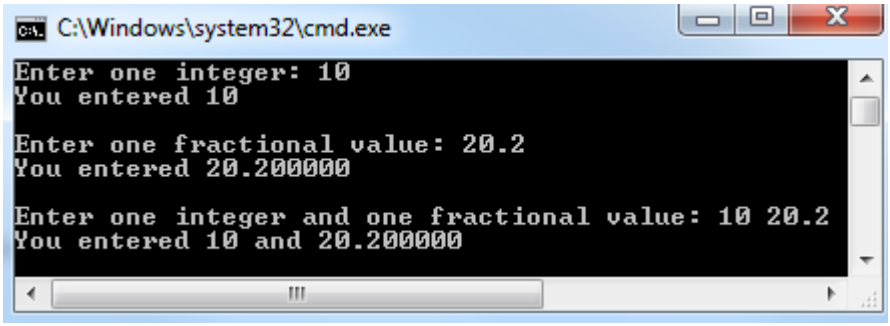
Нека да обобщим какво се случва, когато една функция се извика и като завърши изпълнението си. Когато една функция се извика, изпълнението на програмата се прехвърля в нея. Ако функцията има параметри, то аргументите, с които тя е извикана, се копират в тях. Функцията завършва изпълнението си при достигане на оператор return и програмата продължава с изпълнението на първия оператор след точката на извикване на функцията.

Преди да завършим, ще Ви запозная с библиотечната функция scanf(), с чиято помощ можете да четете данни, въведени чрез клавиатурата.

Пример 8-9:

```
#include <stdio.h>

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

   printf("Enter one integer: ");
   scanf("%d", &i); /* Четене на цяло число от клавиатурата */
   printf("You entered %d\n\n", i);

   printf("Enter one fractional value: ");
   scanf("%lf", &d); /* Четене на дробно число от клавиатурата */
   printf("You entered %f\n\n", d);

   printf("Enter one integer and one fractional value: ");
   scanf("%d%lf", &i, &d); /* Четене на едно цяло и едно дробно число от
                              клавиатурата */
   printf("You entered %d and %f\n\n", i, d);
```

```
    return 0;
}
```



Функцията scanf() чете въведените от клавиатурата данни и ги съхранява в променливи, указани при нейното извикване, т.е. .

scanf("%d%lf", &i, &d);

Когато въвеждате няколко стойности от клавиатурата те трябва да бъдат разделени от поне един интервал. Действителното четене на данните става при натискане на бутона ENTER. Използването на scanf() е аналогично на printf(). Първият аргумент е низ, който съдържа форматни спецификатори, които указват на функцията какъв тип данни ще се въвеждат от клавиатурата. Когато въвеждате цели числа, форматният спецификатор е **%d**, а когато въвеждате дробно число тип double, форматният спецификатор е **%lf** (внимавайте, функцията printf() използва **%f**, когато извежда числа тип double).

Забележете друга важна разлика в сравнение с printf(). Променливите, в които scanf() ще съхранява прочетените от клавиатурата стойности, се предхождат от символа '&'. Защо това е така, ще стане ясно когато се запознаете с концепцията за указатели в C.

8.4 Аритметика в C

В тази точка ще Ви запозная с няколко аритметични операции в езика C:

- \+ (събиране)
- \- (изваждане)
- * (умножение)
- / (деление)

Синтаксисът на тези операции най-общо изглежда така:

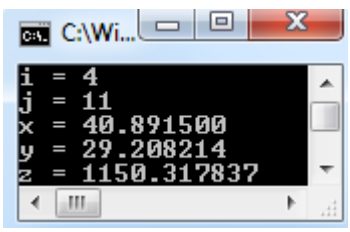
операнд1 + операнд2
операнд1 - операнд2
операнд1 * операнд2
операнд1 / операнд2

Между операндите и операцията може да поставяте произволен брой интервали. Операндите могат да бъдат променливи и константи. Комбинацията от операции и операнди се нарича израз. Всеки израз има стойност, затова изразите могат да се използват навсякъде, където е допустимо използването на стойност например в дясната страна на операцията за присвояване '=', като аргумент на функция, като резултат, връщан от функция и др. Когато израз включва няколко операции, може да използвате скоби, за да поясните реда на изчисление на операциите в него.

Пример: 10 - (2 * 5) = 0, но (10 - 2) * 5 = 40

Пример 8-10:

```
#include <stdio.h>

double multiply (double a, double b);

int main(void)
{
   int i, j; /* Деклариране на две променливи тип int */
   double x, y, z; /* Деклариране на три променливи тип double */

   i = 4;
   j = i + 7; /* Стойността на израза i+7 се присвоява на j */
   x = 9.087;
   x = x * 4.5; /* Стойността на израза x*4.5 се присвоява на x */
   y = x / 1.4; /* Стойността на израза x/1.4 се присвоява на y */
   printf("i = %d\nj = %d\nx = %f\ny = %f\n", i, j, x, y);
   z = multiply(x + 3, y - 3); /* Използване на изразите x+3 и y-3 за аргументи */
   printf("z = %f\n", z);

   return 0;
}

double multiply (double a, double b)
{
    return a * b; /* Връщане на стойността на израза a * b като резултат */
}
```



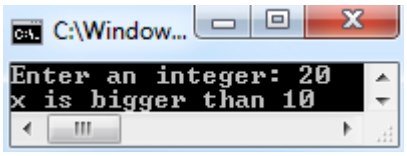
8.5 Вземане на решения

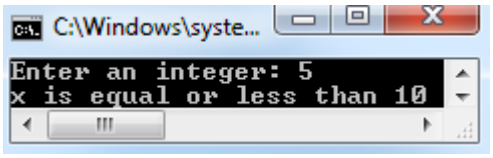
Всички предходни примери се състоят от последователно изпълняване на операторите, изграждащи програмата. Повечето програми обаче ще трябва да вземат решения в определени моменти кои оператори да изпълнят и кои не за постигане на крайната цел. Езикът С предоставя два оператора: **if-else** и **switch**, с чиято помощ посоката на изпълнение на програмата може да се управлява. Те се наричат още **оператори за избор** или **оператори за условен преход.**

Ще започнем с оператора if-else на базата на следващия пример:

Пример 8-11:

```
#include <stdio.h>

int main(void)
{
   int x; /* Деклариране на променлива x от тип int */

   printf ("Enter an integer: "); /* Извеждане на подсказващо съобщение */
   scanf("%d", &x); /* Прочитане на въведената стойност от клавиатурата */

    условие на if
   if (x > 10)
   {
         printf("x is bigger than 10\n");        тяло на if
   }
   else
   {
          printf("x is equal or less than 10\n");        тяло на else
   }

   return 0;
}
```





Когато програмата срещне конструкцията **if (x > 10)**, тя проверява дали условието в скобите е изпълнено. В случая се проверява дали стойността на x е по-голяма от 10. Ако това е така, програмата изпълнява операторите, затворени във фигурните скоби непосредствено след if, след което прескача операторите, затворени във фигурни скоби след else и продължава с първия оператор след if-else. Ако условието не е изпълнено, програмата прескача тялото на if и изпълнява операторите в тялото на else, след което продължава с първия оператор след if-else.

Операцията > е част от операциите за сравнение в езика С. В Табл.11 са

дадени всички операции за сравнение заедно с кратко описание.

операция за сравнение	описание
<	по-малко
<=	по-малко или равно
>	по-голямо
>=	по-голямо или равно
==	равно
!=	различно

Табл. 11 Оператори за сравнение

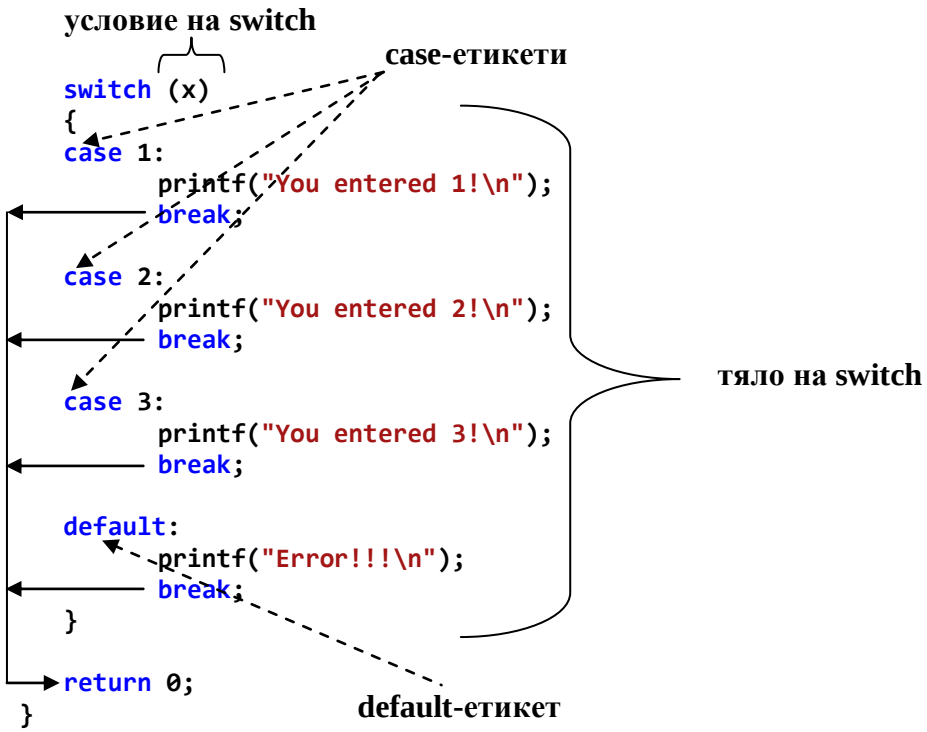
Всички тези операции имат следния общ синтаксис:

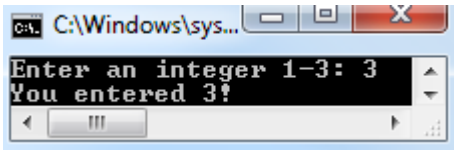
операнд1 **операция-за-сравнение** операнд2

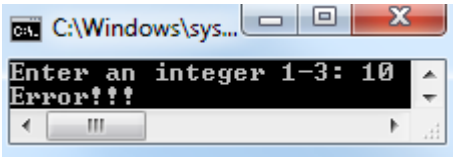
Може да използвате коя да е от тези операции в условието на оператора if-else, за да формирате различни условия. Следващият пример демонстрира оператора за избор switch:

Пример 8-12:

```
#include <stdio.h>

int main(void)
{
   int x; /* Деклариране на променлива x от тип int */

   printf("Enter an integer 1-3: "); /* Извеждане на подсказващо съобщение на
                                        екрана */
   scanf("%d", &x); /* Прочитане на въведената стойност от клавиатурата */

   switch (x)
   {
   case 1:
         printf("You entered 1!\n");
         break;

   case 2:
         printf("You entered 2!\n");
         break;

   case 3:
         printf("You entered 3!\n");
         break;

   default:
         printf("Error!!!\n");
         break;
   }

   return 0;
}
```







Когато програмата срещне конструкцията **switch (x)**, тя сравнява стойността на израза в скобите с константите, указани с етикетите **case**. Ако се срещне съвпадение, програмата се прехвърля за изпълнение на операторите принадлежащи на този case-етикет, прескачайки операторите на всички останали case-етикети. Ако не се открие съвпадение, програмата се прехвърля за изпълнение на операторите принадлежащи на default-етикета. Изпълнението продължава, докато се срещне оператора break. Този оператор принуждава програмата да излезе от тялото на switch и да продължи с изпълнението на първия оператор след него. Условието на оператор switch трябва да бъде целочислен израз, съответно константите след всеки case-етикет също трябва да са цели числа.

8.6 Повторение на код

Сега ще Ви запозная с два други C оператора, които се наричат оператори за повторение или за цикъл. Те принуждават програмата да изпълнява една и съща част от кода определен брой пъти в зависимост от някакво условие.

Ще започнем с оператора while на базата на следващия пример:

Пример 8-13:

```
#include <stdio.h>

int main(void)
{
   int x; /* Деклариране на променлива x от тип int */

   x = 0; /* Присвояване на стойност 0 на x         */

   while (x < 10)
   {
      printf("%d ", x);
      x = x + 1;
   }

   printf("\n");

   return 0;
}
```

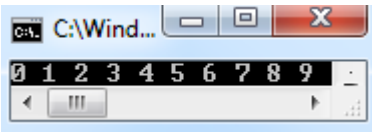
условие на while

тяло на while

/* Отпечатване на стойността на x */
/* Увеличаване на x с 1   */

условието е изпълнено

условието не е изпълнено

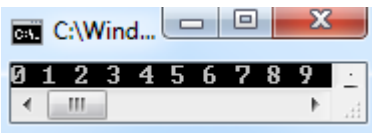


Когато програмата срещне конструкцията **while (x < 10)**, тя проверява дали условието в скобите е изпълнено. Ако това е така, програмата изпълнява операторите, затворени във фигурните скоби непосредствено след while, след което отново проверява условието на while. Този процес се повтаря, докато условието на цикъла е изпълнено, в противен случай програмата спира да изпълнява операторите в тялото на while и се прехвърля за изпълнение на първия оператор след него.

Следващият пример демонстрира друг оператор за повторение, наречен do-while:

Пример 8-14:

```
#include <stdio.h>

int main(void)
{
   int x; /* Деклариране на променлива x от тип int */

   x = 0; /* Присвояване на стойност 0 на x         */

   do                                    условието е изпълнено
   {
      printf("%d ", x);                  тяло на do-while /* Отпечатване на стойността на x */
      x = x + 1;                                          /* Увеличаване на  x с 1          */
   } while (x < 10);                     условието не е изпълнено
        условие на do-while
   printf("\n");

   return 0;
}
```

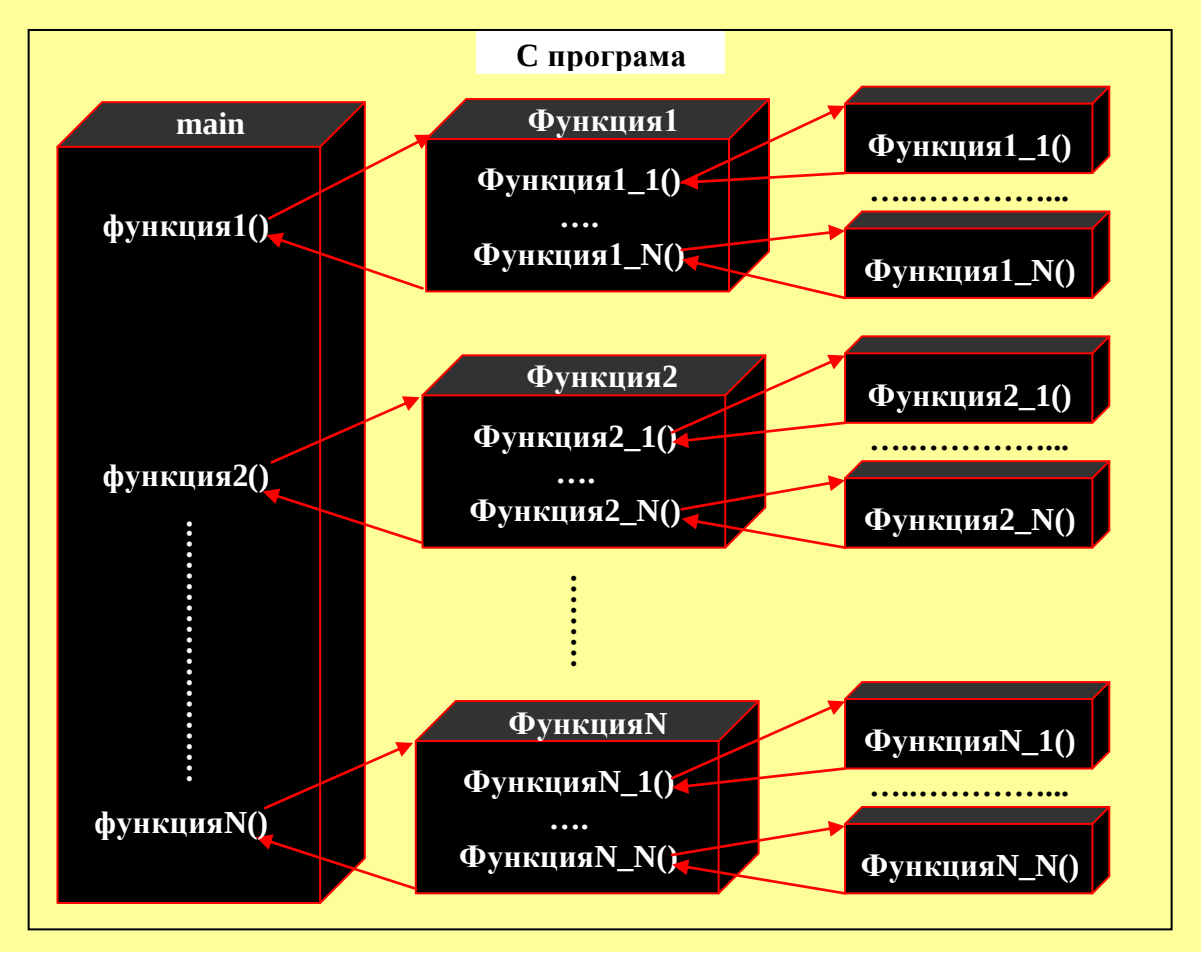


Когато програмата срещне конструкцията **do**, тя изпълнява операторите затворени във фигурните скоби непосредствено след do, след което проверява условието на цикъла **while (x < 10)**. Ако условието е изпълнено, програмата изпълнява отново операторите в тялото на цикъла do-while. Този процес се повтаря, докато условието на цикъла е изпълнено, в противен случай програмата се прехвърля за изпълнение на първия оператор след тялото на do-while.

Единствената разлика между операторите while и do-while е, че тялото на do-while се изпълнява поне веднъж. Обърнете внимание, че променливата, която участва в условието на цикъла, се променя в неговото тяло. Ако това не се правеше, условието винаги щеше да е вярно и програмата щеше да изпълнява операторите в тялото на цикъла безкрайно.

8.7 И така какво е С програма

След като Ви запознах с някои елементи на езика С, нека да направим едно по-пълно обобщение на това какво представлява една С програма. Градивните елементи на една С програма се наричат функции. Всяка функция по същество е подпрограма, която изпълнява конкретна задача. Градивните елементи на функциите се наричат оператори. Операторите са инструкции, които извършват някакви действия. С предоставя разнообразие от оператори като оператор за създаване на променлива, за присвояване на стойност на променлива, за извикване на функция, за преход, за повторение и др. Всяка С програма започва изпълнението си от една специална функция, наречена main(). Тя може да извиква други функции (написани от Вас или библиотечни функции като printf()). Те от своя страна също могат да извикват други функции и т.н. (Фиг.31).



Фиг. 31 Обобщена структура на С програма

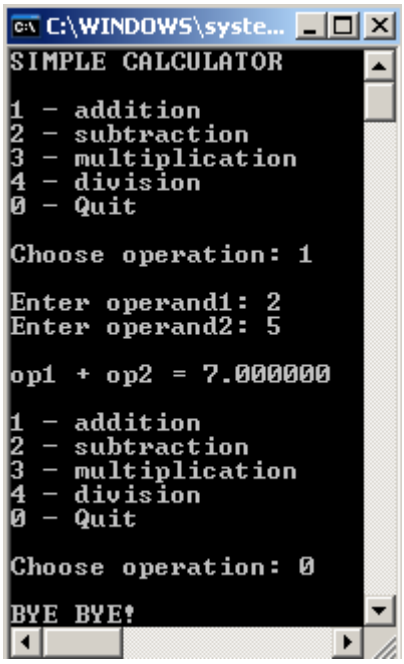
Следващият пример представлява прост калкулатор. Той съдържа повечето разгледани досега конструкции на езика C:

Пример 8-15:

```
#include <stdio.h>

/* Прототипи на функции */
void add(void);
void sub(void);
void mul(void);
void div(void);

/* Входна точка на програмата */
int main(void)
{
	/* Локални променливи */
	int choice;

	/* Отпечатване на кратка информация за програмата */
	printf("SIMPLE CALCULATOR\n\n");

	do
	{
		/* Отпечатване на възможните функции на калкулатора */
		printf("1 - addition\n");
		printf("2 - subtraction\n");
		printf("3 - multiplication\n");
		printf("4 - division\n");
		printf("0 - Quit\n\n");
		printf("Choose operation: ");
		scanf("%d", &choice); /* Прочитане на въведения избор */
		printf("\n");

		if(choice != 0)
		{
		    /* Извършване на избраната аритметична операция */
		    switch (choice)
		    {
		    case 1:
		      add(); /* Събиране */
		      break;
		    case 2:
		      sub(); /* Изваждане */
		      break;
		    case 3:
		      mul(); /* Умножение */
		      break;
		    case 4:
		      div(); /* Деление */
		      break;
		    default:
		      printf("ERROR: Choose between 1 and 4 or 0 to Quit\n\n");
		      break;
		   }/* край на оператора switch*/
		}
		else
		{
			printf("BYE BYE!\n");
		}

	}while (choice != 0);
```

```
	return 0;
}/* Край на main */

/* Дефиниции на функции */
void add(void)
{
	/* Локални променливи */
	double op1;
	double op2;
	double result;

	/* Четене на операндите от клавиатурата */
	printf("Enter operand1: ");
	scanf("%lf", &op1);
	printf("Enter operand2: ");
	scanf("%lf", &op2);

	/* Извършване на операцията събиране */
	result = op1 + op2;

	/* Отпечатване на резултата */
	printf ("\nop1 + op2 = %f\n\n", result);
}

void sub(void)
{
	/* Локални променливи */
	double op1;
	double op2;
	double result;

	/* Четене на операндите от клавиатурата */
	printf("Enter operand1: ");
	scanf("%lf", &op1);
	printf("Enter operand2: ");
	scanf("%lf", &op2);

	/* Извършване на операцията изваждане */
	result = op1 - op2;

	/* Отпечатване на резултата */
	printf ("\nop1 - op2 = %f\n\n", result);
}

void mul(void)
{
	/* Локални променливи */
	double op1;
	double op2;
	double result;

	/* Четене на операндите от клавиатурата */
	printf("Enter operand1: ");
	scanf("%lf", &op1);
	printf("Enter operand2: ");
	scanf("%lf", &op2);

	/* Извършване на операцията умножение */
	result = op1 * op2;

	/* Отпечатване на резултата */
	printf ("\nop1 * op2 = %f\n\n", result);
}
```

```
void div(void)
{
	/* Локални променливи */
	double op1;
	double op2;
	double result;

	/* Четене на операндите от клавиатурата */
	printf("Enter operand1: ");
	scanf("%lf", &op1);
	printf("Enter operand2: ");
	scanf("%lf", &op2);
	/* Извършване на операцията деление */
	result = op1 / op2;

	/* Отпечатване на резултата */
	printf ("\nop1 / op2 = %f\n\n", result);
}
```



8.8 Ключови думи

Стандартът C89(C90) дефинира 32 ключови думи (Табл.12). Те се изписват с малки букви и не могат да се използват за имена на променливи, функции и др.

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. 12 Ключови думи в C89/C90

С част от тези ключови думи се запознахте вече. С останалите ще Ви запозная в следващите глави.

Въпроси за самопроверка

1. Как се казва главната функция, която задължително трябва да присъства във всяка C програма?

2. Ако имате функция с име func, която не приема и не връща стойност, покажете как се извиква в кода.

3. Обяснете как протича изпълнението на програмата при извикване на една функция.

4. Обяснете как протича изпълнението на програмата при завършване на изпълнението на една функция.

5. Как се създава коментар и за какво служи?

6. Довършете следното твърдение: Всеки C оператор трябва да завършва с...

9 Променливи и типове данни в C

9.1 Определение за променлива

В тази глава ще Ви запозная официално с термина "променлива". Променливата е област от паметта на компютъра, която се използва за съхранение на данни. В C една променлива трябва да бъде дефинирана (обявена) преди да бъде използвана някъде в кода. Общата форма на дефиниране на променлива е:

тип име-променлива;

или

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

където

тип
Може да бъде всеки един от типовете в езика C (вижте 9.2 Типове)

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

Пример:

```
int i; /* Дефиниране на една променлива i от тип int */
int x, y, z; /* Дефиниране на три променливи x, y и z от тип int */
double a, b, c; /* Дефиниране на три променливи a, b и c от тип double */
```

Името на една променлива може да се състои от следните символи:

- малки и големи букви от латинската азбука (**a÷z** и **A÷Z)**
- арабски цифри (**0÷9)**
- подчертаващо тире (**_**)

Името на променливата НЕ може да започва с цифра.

Пример:

var1 - валидно име
_var2 - валидно име
3var - невалидно име

Също така C прави разлика между малки и главни букви, т.е. . var и Var се

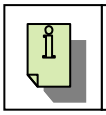
възприемат като две различни имена за С компилатора.

В С дефиницията на променлива е оператор, който води до заделяне на памет за променливата и затова завършва с точка и запетая. Типът на една променлива носи следната информация:

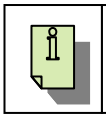
- вида на данните, които тази променлива може да съхранява (цели числа, числа с плаваща запетая, символи и т.н.), а от тук и размера памет, необходим за заделяне за тази променлива;

- 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).

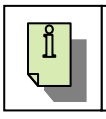
Най-общо казано променливите могат да се дефинират на две места: извън всички функции или в тялото на функция. Променливите, дефинирани извън всички функции, се наричат глобални. Те са достъпни за всички функции в програмата. Променливите, обявени вътре във функция, са локални и могат да се използват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

9.2 Типове

Езикът С включва следните типове:

- базови типове (описани са в следващите точки: 9.2.1 до 9.2.5)

- масиви (описани са в 16 Масиви)

- указатели (описани са в 17 Указатели)

- структури (описани са в 18 Структури)

- обединения (описани са в 19 Обединения)

- изброявания (описани са в 21 Изброявания)

9.2.1 Базови типове данни

Езикът C дефинира няколко базови (вградени в самия език) типове данни. Табл.13 изброява тези типове заедно с кратко описание.

тип	описание	пример
char	Използва се за дефиниране на променливи, които могат да съхраняват символи. Символите в C 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 знакови 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 дробни числа и има почти двойно по-голяма точност.	125464.986537 -1.003455
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 Вече имате представа какво е функция. С указателите ще се запознаете в 17 Указатели.	void foo(void); void *p;

Табл. 13 Базови типове данни

Допълнително езикът C дефинира т.нар. модификатори на базовите типове (Табл.14).

модификатор	описание	пример
unsigned	Може да се прилага към базовите типове char и int, за да укаже, че тези типове могат да приемат само неотрицателни 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	long double long int unsigned long int signed long int

	модификаторите unsigned и signed към тип int.	
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. 14 Модификатори на базовите типове

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

short ≈ signed short ≈ short int ≈ signed short int
unsigned short ≈ unsigned short int
int ≈ signed ≈ signed int
unsigned ≈ unsigned int
long ≈ long int ≈ signed long ≈ signed long int
unsigned long ≈ unsigned long int

Езикът C не дефинира точни размери на базовите типове, а само минимални размери. Размерът на типовете зависи от компилатора и компютъра. В таблицата по-долу са дадени размерите на тези типове в битове съгласно стандарта C89.

тип	минимален размер в битове	числов диапазон	коментар
char	8	-127 ÷ +127 или 0 ÷ 255	Стандартът не дефинира дали по подразбиране тип char е знаков, или беззнаков. 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. 15 Размер на базовите типове данни в C

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. 8-битовите компютри са изключение. При тях тип int е 16 бита.

Винаги се обръщайте към документацията на компилатора, който използвате, за точните размери на тези типове. Засега, ако се придържате към посочените минимални размери, няма да имате проблеми независимо от компилатора, който използвате.

Вероятно се питате защо има толкова много типове данни при условие, че тип long double може да побере доста големи положителни и отрицателни числа. Една от причините е за по-ефикасно използване на паметта. Например за програма, която използва само цели знакови числа в диапазона ±100, тип signed char е напълно достатъчен да представи всички тези стойности. Друга причина е свързана с производителността (бързодействието) на програмата. Операциите върху типове с плаваща запетая са по-бавни от тези върху целочислени типове. Въпреки че възможностите и ресурсите, с които разполагат съвременните компютри, омаловажават тези два проблема, при микроконтролерите, които са с доста по-ограничени възможности, те са фактор, с който трябва да се съобразяваме.

Пример за дефиниране на променливи:

```
signed char    sc;
unsigned char  uc;
short          sh; /* същото като short int и signed short int           */
unsigned short us; /* същото като unsiged short int                      */
signed          i; /* същото като signed int и int                       */
unsigned       ui; /* същото като unsigned int                           */
long            l; /* същото като long int, signed long и signed long int */
unsigned long  ul; /* същото като unsigned long int                      */
float           f;
double          d;
long double    ld;
```

Променливи от един и същ тип могат да се обявят по следния начин:

```
char    ch1, ch2; /* Дефиниране на две променливи от тип char    */
int     i1, i2;   /* Дефиниране на две променливи от тип int     */
```

Това е еквивалентно на следния запис:

```
char    ch1;
char    ch2;
int      i1;
int      i2;
```

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

В зависимост от типа данни, които могат да се съхраняват в базовите типове, те се разделят на три групи:

- целочислени типове
- типове с плаваща запетая
- символни типове

9.2.2 Целочислени типове

Целочислените типове се разделят на знакови (**signed**) и беззнакови (**unsigned**). C89 не дефинира начина на кодиране на знаковите цели числа. Повечето компилатори използват допълнителен код (вижте 3 Представяне на знакови числа в двоична бройна система). Табл.16 изброява всички целочислени типове, минималната ширина в битове и диапазона от стойности. Допуска се, че знаковите числа използват допълнителен код за кодиране на отрицателните числа.

категория	тип	ширина в битове	диапазон
беззнакови	**char**[1]	8	0 ÷ 255
	unsigned char	8	0 ÷ 255
	unsigned short	16	0 ÷ 65535
	unsigned int	16	0 ÷ 65535
	unsigned long	32	0 ÷ 4 294 967 295
знакови	**char**[2]	8	-128 ÷ +127
	signed char	8	-128 ÷ +127
	signed short	16	-32768 ÷ +32767
	signed int	16	-32768 ÷ +32767
	signed long	32	-2 147 483 647 ÷ +2 147 483 647

Табл. 16 Целочислени типове

Забележка[1]: За компилатори, при които тип char е еквивалентен на unsigned char.

Забележка[2]: За компилатори, при които тип char е еквивалентен на signed char.

Следващата таблица показва диапазона от стойности на целочислените типове в средата на Microsoft Visual Studio C++ 2010 Express. Тъй като използваме тази среда за изпълняване на примерите, добре е да знаете как те са имплементирани в нея.

категория	тип	ширина в битове	диапазон
беззнакови	**char**[1]	-	-
	unsigned char	8	0 ÷ 255
	unsigned short	16	0 ÷ 65535
	unsigned int	32	0 ÷ 4 294 967 295
	unsigned long	32	0 ÷ 4 294 967 295
знакови[2]	**char**[1]	8	-128 ÷ +127
	signed char	8	-128 ÷ +127
	signed short	16	-32768 ÷ +32767
	signed int	32	-2 147 483 647 ÷ +2 147 483 647
	signed long	32	-2 147 483 647 ÷ +2 147 483 647

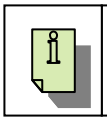
Табл. 17 Целочислени типове в Visual C++ 2010 Express

Забележка[1]: Тип char е signed по подразбиране в тази среда.

Забележка[2]: Знаковите числа се кодират в допълнителен код.

Когато се извършват аритметични операции върху целочислени типове, е възможно резултатът да бъде твърде голям, за да се побере в конкретния целочислен тип. Например, когато аритметична операция се извършва върху две int стойности и резултатът излиза извън диапазона от стойности на тип int, казваме, че е настъпило препълване.

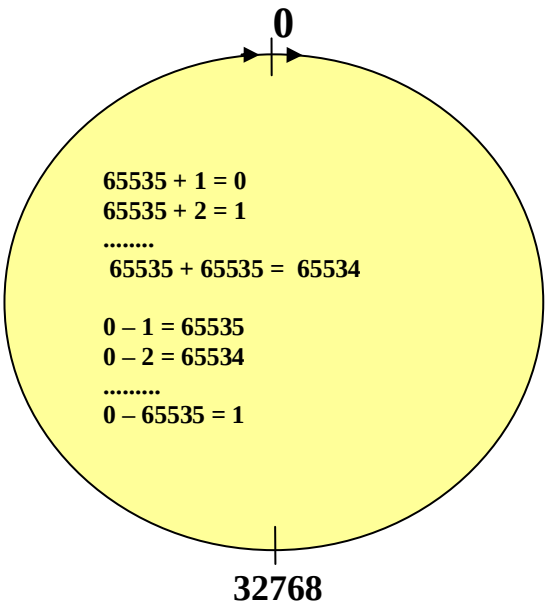
Поведението при целочислено препълване зависи дали операндите са тип signed, или unsigned. Стандартът определя, че ако настъпи целочислено препълване на знакови числа, резултатът е недефиниран.



Резултатът от препълването на знакови цели числа е недефиниран.

Резултатът при препълване на беззнакови числа обаче е ясно дефиниран от стандарта. Следващата фигура показва поведението при препълване на 16

битово беззнаково число.



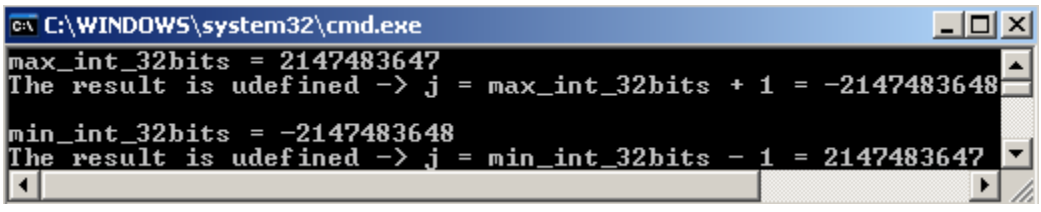
Фиг. 32 Поведение при препълване на беззнакови целочислени типове

Може да направите аналогия с брояча на километри на автомобил. При превъртане (препълване) броячът се нулира и започва от нула.

Следващият пример демонстрира препълване на знакови числа:

Пример 9-1:

```
#include <stdio.h>

int main(void)
{
    signed int max_int_32bits;
    signed int min_int_32bits;
    signed int j;

    max_int_32bits = 2147483647;/* Максимална стойност на 32 битов signed int */
    j = max_int_32bits + 1; /* !!!Препълване с недефиниран резултат */
    printf ("max_int_32bits = %d\n", max_int_32bits);
    printf ("The result is udefined -> j = max_int_32bits + 1 = %d\n\n", j);

    min_int_32bits = -2147483648; /* Минимална стойност на 32 битов signed int */
    j = min_int_32bits - 1; /* !!!Препълване с недефиниран резултат */
    printf ("min_int_32bits = %d\n", min_int_32bits);
    printf ("The result is udefined -> j = min_int_32bits - 1 = %d\n\n", j);

    return 0;
}
```

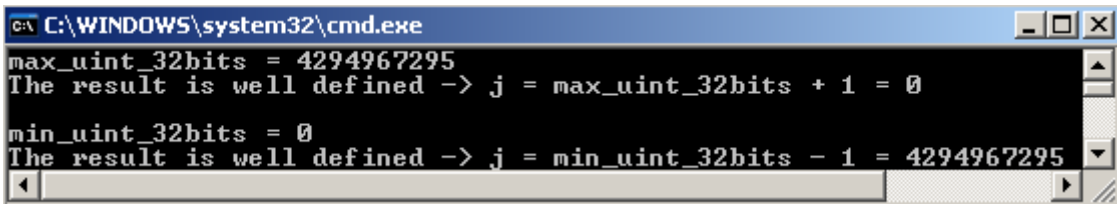


Напомням Ви, че резултатът, съхранен в променливата j, е недефиниран. Други компилатори могат да дадат различен от показания резултат.

Следващият пример демонстрира препълване на беззнакови числа:

Пример 9-2:

```
#include <stdio.h>

int main(void)
{
  unsigned int max_uint_32bits;
  unsigned int min_uint_32bits;
  unsigned int j;

  max_uint_32bits = 4294967295; /* Максимална стойност на 32-битов unsigned int */

  j = max_uint_32bits + 1; /* Препълване с ясно дефиниран резултат */

  printf ("max_uint_32bits = %u\n", max_uint_32bits);
  printf ("The result is well defined -> j = max_uint_32bits + 1 = %u\n\n", j);

  min_uint_32bits = 0; /* Минимална стойност на 32-битов unsigned int */

  j = min_uint_32bits - 1; /* Препълване с ясно дефиниран резултат */

  printf ("min_uint_32bits = %u\n", min_uint_32bits);
  printf ("The result is well defined -> j = min_uint_32bits - 1 = %u\n\n", j);

  return 0;
}
```



Обърнете внимание на използвания форматен спецификатор **%u**. Той се използва, когато отпечатваните на екрана стойности са тип unsigned int.

Когато операндите на една аритметична операция са от тип char (signed или unsigned), препълване не може да се получи. Причината е свързана с това, че тези операнди винаги се преобразуват до тип int. Този въпрос е разгледан подробно в глава 14 Изрази и преобразувания на типове.

9.2.3 Типове с плаваща запетая

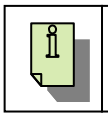
C89 не дефинира стандарта на кодиране на тези типове. Повечето компилатори използват стандарта IEEE 754-1985 и неговия наследник IEEE 754-2008 (вижте 4 Представяне на десетични дробни числа в двоична

бройна система). Табл.18 изброява типовете с плаваща запетая в C. Допуска се, че кодирането е по стандарта IEEE 754.

тип	ширина в битове	диапазон
float	32	$\pm 1.18*10^{-38} \div \pm 3.4*10^{38}$
double	32	$\pm 1.18*10^{-38} \div \pm 3.4*10^{38}$
long double	32	$\pm 1.18*10^{-38} \div \pm 3.4*10^{38}$

Табл. 18 Типове с плаваща запетая

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



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

Следващата таблица показва диапазона от стойности на типовете с плаваща запетая в средата на Microsoft Visual Studio C++ 2010 Express. Използваният стандарт за кодиране е IEEE 754. Тъй като използваме тази среда за изпълняване на примерите, добре е да знаете как те са имплементирани в нея.

тип	ширина в битове	диапазон
float	32	$\pm 1.18*10^{-38} \div \pm 3.4*10^{38}$
double	64	$\pm 2.23*10^{-308} \div \pm 1.80*10^{308}$
long double	64	$\pm 2.23*10^{-308} \div \pm 1.80*10^{308}$

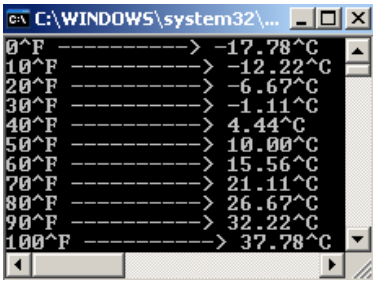
Табл. 19 Типове с плаваща запетая в Visual C++ 2010 Express

Пример 9-3 демонстрира проста програма, която използва числа с плаваща запетая. Програмата преобразува градуси по Фаренхайт (от 0 до 100 със стъпка 10) в градуси по Целзий. Преобразуването се извършва по следната формула $^0C = (5.0/9.0)*(^0F - 32.0)$, където ^0F е температурата по Фаренхайт, а ^0C – по Целзий.

Пример 9-3:

```
#include <stdio.h>

int  main(void)
{
     float fahr, celsius;
     float lower, upper, step;

     lower = 0;      /* Долна граница на температурната скала */
```

```
    upper = 100;    /* Горна граница на температурната скала */
    step  = 10;     /* Размер на стъпката */

    fahr = lower;

    while (fahr <= upper)
    {
        celsius = (5.0/9.0) * (fahr-32.0); /* Изчисляване на температурата */
        printf("%.0f^F ----------> %.2f^C\n", fahr, celsius); /* Отпечатване */
        fahr = fahr + step; /* Увеличаване на температурата с една стъпка */
    }

    return 0;
}
```



Форматните спецификатори **%.0f** и **%.2f** указват колко десетични цифри след десетичната запетая да се отпечатат на екрана. По подразбиране се отпечатват 6 цифри.

9.2.4 Символни типове

Табл.20 изброява символните типове в C.

тип	ширина в битове	диапазон
char	8	ASCII или ASCII + Extended ASCII
signed char	8	ASCII
unsigned char	8	ASCII + Extended ASCII

Табл. 20 Символни типове

Както забелязвате, това са същите 8-битови целочислени типове. Освен за съхраняване на 8-битови цели числа, те се използват и за съхраняване на символи. Символите се затварят в единични кавички, например 'A', 'b', '1', '2' и т.н. Всеки символ се заменя с 8 битов числов код. Този числов код зависи от символната таблица, използвана от компилатора за кодиране на символи. C89 не дефинира използването на конкретна символна таблица. Повечето компилатори използват ASCII (**American Standard Code for Information Interchange**) символната таблица (вижте на http://www.ascii-code.com/). Тя включва 128 символа, от които 95 печатни символа (всички

малки и големи латински букви, десетичните цифри 0÷9 и др.) и 33 управляващи символа (символи, които имат специално значение, като Backspace, Delete и т.н.). Тъй като ASCII символите са 7-битови, повечето компилатори използват Extended ASCII таблицата за кодирани на останалите свободни стойности (от 128 до 255). За разлика от ASCII, Extended ASCII таблицата не е стандартизирана какви символи да включва. Най-разпространените варианти са:

- **ISO 8859-1**, наричан още ISO Latin 1 (съдържа символи, използвани в западноевропейските страни);
- **ISO 8859-2**, наричан още ISO Latin 2 (съдържа символи, използвани в източноевропейските страни);
- **ISO 8859-5**, наричан още Latin/Cyrilic (съдържа символи, използвани в повечето славянски езици като азбуката на кирилица и др.).

Може да използвате следния пример и да отпечатате на екрана символите, съответстващи на числовия диапазон 0÷127.

Пример 9-4:

```
#include <stdio.h>

int  main(void)
{
    unsigned char character;

    character = 0;

    while (character <= 127)
    {
          printf ("The value %d coresponds to symbol %c\n", character, character);
          character = character + 1;
    }

    return 0;
}
```

Форматният спецификатор **%c** се използва за изобразяване на символа, който съответства на подадения числов код.

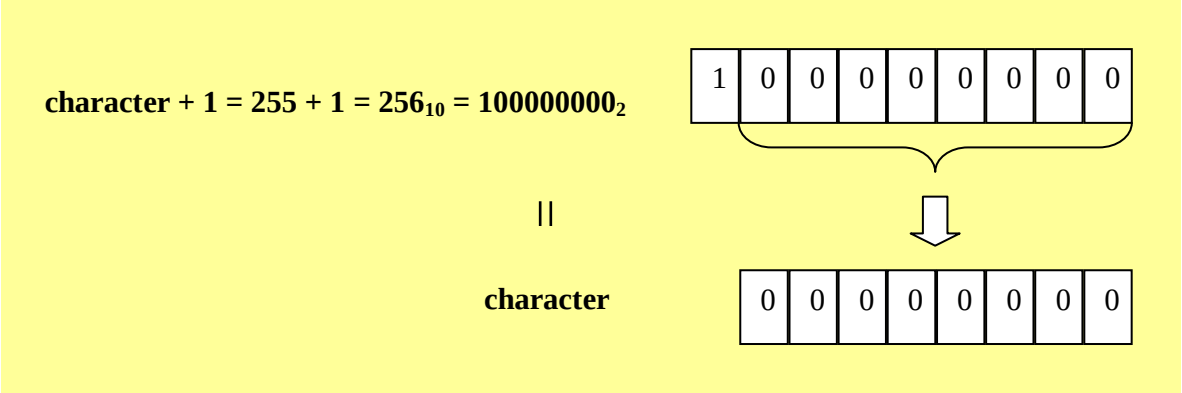
Може да използвате следния пример и да отпечатате на екрана всички символи в числовия диапазон 0÷255, използвани от Вашата система.

Пример 9-5:

```
#include <stdio.h>

int  main(void)
{
    unsigned short character;
```

```
    character = 0;

    while (character <= 255)
    {
          printf ("The value %d coresponds to symbol %c\n", character, character);
          character = character + 1;
    }

    return 0;
}
```

Забележете, че character е дефинирана от тип unsigned short. Това е умишлено, а не по погрешка. Причината е, че ако използваме unsigned char, условието **while (character <= 255)** винаги ще бъде изпълнено, тъй като след отпечатване на последната стойност 255, променливата character се увеличава с 1 и резултатът, който ще се съхрани в нея, ще бъде 0 (Фиг. 33).



Фиг. 33 Илюстрация към Пример 9-5

Както споменах по-рано, аритметични операции с char-типове не водят до препълване. Това означава, че изразът **character + 1** ще даде правилен резултат 256, когато character е 255. В двоичен код стойността 256 се представя с минимум 9 бита (Фиг.33). Тази стойност после се съхранява в character, която е само 8 бита. По тази причина само младшите 8 бита (които са нули) ще се копират в променливата character.

Ако това обяснение не Ви е съвсем ясно, не се тревожете. Всички тези особености са разгледани в глава 14 Изрази и преобразувания на типове.

9.2.5 Присвояване на стойност на променлива

Задаването на стойност на променлива става с помощта на операцията присвояване '='. Тя има следния обобщен синтаксис:

име-променлива = стойност;

където

стойност

Може да е друга променлива, константа или дори израз, чийто тип е съвместим с типа на променливата.

Пример:

```
/* Дефиниране на променлива с име i от тип int */
int i;

/* Дефиниране на променлива с име ch от тип char */
char ch;

/* Присвояване на символна константа 'A' на променливата ch */
ch = 'A';

/* Присвояване на целочислена константа 10 на променливата i */
i = 10;
```

Променливите могат да получават стойности и по време на дефинирането си. Това се нарича инициализация на променливата, т.е. .

тип име-променлива = инициализатор;

Пример:

```
int i   = 10; /* инициализаторът е константа          */
int y   = i;  /* инициализаторът е друга променлива   */
char ch ='A'; /* инициализаторът е символна константа */
```

Преди да продължим по-нататък ще Ви запозная 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. Константите обаче нямат адрес и не са L-стойности. Терминът L-стойност се е появил в началото от израз за присвояване E1 = E2, където операндът 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 ще видим в следващата точка, 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.

9.3 Квалификатори на типа

C89 допълнително въвежда така наречените квалификатори на типа – const и volatile. Те могат да се прилагат към дефиниции на променливи и параметри на функции.

9.3.1 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.

const тип име-променлива = инициализатор;

Пример 9-6:

```
int main (void)
{
               int  x;
       const   int  y =  10;
       const   char c = 'A';

      x = 20;            /* Позволено, x може да се променя    */
   /* y = 20; */         /* !!!Грешка, y не може да се променя */
   /* c = 'B' */         /* !!!Грешка, c не може да се променя */

       return 0;
}
```

Всеки опит в кода 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 квалификатора 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 горе), но const-променливите могат да се променят от хардуера. Например една променлива може да се дефинирана като const и в същото време да 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.

9.3.2 Квалификатор volatile

Квалификаторът volatile информира компилатора, че променливата може

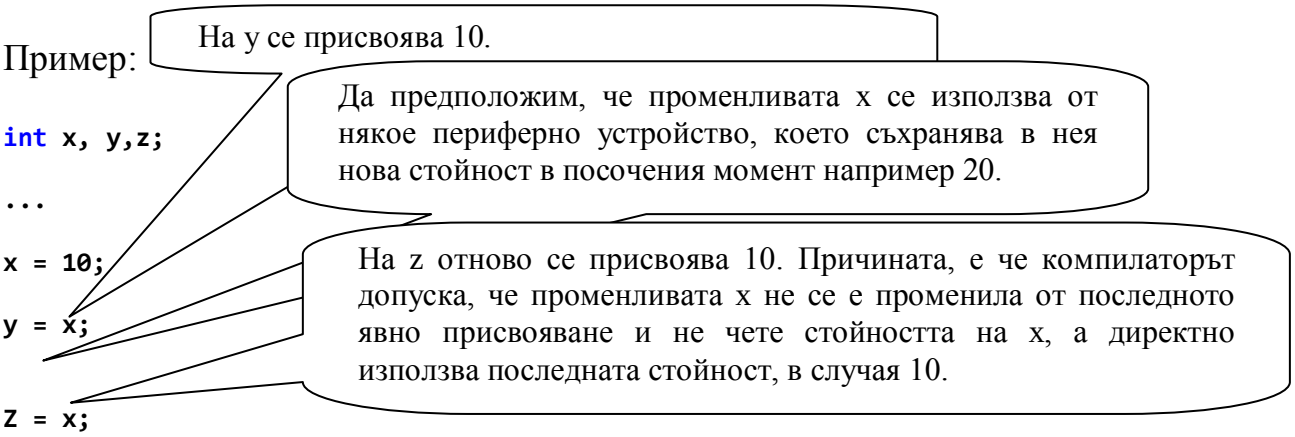
да се измени неявно, т.е. . без намесата на програмни инструкции например от хардуера.

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, тя трябва да се дефинира като volatile. Тогава компилаторът ще изключи оптимизацията.

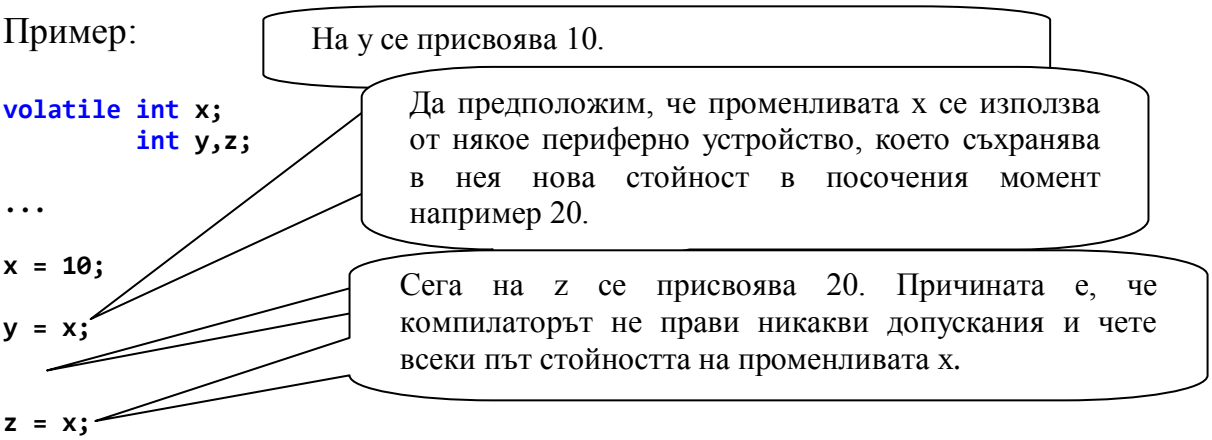
Тъй като е трудно да се даде практически пример, демонстриращ квалификатора volatile, по-долу е даден само теоретичен пример, описващ възможния проблем, който може да възникне, ако една променлива не е квалифицирана с volatile.

Пример:



Този проблем се избягва, ако променливата x се квалифицира с volatile.

Пример:



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

Пример:

```
const volatile int x = 10;
volatile const int y = 10;
```

9.4 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 се определя от мястото, където променливата е дефинирана.

В C89 променливите могат да се дефинират на две места:

- Извън тяло на функция

Такива променливи се наричат глобални. Глобалните променливи са видими от точката на дефиниране до края на файла, в който са дефинирани. Казано с други думи, една глобална променлива е видима за всички функции, които са разположени след точката, в която променливата е дефинирана.

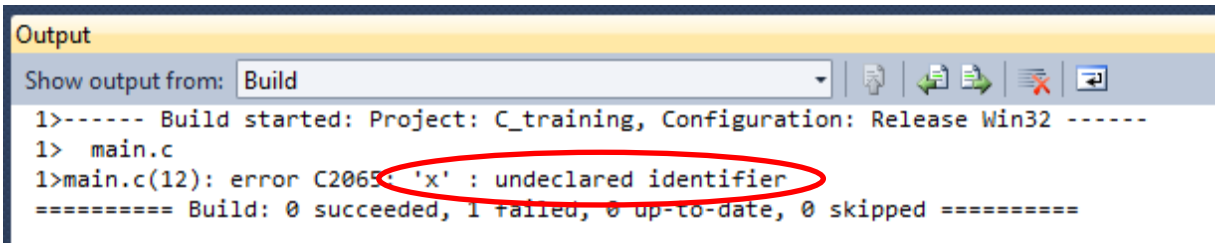
Пример 9-7:

```
void set_x(void);
int  get_x(void);

int main(void)
{
      int i;

      set_x();
      i = get_x();
       /* i = x; */ /*!!!Грешка, x не се вижда в тази част на програмата */

      return 0;
}

int x; /* Точка на дефиниране на x */

void set_x(void)
{
      x = 10;
}

int get_x(void)
{
      return x;
}
```

Област на видимост на x

В пример 9-7 променливата x е дефинирана след функцията main(). Това означава, че тя не е видима в main(). За да се убедите в това, махнете

коментара около конструкцията **i = x;** и компилирайте кода. Ще получите следния резултат:



Функциите set_x() и get_x() са дефинирани след променливата x и следователно имат достъп до тази променлива.

Глобалните променливи, дефинирани в един файл, могат да се използват и от други файлове. За целта е необходимо променливата да бъде декларирана във файла, в който ще се използва. Това е обяснено по-подробно в 9.6 Клас памет на променливите.

	**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.**

- В началото на съставен оператор (блок с код)

Съставен оператор, наричан още блок с код, представлява група оператори, затворени във фигурни скоби, т.е.

```
{/* Съставен оператор */
      тип   име-локална-променлива1;   /* Дефиниране на          */
      ...                              /* локални променливи */
      тип   име-локална-променливаN;

      оператор1;
      ...                              /* изпълними оператори */
      операторN;
}
```

Съставният оператор може да се използва навсякъде, където единичен оператор може да се използва.

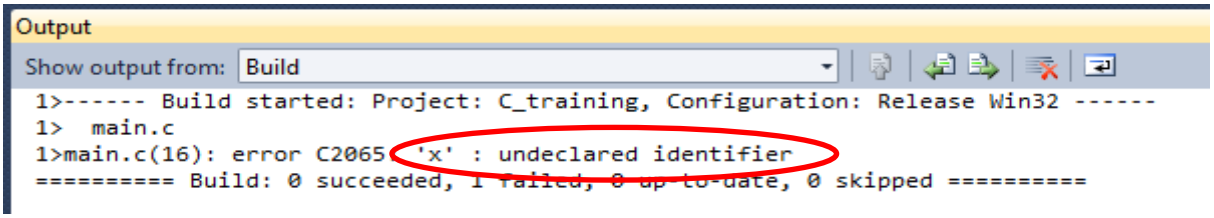
Променливите, дефинирани в блок с код, се наричат локални. Те са видими от точката си на дефиниране до затварящата фигурна скоба на блока с код, в който са дефинирани. В частност тялото на функция е съставен оператор. Ето защо променливите се дефинират в началото на функцията преди всеки друг изпълним оператор.

Пример 9-8:

```
void set_x(void);

int main(void)
{
      int x; /* Точка на дефиниране на x */

      x = 10;
      /*...*/

      return 0;
}

void set_x(void)
{
      /* x = 10; */ /*!!!Грешка, x не е видима извън main() */
}
```

Област на видимост на x

В Пример 9-8 променливата x е дефинирана в началото на функцията main(). Това означава, че тя е видима само от точката на дефиниране до края на функцията main(). За да се убедите в това, махнете коментара около конструкцията **x = 10;** във функцията set_x() и компилирайте кода. Ще получите следния резултат:



Както споменах, съставният оператор може да се използва навсякъде, където се използва единичен оператор, т.е. . всеки един оператор в тяло на функция може да бъде и съставен оператор. Променливите дефинирани в съставния оператор обаче са видими само в него и не са достъпни за кода извън него.

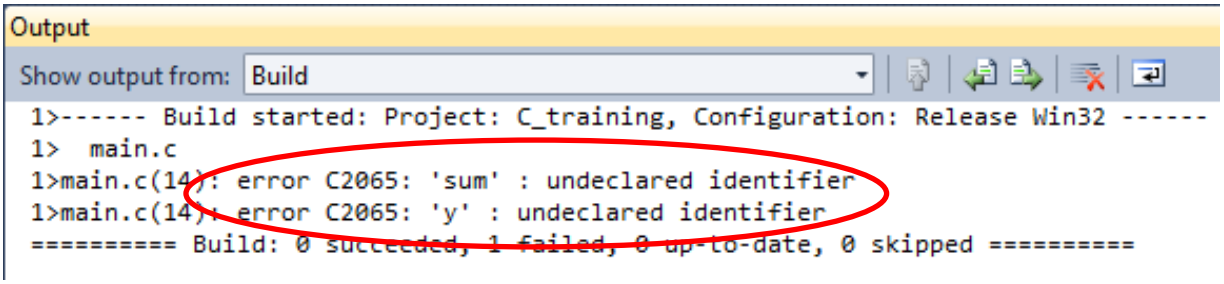
Пример 9-9:

```
int main(void)
{
      int x; /* Точка на дефиниране на x */

      x = 10;

      {/* Съставен оператор */
            int y = 20, sum;
            sum = x + y;
      }

      /* sum = x + y; */ /*!!!Грешка, y и sum не са видими тук */

      return 0;
}
```

Област на видимост на y и sum

Област на видимост на x

В Пример 9-9 във функцията main() е дефиниран съставен оператор. Променливите y и sum, дефинирани в него, не са видими извън отварящата и затварящата скоба на оператора. За да се убедите в това, махнете коментара около конструкцията **sum = x + y;** и компилирайте кода. Ще получите следния резултат.



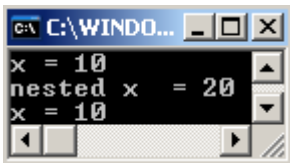
Обърнете внимание, че променливите, дефинирани в един съставен оператор, са видими във вложен в него друг съставен оператор. Обратното обаче, както разбрахте, не е вярно. В горния пример съставният оператор се явява вложен по отношение на тялото на функцията.

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.

C89 допуска име на променлива във вложен съставен оператор да съвпада с име на променлива, дефинирана в обхващащия съставен оператор. В този случай, променливата от вложения блок "скрива" променливата от обхващащия блок в рамките на вложения блок.

Пример 9-10:

```
#include <stdio.h>

int main(void)
{
	int x = 10; /* Точка на дефиниране на x */

	printf("x = %d\n", x); /* Отпечатва 10 на екрана */

	{/* Съставен оператор */
		int x = 20; /* Точка на дефиниране на вложена променлива x */
		printf ("nested x  = %d\n", x); /* Отпечатва 20 */
	}

	printf("x = %d\n", x); /* Отпечатва 10 */

	return 0;
}
```



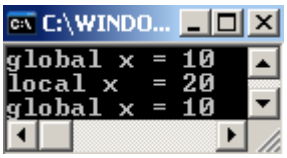
Избягвайте използването на еднакви имена на локални променливи във вложен и обхващащия го блок. Това води до объркване.

Параметрите на функция също се явяват локални променливи. Тяхната област на видимост е тялото на функцията.

C89 допуска също име на локална променлива да съвпада с име на глобална променлива. В този случай локалната променлива "скрива" глобалната променлива в рамките на блока, в който е дефинирана.

Пример 9-11:

```
#include <stdio.h>

int x = 10; /* Точка на дефиниране на глобална променлива x */

int main(void)
{
	printf("global x = %d\n", x); /* Отпечатва 10 */

	{/* Съставен оператор */
		int x = 20; /* Точка на дефиниране на локална променлива x */

		printf ("local x  = %d\n", x); /* Отпечатва 20 */
	}

	printf("global x = %d\n", x); /* Отпечатва 10 */

	return 0;
}
```



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

9.5 Време на живот на променливите

Времето на живот на променливите представлява времето, през което една променлива се съхранява в паметта на компютъра. Паметта за глобалните променливи се заделя при стартиране на програмата и се съхранява, докато трае изпълнението. След завършване на програмата тази памет се освобождава. Паметта за локалните променливи се заделя при влизане в блока с код, където са дефинирани, и се съхранява само докато се изпълнява блока с код, т.е. . времето на живот на локалните променливи

съвпада с областта им на видимост. При излизане от блока с код тази памет се освобождава. Времето на живот на локалните променливи може да се променя с помощта на т.нар. спецификатор за клас памет static. Това е обяснено по-подробно в 9.6 Клас памет на променливите.

9.6 Клас памет на променливите

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

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.

Класът памет се задава в дефиницията на променливата, т.е.

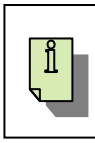
клас-памет тип име-променлива;

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

9.6.1 Клас памет auto

Клас памет auto може да се прилага само към локални променливи (без параметри на функции). 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 (**stack**). Когато програмата влезе в локална област

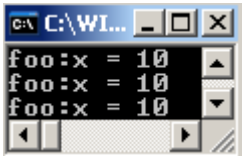
на действие, каквато е тялото на функция и блок с код, компилаторът заделя памет за тях в стека. При излизане от функцията (или блока с код) паметта се освобождава и може да се използва от компилатора за заделяне на памет за други автоматични локални променливи. Ако дефиницията на локалната auto-променлива е съпроводена и с инициализация, то променливата се инициализира всеки път при влизане във функцията (или блока с код), в противен случай локалната auto-променлива съдържа произволна стойност. Инициализаторът на auto-променливите може да бъде всякакъв израз, чийто тип е съвместим с типа на променливата.



Локалните auto-променливи не се инициализират автоматично от компилатора. При липса на явен инициализатор те съдържат произволни стойности.

Пример 9-12:

```
#include <stdio.h>

void foo(void);

int main(void)
{
	foo();
	foo();
	foo();

	return 0;
}

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

	printf("foo:x = %d\n", x);
	x = x + 1; /* Безсмислена промяна на x */
}
```

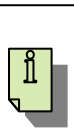


Както показва последния коментар, тази промяна на променливата x е безсмислена. Паметта, заделена за x се освобождава при излизане от функцията и всяка съхранена в нея стойност се губи. При всяко влизане в foo() се заделя памет за x в стека и x се инициализира с 10.

9.6.2 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 е дефиниран 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 (вижте отново Фиг.17). Ако компилаторът не може да удовлетвори тази заявка, променливата се съхранява като обикновена auto-променлива. Ако дефиницията на локалната register-променлива е съпроводена и с инициализация, то променливата се инициализира всеки път при влизане във функцията (или блока с код). Инициализаторът на register-променливите може да бъде всякакъв израз, чийто тип е съвместим с типа на променливата.

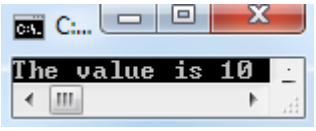


Локалните register-променливи не се инициализират автоматично от компилатора. При липса на явен инициализатор те съдържат произволни стойности.

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.

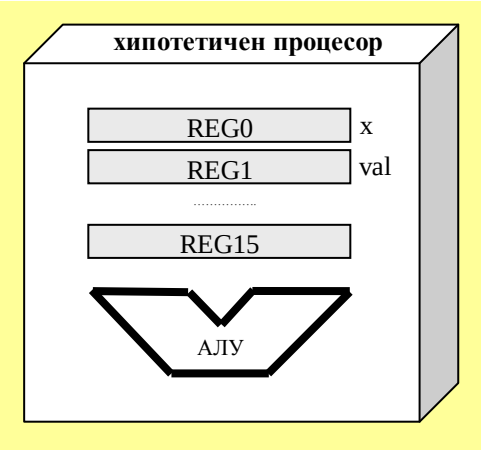
Пример 9-13:

```
#include <stdio.h>

void print_value (register int val);

int main(void)
{
	register int x;

	x = 10;
	print_value(x);

	return 0;
}

void print_value (register int val)
{
	printf("The value is %d\n", val);
}
```



Когато компилаторът срещне конструкцията **register int x;** той ще се опита да използва вътрешните регистри на процесора, за да представи променливата x. Същите действия ще бъдат извършени при извикване на функцията print_value(). Компилаторът ще се опита да представи параметъра val чрез някой от регистрите на процесора. Фиг.34 илюстрира това на базата на хипотетичния процесор от глава 5 Архитектура на

компютъра.



Фиг. 34 Използване на вътрешните регистри на процесора за представяне на register-променливи

9.6.3 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 програмата.

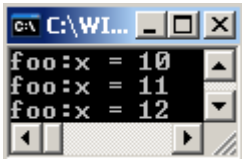
9.6.3.1 Локални статични променливи

Когато една локална променлива се дефинира като static, се променя времето 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. Това означава, че при излизане от блока променливата не е достъпна, но стойността се запазва.

Пример 9-14:

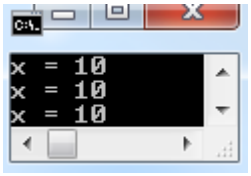
```
#include <stdio.h>

void foo(void);

int main(void)
{
	foo();
	foo();
	foo();

	return 0;
}

void foo(void)
```

```
{
	static int x = 10;

	printf("foo:x = %d\n", x);
	x = x + 1;
}
```



Паметта за статичните локални променливи се заделя още при стартирането на програмата. Тази памет е различна от мястото, където се съхраняват локалните auto-променливи, т.е. . статичните локални променливи се съхраняват извън стека. Ако променливата има и инициализатор, то стойността на инициализатора се присвоява на променливата също при стартирането на програмата. Инициализаторът на локалните статични променливите може да бъде само константен израз, чийто тип е съвместим с типа на променливата. В нашия пример, при стартирането на програмата се заделя памет за променливата x, дефинирана във foo(), и x се инициализира с 10. Обърнете внимание, че тази инициализация се извършва само веднъж (при стартирането на програмата), а не при всяко влизане във функцията. След излизане от функцията статичните локални променливи запазват последно съхранената в тях стойност. Променливата x се увеличава с 1 при всяко извикване на foo() и стойността се съхранява и при излизане от foo().

Нека да направим един експеримент като махнем ключовата дума static от дефиницията на променливата x. Сега x е обикновена локална (автоматична) променлива. Резултатът от изпълнението на програмата е:



Причината е, че паметта за променливата x се заделя всеки път при влизане във функцията и всеки път се инициализира с 10. При излизане от функцията тази памет се освобождава, т.е. . променливата се "изтрива" и програмата повече "нищо не знае" за нея до следващото влизане във функцията.

Ако липсва инициализатор, статичните локални променливи се инициализират с 0 автоматично от компилатора.

Локалните статични променливи се инициализират с 0 автоматично от компилатора при липса на явен инициализатор.

Препоръчително е да не разчитате на компилатора, а винаги явно да инициализирате локалните статични променливи.

9.6.3.2 Глобални статични променливи

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

Пример 9-15:

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

void foo(void);

static int x;

int main(void)
{
    x = 10;
    foo();

    return 0;
}
```

```
/* сорс-файл2 */
void foo(void)
{

    x = 20; /*!!!Грешка, x не е достъпна тук */

}
```

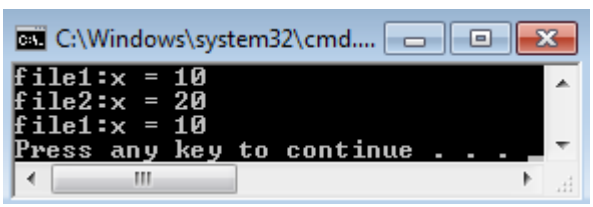
Примерът показва, че ако една глобална променлива е дефинирана като статична, тя е достъпна само във файла, където се намира дефиницията. Всеки опит за достъп до тази променлива в други файлове води до генериране на грешка при компилиране.

Имената на две или повече 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 9-16:

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

static int x;

void foo(void);

int main(void)
{
      x = 10;

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

      return 0;
}
```

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

static int x;

void foo(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.

Глобалните статични променливи следват същите правила за създаване и инициализиране като локалните статични променливи.

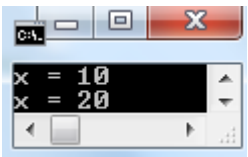
9.6.4 Клас памет extern

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 в 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.

Пример 9-17:

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

void foo(void);

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

int main(void)
{
      x = 10;

      foo();

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

      return 0;
}
```

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

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

void foo(void)
{
      /* x вече е достъпна тук */
      printf("x = %d\n", x);

      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 горния пример), или локално в блока, в който тази променлива се използва.

Пример 9-18:

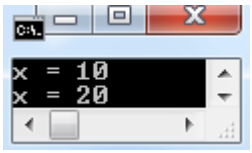
```
/* сорс-файл1 */
#include <stdio.h>

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

void foo(void);

int main( void )
{
    x = 10;

    foo();

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

    return 0;
}
```

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

void foo(void);
void foo1(void);

void foo(void)
{
    /* Локална декларация на
       глобалната променлива x */
      extern int x;

      /* x е достъпна тук */
      printf("x = %d\n", x);

      x = 20;
}

void foo1(void)
{
   /* Грешка, x не е достъпна тук */
   /* x = 30; */
}
```

В показания пример, променливата x е достъпна в foo(), но не е достъпна в foo1(). Причината е, че декларацията на променливата не се вижда извън foo().

Пример 9-19:

```
#include <stdio.h>

void func(void);

int main(void)
{
      /* Декларация на x */
      extern int x; /* x вече е видима и в main() */

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

      return 0;
}

/* Точка на дефиниране на x */
int 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. Тази декларация указва на компилатора, че променливата е дефинирана някъде другаде в същия или в друг файл.

Глобалните нестатични променливи следват същите правила за създаване и инициализиране като статичните променливи (локални и глобални).

	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.

9.6.5 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, ще Ви припомня как изглежда обобщената дефиницията на променлива:

T име-променлива;

където **T** е някой от типовете данни в C.

Ако пред тази дефиниция 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. Горната декларация може да се запише като

typedef T ново-име-на-тип;

където

ново-име-на-тип е синоним на типа **T**.

Пример 9-20:

```
#include <stdio.h>

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

int main(void)
{
      uint8  x; /* Същото като unsigned char x */

      x = 10;

      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 размерът на базовите типове данни не е фиксиран от стандарта C, то typedef може да се използва да се декларират 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;
```

Забележете, че имената на унифицираните типове съдържат броя на битовете, заемани от тях в паметта. Това подобрява самодокументирането на кода.

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

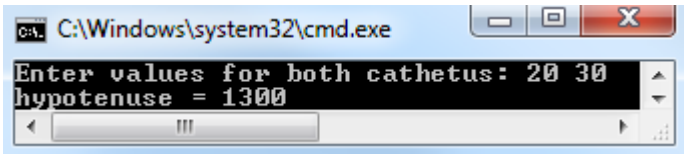
За предпочитане е да използвате унифицирани имена на типовете вместо тези указани в стандарта. Декларациите на тези унифицирани имена е най-добре да се поставят в хедър-файл, който да се включва във всеки сорс-файл. Това подобрява самодокументирането на кода и повишава преносимостта на програмата.

Следващият пример демонстрира това. Съдържанието на файла stdtypes.h се добавя в сорс-файла с помощта на директивата #include. Обърнете внимание, че имената на потребителските хедър-файлове обикновено се затварят в двойни кавички, а не в < >.

Пример 9-21:

```
/* stdtypes.h */

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;
```

```
/* main.c */

#include <stdio.h>
#include "stdtypes.h" /* Добавя съдържанието на файла stdtypes.h */

int main(void)
{
      uint16 cathetus1;
      uint16 cathetus2;
      uint32 hypotenuse;

      /* Въведете стойности за двата катета */
      printf("Enter values for both cathetus: ");
      scanf("%u%u", &cathetus1, &cathetus2);

      /* Изчисляване на хипотенузата */
      hypotenuse = cathetus1*cathetus1 + cathetus2*cathetus2;

      printf("hypotenuse = %u\n", hypotenuse);

      return 0;
}
```



9.7 Подравняване на променливите в паметта

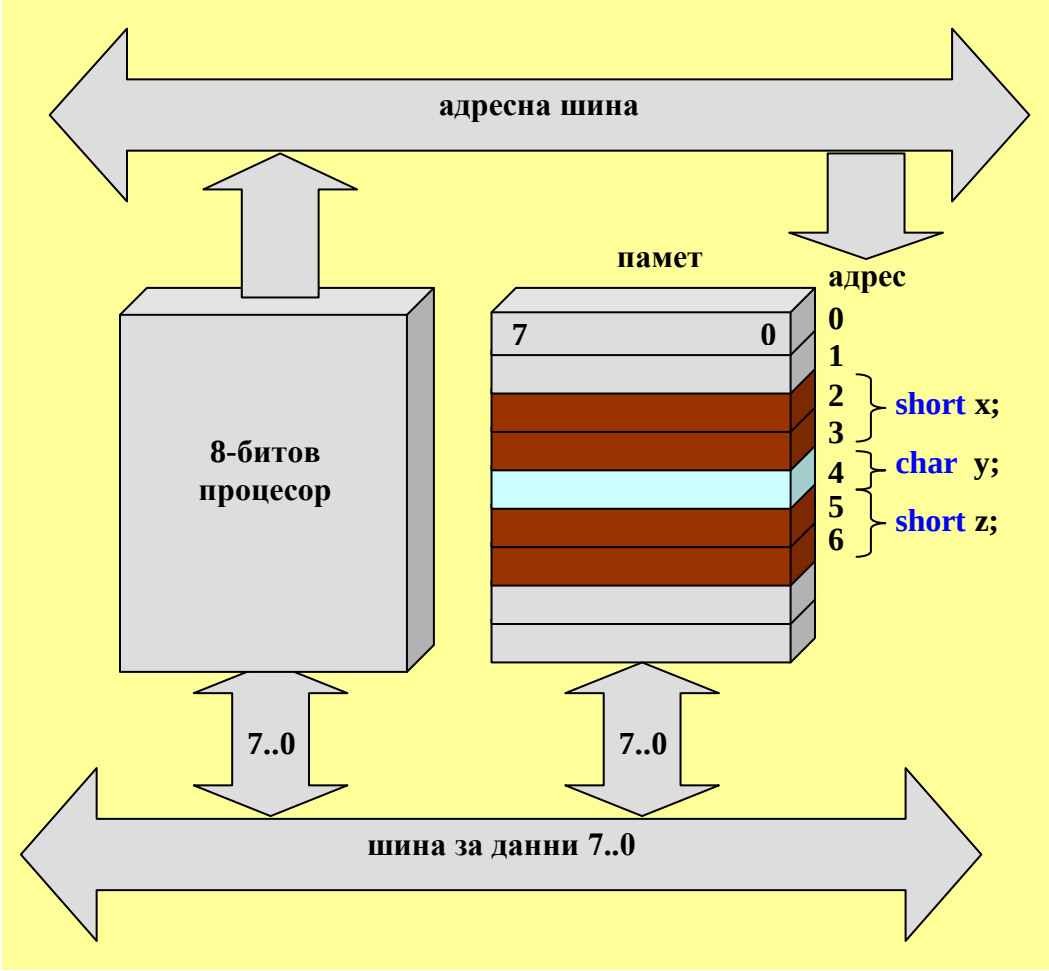
Подравняването представлява разполагане на променливите на определени адреси в паметта. Подравняването на променливите в паметта не е изискване на езика C, а на различните компютърни архитектури. Всяка имплементация на компилатора обаче трябва да се съобразява с това изискване. Когато дефинирате променливи, Вие не трябва да се тревожите как те се разполагат в паметта. Компилаторът знае изискванията на

компютъра и разполага данните според тях. Причината да Ви запозная с подравняването на променливите в паметта е свързана с факта, че езикът C предоставя средство (което ще стане ясно в главата, която разглежда указателите), което позволява неподравнен достъп до паметта. Непознаването на проблема, който може да възникне, би довело до трудно откриваеми логически грешки.

За да разберете какво се има предвид под подравняване на променливите в паметта (**memory alignment**), ще разгледаме типична организация на паметта на 8- и 16- битови компютри. Фиг.35 показва типичната организация на паметта на 8-битов компютър. На фигурата е показано също и паметта, заделена за три променливи, декларирани по следния начин:

```
short x;
char  y;
short z;
```

Паметта е организирана по байтове и всеки байт си има адрес.

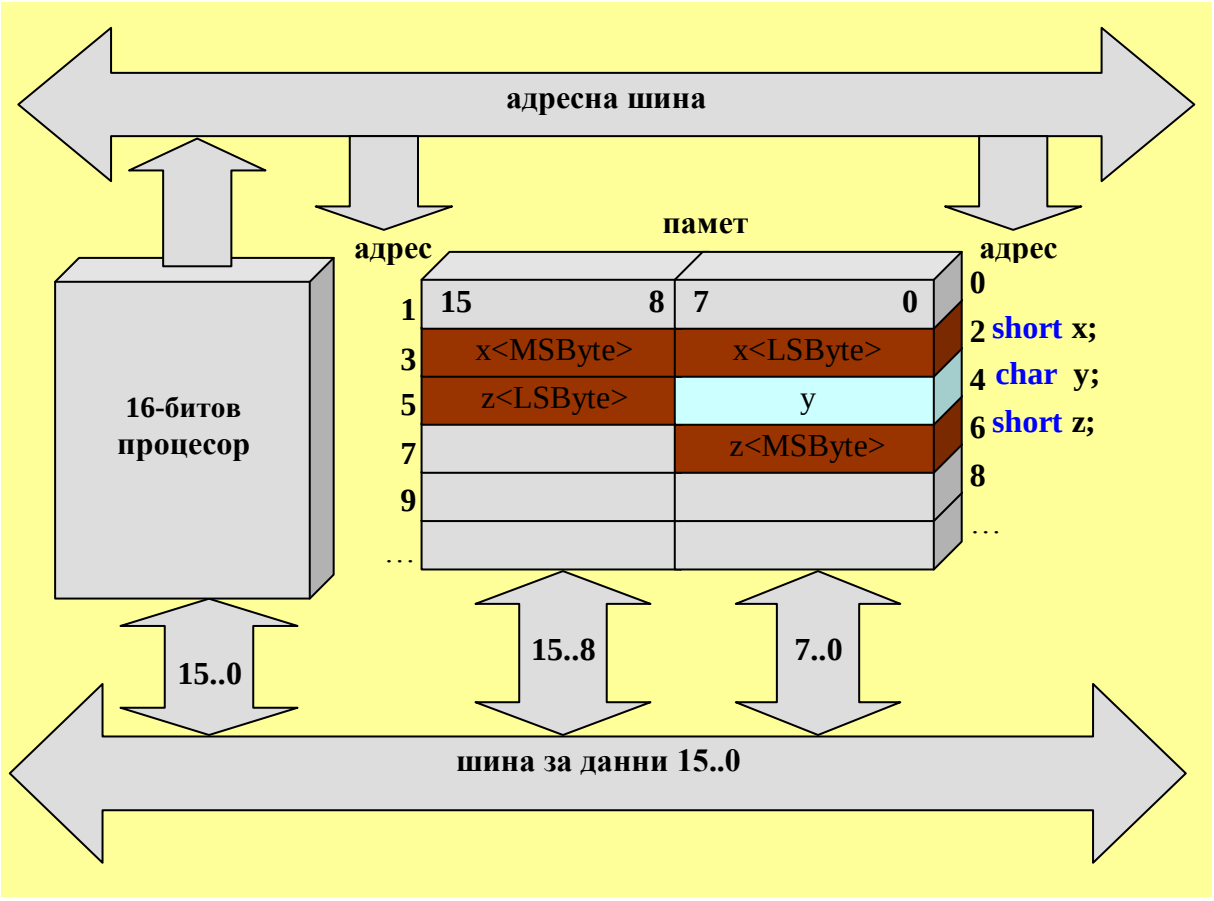


Фиг. 35 Организация на паметта на 8-битов компютър

Компютър с 8-битова шина за данни може да трансферира по 8 бита (1 байт) с една операция. За трансфер на многобайтови данни като тип short,

int, long, float и double са необходими толкова на брой операции, колкото е размера в байтове на данните. 8-битовите компютри могат да осъществяват достъп до всеки байт, т.е. . те нямат специални изисквания за разполагане на променливите в паметта на определени адреси.

Фиг.36 показва типичната организация на паметта на 16-битов компютър. Паметта се състои от 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, т.е. . ако данните са подравнени в паметта (вижте променливата x).



Фиг. 36 Организация на паметта на 16 битов-компютър

Ако данните са неподравнени (вижте променливата z), са възможни две ситуации.

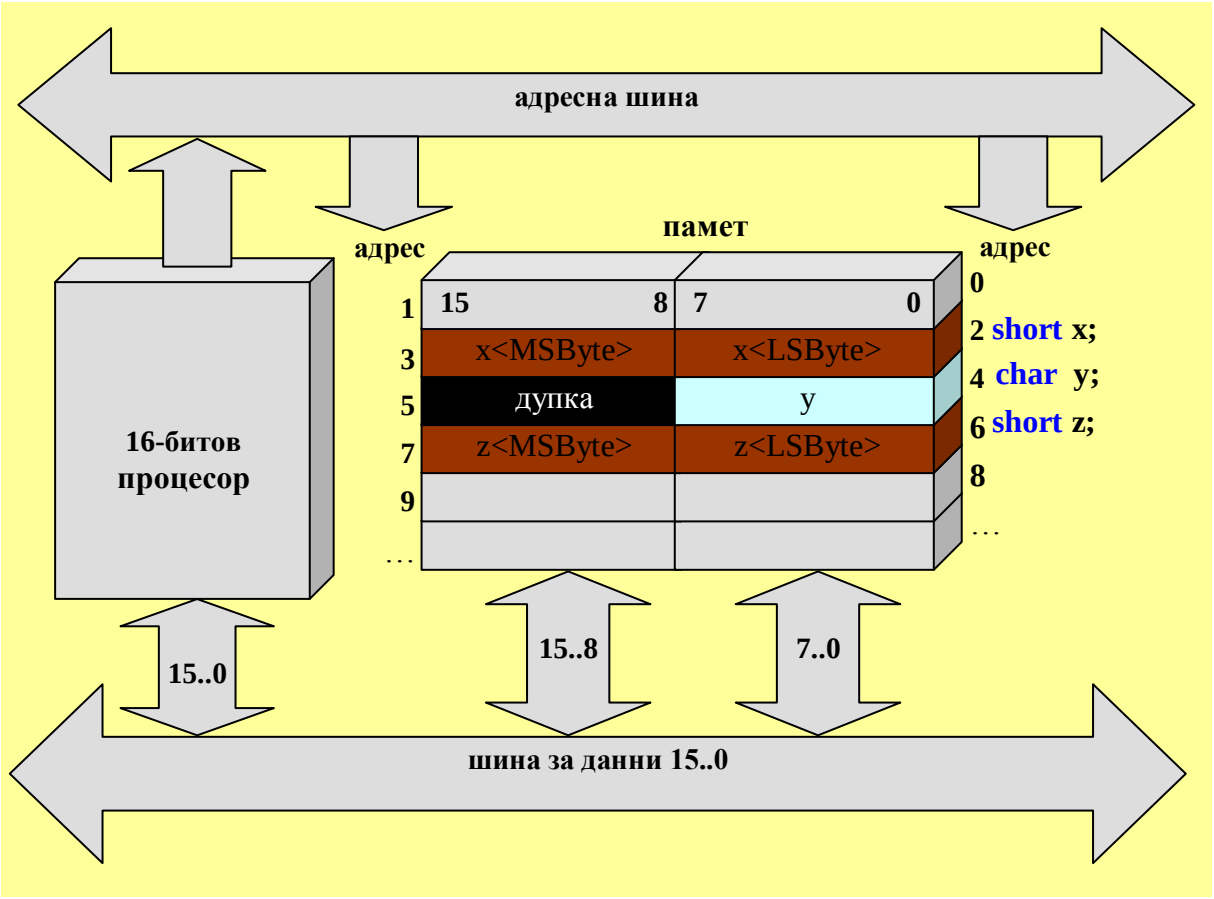
- Компютри, които не са проектирани да осъществяват достъп до неподравнена машинна дума, ще генерират изключение[1] при такава ситуация, което вероятно ще причини аварийно завършване на програмата.

Забележка[1]: Изключението е събитие, което се случва само в специални, обикновено

- Компютри, които са проектирани да осъществяват достъп до неподравнени думи, извършват това с две операции. Първата операция извлича машинната дума, в която е разположен младшия байт (LSByte), а с втората операция се извлича машинната дума, в която е разположен старшия байт (MSByte).

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.

Изискването за подравняване на променливите в паметта може да доведе до появата на "дупки" (неизползвани байтове) в паметта. Например, ако компилаторът подравни променливата z (Фиг.37), байт с адрес 5 остава неизползван.



Фиг. 37 Дупки в паметта

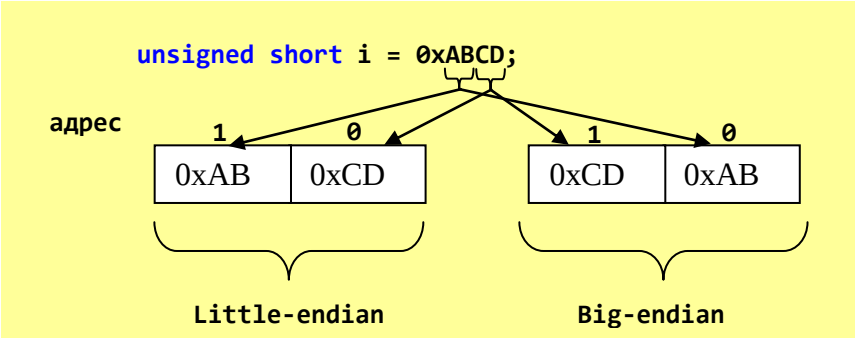
9.8 Big-endian и Little-endian

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.38 показва нагледно това.



Фиг. 38 Big-endina и Little-endian

Някои компютри позволяват това да се конфигурира.

Въпроси за самопроверка

1. Какво е променлива?

2. Напишете общата форма за дефиниране на променлива.

3. Каква информация носи типа на една променлива за компилатора?

4. Къде се дефинират променливите?

5. Кои са базовите типове данни в езика C?

6. Кои са модификаторите на базовите типове и как се прилагат към тях?

7. Опишете правилата за създаване на име на променлива (от какви символи може да се състои и т.н.)

8. Ако имате променлива с име **symbol**, покажете как ще изглежда дефиницията от тип char.

9. Как се присвоява стойност на една променлива?

10. Опишете областта на видимост и времето на живот на локалните променливи (auto и register). Каква е стойността на локалната променлива след дефинирането при липса на явен инициализатор?

11. Опишете областта на видимост и времето на живот на локалните статични променливи. Каква е стойността на локалната статична променлива след дефинирането при липса на явен инициализатор?

12. Опишете областта на видимост и времето на живот на глобалните статични променливи. Каква е стойността на глобалната статична променлива след дефинирането при липса на явен инициализатор?

13. Опишете областта на видимост и времето на живот на глобалните външни променливи. Каква е стойността на глобалната външна променлива след дефинирането при липса на явен инициализатор?

14. За какво служи спецификатора typedef?

10 Константи

10.1 Определение и видове

Константата е фиксирана стойност, която не може да се изменя в програмата. В зависимост от типа, константите могат да се разделят на:

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

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

10.2 Целочислени константи

C89 поддържа 3 формата на представяне на целочислените константи: десетични, осмични и шестнайсетични.

10.2.1 Десетични

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

Пример: 10, 123, 64674, -100 и т.н.

Типът на десетичната константа е първия целочислен тип, започвайки от int, който може да представи стойността на тази константа (Табл.21).

вид константа	тип
Десетична целочислена константа	int
	long
	unsigned long

Табл. 21 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.

10.2.2 Шестнайсетични

Шестнайсетичните 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 шестнайсетичната константа е първият целочислен тип, започвайки от int , който може да представи стойността на тази константа (Табл.22).

вид константа	тип
Шестнайсетична константа	int
	unsigned
	long
	unsigned long

Табл. 22 Подразбиращ се тип на шестнайсетична константа

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

10.2.3 Осмични

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

Пример: 012, 023,077 и т.н.

Типът на осмичната константа е първият целочислен тип, започвайки от int, който може да представи стойността на тази константа (Табл.23).

вид константа	тип
Осмична константа	int
	unsigned
	long
	unsigned long

Табл. 23 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.

Забележка: Осмичните константи на практика не се използват. Тук са дадени само за пълнота.

10.3 Константи с плаваща запетая

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

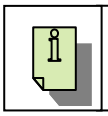
цяла-част. дробна-част

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
123.0E2 ≈ 123.0E+2 ≈ 123.E2

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:

3.4 - на числото 3.4 се назначава тип double (по подразбиране)
3.4f - на числото 3.4 се назначава тип float
3.4l - на числото 3.4 се назначава тип long double

10.4 Символни константи

Символната константа представлява символ, затворен в апострофи.

'символ'

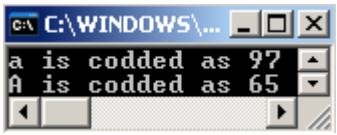
Пример: 'a', '!', '-' и т.н.

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. За отпечатване на символи на екрана с помощта на printf() се използва форматният спецификатор **%c**. Ако искате да видите числовия код, който съответства на символа, може да използвате спецификатора **%d**.

Пример 10-1:

```
#include <stdio.h>

int main(void)
{
	printf("%c is codded as %d\n", 'a', 'a');
	printf("%c is codded as %d\n", 'A', 'A');
```

```
	return 0;
}
```



С стандартът 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 С (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 шестнайсетично число

Табл. 24 escape-последователности

Пример 10-2:

```
#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;
}
```

Фактът, че символните константи се заменят с числов код, означава, че можем да ги използваме навсякъде, където могат да се използват целочислени константи например в аритметични изрази.

Пример: `char character = 'a' + 1;`

Ако символите се кодират в ASCII код, то резултатът ще бъде 97 + 1 = 98, което отговаря на символа 'b'.

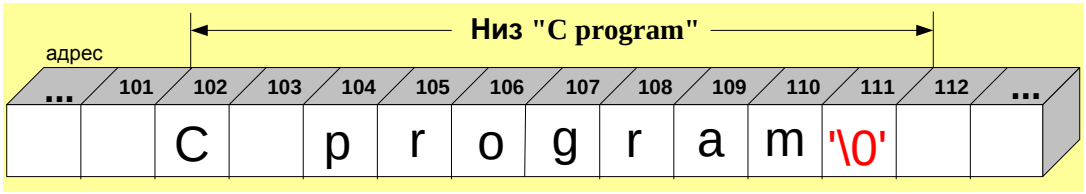
10.5 Низови константи

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

"последователност-символи"

Пример: **"C program"**

Низовите константи се съхраняват в паметта на компютъра, като символите се разполагат последователно един след друг в паметта. Компилаторът автоматично добавя нулев символ (нарича се още нулев байт) след последния символ. В C нулевият символ се използва за обозначаване на край на низ и се означава като '\0'. Фиг.39 дава ясна представа как се разполагат символите на низа в паметта.



Фиг. 39 Съхраняване на низ в паметта

Низова константа, състояща се само от двойни кавички без символи между тях, се нарича нулев низ. Такъв низ се състои само от нулев символ.

Пример: **""**

Досега Вие използвахте низови константи при отпечатване на съобщения с printf(). В 16 Масиви и 17 указатели ще научите повече за употребата им, а в Част V: Стандартна Библиотека ще Ви покажа функции за обработка на низове.

Въпроси за самопроверка

1. Какво е константа ?

2. Какви видове константи има?

3. Ако тип int е 16 бита, а long е 32 бита, какъв тип ще назначи компилатора на следните целочислени константи?

   a) 10
   b) 10u
   c) 10L
   d) 10ul
   e) 50000
   f) 0xC350 (десетично 50000)

4. Какъв е разделителният символ в константите с плаваща запетая?

5. Какъв тип ще назначи компилатора на следните константи с плаваща запетая?

   a) 10.2
   b) 10.2f
   c) 10.2L

6. Какво е низова константа?

7. Как се обозначава край на низова константа?

11 Извеждане на данни на екран

В тази глава ще Ви запозная официално с функцията printf() и още няколко библиотечни функции, с чиято помощ може да извеждате данни на екрана. Всички тези функции изискват включването на хедър-файла <stdio.h> в сорс-файла, в който се използват.

Не забравяйте да включите хедър-файла <stdio.h>.

11.1 Извеждане на форматирани данни с printf()

Функцията printf() позволява извежданите данни да бъдат форматирани по-определен начин, например може да изведете десетично цяло число в шестнайсетичен вид и много други.

Обобщената форма на извикване на тази функция е:

printf(форматиращ-низ, списък-аргументи);

Първият аргумент на функцията е **форматиращ-низ**. Той представлява последователност от символи, затворени в кавички. Тези символи се отпечатват на екрана както се въвеждат с изключение на следните последователности:

- форматиращи-последователности (форматни спецификатори)

%форматиращ-символ

Използвайте следните форматиращи последователности:

Форматираща - последователност	Описание
%c	Отпечатване на символ
%hd	Отпечатване на данни от тип short int
%hu	Отпечатване на данни от тип unsigned short int
%d	Отпечатване на данни от тип int
%u	Отпечатване на данни от тип unsigned int
%ld	Отпечатване на данни от тип long int
%ul	Отпечатване на данни от тип unsigned long int
%f	Отпечатване на данни от тип float и double
%Lf	Отпечатване на данни от тип long double
%s	Отпечатване на низ

%#x	Отпечатване на цяло число в шестнайсетичен формат
%%	Отпечатване на символа %

Табл. 25 printf форматиращи-последователности

- escape-последователности

\escape-символ

Използвайте следните escape-последователности:

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 шестнайсетично число

Табл. 26 escape-последователности

Вторият аргумент е **списък-аргументи**. Този списък има следния синтаксис:

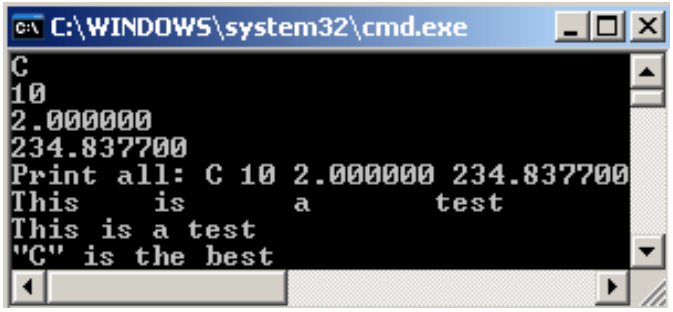
аргумент1, аргумент2, ... , аргументN

Списък-аргументи се състои от всички стойности (променливи, константи или изрази), разделени със запетая, които ще се отпечатват на екрана. На всяка форматираща последователност във форматиращия низ трябва да съответства аргумент в **списък-аргументи**. Форматиращата-последователност "съобщава" на функцията printf() какъв е типа на съответния аргумент. Това е необходимо, тъй като данните, отпечатвани на екрана, представляват последователност от символи. Например, ако се отпечатва десетичното число 10 (форматният спецификатор е %d), printf ще го преобразува в последователност от два символа '1' и '0' и ще ги изпрати към екрана.

Пример 11-1:

```
#include <stdio.h> /* Съдържа прототипа на функцията printf() и други  */
```

```c
int main(void)
{
	/* Дефиниране на променливи */
	char    c;
	int     i;
	float   f;
	double d;

	/* Присвояване на стойности на променливите */
	c = 'C';
	i = 10;
	f = 2.0;
	d = 234.8377;

	/* Отпечатване на съдържанието на променливите на екрана */
	printf("%c\n", c);
	printf("%i\n", i);
	printf("%f\n", f);
	printf("%f\n", d);

	/* Отпечатване на всички променливи наведнъж */
	printf("Print all: %c %d %f %f\n", c, i, f, d);

	/* Демонстрация на някои escape-последователности */
	printf("This\tis\ta\ttest\n");
	printf("This \ais \aa \atest\n");
	printf("\"C\" is the best\n");

	return 0;
}
```



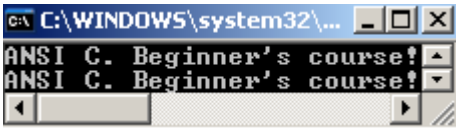
Както се вижда от резултата, escape- и форматиращите-последователности не се изобразяват на екрана така както се въвеждат в програмата, а имат специално предназначение.

Забележете, че ако искате да отпечатате низова константа, може да го направите по два начина: или чрез директно подаване на константата в скобите след printf() или чрез използване на форматния спецификатор %s.

Пример 11-2:

```c
#include <stdio.h>

int main(void)
{
	printf("ANSI C. Beginner\'s course!\n");
	printf("%s", "ANSI C. Beginner\'s course!\n");
```

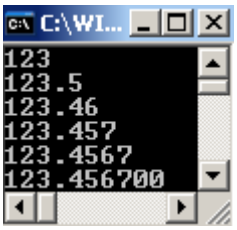
```
    return 0;
}
```



По подразбиране числата с плаваща запетая се отпечатват с 6 цифри след десетичната точка. Вие може изрично да укажете на printf да извежда само определен брой цифри след десетичната точка.

Пример 11-3:

```
#include <stdio.h>

int main(void)
{
    printf("%.0f\n", 123.4567); /* Отпечатай само цялата част */
    printf("%.1f\n", 123.4567); /* Отпечатай 1 цифра след десетичната точка */
    printf("%.2f\n", 123.4567); /* Отпечатай 2 цифри след десетичната точка */
    printf("%.3f\n", 123.4567); /* Отпечатай 3 цифри след десетичната точка */
    printf("%.4f\n", 123.4567); /* Отпечатай 4 цифри след десетичната точка */
    printf("%f\n", 123.4567);   /* Отпечатай 6 цифри след десетичната точка */

    return 0;
}
```



Уверете се, че използваният форматен спецификатор отговаря на типа на извежданите данни.

11.2 Извеждане на низ с puts()

Функцията puts() също позволява извеждане на низ на екрана, но за разлика от printf() данните не могат да се форматират.

Обобщената форма на извикване на тази функция е:

puts("низ");

Пример 11-4:

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

```
int main(void)
{
	puts("Hello C programmers.");

	return 0;
}
```

Тази функция автоматично извежда и символ за нов ред след подадения за отпечатване низ. Както и при printf(), отпечатваният низ може да съдържа escape-последователности.

Пример 11-5:

```
#include <stdio.h>

int main(void)
{
	puts("Say beep\a\a\a.!");

	return 0;
}
```

11.3 Извеждане на символ с putchar()

С функцията putchar() може да извеждате символ по символ на екрана. Извежданият символ може да бъде и escape-последователност.

Обобщената форма на извикване на тази функция е:

putchar('символ');

Пример 11-6:

```
#include <stdio.h>

int main( void )
{
	putchar('C');
	putchar('8');
	putchar('9');
	putchar('(');
	putchar('9');
	putchar('0');
	putchar(')');
	putchar('\n');
```

```
        return 0;
}
```

Въпроси за самопроверка

1. Напишете обобщената форма на извикване на функцията **printf()**.

2. Кои са форматиращите последователности за отпечатване на char, int, float и double с помощта на **printf()**?

3. Изведете на екрана съобщението "ANSI C. Beginner's course!" с puts().

12 Четене на данни от клавиатурата

В тази глава ще Ви запозная официално с функцията scanf() и getchar(), с чиято помощ може да въвеждате данни от клавиатурата. Тези функции изискват включването на хедър-файла <stdio.h> в сорс-файла, в който се използват.

Не забравяйте да включите хедър-файла <stdio.h>.

12.1 Въвеждане на данни със scanf()

Обобщената форма на извикване на тази функция е:

scanf(форматиращ-низ, списък-аргументи);

Първият аргумент на функцията е **форматиращ-низ**. Той представлява низ от форматиращи-последователности, затворен в кавички.

Използвайте следните форматиращи-последователности:

Форматираща-последователност	Описание
%c	Въвеждане на символ
%hd	Въвеждане на данни от тип short int
%hu	Въвеждане на данни от тип unsigned short int
%d	Въвеждане на данни от тип int
%ld	Въвеждане на данни от тип long int
%ul	Въвеждане на данни от тип unsigned long int
%f	Въвеждане на данни от тип float
%lf	Въвеждане на данни от тип double
%Lf	Въвеждане на данни от тип long double
%s	Въвеждане на низ

Табл. 27 scanf форматиращи-последователности

Вторият аргумент е **списък-аргументи**. Този списък има следния синтаксис:

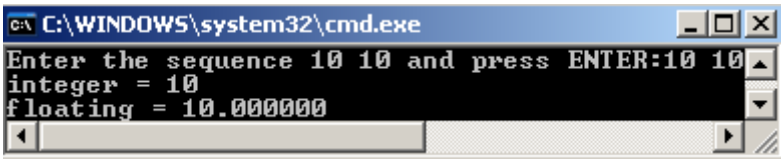
&аргумент1, &аргумент2, ... , &аргументN

и се състои от имената на всички променливи, в които се съхраняват данните, въведени от клавиатурата, разделени със запетаи. Както се вижда, пред всяко име на променлива стои символът '&'. Значението на този

символ ще стане ясно след изучаване на указателите в езика C. Засега просто запомнете, че когато четете данни от клавиатурата пред името на всяка променлива, в която се съхраняват данните, се поставя символа '&'. Запомнете също, че на всяка форматираща последователност във форматиращия низ трябва да съответства променлива в **списък-аргументи**. Форматиращата последователност "съобщава" на функцията scanf() типа, към който въведените данни трябва да се преобразуват. Това е необходимо, тъй като всички данни, въведени с клавиатурата, представляват последователност от символи. Например, ако сте въвели числото 10, това в действителност е поредица от два символа '1' и '0'. Ако указаният форматен спецификатор е %d, scanf() преобразува тези два символа в десетичното число 10 и го съхранява в указаната променлива. Ако укажете обаче друг спецификатор, например %f или %s, то scanf() ще преобразува тези символи в число 10.0 или низ "10" съответно. Следващият пример демонстрира това.

Пример 12-1:

```
#include <stdio.h>

int main(void)
{
	int   integer;
	float floating;

	printf("Enter the sequence 10 10 and press ENTER:");
	scanf ("%d%f", &integer, &floating);

	printf("integer = %d\nfloating = %f\n", integer, floating);

	return 0;
}
```

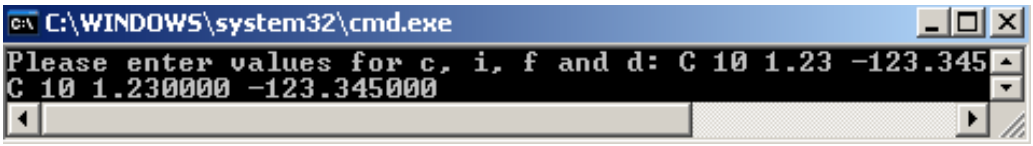


При въвеждане на данни от клавиатурата е необходимо те да се разделят с интервал или табулация (интервалите и табулациите се наричат под общото име бели-символи (**white-symbols**)). По това функцията **scanf()** разпознава отделните данни въведени чрез клавиатурата. Действителното прочитане на данните става след натискане на клавиша ENTER. След натискане на този клавиш, функцията **scanf()** започва четене на данните, преобразува ги в указания тип и ги съхранява в съответните променливи.

Пример 12-2:

```
#include <stdio.h> /* Съдържа прототипа на функцията scanf() и други  */

int main(void)
{
```

```
  /* Дефиниране на променливи */
  char   c;
  int    i;
  float  f;
  double d;

  /* Отпечатване на подсказващо съобщение */
  printf("Please enter values for c, i, f and d: ");

  /* Четене на въведените данни от клавиатурата */
  scanf("%c%d%f%lf", &c, &i, &f, &d);

  /* Отпечатване на въведените данни на екрана */
  printf("%c %d %f %f\n", c, i, f, d);

  return 0;
}
```

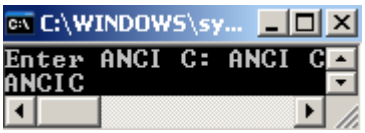


	Уверете се, че използваният форматен спецификатор отговаря на типа на променливата, в която ще се съхраняват въведените данни.

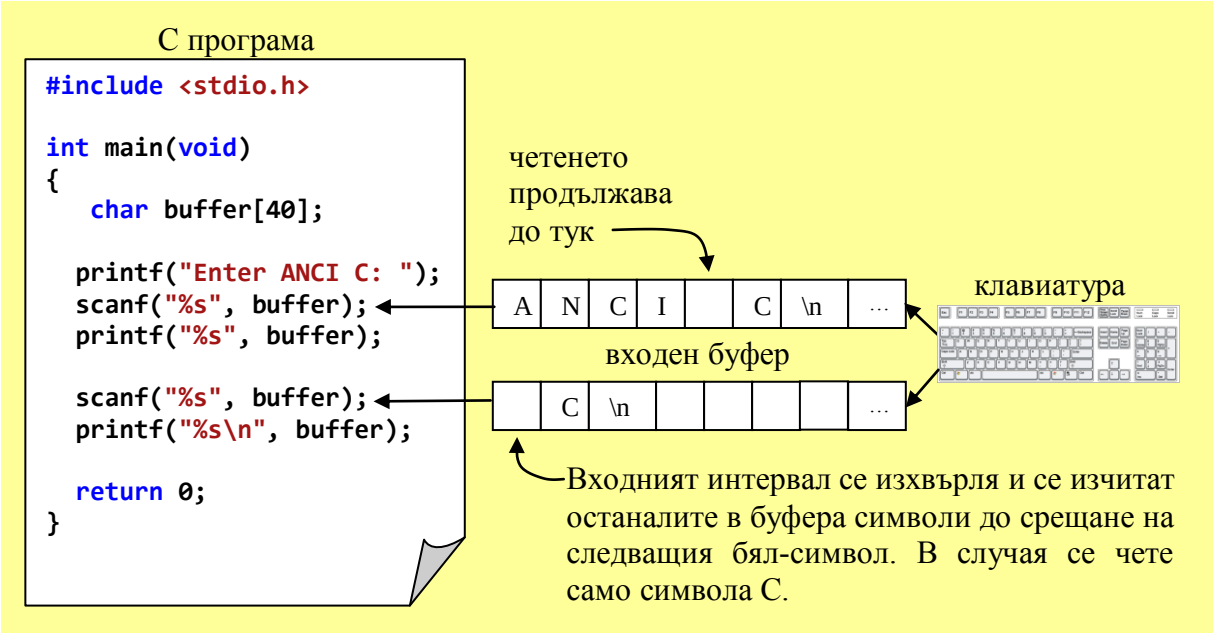
Когато въвеждате символи от клавиатурата, трябва да имате предвид следната особеност. Въведените символи постъпват първо във входен буфер. При натискане на ENTER в буфера се записва символа '\n' и функцията scanf() започва да чете символите от буфера. Четенето продължава, докато се прочете бял-символ (' ', '\t', '\v', '\r', '\f' и '\n'). Белият символ не се чете и остава в буфера заедно с всички въведени след него символи. При последващо извикване scanf() (или друга функция за четене от клавиатурата) ще прочете останалите във входния буфер символи. Ако при извикване на scanf() във входния буфер има останали бели символи (или първите въведени символи от клавиатурата са бели символи), те се изхвърлят.

Пример 12-3:

```
#include <stdio.h>

int main(void)
{
	char buffer[40];

	printf("Enter ANCI C: "); /* Enter ANCI C: ANCI C */
	scanf("%s", buffer);      /* Прочита само ANCI    */
	printf("%s", buffer);     /* Отпечатва ANCI       */

	scanf("%s", buffer);      /* Прочита останалите символи, в случая само C */
	printf("%s\n", buffer);   /* Отпечатва C */

	return 0;
}
```



Фиг.40 илюстрира пример 12-3.



Фиг. 40 Четене на данни с scanf()

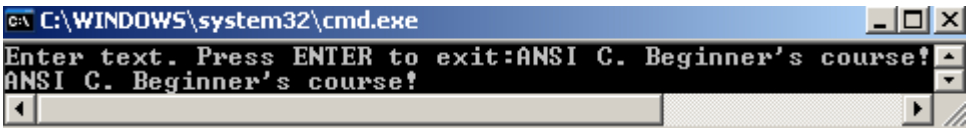
12.2 Четене на символи с getchar()

Функцията getchar() връща като резултат символ, въведен с клавиатурата. Действителното прочитане на символа става след натискане на клавиша ENTER. Следващата програма чете символ по символ низа **"ANSI C. Beginner's course!"**, въведен с клавиатурата, и го отпечатва на екрана отново символ по символ.

Пример 12-4:

```
#include <stdio.h>

int main(void)
{
  char c;

  printf ("Enter text. Press ENTER to exit:");

  /* В този цикъл се изчитат всички въведени символи преди натискане на ENTER */
  do
  {
    c = getchar(); /* Четене на символ от клавиатурата */
    putchar (c);   /* Извеждане на символа на екрана   */
  } while (c != '\n'); /* Четенето спира когато се прочете символ за нов ред */
```

```
  putchar('\n'); /* Отпечатване на нов ред */

  return 0;
}
```



Обърнете внимание, че символите на въведения низ първо постъпват във входен буфер. При натискане на клавиша ENTER в буфера се записва и символ за нов ред '\n'. Всяко извикване на функцията getchar(), в цикъла do-while, извлича (чете и премахва) поредния символ от буфера. При прочитане на символа за нов ред изпълнението на цикъла се прекратява.

Въпроси за самопроверка

1. Напишете обобщената форма на извикване на функцията scanf().

2. Кои са форматиращите последователности за четене на данни тип char, int, float и double с помощта на scanf()?

3. Какъв е синтаксисът на **списък-аргументи** на функцията scanf()?

13 Операции в езика C

13.1 Въведение

C предоставя голям набор от операции, които могат да се разделят на следните категории в зависимост от функционалността си:

- аритметични операции
- операции за сравнение
- логически операции
- битови операции
- операции за присвояване
- други

В зависимост от броя на операндите[1], операциите се разделят на:

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

Забележка[1]: Операнд се нарича единицата, върху която една операция се прилага. В най-простия случай операнд може да бъде променлива или константа.

Повечето бинарни операции изискват операндите да са от един и същ тип. Ако това не е така, компилаторът ще се опита да ги преобразува до един общ тип преди да извърши операцията. Езикът C дефинира определен набор от правила, по които се извършват тези преобразувания. За простота примерите в тази глава използват операнди от един и същ тип за всички такива операции. Правилата за преобразуване на типове са разгледани в следващата глава.

13.2 Аритметични операции

В C има 7 аритметични операции (Табл.28).

операция	описание
+	събиране
-	изваждане отрицание
*	умножение
/	деление
%	деление по модул
++	увеличаване с 1
--	намаляване с 1

Табл. 28 Аритметични операции

Операция събиране +

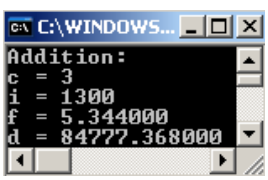
Операцията събиране има следния синтаксис:

операнд1 + операнд2

Резултатът от тази операция е сумата на двата му операнда. Операндите могат да бъдат променливи и константи от всеки един от базовите типове в C.

Пример 13-1:

```
#include <stdio.h>

int main(void)
{
        /* Дефиниране на променливи */
        char   c;
        int    i;
        float  f;
        double d;

        /* Извършване на сумиране */
        c = 1 + 2;
        i = 1000 + 300;
        f = 2.344 + 3.0;
        d = 123.234 + 84654.1340;

        /* Отпечатване на резултатите от сумирането на екрана */
        printf("Addition:\n");
        printf("c = %d\ni = %d\nf = %f\nd = %f\n", c, i, f, d);

        return 0;
}
```



Забележете, че форматният спецификатор, използван за извеждане на стойността на променливата с на екрана, е %d, а не %c. Причината е, че искаме да отпечатаме числовата стойност на тази променлива, а не символа, отговарящ на тази стойност.

Операция изваждане -

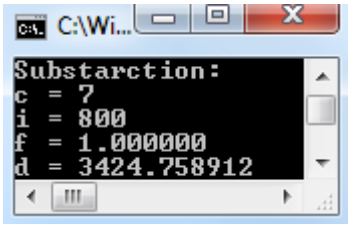
Операцията изваждане има следния синтаксис:

операнд1 - операнд2

Резултатът от тази операция е разликата на двата му операнда. Операндите могат да бъдат променливи и константи от всеки един от базовите типове в C.

Пример 13-2:

```
#include <stdio.h>

int main(void)
{
        /* Дефиниране на променливи */
        char   c;
        int    i;
        float  f;
        double d;

        /* Извършване на изваждане */
        c = 10 - 3;
        i = 1000 - 200;
        f = 2.23 - 1.23;
        d = 3424.857612 - 0.0987;

        /* Отпечатване на резултатите от изваждането на екрана */
        printf("Substarction:\n");
        printf("c = %d\ni = %d\nf = %f\nd = %f\n", c, i, f, d);

        return 0;
}
```



Операция отрицание –

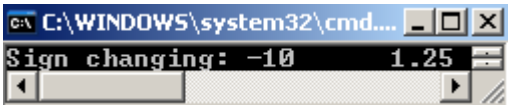
Символът за изваждане също се използва и за отрицание. Тя има следния синтаксис:

- операнд

Резултатът от тази операция е промяна на знака на операнда. Тъй като има само един операнд тази операция се явява унарна. Операндът може да бъде променлива или константа от всеки един от базовите типове в C.

Пример 13-3:

```
#include <stdio.h>

int main(void)
{
	/* Дефиниране на променливи                  */
	int    i;
	float  f;

	/* Присвояване на стойности на променливите */
	i = 10;
	f = -1.25;

	/* Отпечатване на данните на екрана          */
	printf("Sign changing: %d\t%.2f\n", -i, -f);

	return 0;
}
```



Операция умножение *

Операцията умножение има следния синтаксис:

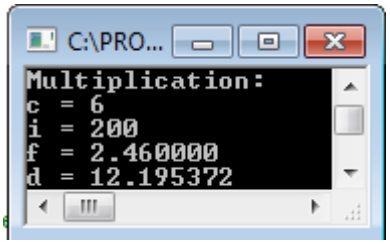
операнд1 * операнд2

Резултатът от тази операция е произведението на двата операнда. Операндите могат да бъдат променливи и константи от всеки един от базовите типове в C.

Пример 13-4:

```
#include <stdio.h>

int main(void)
{
	/* Дефиниране на променливи */
	char   c;
	int    i;
	float  f;
	double d;

	/* Извършване на умножение */
	c = 2 * 3;
```

```
        i = 100 * 2;
        f = 2.0 * 1.23;
        d = 123.56 * 0.0987;

        /* Отпечатване на резултатите от умножението на екрана */
        printf("Multiplication:\n");
        printf("c = %d\ni = %d\nf = %f\nd = %f\n\n\n", c, i, f, d);

        return 0;
}
```



Операция деление /

Операцията деление има следния синтаксис:

операнд1 / операнд2

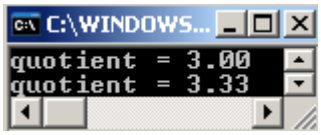
Резултатът от тази операция е частното на двата операнда. Операндите могат да бъдат променливи и константи от всеки един от базовите типове в C.

Операцията деление има следната особеност: когато операндите са цели числа и резултатът е цяло число. Дробната част на резултата се изхвърля в този случай.

Пример 13-5:

```
#include <stdio.h>

int main(void)
{
    int    i_numerator;
    int    i_denominator;
    double quotient, d_numerator, d_denominator;

    /* целочислено деление */
    i_numerator   = 10;
    i_denominator = 3;
    quotient = i_numerator / i_denominator;
    printf("quotient = %.2f\n", quotient);

    /* деление с плаваща запетая */
    d_numerator   = 10.0;
    d_denominator = 3.0;
    quotient = d_numerator / d_denominator;
    printf("quotient = %.2f\n", quotient);
```

```
    return 0;
}
```



Забележете какъв е резултатът при извършване на следното деление: 10 / 3. Вместо 3.33 на екрана се отпечатва 3.00. Причината е, както беше споменато, че при деление на цели числа, дробната част на резултата (ако има такава) се изхвърля, т.е. . резултатът отново е цяло число.

	**При деление на цели числа, дробната част на резултата (ако има такава) се изхвърля.**
	**Ако десният операнд е 0, резултатът е недефиниран.**
	**Винаги проверявайте дали десният операнд е нула преди да извършите операцията деление.**

Ако операндите са от целочислен тип и единият 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 резултатът 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) */

Операция деление по модул %

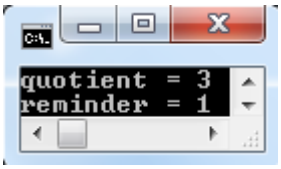
Операцията деление по модул има следния синтаксис:

операнд1 % операнд2

Резултатът от тази операция е остатъкът от делението на двата операнда. Операндите могат да бъдат променливи и константи само от целочислен тип.

Пример 13-6:

```
#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;
}
```



Показаните по-долу примери показват как се получава остатъка при деление на две цели числа.

```
_ 10  %  3 = 3
  9
___
  1  ← Остатък
```

```
_ 10  %  2 = 5
  10
___
  0  ← Остатък
```

Вторият пример показва, че когато делимото се дели точно на делителя, остатъкът е нула.

	**Операндите на операцията % могат да бъдат само от целочислен тип (char, short, int и т.н).**
	**Ако десният операнд е 0, резултатът е недефиниран.**
	**Винаги проверявайте дали десният операнд е нула преди да извършите операцията деление по модул.**

Ако левият операнд е по-малък от десния, резултатът е равен на стойността

на левия операнд.

Пример:

0 % 2 = 0
1 % 3 = 1

Операция увеличаване с 1 (инкрементиране) ++

Операцията инкрементиране има име две форми:

- префиксна форма

++операнд

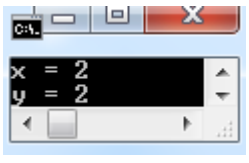
- постфиксна форма

операнд++

Резултатът от тази операция е увеличаване на операнда с 1. Операндът може да е променлива от всеки базов тип. Двете форми са еквивалентни, когато се използват самостоятелно.

Пример 13-7:

```
#include <stdio.h>

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

    x = 1;
    y = 1;

    x++;
    ++y;

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

    return 0;
}
```



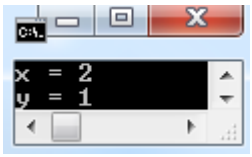
Разликите между двете форми се проявяват, когато операцията ++ е част от

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 с 1.

Пример 13-8:

```
#include <stdio.h>

int main(void)
{
        int  x;
        int  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:

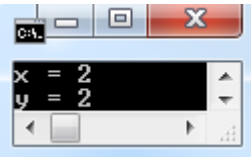
```
y = x;       /* Стойността на x се присвоява на y */
x = x +1;    /* x се увеличава с 1                 */
```

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

Пример 13-9:

```
#include <stdio.h>

int main(void)
{
    int  x;
    int  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 */
```

Операция намаляване с 1 (декрементиране) --

Операцията декрементиране е аналогична на операцията инкрементиране с тази разлика, че резултатът от операцията е намаляване на операнда с 1.

13.3 Операции за сравнение

Табл.29 изброява операциите за сравнение в C.

операция	описание
<	по-малко
<=	по-малко или равно
>	по-голямо
>=	по-голямо или равно
==	равно
!=	различно

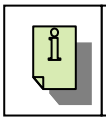
Табл. 29 Операции за сравнение

Всички операции за сравнение са бинарни. Те имат следния синтаксис:

операнд1 > операнд2
операнд1 < операнд2
операнд1 >= операнд2
операнд1 <= операнд2
операнд1 == операнд2
операнд1 != операнд2

Операндите могат да бъдат променливи и константи от всеки един от базовите типове в C. Резултатът от всички операции за сравнение е 0 или 1.

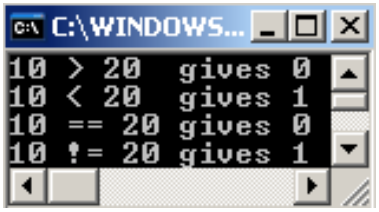
Ако условието е изпълнено, резултатът е 1, в противен случай е 0.



Резултатът от всички операции за сравнение е 0 или 1.

Пример 13-10:

```
#include <stdio.h>

int main(void)
{
	int op1, op2;
	int res;

	op1 = 10;
	op2 = 20;

	res = op1 > op2; /* Резултатът от op1 > op2 е 0 */
	printf("10 > 20  gives %d\n", res);

	res = op1 < op2; /* Резултатът от op1 < op2 е 1 */
	printf("10 < 20  gives %d\n", res);

	res = op1 == op2; /* Резултатът от op1 == op2 е 0 */
	printf("10 == 20 gives %d\n", res);

	res = op1 != op2; /* Резултатът от op1 != op2 е 1 */
	printf("10 != 20 gives %d\n", res);

	return 0;
}
```

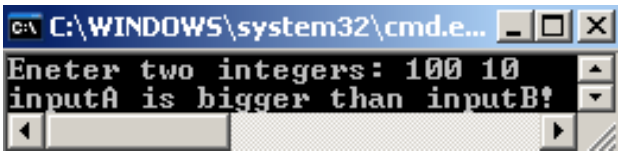


Тези операции типично се употребяват в управляващия израз на оператори за избор и цикъл.

Пример 13-11:

```
#include <stdio.h>

int main(void)
{
	int	inputA;
	int	inputB;

	printf("Eneter two integers: ");
	scanf("%d%d", &inputA, &inputB);

	if (inputA > inputB)
	{
```

```
            printf ("inputA is bigger than inputB!\n");
        }
        else
        {
            printf ("inputA is less than or equal inputB!\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.

13.4 Логически операции

Табл.30 изброява логическите операции в C.

операция	описание
&&	Логическо И
||	Логическо ИЛИ
!	Логическо НЕ

Табл. 30 Логически операции

Логическите операции && и || са бинарни, а ! е унарна. Те имат следния синтаксис:

операнд1 && операнд2
операнд1 || операнд2
! операнд

Операндите могат да бъдат променливи и константи от всеки един от базовите типове в C. Резултатът от всички логически операции е 0 или 1.

Резултатът от всички логически операции е 0 или 1.

В С всяка стойност, различна от нула, се третира като ИСТИНА (**TRUE**), а нулата като ЛЪЖА (**FALSE**).

Логическо И &&

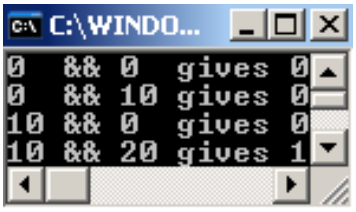
Операцията && се извършва съгласно следната таблица на истинност:

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

Табл. 31 Таблица на истинност на операцията &&

Пример 13-12:

```
#include <stdio.h>

int main( void )
{
	int res;

	res = 0 && 0; /* Резултатът е 0 */
	printf("0  && 0  gives %d\n", res);

	res = 0 && 10; /* Резултатът  е 0 */
	printf("0  && 10 gives %d\n", res);

	res = 10 && 0; /* Резултатът  е 0 */
	printf("10 && 0  gives %d\n", res);

	res = 10 && 20; /* Резултатът  е 1 */
	printf("10 && 20 gives %d\n", res);

	return 0;
}
```



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

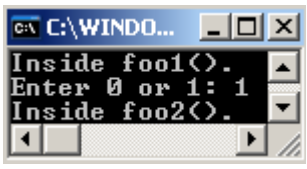
1) Първо се изчислява стойността на левия операнд;

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 крайния резултат се изчислява съгласно таблицата на истинност.

Пример 13-13:

```
#include <stdio.h>

int foo1(void);
int foo2(void);

int  main(void)
{
       int i;

       i = foo1() && foo2();

      return 0;
}

int foo1(void)
{
       int res;

       printf("Inside foo1().\n");
       printf("Enter 0 or 1: ");
       scanf("%d", &res);

       return res;
}

int foo2(void)
{
       printf("Inside foo2().\n");

       return 0;
}
```



Както показват резултатите, ако foo1() върне 0, foo2() не се извиква.

Логическо ИЛИ ||

Операцията || се извършва съгласно следната таблица на истинност:

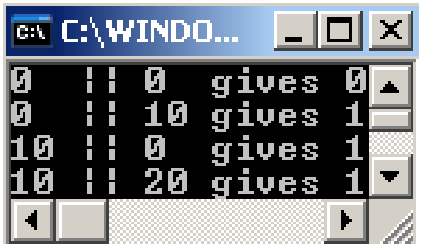
операнд1	операнд2	резултат
FALSE	FALSE	0

FALSE	TRUE	1
TRUE	FALSE	1
TRUE	TRUE	1

Табл. 32 Таблица на истинност на операцията ||

Пример 13-14:

```
#include <stdio.h>

int main(void)
{
	int res;

	res = 0 || 0; /* Резултатът е 0 */
	printf("0  || 0  gives %d\n", res);

	res = 0 || 10; /* Резултатът  е 1 */
	printf("0  || 10 gives %d\n", res);

	res = 10 || 0; /* Резултатът  е 1 */
	printf("10 || 0  gives %d\n", res);

	res = 10 || 20; /* Резултатът  е 1 */
	printf("10 || 20 gives %d\n", res);

	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:

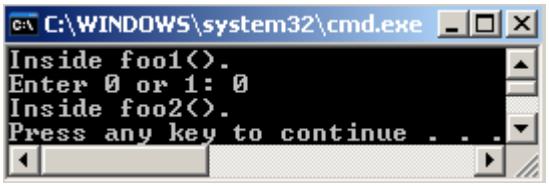
1) Първо се изчислява стойността на левия операнд;

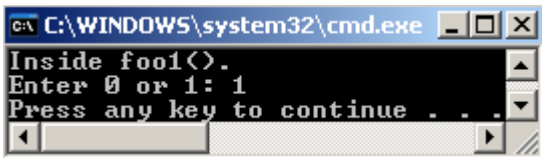
2) Ако стойността на левия операнд е различна от 0, то резултатът от цялата операция е 1 и десният операнд не се изчислява;

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

Пример 13-15:

```
#include <stdio.h>

int foo1(void);
int foo2(void);

int  main(void)
{
      int i;

      i = foo1() || foo2();

     return 0;
}

int foo1(void)
{
      int res;

      printf("Inside foo1().\n");
      printf("Enter 0 or 1: ");
      scanf("%d", &res);

      return res;
}

int foo2(void)
{
      printf("Inside foo2().\n");

      return 0;
}
```





Както показват резултатите, ако foo1() върне 1, foo2() не се извиква.

Логическо НЕ !

Операцията ! се извършва съгласно следната таблица на истинност:

операнд	резултат
FALSE	1
TRUE	0

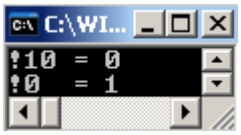
Табл. 33 Таблица на истинност на операцията !

Както показва таблицата на истинност на тази операция, ако операндът има

ненулева стойност (TRUE), резултатът от операцията е 0. Ако операндът е 0 (FALSE), резултатът е 1.

Пример 13-16:

```
#include <stdio.h>

int main(void)
{
	int res;

	res = !10; /* Резултатът е 0 */
	printf("!10 = %d\n", res);
	res = !0; /* Резултатът е 1 */
	printf("!0  = %d\n", res);

	return 0;
}
```



Логическите операции && и || типично се използват да свържат резултатите от две или повече операции за сравнение.

Пример 13-17:

```
#include <stdio.h>

int main(void)
{
	int op1, op2;
	int res;

	op1 = 20;
	op2 = 10;
	res = (op1 > op2) && (op2 != 0); /* Резултатът от (op1 > op2) && (op2 != 0)
	                                    е 1  */
	printf("res = %d\n", res);

	return 0;
}
```

13.5 Битови операции

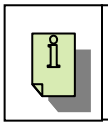
Табл.34 описва всички битови операции в езика C.

операция	описание
&	Битово И (Bitwise AND)
|	Битово ИЛИ (Bitwise OR)
^	Битово Изключващо ИЛИ (Bitwise Exclusive OR)
~	Битово инвертиране
<<	Битово преместване наляво
>>	Битово преместване надясно

Табл. 34 Битови операции

Както показва самото име, тези операции въздействат върху битовете на своите операнди. Битовите операции могат да се прилагат само върху целочислени операнди. Те имат следния синтаксис:

операнд1 & операнд2
операнд1 | операнд2
операнд1 ^ операнд2
операнд1 << операнд2
операнд1 >> операнд2
~ операнд



Битовите операции могат да се прилагат само върху целочислени операнди (char, short, int и т.н.).

Битово И &

Операцията & 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. 35 Таблица на истинност на операцията &

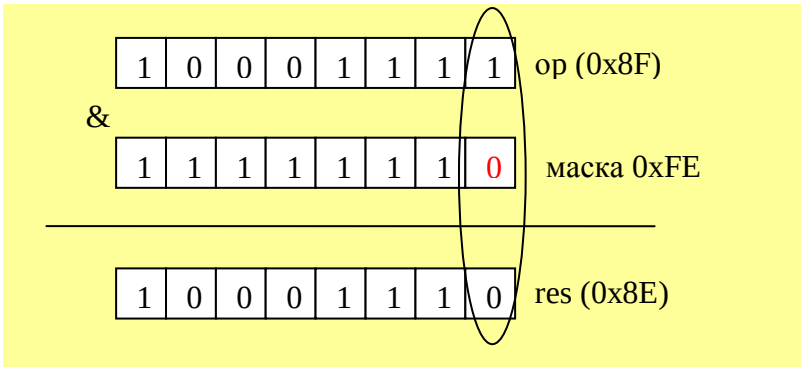
Типична употреба на тази операция е да нулира определени битове на даден операнд.

Пример 13-18:

```
#include <stdio.h>

int main(void)
{
```

```
        unsigned char op;
        unsigned char res;

        op = 0x8F;
        res = op & 0xFE;
        printf("res = %#x\n", res);

        return 0;
}
```

Фиг.41 показва вътрешното двоично представяне на операндите op и 0xFE и извършването на битово И между тях. В случая се нулира бит 0 на op и резултатът се съхранява в променливата res.



Фиг. 41 Нулиране на определени битове

Операндът, който определя кои битове ще се нулират, се нарича маска. В горния пример това е десният операнд, представен чрез шестнайсетичната константа 0xFE. Маската съдържа нули в тези позиции, които трябва да се нулират и единици във всички останали позиции, които не трябва да се променят.

Битово ИЛИ |

Операцията | се извършва съгласно следната таблица на истинност:

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

Табл. 36 Таблица на истинност на операцията |

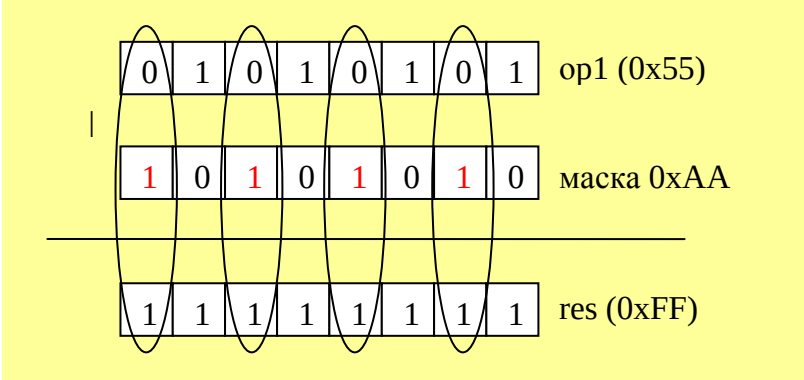
Типична употреба на тази операция е да установи в 1 определени битове на

даден операнд.

Пример 13-19:

```
#include <stdio.h>

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

	op = 0x55;
	res = op | 0xAA;
	printf("res = %#x\n", res);

	return 0;
}
```

Фиг.42 показва вътрешното двоично представяне на операндите op и 0xAA и извършването на битово ИЛИ между тях. В случая се установяват битове 7, 5, 3 и 1 на op и резултатът се съхранява в променливата res.



Фиг. 42 Установяване на определени битове в 1

Маската съдържа единици в тези позиции, които трябва да се установят в 1 и нули във всички останали позиции, които не трябва да се променят.

Битово Изключващо ИЛИ ^

Операцията ^ се извършва съгласно следната таблица на истинност:

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

1	1	0

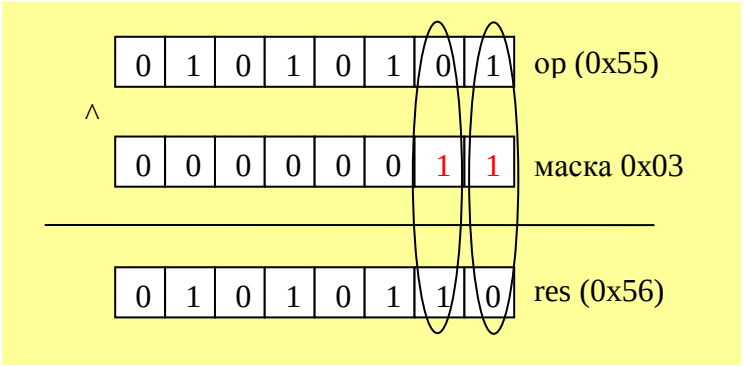
Табл. 37 Таблица на истинност на операцията ^

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

Пример 13-20:

```
#include <stdio.h>

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

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

	return 0;
}
```

Фиг.43 показва вътрешното двоично представяне на операндите op и 0x03 и извършването на битово Изкл. ИЛИ между тях. В случая се инвертират битове 1 и 0 на op и резултатът се съхранява в променливата res.



Фиг. 43 Инвертиране на определени битове

Маската съдържа единици в тези позиции, които трябва да се инвертират и нули във всички останали позиции, които не трябва да се променят.

Битово Инвертиране ~

Тази операция е единствената унарна битова операция и се извършва съгласно следната таблица на истинност:

бит n - операнд	резултат
0	1
1	0

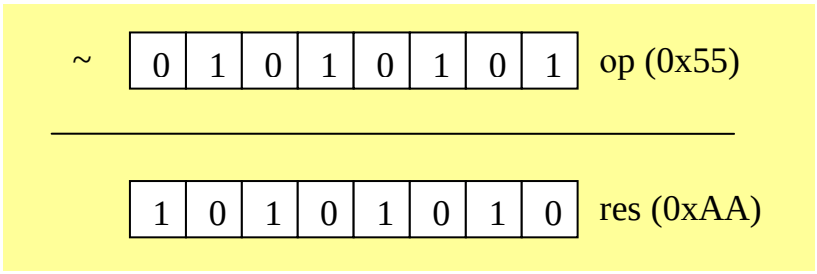
Табл. 38 Таблица на истинност на операцията ~

Както се вижда операцията ~ инвертира битовете на своя операнд.

Пример 13-21:

```
#include <stdio.h>

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

	op  = 0x55;
	res = ~op;
	printf("res = %#x\n", res);

	return 0;
}
```

Фиг.44 показва вътрешното двоично представяне на операнда op и извършването на битово инвертиране ~ върху него. Резултатът се съхранява в променливата res.



Фиг. 44 Инвертиране на всички битове

Битово преместване наляво <<

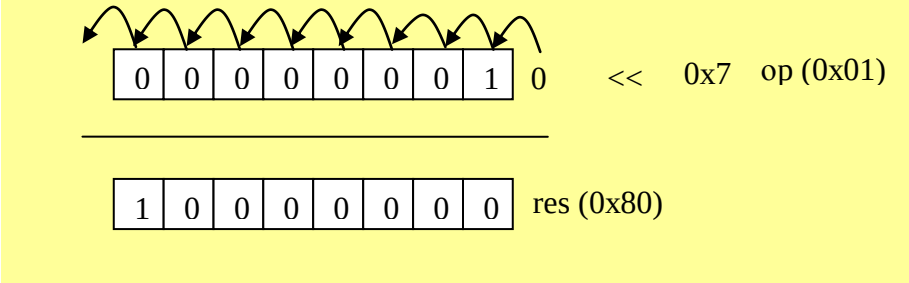
Операцията << премества левия операнд наляво с толкова бита, колкото е указано с десния операнд. При преместването най-левият бит се губи, а позицията на най-десния бит се попълва с 0.

Пример 13-22:

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

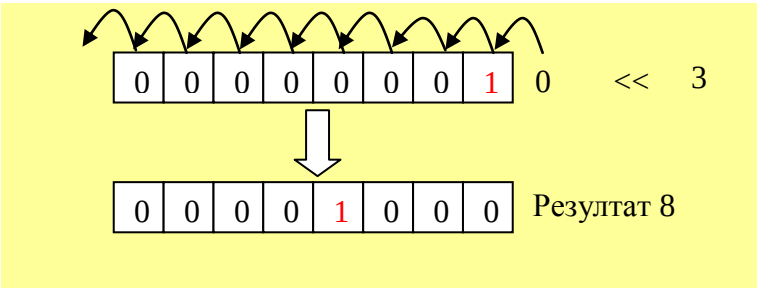
```
int main(void)
{
	unsigned char op;
	unsigned char res;

	op = 0x01;
	res = op << 7;
	printf("res = %#x\n", res);

	return 0;
}
```

Фиг.45 показва вътрешното двоично представяне на операнда op и извършването на битово преместване наляво върху него. Резултатът се съхранява в променливата res.



Фиг. 45 Преместване на ляво

Ако стойността на левия операнд е от беззнаков тип, битовото преместване наляво с n бита е еквивалентно на умножение на операнда с 2^n. Например изразът 1 << 3 дава като резултат $1*2^3 = 8$ (Фиг.46)



Фиг. 46 Умножение на операнд с 2^3

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

	**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.**

Битово преместване надясно >>

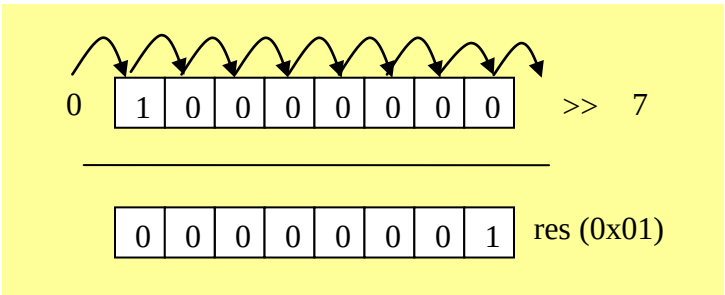
Операцията >> премества левия операнд надясно с толкова бита, колкото е указано с десния операнд. При преместването най-десният бит се губи, а позицията на най-левия бит се попълва съгласно следните правила:

- Ако левият операнд е от беззнаков тип, освободените най-леви битове се запълват с нули;
- Ако левият операнд е от знаков тип и стойността му е неотрицателна, освободените най-леви битове 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 битове с нули (логическо преместване надясно), а други ще ги запълнят със стойността на знаковия бит (знаково преместване на дясно).

Пример 13-23:

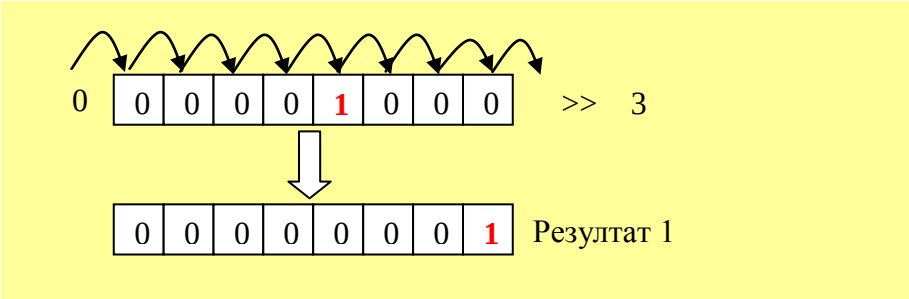
```
#include <stdio.h>

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

	op = 0x80;
	res = op >> 7;
	printf("res = %#x\n", res);

	return 0;
}
```

Фиг.47 показва вътрешното двоично представяне на операнда op и извършването на битово преместване надясно върху него. Резултатът се съхранява в променливата res.



Фиг. 47 Битово преместване надясно

Ако стойността на левия операнд е от беззнаков тип или е от знаков тип, но има неотрицателна стойност, битовото преместване надясно с n бита е еквивалентно на деление на операнда с 2^n. Например изразът 8 >> 3 дава като резултат $8/2^3 = 1$ (Фиг.48).



Фиг. 48 Деление на операнд с 2^3

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.

13.6 Операции за присвояване

Табл.39 показва операциите за присвояване в езика C.

присвояване	еквивалентна операция
операнд1 = операнд2	-
операнд1 += операнд2	операнд1 = операнд1 + операнд2
операнд1 -= операнд2	операнд1 = операнд1 - операнд2
операнд1 *= операнд2	операнд1 = операнд1 * операнд2
операнд1 /= операнд2	операнд1 = операнд1 / операнд2
операнд1 %= операнд2	операнд1 = операнд1 % операнд2
операнд1 &= операнд2	операнд1 = операнд1 & операнд2
операнд1 |= операнд2	операнд1 = операнд1 | операнд2
операнд1 ^= операнд2	операнд1 = операнд1 ^ операнд2
операнд1 <<= операнд2	операнд1 = операнд1 << операнд2
операнд1 >>= операнд2	операнд1 = операнд1 >> операнд2

Табл. 39 Операции за присвояване

Левият операнд трябва да бъде модифицируема L-стойност (променлива, която не е квалифицирана с const). Десният може да бъде променлива,

константа или израз. Типът на десния операнд трябва да бъде съвместим с типа на левия. Ако това не е така, компилаторът ще се опита да го преобразува, съгласно правилата, разгледани в следващата глава.

Пример 13-24:

```
#include <stdio.h>

int main(void)
{
      unsigned char op;

      op   = 10;
      op  += 3;  /* Еквивалентно е на op = op + 3 */
      printf("op = %d\n", op);

      return 0;
}
```

Операцията = се нарича просто присвояване. Всички останали се наричат съставни операции за присвоявания.

	**Използването на съставните оператори за присвояване подобрява производителността на програмата, тъй като левият операнд се изчислява само веднъж.**

13.7 Условна операция

Това е единствената тринарна (с три операнда) операция в C. Тя има следния синтаксис:

операнд1 ? операнд2 : операнд3

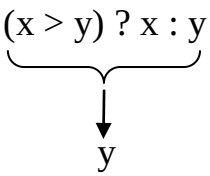
Ако стойността на операнд1 е различна от нула, се изчислява операнд2 и неговата стойност става резултат от цялата операция, в противен случай се изчислява операнд3 и неговата стойност става резултат от цялата операция.

Пример 13-25:

```
#include <stdio.h>

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

    x = 10;
```

```
    y = 20;
    z = (x > y) ? x : y;
    printf("z = %d\n", z);

    return 0;
}
```

В горния пример операнд1 се явява израза (x > y), операнд2 е x и операнд3 е y. Тъй като стойността на операнд1 е 0, то операнд3 става резултат на цялата операция, т.е.



след което y се присвоява на z.

13.8 Операция sizeof

Операцията sizeof се използва за изчисляване на броя байтове, заемани от операнда. Тя има следния синтаксис:

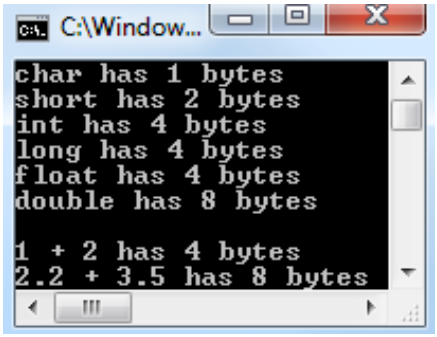
sizeof (тип)
sizeof израз

Операндът може да бъде всеки тип (char, int и т.н.) или израз. Ако операндът е тип, той трябва да бъде сложен в скоби, докато скобите не са задължителни, ако е израз. Ако операндът е израз, операцията sizeof не изчислява израза, а само определя типа на резултата, а от там и броя байтове необходим за съхраняване на този тип.

Пример 13-26:

```
#include <stdio.h>

int main(void)
{
	printf("char has %d bytes\n", sizeof(char));
	printf("short has %d bytes\n", sizeof(short));
	printf("int has %d bytes\n", sizeof(int));
	printf("long has %d bytes\n", sizeof(long));
	printf("float has %d bytes\n", sizeof(float));
	printf("double has %d bytes\n\n", sizeof(double));
```

```
        printf("1 + 2 has %d bytes\n", sizeof(1 + 2));
        printf("2.2 + 3.5 has %d bytes\n", sizeof(2.2 + 3.5));

        return 0;
}
```



Друго важно нещо, което трябва да запомните, е, че резултатът на операцията sizeof се изчислява по време на компилиране, а не по време на изпълнение на програмата.

	Резултатът от операцията sizeof се изчислява по време на компилиране, а не по време на изпълнение на програмата.

13.9 Операция за последователно изчисляване на изрази

В C символът за запетая играе роля на разделител (пунктуатор) в един контекст (например в списъка с аргументи на функция) и на операция за последователно изчисление на изрази в друг. Тя има следния синтаксис:

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 надясно 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.

Пример 13-27:

```
#include <stdio.h>

int main(void)
{
    int x, y;

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

```
    return 0;
}
```

В показания пример изразът x = 10 е операнд1, израза y = x е операнд2 и изразът printf("y = %d\n",y) е операнд3, който в крайна сметка става резултат от цялата операция. Тъй като целият този израз представлява оператор, той завършва с точка и запетая.

13.10 Други операции

Табл.40 показва в резюме накратко всички останали C операции. Тъй като разбирането на тези операции изисква допълнителни познания, те са описани накратко тук за пълнота. По-подробно описание ще намерите в съответните глави както е показано в таблицата.

операция	описание	коментар
*	косвен достъп	вижте 17 Указатели
&	извличане на адрес	вижте 17 Указатели
()	извикване на функция	вижте 22 Функции
[]	индексиране на масив	вижте 16 Масиви
.	директен достъп до член на struct-/union- променлива	вижте 18 Структури 19 Обединения
->	косвен достъп до член на struct-/union- променлива	вижте 18 Структури 19 Обединения
(тип)	преобразуване на тип	вижте 14 Изрази и преобразувания на типове

Табл. 40 Други C операции

13.11 Приоритет и асоциативност на операциите

Всяка операция в C има приоритет и асоциативност. Приоритетът указва коя операция в един израз се извършва първа измежду операции с различен приоритет, а асоциативността указва коя операция се извършва първа измежду операции с еднакъв приоритет. Табл.41 показва приоритета и асоциативността на всички C операции. Операциите, разположени в една група, имат еднакъв приоритет.

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 постинкрементиране, постдекрементиране	от ляво надясно	най-висок
++x, --x **sizeof** **~** **!** **-** **+** **&** ***** **(тип)**	преинкрементиране, предекрементиране размер на обект битово отрицание логическо отрицание унарен минус унарен плюс извличане на адрес косвен достъп до обект преобразуване на тип	**от дясно наляво**	
*** / %**	умножение, деление, деление по модул	от ляво надясно	
+ -	събиране, изваждане	от ляво надясно	
<< >>	битово преместване наляво и надясно	от ляво надясно	
< <= > >=	оператори за сравнение	от ляво надясно	
== !=	оператори за равенство	от ляво надясно	
&	битово И	от ляво надясно	
^	битово Изкл. ИЛИ	от ляво надясно	
|	битово ИЛИ	от ляво надясно	
&&	логическо И	от ляво надясно	
||	логическо ИЛИ	от ляво надясно	
?:	условен оператор	**от дясно наляво**	
= **+= -=** ***= /= %=** **<<= >>=** **&= |= ^=**	просто присвояване, съставни присвоявания	**от дясно наляво**	
,	последователно изчисление	от ляво надясно	най-нисък

Табл. 41 Приоритет и асоциативност на операциите

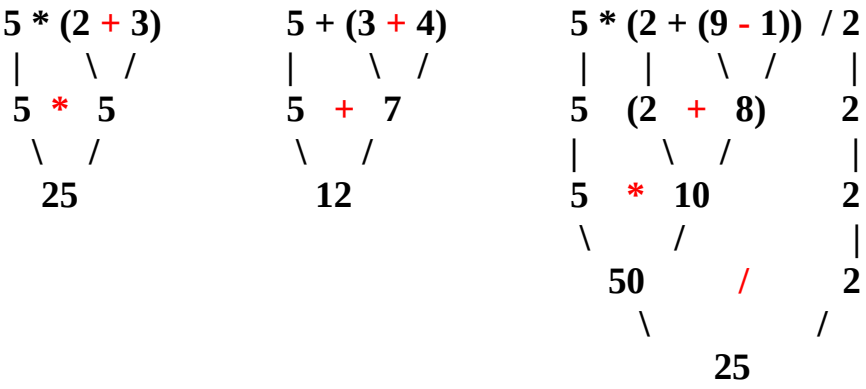
Асоциативност "от ляво надясно" означава, че от всички операции с еднакъв приоритет първа ще се изчисли операцията, която се намира най-вляво в израза. Асоциативност "от дясно наляво" означава, че от всички операции с еднакъв приоритет първа ще се изчисли операцията, която се намира най-вдясно в израза.

Пример:

```
5 * 2 + 3          5 + 3 + 4
 \  /    |         \   /   |
  10  + 3            8  + 4
   \    /              \   /
    13                  12
```

Приоритетът на операциите може да се променя с използването на скоби. Изразът в скоби се изчислява винаги пръв. Ако изразът в скоби съдържа друг израз в скоби, той се изчислява пръв.

Пример:



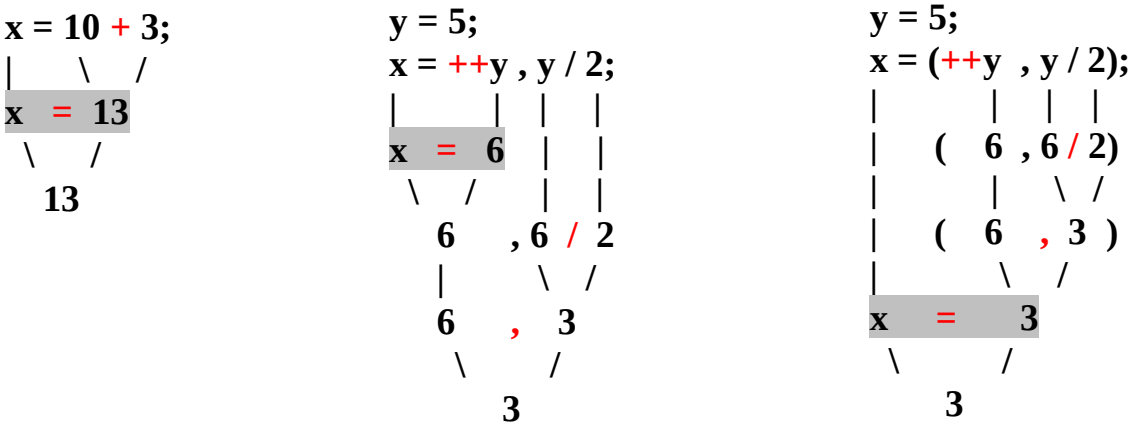
Както се вижда от горните примери, стойността на един израз може да зависи от реда на изчисление на операциите в него. Важно нещо, което трябва да запомните, е, че редът на изчисление на операндите на една бинарна операция не е дефиниран от стандарта. Един компилатор може първо да изчисли левия операнд и след това десния, докато друг компилатор може да ги изчисли в обратен ред. Изключение от това правило са бинарните операциите && и ||, при които левият операнд се изчислява винаги пръв. Код, който разчита на определен ред на изчисление на операндите е непреносим.

Пример: x = f1() + f2();

Редът на извикване на функциите f1() и f2() зависи от имплементацията на компилатора.

Забележете, че операциите за присвояване имат най-нисък приоритет след операцията запетая. Ето защо ако десният операнд е израз, той ще се изчисли и след това получената стойност ще се присвои на левия операнд. Внимавайте обаче, ако десният операнд се състои от изрази, разделени със запетаи. В този случай операцията присвояване ще се извърши първа. Ако е необходимо обаче всички изрази на десният операнд да се изчислят преди операцията присвояване, е необходимо използването на скоби.

Пример:



Редът на изчисление на операндите на една бинарна операция не е дефиниран от стандарта. Изключение са бинарните операциите && и ||, при които левият операнд се изчислява винаги пръв.

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.

Винаги използвайте скоби, за да укажете явно реда на изчисление на операциите в един израз. Това подобрява четимостта на израза и намалява риска от грешки.

Въпроси за самопроверка

1. Избройте категориите операции в езика C.

2. Какъв ще бъде резултатът от делението на две цели числа?

3. Какъв ще е резултатът от делението 13 / 2? Напишете примерна програма, която изобразява резултата на екрана.

4. Операторът % може да се използва с цели и дробни числа. Вярно или невярно?

5. Какъв ще е резултатът от делението по модул 121 % 21? Напишете примерна програма, която изобразява резултата на екрана.

6. Какъв е резултатът от логическите операции?

7. Каква ще бъде стойността на y в следната програма?

```
#include <stdio.h>

int main(void)
{
	int x,y;

	x = 10;
	y = 0;

	(x < 0) && (y = --x + 1);

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

	return 0;
}
```

8. Ако имате променлива x тип int със стойност 0x2355, напишете код, който установява в 1 бит 7 и нулира бит 0, без да променяте останалите битове.

9. Каква ще бъде стойността на x в следната програма?

```
#include <stdio.h>

int main(void)
{
	int x;

	x = (3 != 0) ? 1 : 0;

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

	return 0;
}
```

10. Ако тип int е 4 байта, определете стойностите на променливите x и y в следния пример:

```
int y, x = 1;
y = sizeof(++x);
```

11. Обяснете за какво се използва приоритетът и асоциативността на операциите?

12. Какъв е редът на изчисление на операциите в следния израз?

    2 * (10 + 3) - 3

14 Изрази и преобразувания на типове

14.1 Въведение

Израз е комбинация от операнди и операции. Операндите могат да бъдат променливи, константи, функции връщащи резултат и т.н. В най-простия случай израз може да бъде само променлива или само константа. Всеки израз има тип. Това е типът на стойността, получена след изчисляване на израза. Ако един израз не връща стойност, неговият тип е void. В C е допустимо в един израз да участват операнди (променливи, константи и т.н.) от различни типове. В този случай компилаторът преобразува автоматично типовете на всички операнди до един общ тип по определени правила по време на изчисление на израза.

14.2 Обичайни унарни преобразувания

Табл.42 описва обичайните унарни преобразувания. Засега може да игнорирате защрихованите полета. Тези преобразувания ще станат ясни, когато разгледаме масивите, указателите, битовите полета и изброяванията.

ако операндът е	той се преобразува в
char, unsigned char, signed char	int
signed short	int
unsigned short	▪ int (ако int може да представи всички стойности на signed short) ▪ unsigned int (ако int не може да представи всички стойности на signed short)
масив от тип T	указател към тип T
функция връщаща тип T	указател към функция връщаща тип T
битово поле int	▪ 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	int

Табл. 42 Обичайни унарни преобразувания

Обичайните унарни преобразувания се прилагат автоматично върху:

- операндите на унарните операции !, -, +, ~, * [1]
- по отделно върху операндите на битовите операции << и >>
- по отделно върху операндите на логическите операции **&&** и ||
- върху операндите на всички останали бинарни операции
- име на масив (Масивите са разгледани в 16 Масиви)
- име на функция (Функциите са разгледани в 22 Функции).

Забележка[1] : Има се предвид операцията за косвен достъп, а не операцията за умножение (тя е бинарна). Тези две операции използват един и същ символ.

Както се вижда от таблицата, операндите от тип **char** (signed или unsigned) и **short** (signed или unsigned) в един израз се преобразуват автоматично до тип int или unsigned int. Ако стойността на типа може да се представи от тип int, то тя се преобразува в тип int, в противен случай се преобразува в тип unsigned int[1]. Това преобразуване се нарича още **целочислено повишаване на типа.**

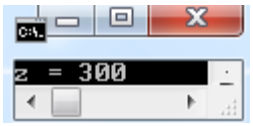
Забележка[1] : Ако тип unsigned short и int са с еднакъв размер (16 бита), то int не може да представи всички стойности на unsigned short. В този случай компилаторът преобразува операнда в тип unsigned int.

Обърнете внимание, че левият операнд на операцията присвояване не се подлага на целочислено повишаване.

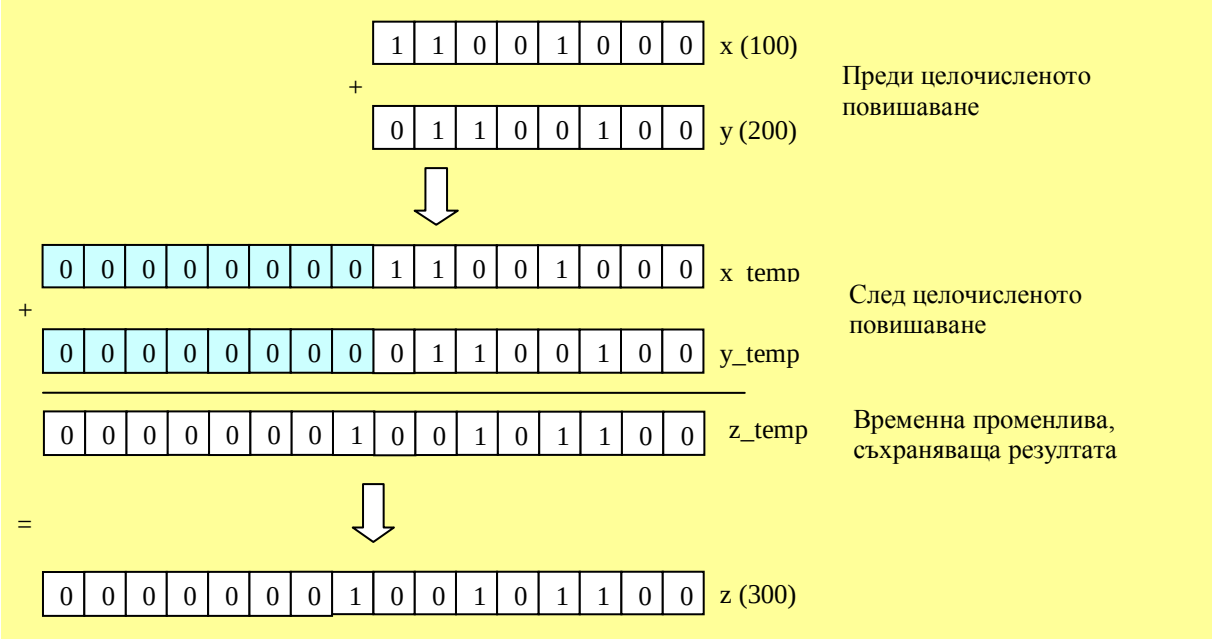
Именно целочисленото повишаване е причина да не се получава препълване, когато сумирате два операнда тип char (signed или unsigned).

Пример 14-1:

```
#include <stdio.h>

int main (void)
{
	unsigned char x = 200, y = 100;
	int           z;

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

	return 0;
}
```

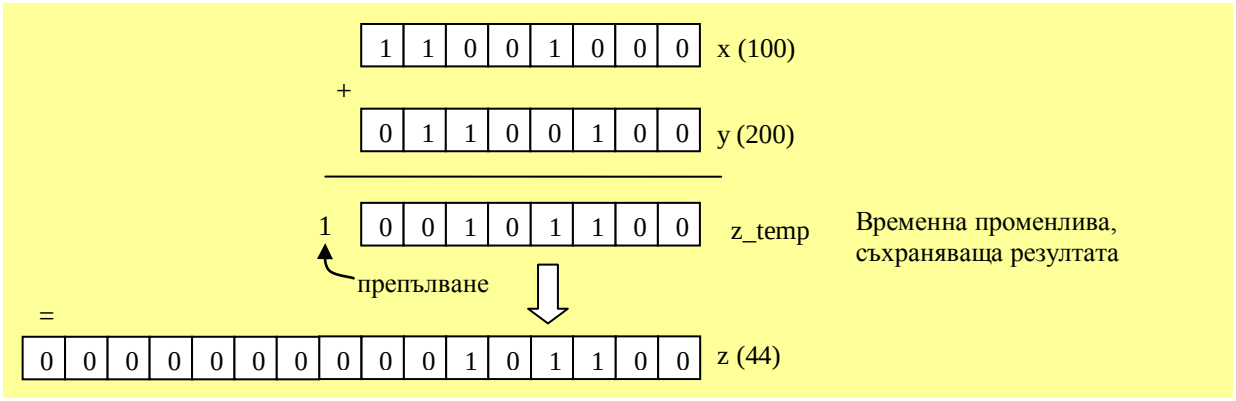


Нека разгледаме израза z = x + y подробно. Фиг.49 показва вътрешното представяне на променливите x, y преди и след целочисленото повишаване. В примера се приема, че тип int е 16 бита.



Фиг. 49 Вътрешно представяне на израза z = x + y

Обърнете внимание, че ако операндите x и y не се повишаваха до тип int, резултатът от сумирането щеше да надхвърли горната граница на числовия диапазон, който може да се представи от тип unsigned char (255) и в променливата z щеше да се съхрани резултат, различен от очаквания (Фиг.50).



Фиг. 50 Резултат при липса на целочислено повишаване

Резултатът от сумирането на 8 битовите променливи x и y генерира препълване, което няма къде да се съхрани и се губи.

Забележете също, че при целочисленото повишаване се създава временна променлива, която съдържа преобразуваната стойност. Оригиналната променлива не се променя.

14.3 Обичайни бинарни преобразувания

Обичайните бинарни преобразувания се прилагат автоматично върху:

- операндите на аритметичните операции +, -, *, /, %
- операндите на операции за сравнение <, <=, >, >=, ==, !=
- операндите на битовите операции &, |, ^
- втория и третия операнд на условната операция ?:

При изчисляването на израз, компилаторът първо прилага обичайните унарни преобразувания съгласно Табл.42. След това започва изчислението от операцията с най-висок приоритет. Ако операндите на тази операция са от различен тип, компилаторът ги преобразува до един общ тип съгласно следните правила:

№	ако единият операнд е	а другият е	компилаторът преобразува операндите в
1	long double	double, float, unsigned long, signed long, unsigned int, signed int	long double
2	double	float unsigned long signed long unsigned int signed int	double
3	float	unsigned long signed long unsigned int signed int	float
4	unsigned long	signed long unsigned int signed int	unsigned long
5	signed long	unsigned int	signed long ако signed long може да

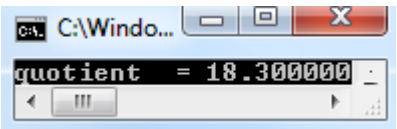
			представи всички стойности на unsigned int, иначе двата операнда се преобразуват в unsigned long[1]
6	signed long	signed int	signed long
7	unsigned int	signed int	unsigned int
8	signed int	char (signed или unsigned) short (signed или unsigned) битово поле enum	signed int или unsigned int съгласно Табл.42 и правило 7
9	char (signed или unsigned) short (signed или unsigned) битово поле enum	char (signed или unsigned) short (signed или unsigned) битово поле enum	signed int или unsigned int съгласно Табл.42 и правило 7

Табл. 43 Обичайни бинарни преобразувания

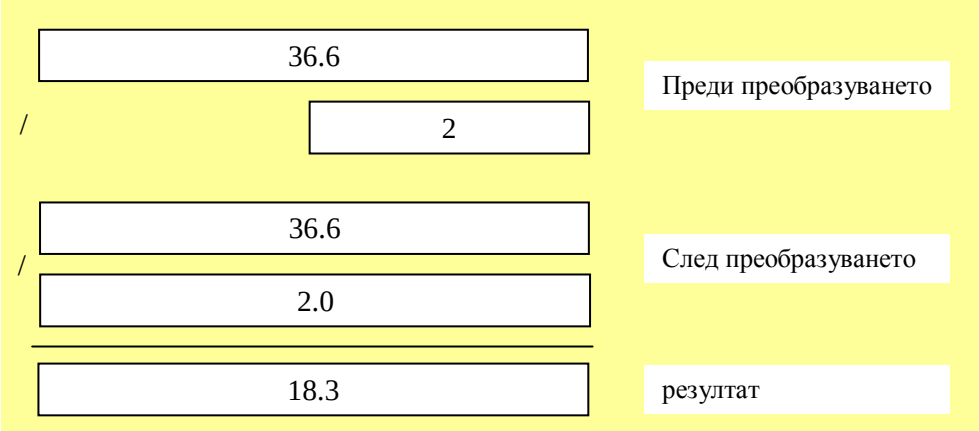
Забележка[1] : Ако тип long и int са с еднакъв размер (32 бита), signed long не може да представи всички стойности на unsigned int.

Пример 14-2:

```
#include <stdio.h>

int main(void)
{
	double quotient;

	quotient = 36.6 / 2;
	printf("quotient  = %f\n", quotient);

	return 0;
}
```

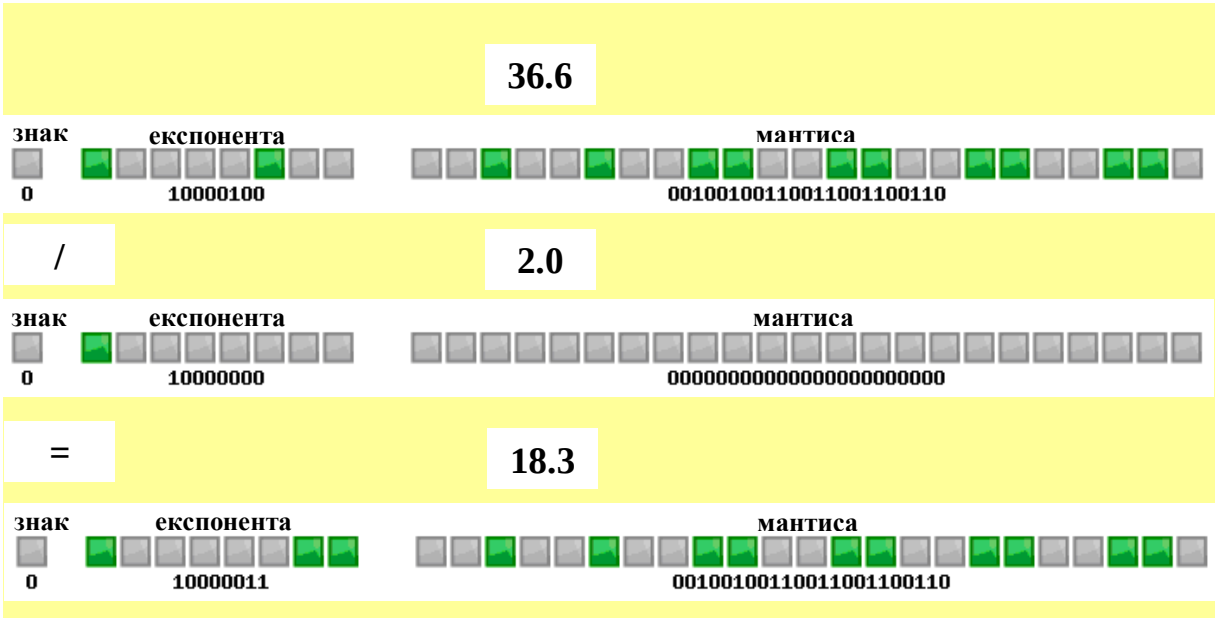


Нека разгледаме израза 36.6 / 2 по-подробно. Съгласно Табл.43, компилаторът автоматично преобразува операнд2 (тип int) в тип double. Преобразуването се състои в представяне на целочисления операнд2 в формата, използван за представяне на числата с плаваща запетая за конкретната система. Фиг.51 илюстрира образно процеса на преобразуване. Приема се, че тип int е 2 байта, а double – 4 байта.



Фиг. 51 Преобразуване на тип int в double

Фиг.52 показва вътрешното битово представяне на операндите след преобразуването и резултата в стандарта IEEE 754 с единична точност.



Фиг. 52 Вътрешно представяне на 36.6/2 = 18.3 в IEEE 754 с единична точност

Внимавайте, когато смесвате знакови и беззнакови целочислени операнди. Разгледайте внимателно следния пример, за да разберете проблема, който може да възникне.

Пример 14-3:

```
#include <stdio.h>

int main(void)
{
    int value = -1;
    unsigned int limit = 200U;

    if ( value < limit ) /* <--- коварно преобразуване на типовете */
    {
       printf("This will not be printed on the screen!\n");
    }
```

```
    else
    {
       printf("Be careful when mix unsigned and signed values!\n");
    }

    return 0;
}
```

На пръв поглед може би очаквате, че условието value < limit ще бъде вярно при дадените стойности на променливите value и limit и програмата ще отпечата съобщението **This will not be printed on the screen!**. Но ако изпълните програмата, ще видите, че това не е така. Отговорът се крие в правилото за обичайните бинарни преобразувания, че ако единият операнд е unsigned int, а другият е signed int, то и двата операнда се преобразуват в unsigned int. В нашия пример value е тип signed int и ще се преобразува в unsigned int автоматично от компилатора. На практика при това преобразуване вътрешното представяне няма да се промени, а ще се промени само интерпретацията на битовете. За да разберете какво се има предвид, нека допуснем, че компилаторът използва допълнителен код за представяне на отрицателни числа. Тогава value = -1 ще изглежда така (приема се, че тип int е 16 бита): 1111111111111111. Ако това число се интерпретира като беззнаково, неговата стойност е 65535_{10}. Тогава се получава условието 65535 < 200, което не е вярно и компилаторът прескача тялото на if и изпълнява оператора в тялото на else.

Внимавайте, когато смесвате знакови и беззнакови целочислени операнди в изрази.

14.4 Преобразувания при присвояване на аритметични типове

При просто присвояване нормално типът на десния операнд е същият 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.

Най-общо казано, ако стойността на десния операнд се побира в диапазона от стойности на левия операнд, то стойността на десния операнд се копира в левия без промяна.

Ако левият операнд е от знаков тип, а десният операнд е от знаков или беззнаков тип и стойността му излиза извън диапазона от стойности на

левия операнд, то резултатът зависи от имплементацията. Тази ситуация е известна като целочислено препълване.

Ако левият операнд е от беззнаков тип, а десният операнд е от беззнаков тип и стойността му излиза извън диапазона от стойности на левия операнд, то излишните старши байтове на десния операнд се изхвърлят, а младшите се копират в левия операнд.

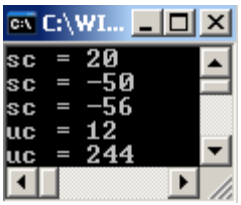
Ако левият операнд е от беззнаков тип, а десният операнд е от знаков тип и стойността му излиза извън диапазона от стойности на левия операнд, то са възможни два сценария:

1. Ако стойността на десния операнд е неотрицателна, от нея се изважда число с 1 по-голямо от максималното число, което може да се представи от левия операнд, толкова пъти, докато получената стойност попадне в обхвата му от стойности;

2. Ако стойността на десния операнд е отрицателна, към нея се добавя число с 1 по-голямо от максималното число, което може да се представи от левия операнд, толкова пъти, докато получената стойност попадне в обхвата му от стойности.

Пример 14-4:

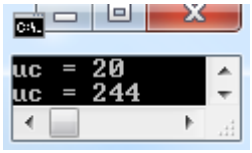
```
#include <stdio.h>

int main(void)
{
	unsigned char    uc; /* диапазон от стойности     0  -   +255 */
	signed   char    sc; /* диапазон от стойности   -127 -   +128 */
	signed   short   ss; /* диапазон от стойности -32767 - +32768 */

	uc = 20;
	sc = uc; /* 20 -> sc */
	printf("sc = %d\n", sc);

	ss = -50;
	sc = ss; /* -50 -> sc */
	printf("sc = %d\n", sc);

	uc = 200;
	sc = uc; /* Препълване с имплементационно-дефиниран резултат, ??? -> sc */
	printf("sc = %d\n", sc);

	ss = -500;
	uc = ss; /* ss + (uc_max + 1)x2 -> uc */
	printf("uc = %u\n", uc);

	ss = 500;
	uc = ss; /* ss - (uc_max + 1)x1 -> uc */
	printf("uc = %u\n", uc);

	return 0;
}
```



Ако левият операнд е от целочислен тип, а десният е тип с плаваща запетая, то дробната част на десния операнд се изхвърля. Ако цялата част на десния операнд излиза извън диапазона от стойности на левия операнд, резултатът е недефиниран. Това е вид препълване.

Пример 14-5:

```
#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;
}
```



Ако левият операнд е тип с плаваща запетая, а десният е от целочислен тип, то 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.

Пример 14-6:

```
#include <stdio.h>

int main(void)
{
	float f;

	f = 1278459999;
	printf("f = %.0f\n", f);

	return 0;
}
```

14.5 Явно преобразуване на типове

Езикът C поддържа операция за явно (изрично) преобразуване на един тип в друг. Тя има следния синтаксис:

(тип) операнд

където

тип
: Указва към какъв тип трябва да се преобразува **операнд**.

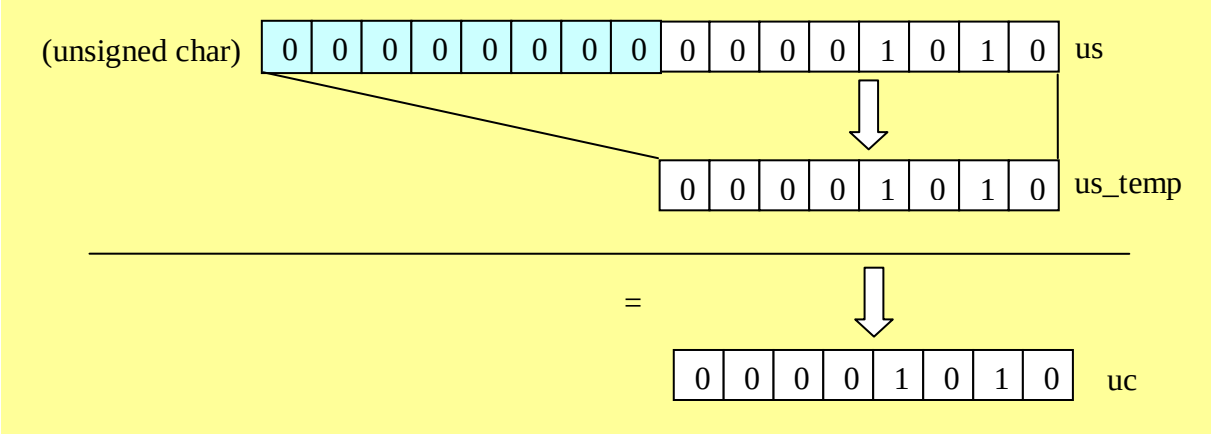
операнд
: Може да бъде променлива или константа от всеки един от базовите типове.

Явното преобразуване на аритметични типове следва същите правила като присвояване.

Пример 14-7:

```
#include <stdio.h>

int main(void)
{
	unsigned char	uc;
	unsigned short	us;

	us = 10;
	uc = (unsigned char)us; /* unsigned short -> unsigned char */

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

	return 0;
}
```

Фиг.53 показва какво става под "капака", когато компилаторът срещне израза `uc = (unsigned char)us;`



Фиг. 53 Явно преобразуване на типове

Както се вижда от Фиг.53 преобразуването на тип unsigned short в unsigned char се състои просто в изхвърляне на старшия байт на променливата us. Младшият байт на us се копира във временна променлива us_temp, която представлява и резултат от преобразуването. Именно тази временна променлива се присвоява на променливата uc.

Обърнете внимание, че при преобразуване на типа, оригиналният тип на променливата не се променя. Преобразуването се извършва само вътрешно по време на операцията преобразуване и резултатът се съхранява във временна променлива. Именно тази временна променлива се използва, ако преобразуваната променлива е част от израз, например в присвояване.

	**Когато смесвате операнди с различни типове в един израз, за предпочитане е да използвате явно преобразуване на операндите вместо да разчитате на компилатора. Това подсказава, че смесването на типове е умишлено, а не по невнимание.**

	**Внимавайте, когато преобразувате целочислен "по-голям" тип към целочислен "по-малък" тип. Ако по-малкият тип е знаков, е възможно да получите недефиниран резултат, а ако е беззнаков е възможно загуба на информация поради изхвърляне на старшите байтове на по-големия тип.**

Въпроси за самопроверка

1. Какво е израз в езика C ?

2. Какво представлява целочисленото повишаване и кога се прилага ?

3. Опишете правилата за обичайните бинарни преобразувания.

4. Ако променливата d е тип double, а i е тип int, какъв ще е типът на резултата от изчислението на израза d + i?

5. Каква ще е стойността на променливата res в следната програма?

```
#include <stdio.h>

int main(void)
{
	unsigned char op1, op2, res;

	op1 = 200;
	op2 = 100;

	res = op1 + op2;

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

	return 0;
}
```

6. Опишете правилата за преобразуване при присвояване на аритметични типове.

7. Каква ще е стойността на променливата res в следната програма?

```
#include <stdio.h>

int main(void)
{
	unsigned char res;

	res = 10.345;

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

	return 0;
}
```

8. Какъв е резултатът от следните явни преобразувания?

```
(unsigned char)0xaa55
(int)20.34
```

15 Управляващи оператори

15.1 Въведение

Нормално една C програма изпълнява един оператор, след което преминава към следващия оператор и т.н. Понякога е необходимо последователното изпълнение на операторите да бъде нарушено и една група оператори да бъде изпълнена при едни условия, а друга група оператори при други условия, или група оператори да бъде изпълнена многократно преди по-нататъшно изпълнение на програмата. За тази цел езикът C предоставя две категории оператори: оператори за избор и оператори за цикъл.

15.2 Оператори за избор (условни оператори)

15.2.1 Оператор if-else

Операторът if-else се използва за отклонение на програмата по един от два възможни пътя в зависимост от стойността на управляващ израз (условие). Той има следния синтаксис:

```
if (управляващ-израз)
{
     if-оператори
}
else
{
     else-оператори
}
```

където

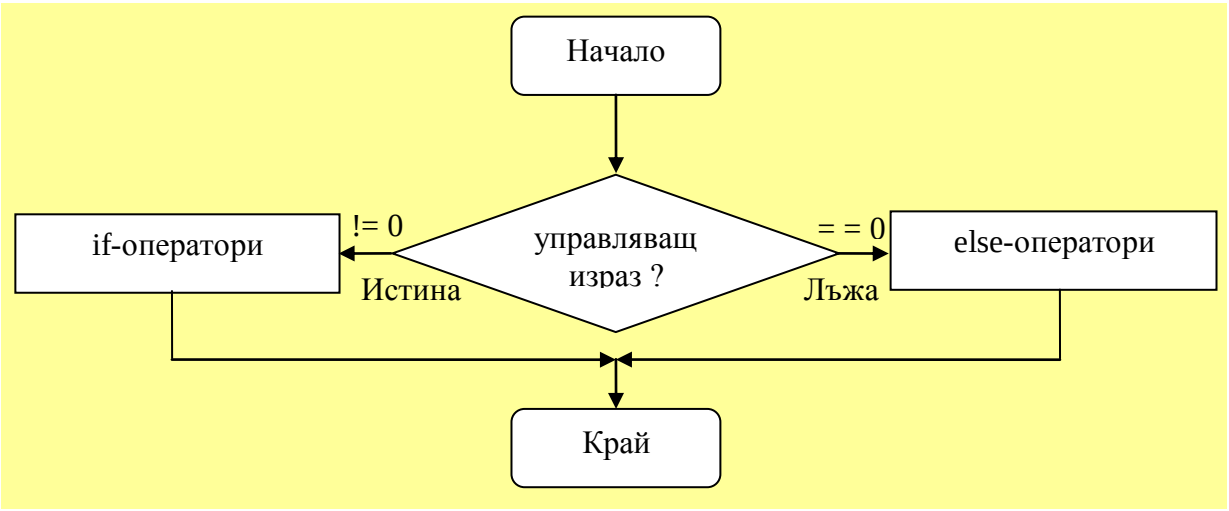
управляващ-израз
Може да бъде всеки израз, който връща стойност различна от void.

if-оператори / else-оператори
Може да бъде всеки C оператор, включително и друг if-else оператор.

Ако стойността на **управляващ-израз** е различна от нула (е ИСТИНА), програмата преминава към изпълнение на операторите, означени като if-оператори. След изпълнението на тези оператори програмата преминава към изпълнение на първия оператор след if-else, прескачайки else-операторите.

Ако стойността на **управляващ-израз** е нула (е ЛЪЖА), програмата преминава към изпълнение на операторите, означени като else-оператори, прескачайки if-операторите. След изпълнението на тези оператори програмата преминава към изпълнение на първия оператор след if-else.

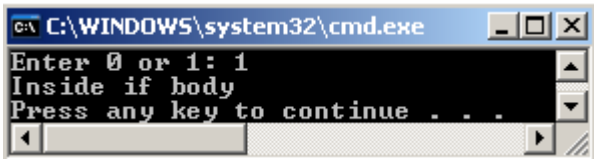
Фиг.54 показва if-else в графичен вид.

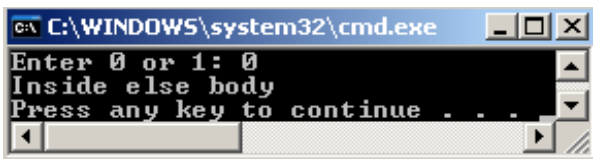


Фиг. 54 if-else оператор

Пример 15-1:

```
#include <stdio.h>

int main(void)
{
	int choice;

	printf("Enter 0 or 1: ");
	scanf("%d", &choice);

	if (choice)
	{
		printf("Inside if body\n");
	}
	else
	{
		printf("Inside else body\n");
	}

	return 0;
}
```



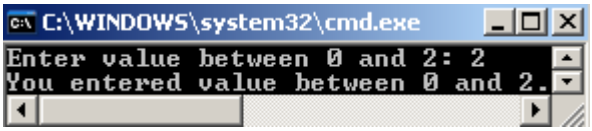


Операторът else не е задължителен и може да се пропусне, т.е. . следната конструкция е напълно валидна:

```
if (управляващ-израз)
{
     if-оператори
}
```

Пример 15-2:

```
#include <stdio.h>

int main(void)
{
      int choice;

      printf("Enter value between 0 and 2: ");
      scanf("%d", &choice);

      if ( (choice >= 0) && (choice <= 2) )
      {
            printf("You entered value between 0 and 2.\n");
      }

      return 0;
}
```



Операторите в тялото на if и else могат също да бъдат if-else оператори.

Пример 15-3:

```
#include <stdio.h>

int main(void)
{
      int choice;

      printf("Enter 0 or 1: ");
      scanf("%d", &choice);

      if ((0 == choice) || (1 == choice) )
      {
            if (choice)
            {
                  printf("Inside if body\n");
                  return 0;
            }
            else
            {
                  printf("Inside else body\n");\
                  return 0;
            }
      }

      printf ("You made a wrong choice! Bye Bye!\n");

      return 0;
```

```
}
```

Когато тялото на if или else се състои от единичен оператор, фигурните скоби могат да се пропуснат.

Пример 15-4:

```
#include <stdio.h>

int main(void)
{
	int choice;

	printf("Enter 0 or 1: ");
	scanf("%d", &choice);

	if (choice)
		printf("Inside if body\n");
	else
		printf("Inside else body\n");

	return 0;
}
```

Препоръчително е винаги да използвате фигурни скоби дори тялото на if или else да се състои от единичен оператор.

Въпреки че операторът if-else позволява изпълнението на програмата да продължи по един от два възможни пътя, възможно е 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.

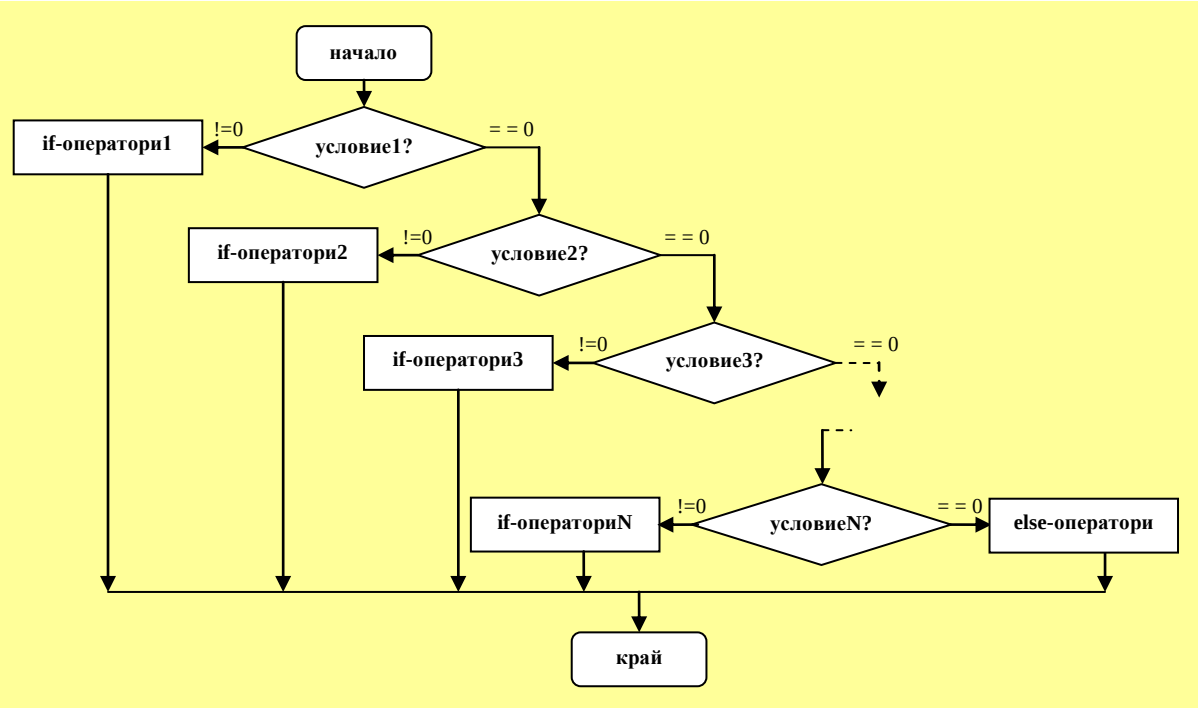
if-else-if стълбица

```
if(условие1)
	{if-оператори1}
else
	if(условие2)
		{if-оператори2}
	else
		if(условие3)
			{if-оператори3}
		else
			...
				else
					{else-оператори}
```

Тази конструкция може да се запише компактно по следния начин:

```
if(условие1)
     {if-оператори1}
else if(условие2)
     {if-оператори2}
else if(условие3)
     {if-оператори3}
...
else
     {else-оператори}
```

Фиг.55 представя if-else-if стълбицата в графичен вид.



Фиг. 55 if-else-if стълбица

При if-else-if стълбицата се изпълнява 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.

15.2.2 Оператор switch

Операторът switch служи за отклонение на програмата по един от няколко възможни пътя в зависимост от стойността на **целочислен управляващ израз**. Той има следния синтаксис:

```
switch (управляващ-израз)
{
case константа1:
     оператори1
     break;

case константа2:
     оператори2
     break;
...
case константаN:
     операториN
     break;

default:
     default-оператори
     break;
}
```

където

управляващ-израз
Може да бъде само израз, който връща цяло число като резултат.

константа1,...константаN
Може да бъде само целочислен константен израз.

оператори1 / оператори2..../ операториN
Може да бъде всеки C оператор, включително и друг switch оператор.

Ако стойността на целочисления **управляващ-израз** съвпада със стойността на някоя от целочислените константи **константа1,...константаN**, програмата преминава към изпълнение на операторите, които се намират след нея. При достигане на оператора break програмата излиза от тялото на оператора switch и преминава към изпълнение на първия оператор след него.

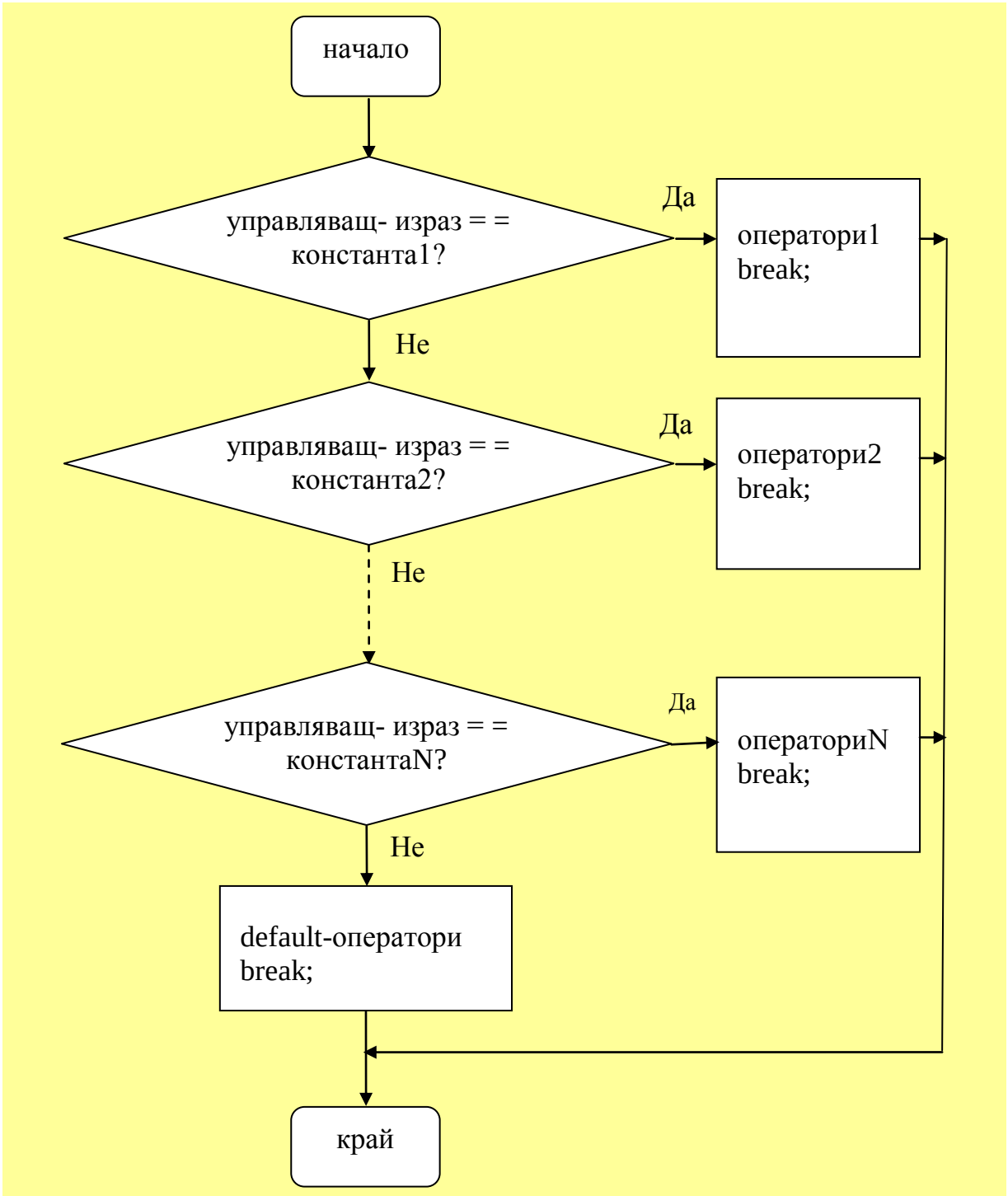
Ако стойността на целочисления **управляващ-израз** не съвпада със стойността на някоя от целочислените константи **константа1,...константаN**, програмата преминава към изпълнение на операторите, които се намират след default. При достигане на оператора break програмата излиза от тялото на оператора switch и преминава към изпълнение на първия оператор след него.

Операторът default не е задължителен. Ако стойността на целочисления

управляващ-израз не съвпада със стойността на някоя от целочислените константи **константа1**,...**константаN** и секцията default липсва, операторът switch се прескача и програмата преминава към изпълнение на първия оператор след switch.

	Управляващият израз на оператора switch може да бъде само целочислен израз.

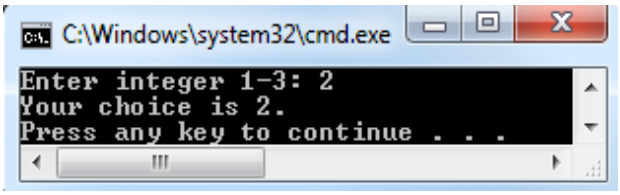
Фиг.56 представя оператора switch в графичен вид.

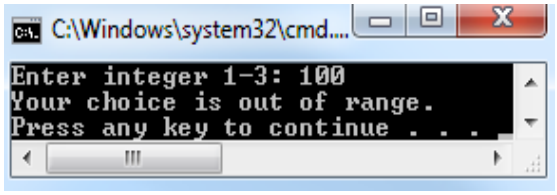


Фиг. 56 Оператор switch

Пример 15-5:

```
#include <stdio.h>

int main(void)
{
	int choice;

	printf("Enter integer 1-3: ");
	scanf("%d", &choice);

	switch(choice)
	{
	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 out of range.\n");
		break;
	}

	return 0;
}
```



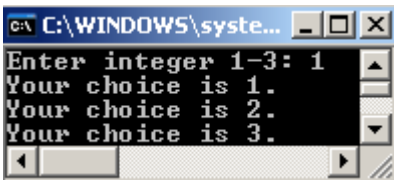


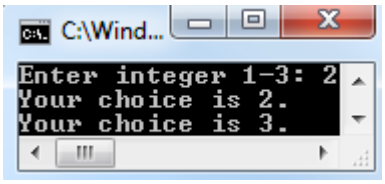
Ако в някой от case-етикетите е пропуснато да се добави оператор break след поредицата оператори, принадлежащи на етикета, програмата продължава с изпълнението на операторите на следващия етикет. Това продължава, докато се срещне оператор break или се стигне до затварящата фигурна скоба на оператора switch.

Пример 15-6:

```
#include <stdio.h>

int main(void)
{
	int choice;

	printf("Enter integer 1-3: ");
	scanf("%d", &choice);

	switch(choice)
```

```
	{
	case 1:
		printf("Your choice is 1.\n");
		/* <--- липсва break */
	case 2:
		printf("Your choice is 2.\n");
		/* <--- липсва break */
	case 3:
		printf("Your choice is 3.\n");
		break;
	default:
		printf("Your choice is out of range.\n");
		break;
	}

	return 0;
}
```

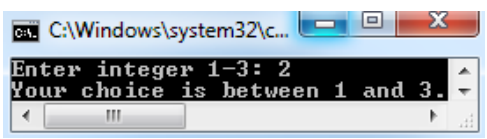




Няколко case-етикета могат да се добавят един след друг. Това означава, че един и същ код може да се изпълни при различни стойности на управляващия израз.

Пример 15-7:

```
#include <stdio.h>

int main(void)
{
	int choice;

	printf("Enter integer 1-3: ");
	scanf("%d", &choice);

	switch(choice)
	{
	case 1:
	case 2:
	case 3:
		printf("Your choice is between 1 and 3.\n");
		break;
	default:
		printf("Your choice is out of range.\n");
		break;
	}

	return 0;
}
```



15.3 Оператори за цикъл (итеративни оператори)

15.3.1 Оператор while

Операторът while служи за повторение на изпълнението на един или повече оператори. Той има следния синтаксис:

```
while (управляващ-израз)
{
     оператори
}
```

където

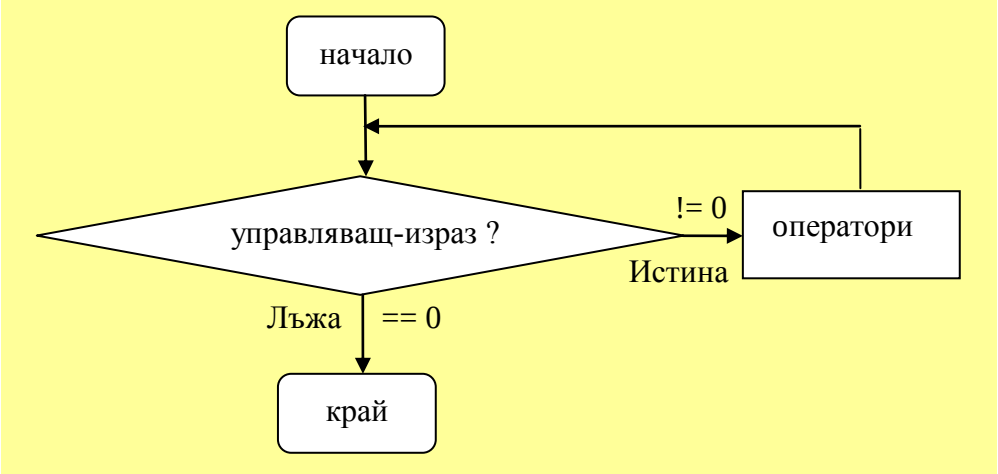
управляващ-израз
Може да бъде всеки израз, който връща стойност различна от void.

оператори
Може да бъде всеки C оператор, включително и друг while оператор.

Ако стойността на **управляващ-израз** е различна от нула, програмата преминава към изпълнение на операторите в тялото на while. След това отново се проверява стойността на **управляващ-израз**. Ако стойността е различна от нула, операторите в тялото на while се изпълняват отново. Процесът се повтаря, докато стойността на **управляващ-израз** е различна от нула.

Ако стойността на **управляващ-израз** е нула, операторът while се прескача и програмата преминава към изпълнение на първия оператор след него.

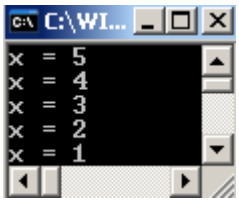
Фиг.57 представя оператора while в графичен вид.



Фиг. 57 Оператор while

Пример 15-8:

```
#include <stdio.h>

int main(void)
{
	char x = 5;

	while(x != 0)
	{
		printf("x = %d\n", x);
		x--;
	}

	return 0;
}
```



Управляващият израз на оператора while в показания пример е x != 0. При първото изчисление този израз дава като резултат 1 (тъй като 5 != 0 е вярно), което води до изпълнение на операторите в тялото на while, след което отново се преминава към изчисление на управляващия израз. При всяко изпълнение на операторите в тялото на while, променливата x се намалява с 1. Когато стойността на x стане 0, изразът x != 0 дава като резултат 0 (тъй като 0 != 0 не е вярно) и програмата преминава към изпълнение на оператора след while, в нашия случай това е операторът return 0;.

Ако стойността на **управляващ-израз** винаги се изчислява различна от нула, цикълът е безкраен. Закоментирайте израза x--; в горния пример и вижте какъв ще бъде резултатът.

Когато тялото на while се състои от единичен оператор, фигурните скоби могат да се пропуснат.

	**Препоръчително е винаги да използвате фигурни скоби дори тялото на while да се състои от единичен оператор.**

15.3.2 Оператор do-while

Операторът do-while служи за повторение на изпълнението на един или

повече оператори. Той има следния синтаксис:

```
do
{
    оператори
} while (управляващ-израз);
```

където

управляващ-израз
Може да бъде всеки израз, който връща стойност различна от void.

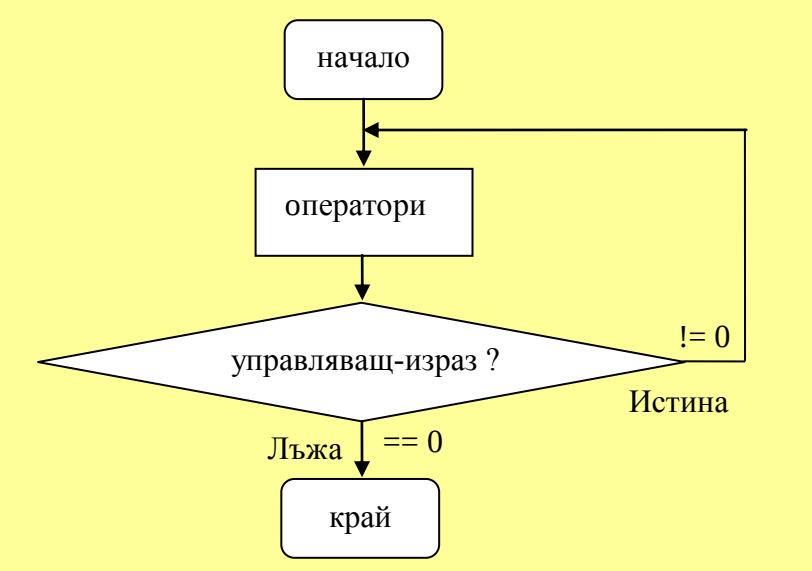
оператори
Може да бъде всеки C оператор, включително и друг do-while оператор.

При достигане до оператора do-while, програмата първо изпълнява операторите в тялото на цикъла, след което се изчислява стойността на **управляващ-израз.**

Ако стойността на **управляващ-израз** е различна от нула, програмата отново преминава към изпълнение на операторите в тялото на do-while. Процесът се повтаря, докато стойността на **управляващ-израз** е различна от нула.

Ако стойността на **управляващ-израз** е нула, операторът do-while се прескача и програмата преминава към изпълнение на първия оператор след do-while.

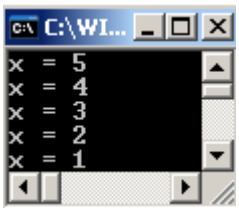
Фиг.58 представя оператора do-while в графичен вид.



Фиг. 58 Оператор do-while

Пример 15-9:

```
#include <stdio.h>

int main(void)
{
	int x = 5;

	do
	{
		printf("x = %d\n", x);
		x--;
	}while(x != 0);

	return 0;
}
```



Единствената разлика между while и do-while е, че операторите в тялото на do-while се изпълняват задължително поне един път.

Ако стойността на **управляващ-израз** винаги се изчислява различна от нула, цикълът е безкраен. Закоментирайте израза x--; в горния пример и вижте какъв ще бъде резултатът.

Когато тялото на do-while се състои от единичен оператор, фигурните скоби могат да се пропуснат.

Препоръчително е винаги да използвате фигурни скоби дори тялото на do-while да се състои от единичен оператор.

15.3.3 Оператор for

Операторът for служи за повторение на изпълнението на един или повече оператори. От всички оператори за цикли той е най-гъвкав и има следния синтаксис:

```
for(израз1; управляващ-израз ; израз3)
{
	операции
}
```

където

управляващ-израз
Може да бъде всеки израз, който връща стойност, различна от void.

израз1/израз3
Може да бъде произволен израз.

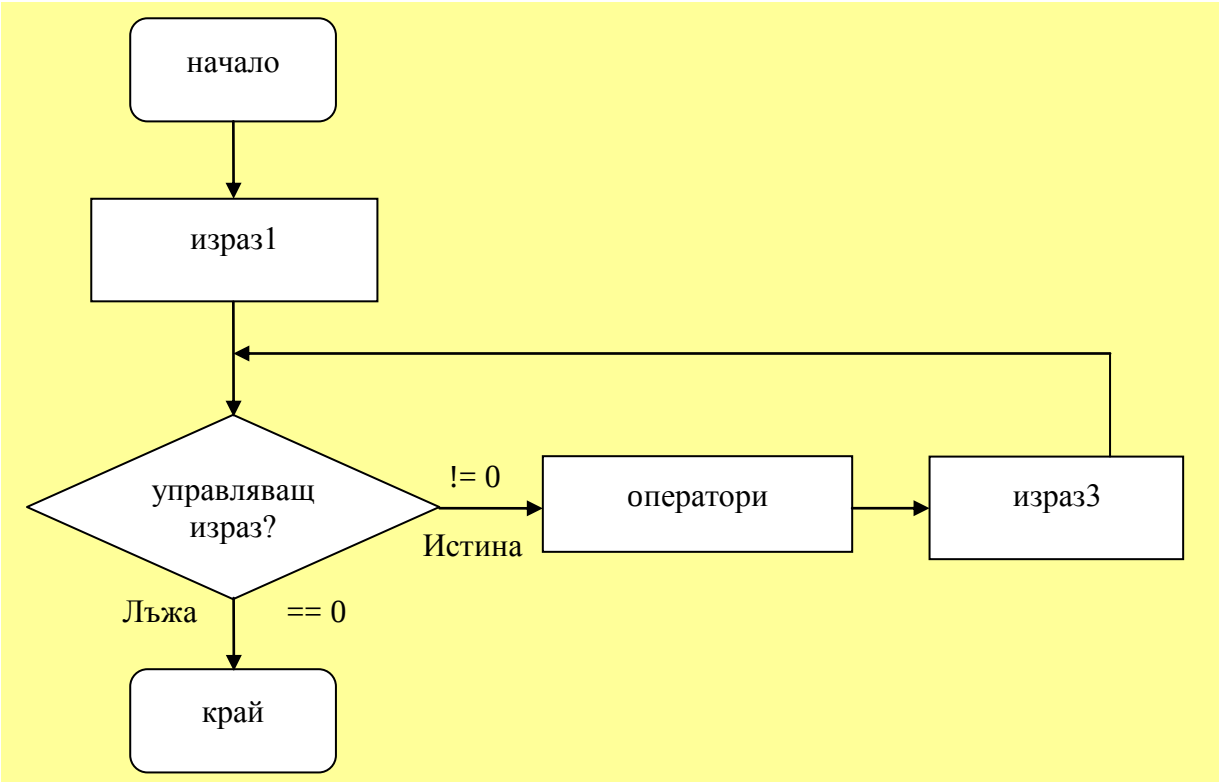
оператори
Може да бъде всеки C оператор, включително и друг for оператор.

При достигане до оператора for програмата първо изчислява **израз1**, след което преминава към изчисляване **управляващ-израз.**

Ако стойността на **управляващ-израз** е различна от нула, програмата преминава към изпълнение на операторите в тялото for. След като изпълни операторите, програмата изчислява стойността на израз3, след което отново изчислява **управляващ-израз**. Ако стойността е различна от нула, операторите в тялото на for се изпълняват отново. Процесът се повтаря, докато стойността на **управляващ-израз** е различна от нула.

Ако стойността на **управляващ-израз** е нула, операторът for се прескача и програмата преминава към изпълнение на първия оператор след for.

Фиг.59 представя оператора for в графичен вид.

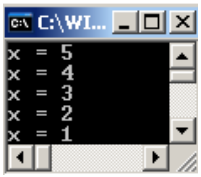


Фиг. 59 Оператор for

Типично израз1 се използва за инициализиране на променливата, която управлява цикъла, а израз3 променя тази променлива по някакъв начин.

Пример 15-10:

```
#include <stdio.h>

int main(void)
{
	int x;

	for(x = 5; x != 0 ; x--)
	{
		printf("x = %d\n", x);
	}

	return 0;
}
```



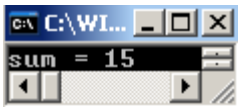
Позициите, в които се разполагат израз1, управляващ-израз и израз3 могат да се оставят празни, но точките и запетаите задължително трябва да присъстват. Цикълът for има следната особеност: ако **управляващ-израз** е празен това се интерпретира от компилатора като ИСТИНА и цикълът става безкраен. Закоментирайте или изтрийте израза x != 0 в горния пример и вижте какъв ще бъде резултатът.

	Ако управляващ-израз на for е празен, това се интерпретира от компилатора като ИСТИНА и цикълът for става безкраен.

Също така израз1 и израз3 могат да се състоят от няколко израза разделени със запетая.

Пример 15-11:

```
#include <stdio.h>

int main(void)
{
	int x, sum;

	for(sum = 0, x = 5 ; x != 0 ; sum += x , x--)
	{
		/* празно тяло */
	}

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

	return 0;
}
```



Пример 15-11 сумира числата от 0 до 5. Забележете, че променливата sum, в която се натрупва резултатът от сумирането, се нулира в израз1 и се увеличава в израз3. Подобни действия обаче не подобряват с нищо кода, затова избягвайте да го правите. Следващата програма например е много по-нагледна от предходната и позволява добавянето на коментари.

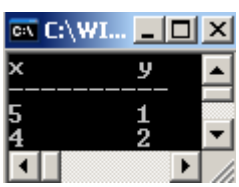
Пример 15-12:

```
#include <stdio.h>

int main(void)
{
	int x, sum;

	sum = 0; /* Нулиране на променливата, в която ще се съхранява резултата */

	for(x = 5 ; x != 0 ; x--)
	{
		sum += x; /* Добави следващото число */
	}

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

	return 0;
}
```

Въпреки това обаче, ако цикълът съдържа две управляващи променливи, е удачно те да се инициализират и променят в израз1 и израз3 съответно.

Пример 15-13:

```
#include <stdio.h>

int main(void)
{
	int x, y;

	printf("x\ty\n");
	printf("----------\n");

	for(y = 1, x = 5 ; x != y ; x--, y++)
	{
		printf("%d\t%d\n", x, y);
	}

	 return 0;
}
```



Използвайте израз1 и израз3 на цикъла for само за да инициализирате и променяте управляващи променливи на цикъла, в противен случай рискувате да направите кода си по-объркан и неясен.

Когато тялото на for се състои от единичен оператор, фигурните скоби могат да се пропуснат.

Препоръчително е винаги да използвате фигурни скоби дори тялото на for да се състои от единичен оператор.

15.4 Оператор break

Операторът break може да се появява само в тялото на оператор switch или в тялото на операторите за цикли while, do-while и for. Използването на break извън този контекст ще доведе до грешка при компилиране. Резултатът от изпълнението на оператора break е прекратяване на изпълнението на съответния оператор към когото принадлежи и преход към следващия оператор, намиращ се след него, т.е. .

```
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
```

Операторът break може да се намира дори в блок с код (съставен оператор), който се намира в тялото на оператор switch или цикъл, и пак ще предизвика излизане от съответния оператор.

Пример 15-14:

```
#include <stdio.h>

int main(void)
{
        int x = 0;

        while(1) /* безкраен цикъл, управляващият израз винаги е различен от 0 */
        {
                printf ("x = %d\n", x);
                x++;

                if(6 == x) {
                        break;
                }
        }/* Затваряща фигурна скоба на while */

        return 0;
}
```

Операторът break е разположен в тялото на if и предизвиква излизане от цикъла while, в който е разположен оператора if.

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
```

Операторът while се явява вложен по отношение на оператора swtich

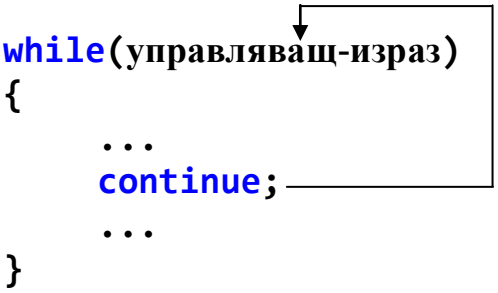
```
for(израз1 ; управляващ-израз ; израз3)
{
      ...
      for(израз1 ; управляващ-израз ; израз3)
      {
            ...
            break;
            ...
      }
      следващ-оператор
      ...
      break;
      ...
}
следващ-оператор
```

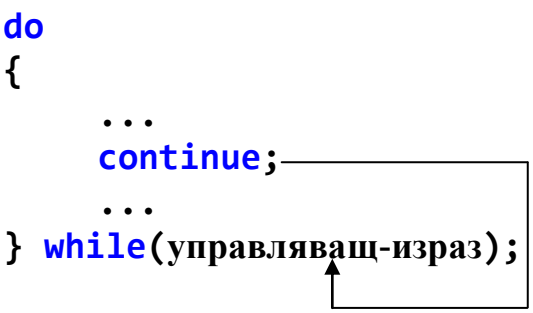
оператор for, вложен в друг оператор for

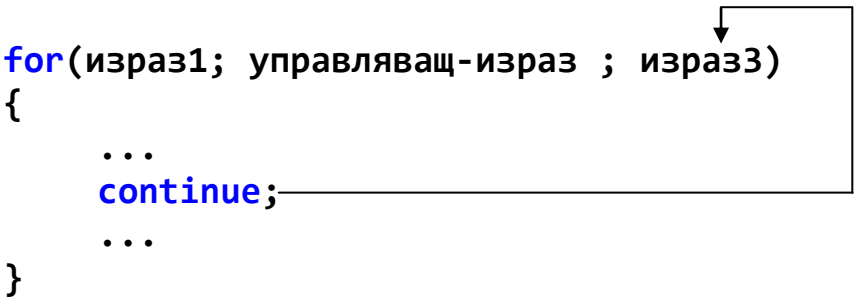
15.5 Оператор continue

Операторът continue може да се появява само в тялото на операторите за цикли while, do-while и for. Използването на continue извън този контекст

ще доведе до грешка при компилиране. 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 изчисление на израз3, т.е. .







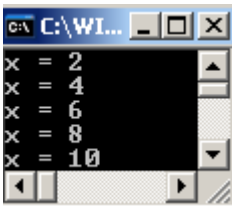
Операторът continue може да се намира дори в блок с код (съставен оператор), който се намира в тялото на оператор за цикъл и пак ще предизвика нова итерация на цикъла.

Следващият пример отпечатва на екрана само четните числа от 1 до 10 с помощта на оператора continue.

Пример 15-15:

```
#include <stdio.h>

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

```
        for (x = 1; x <= 10 ; x++)
        {
                if (0 != (x % 2))
                {
                        continue;       /* Прескача всички оператори след него,
                                           принадлежащи на цикъла, в който се извиква */
                 }

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

        return 0;
}
```



Операторът continue е разположен в тялото на if и предизвиква нова итерация на цикъла for, в който е разположен оператора if.

Обърнете внимание, че ако операторът, 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. .

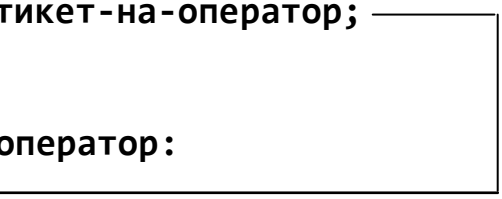
```
for(израз1; управляващ-израз ; израз3)
{
      ...
      while(управляващ-израз)
      {
            ...
            continue;
            ...
      }
      ...
      continue;
      ...
}
```

оператор while, вложен в оператор for

15.6 Оператор goto

Операторът goto е оператор за безусловен преход към друг оператор, указан с етикет. Областта на действие на goto е само тялото на функцията, в която операторът се използва. Преход между функции с помощта на goto е невъзможен.

```
тип-резултат име-функция(списък-параметри)
{
    ...
    goto име-етикет-на-оператор;
    ...
    ...
име-етикет-на-оператор:
    оператор;
    ...
    return израз;
}
```



където

име-етикет-на-оператор:
Следва същите правила като име на променлива.

На всеки оператор може да се назначи етикет. Етикетът завършва с двоеточие. Операторът може да се намира непосредствено след двоеточието, но за по-добра нагледност обикновено се поставя на следващия ред, т.е.

```
име-етикет:оператор;
```

или

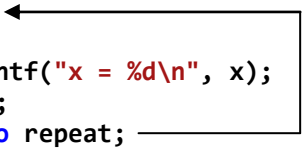
```
име-етикет:
        оператор;
```

Етикираният оператор може да се намира преди или след goto инструкцията, която прави преход към него.

Следващият пример симулира цикъл с помощта на goto.

Пример 15-16:

```
#include <stdio.h>

int main(void)
{
    int x = 0;

repeat:
    if (x < 6)
    {
        printf("x = %d\n", x);
        x++;
        goto repeat;
    }
```



```
            return 0;
}
```

	**Операторът goto може да направи програмата доста заплетена. Повечето специалисти препоръчват този оператор да не се използва изобщо.**

Въпроси за самопроверка

1. Обяснете как работи операторът if-else?

2. Обяснете как работи операторът switch?

3. Обяснете как работи операторът за цикъл while?

4. Обяснете как работи операторът за цикъл do-while?

5. Обяснете как работи операторът за цикъл for?

6. Коригирайте следната програма така, че да отпечата само числата от 0 до 5, без да променяте управляващия израз на цикъла.

```
#include <stdio.h>

int main(void)
{
	int x;

	for (x = 0; x < 100 ; x++)
	{
		printf("x = %d\n", x);
	}

	return 0;
}
```

16 Масиви

16.1 Въведение

Масивът е съвкупност от променливи от един тип и с общо име. Променливите, които принадлежат на масива, се наричат още елементи. Масивите могат да бъдат едномерни и многомерни. В практиката най-често се използват едномерни и двумерни масиви, по-рядко и тримерни.

16.2 Едномерни масиви

Дефинирането на едномерен масив изглежда така:

тип име-масив[размер];

където

размер
: Определя броя на елементите на масива. Може да бъде само целочислен константен израз.

тип
: Определя типа на елементите на масива.

Квадратните скоби след името указват на компилатора, че това е масив. Вътре в тях се указва броя на елементите на масива, а **тип** определя типа на тези елементи и може да бъде един от следните:

- всеки един от базовите типове
- указателен тип (указателните типове са разгледани в 17 Указатели)
- структурен тип (структурните типове са разгледани в 18 Структури)
- union-тип (union-типовете са разгледани в 19 Обединения)
- enum-тип (enum-типовете са разгледани в 21 Изброявания)

Достъпът до елементите на масива става чрез неговото име, последвано от номера (индекса) на съответния елемент в квадратни скоби, т.е. .

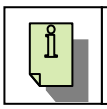
име-масив[индекс]

където

индекс

Може да бъде всеки целочислен израз.

В С елементите на масива се номерират от 0 нагоре, т.е. . първият елемент има индекс 0, вторият индекс 1 и т.н. Когато компилаторът срещне дефиниция на масив, той разполага последователно неговите елементи в паметта на компютъра. Елементите на масива могат да се използват навсякъде където обикновени променливи от същия тип могат да се използват, т.е. . представете си, че цялото обозначение **име-масив[индекс]** е името на дадена променлива.



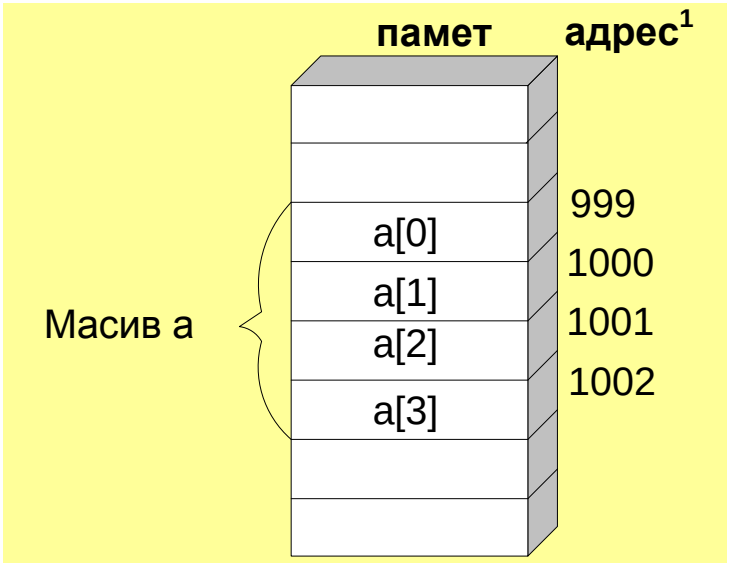
Първият елемент на масива има индекс 0, вторият 1 и т.н.

Пример:

```
/* Дефиниране на масив от 4 елемента тип char */
char a[4];
```

Елементите на този масив са a[0], a[1], a[2] и a[3].

Схематично масивът a може да се представи в паметта на компютъра по следния начин:



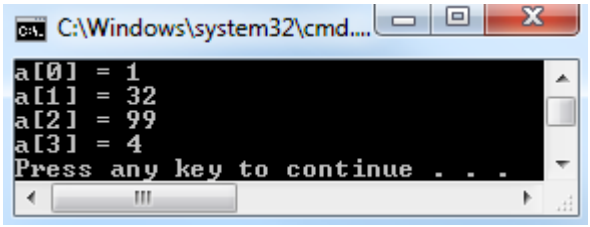
Фиг. 60 представяне на масива a в паметта

Забележка[1]: Адресите, на които се разполагат елементите на масива, са произволно избрани в примера и имат само илюстративен характер.

Обърнете внимание, че последният елемент има индекс с единица по-малък от размера на масива.

Пример 16-1:

```
#include <stdio.h>

int main(void)
{
        /* Дефиниране на масив от 4 елемента тип char */
        char   a[4];

        /* Присвояване на стойности на елементите на масива */
        a[0] = 1;
        a[1] = 32;
        a[2] = 99;
        a[3] = 4;

        /* Отпечатване на елементите на масива на екрана */
        printf("a[0] = %d\n", a[0]);
        printf("a[1] = %d\n", a[1]);
        printf("a[2] = %d\n", a[2]);
        printf("a[3] = %d\n", a[3]);

        /* printf("a[4] = %d\n", a[4]); */ /* !!!Внимание, индексът излиза извън
                                              масива*/

        return 0;
}
```



	**Езикът C не следи дали индексът е в границите на масива. Ако индексът излезе изън границите на масива, поведението е недефинирано.**

	**Внимавайте при индексиране на масива да не излезете извън границите на масива. Винаги проверявайте индексът дали е валиден.**

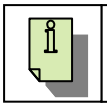
Масивите могат да се инициализират по време на дефинирането си. Общата форма на инициализация на едномерен масив изглежда така:

тип име-масив[размер] = {инициализатор1, … , инициализаторN};

Инициализаторите могат да бъдат само константи. Броят на инициализаторите не трябва да превишава броя на елементите на масива, зададен с **размер**. Ако броят на инициализаторите е по-малък, елементите, за които недостигат инициализатори, се инициализират с нула.

Пример 16-2:

```
#include <stdio.h>

int main(void)
{
        /* Дефиниране и инициализиране на масив от 4 елемента тип char */
        char   a[4] = {1,32,99,4};

        /* Отпечатване на елементите на масива на екрана */
        printf("a[0] = %d\n", a[0]);
        printf("a[1] = %d\n", a[1]);
        printf("a[2] = %d\n", a[2]);
        printf("a[3] = %d\n", a[3]);

        return 0;
}
```

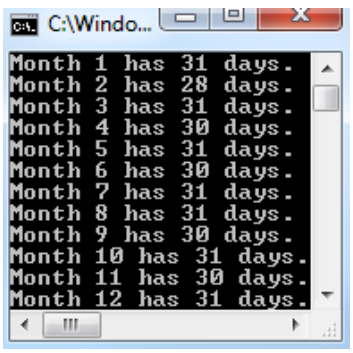


Ако броят на инициализаторите на масив е по-малък от размера, елементите, за които недостигат инициализатори, се инициализират с 0.

Следващият пример въвежда нова конструкция, която намира широка употреба при задаване размер на масив. Това е директивата **#define** (директивите са разгледани подробно в глава 23 Предпроцесор).

Пример 16-3:

```
#include <stdio.h>

#define MONTHS 12

int main(void)
{
      int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
      int index;

      for (index = 0; index < MONTHS ; index++)
      {
            printf("Month %d has %d days.\n", index + 1 , days[index]);
      }

      return 0;
}
```



С директивата #define може да дефинирате макроси. Макросът е текстов фрагмент, на който е дадено някакво име. Всяко използване на това име някъде в кода се заменя с текстовия фрагмент. Текстовият фрагмент може да бъде произволен текст стига след замяната да се получава валидна С конструкция. Директивата #define има следния синтаксис:

#define име-макрос текстов-фрагмент

В нашия пример имаме макрос MONTHS, а текстовият фрагмент се състои просто от числото 12. Когато компилаторът срещне макрос MONTHS някъде в кода, той го заменя с числото 12. Забележете, че ако по-късно решите да промените размера на масива по някаква причина, всичко което е необходимо да направите е да го промените в дефиницията на макроса.

Когато масив се инициализира по време на дефинирането си, размерът на масива може да се пропусне. В този случай компилаторът определя броя на елементите на масива от броя на инициализаторите.

Пример 16-4:

```
#include <stdio.h>

int main(void)
{
        /* Дефиниране и инициализиране на масив от 4 елемента тип char */
        char   a[] = {1,32,99,4};

        /* Отпечатване на елементите на масива на екрана */

        printf("a[0] = %d\n", a[0]);
        printf("a[1] = %d\n", a[1]);
        printf("a[2] = %d\n", a[2]);
        printf("a[3] = %d\n", a[3]);

        return 0;
}
```

Спомнете си, че в глава 10 Константи се запознахте с низовите константи. По същество низовата константа е едномерен масив от символи. Низовите константи могат да се използват за инициализиране на символни масиви.

char име-масив[размер] = "низова константа";

или

char име-масив[ ] = "низова константа";

Обърнете внимание, че размерът на символния масив трябва да е с единица по-голям от броя на символите в низа заради нулевия символ.

Пример 16-5:

```
#include <stdio.h>

int main(void)
{
        /* Дефиниране и инициализиране на символен масив */
        char   str[10] = "C program";

        printf("%s\n", str); /* Отпечатване на низа на екрана */

        return 0;
}
```

Символният масив може да се инициализира с низ и символ по символ.

Пример 16-6:

```
#include <stdio.h>

int main(void)
{
        /* Дефиниране и инициализиране на символен масив */
        char   str[10] = {'C',' ','p','r','o','g','r','a','m','\0'};

        printf("%s\n", str); /* Отпечатване на низа на екрана */

        return 0;
}
```

Ако инициализирате символен масив с низ символ по символ, не забравяйте да добавите и нулевия символ накрая. В противен случай масивът се инициализира просто със символи , а не с низ.

Следващият пример отпечатва символите на низ в обратен ред.

Пример 16-7:

```
#include <stdio.h>

int main(void)
{
      char StringBuffer[ ] = "ANSI C. Beginner\'s course";
      int  IndexCounter;

      /* Намиране на индекса на нулевия символ */
      for( IndexCounter = 0; StringBuffer[IndexCounter] != '\0' ; IndexCounter++ )
      {
            /* тяло без оператори */
      }
```

```
	IndexCounter--; /* Намиране на индекса на последния символ на низа */

	/* Отпечатване на низа в обратен ред*/
	while (IndexCounter >= 0)
	{
		putchar(StringBuffer[IndexCounter]);
		IndexCounter--;
	}

	putchar('\n'); /* Отпечатване на нов ред */

	return 0;
}
```

Нека да разгледаме някои части от примера по-отблизо. Ще започнем с

```
for( IndexCounter = 0; StringBuffer[IndexCounter] != '\0' ; IndexCounter++ )
{
	/* тяло без оператори */
}
```

Единствената роля на този цикъл е да намери индекса на нулевия символ '\0' и да го съхрани в променливата IndexCounter. За целта тази променлива се нулира и се използва за индексиране на масива StringBuffer. Поредният елемент на масива се проверява дали е нулев символ. Ако не е, променливата IndexCounter се увеличава с 1. Всички тези действия се извършват в кръглите скоби след for и в тялото на цикъла не е нужно да извършваме никакви действия. Цикълът for може да се напише и по следния начин:

```
for( IndexCounter = 0; StringBuffer[IndexCounter] != '\0' ; /* празен израз */)
{
	IndexCounter++;
}
```

Езикът C дефинира т.нар. празен оператор, който не върши нищо. Той се състои само от точка и запетая. Когато тялото на даден цикъл е празно, то може да се замени с празен оператор, т.е.

```
for( IndexCounter = 0; StringBuffer[IndexCounter] != '\0' ; IndexCounter++ )

	; /* празен оператор */
```

Празният оператор може да се постави и веднага след затварящата кръгла скоба.

```
for( IndexCounter = 0; StringBuffer[IndexCounter] != '\0' ; IndexCounter++ );
```

Следващата конструкция, която ще разгледаме, е

```
while (IndexCounter >= 0)
{
	putchar(StringBuffer[IndexCounter]);
	IndexCounter--;
}
```

Напредналите С програмисти биха я написали по следния начин:

```
while (IndexCounter >= 0)
{
	putchar(StringBuffer[IndexCounter--]);
}
```

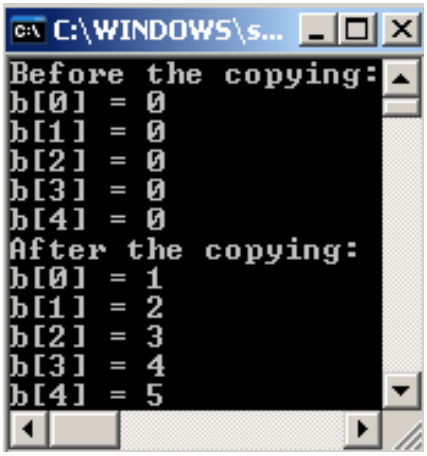
Спомнете си начина на работа на постфиксната операцията --. Първо ще се извлече текущата стойност на операнда IndexCounter и тя ще се използва за индексиране на масива StringBuffer, след което IndexCounter ще се намали с 1.

Масивите не могат да се копират директно, т.е. . ако имате масиви a и b с еднакъв брой елементи, конструкцията b = a е невалидна. Ако искате да копирате един масив в друг, ще трябва да го направите елемент по елемент.

Пример 16-8:

```
#include <stdio.h>

void copy_array(void);
void print_array_b(void);

#define SIZE  5

int a[SIZE] = {1,2,3,4,5};
int b[SIZE];

int main(void)
{
	printf("Before the copying:\n");
	print_array_b();
	copy_array();
	printf("After the copying:\n");
	print_array_b();

	return 0;
}

/* Копира масива a в масива b */
void copy_array(void)
{
	int i;

	for(i = 0 ; i < SIZE ; i++)
	{
		b[i] = a[i];
	}
}

/* Отпечатва масива b */
```

```
void print_array_b(void)
{
	int i;

	for(i = 0 ; i < SIZE ; i++)
	{
		printf("b[%d] = %d\n",i , b[i]);
	}
}
```



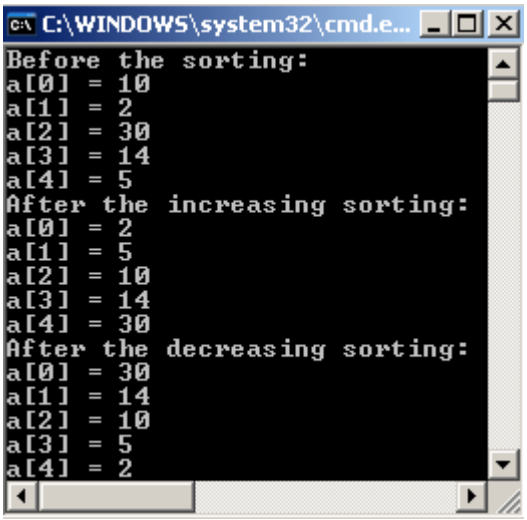
Следващата програма демонстрира прост начин за сортиране на масиви от числа. Може да сортирате числата във възходящ или низходящ ред.

Пример 16-9:

```
#include <stdio.h>

void sort_array(int dir);
void print_array(void);

#define SIZE       5
#define INCREASING 1
#define DECREASING 0

int a[SIZE] = {10,2,30,14,5};

int main(void)
{
	printf("Before the sorting:\n");
	print_array();
	sort_array(INCREASING); /* Възходящо сортиране */
	printf("After the increasing sorting:\n");
	print_array();
	sort_array(DECREASING); /* Низходящо сортиране */
	printf("After the decreasing sorting:\n");
	print_array();

	return 0;
}

/* Сортира масива a */
void sort_array(int dir)
{
	int i, j, temp;
```

```
        for(i = 0 ; i < SIZE ; i++)
        {
                for(j = i+1 ; j < SIZE ; j++)
                {
                        if(INCREASING == dir) /* Сортирай масива по възходящ ред */
                        {
                                if(a[i] > a[j])
                                {
                                        temp = a[i];
                                        a[i] = a[j];
                                        a[j] = temp;
                                }
                        }
                        else                    /* Сортирай масива по низходящ ред */
                        {
                                if(a[i] < a[j])
                                {
                                        temp = a[i];
                                        a[i] = a[j];
                                        a[j] = temp;
                                }
                        }
                }/* край на вложения for*/
        }/* край на for*/

}

/* Отпечатва масива a */
void print_array(void)
{
        int i;

        for(i = 0 ; i < SIZE ; i++)
        {
                printf("a[%d] = %d\n",i , a[i]);
        }
}
```

При първото „извъртане“ на вложения цикъл, в първата позиция на масива се появява най-малкият елемент (при възходящо сортиране) или най-големият елемент (при низходящо сортиране). При всяко следващо „извъртане“ на вложения цикъл в следващата позиция на масива се появява поредният по големина елемент.



16.3 Двумерни масиви

Двумерният масив е масив, чиито елементи са едномерни масиви.

Дефинирането на двумерен масив изглежда така:

тип име-масив[размер1][размер2];

където

размер1

Определя броя на елементите на двумерния масив, т.е. . броя на едномерните масиви. Може да бъде само целочислен константен израз.

размер2

Определя броя на елементите в едномерните масиви. Може да бъде само целочислен константен израз.

тип

Определя типа на елементите на едномерните масиви.

Елементите на двумерните масиви се достигат чрез индексиране.

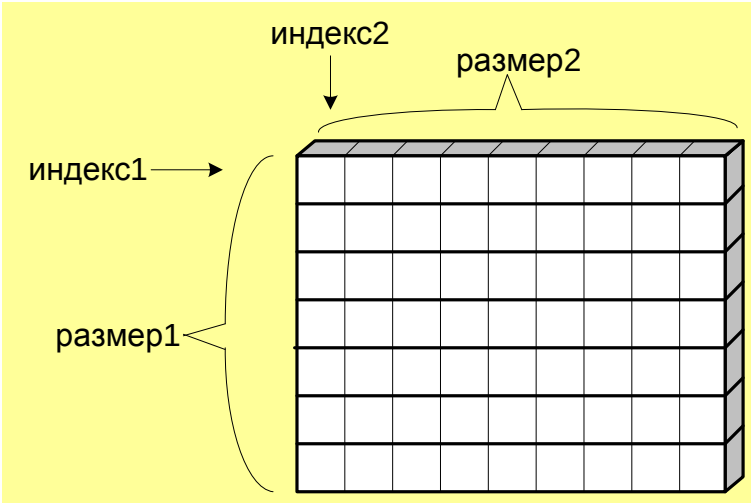
име-масив[индекс1] [индекс2]

където

индекс1/индекс2

Може да бъде всеки целочислен израз.

Двумерният масив може да се представи графично като таблица, в която **размер1** определя броя на редовете, а **размер2** определя броя на колоните (Фиг.61).



Фиг. 61 Двумерен масив

С **индекс1** се избира едномерния масив (т.е. реда), а с **индекс2** се избира конкретния елемент от този масив (т.е. колоната).

Двумерните масиви също могат да се инициализират при дефинирането си. За целта трябва да се осигури инициализатор за всеки елемент едномерен масив, т.е. .

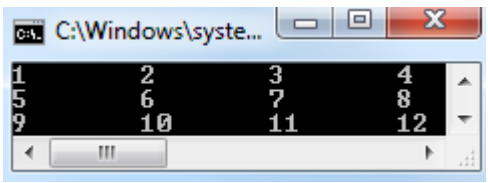
тип име-масив[размер1][размер2] = {
{инициализатор-масив1},
{инициализатор-масив2},
…
{инициализатор-масивN}
};

където

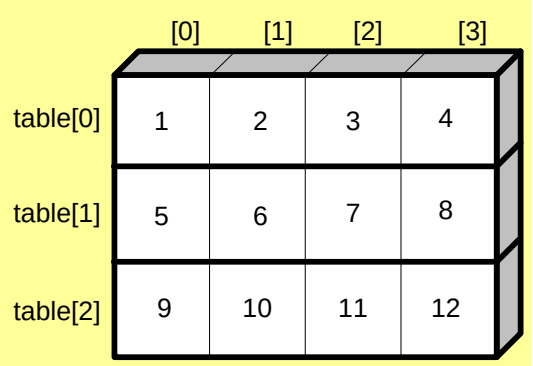
инициализатор-масив1, ... ,инициализатор-масивN
Представляват константи разделени със запетая.

Пример 16-10:

```
#include <stdio.h>

int main(void)
{
	int table[3][4] = {
			  {1,2,3,4},   /* инициализира масива table[0] */
			  {5,6,7,8},   /* инициализира масива table[1] */
			  {9,10,11,12} /* инициализира масива table[2] */
			 };
	int row, column;

	for(row = 0 ; row < 3; row++) /* обхожда масива по редове */
	{
		for(column = 0; column < 4; column++) /* обхожда масива по колони */
		{
			printf("%d\t", table[row][column]);

			if(3 == column)
			{
				printf("\n");
			}
		}
	}

	return 0;
}
```



Фиг.62 показва как изглежда масивът table[3][4].



Фиг. 62 table[3][4]

Когато двумерен масив се инициализира по време на дефинирането си, размер1 може да се пропусне. В този случай компилаторът определя броя на елементите на масива от броя на инициализаторите.

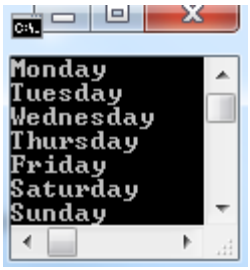
Пример:

```
int table[][4] = {
                   {1,2,3,4},   /* инициализира масива table[0] */
                   {5,6,7,8},   /* инициализира масива table[1] */
                   {9,10,11,12} /* инициализира масива table[2] */
                 };
```

Следващият пример използва двумерен масив за създаване на низова таблица, която съдържа дните на седмицата.

Пример 16-11:

```
#include <stdio.h>

#define WEEK_DAYS     7
#define LENGTH       10

int main(void)
{
	char StringTable[WEEK_DAYS][LENGTH] =
	{
		"Monday",
		"Tuesday",
		"Wednesday",
		"Thursday",
		"Friday",
		"Saturday",
		"Sunday"
	};
	int index;

	/* Отпечатване на всеки низ от таблицата */
	for(index = 0 ; index < WEEK_DAYS ; index++)
	{
		printf("%s\n", StringTable[index]);
	}

	return 0;
}
```



Фиг.63 показва как изглежда низовата таблица StringTable в паметта.

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
StringTable[0]	'M'	'o'	'n'	'd'	'a'	'y'	'\0'			
StringTable[1]	'T'	'u'	'e'	's'	'd'	'a'	'y'	'\0'		
StringTable[2]	'W'	'e'	'd'	'n'	'e'	's'	'd'	'a'	'y'	'\0'
StringTable[3]	'T'	'h'	'u'	'r'	's'	'd'	'a'	'y'	'\0'	
StringTable[4]	'F'	'r'	'i'	'd'	'a'	'y'	'\0'			
StringTable[5]	'S'	'a'	't'	'u'	'r'	'd'	'a'	'y'	'\0'	
StringTable[6]	'S'	'u'	'n'	'd'	'a'	'y'	'\0'			

Фиг. 63 Низова таблица StringTable

Забележете, че не всички елементи, заделени за таблицата, се използват. Освен това, ако трябва да сортирате низовете в някакъв ред (например лексикален), ще трябва да размените местата им в таблицата. В глава 17 Указатели ще Ви покажа по-ефективен начин за създаване на низови таблици.

16.4 Многомерни масиви

Един N-мерен масив е масив, чиито елементи са (N-1)-мерни масиви. В практиката рядко се използват масиви с размерност над 2. Ако приложите всички разсъждения за едномерни и двумерни масиви, описани по-рано, сами може да разберете как работи един многомерен масив.

N-мерен масив е масив, чиито елементи са (N-1)-мерни масиви.

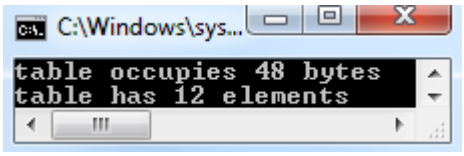
16.5 Операция sizeof и масиви

Ако операнд на операцията sizeof е име на масив (едномерен или многомерен), резултатът е броя на байтовете заемани в паметта от масива. Броят на елементите на масива може да бъде изчислен по време на изпълнение на програмата, като броят на байтовете, заемани от целия

масив, се раздели на броя байтове, заемани от един елемент на масива. Броят на байтовете на един елемент зависи от неговия тип.

Пример 16-12:

```
#include <stdio.h>

int main(void)
{
	int table[3][4];

	printf("table occupies %d bytes\n", sizeof(table));

	printf("table has %d elements\n", sizeof(table)/sizeof(int));

	return 0;
}
```



Въпроси за самопроверка

1. Покажете как се дефинира едномерен масив?

2. Дефинирайте масив с име digits, който има 10 елемента от тип int.

3. Номерата (индексите) на елементите на масива започват от 1. Вярно или невярно?

4. Как се достигат елементите на масив?

5. Ако е дадена дефиницията int a[5] = {1};, какви ще бъдат стойностите на елементите a[0], ... , a[4]?

6. Какво е двумерен масив?

7. Създайте двумерен символен масив и го инициализирайте с имената на месеците. Отпечатайте масива на екрана.

17 Указатели

17.1 Въведение

Указателят е променлива, която съдържа адреса на друга променлива, т.е. . указателят е променлива, която сочи към друга променлива. Стойностите, които един указател може да съдържа, се явяват адреси. Дефинирането на указател изглежда така:

тип * име-указател;
тип* име-указател;
тип *име-указател;

Символът * пред името указва на компилатора, че това, което се дефинира, е указател, **тип** указва типа на променливата, към която указателят може да сочи и може да бъде един от следните:

- всеки един от базовите типове, включително и void
- друг указател
- структурен тип (структурните типове са разгледани в 18 Структури)
- union-тип (union-типовете са разгледани в 19 Обединения)
- enum-тип (enum-типовете са разгледани в 21 Изброявания)

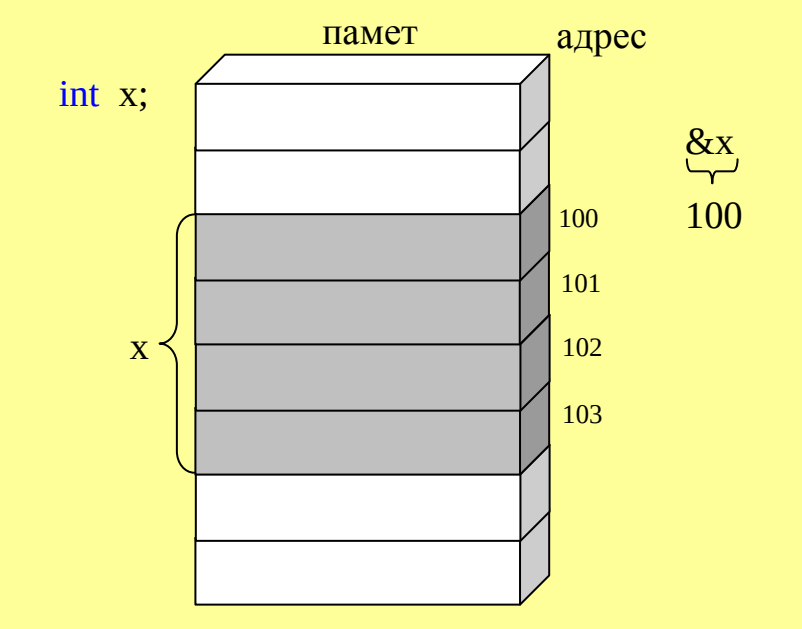
След като се създаде един указател, той сочи към произволно място в паметта[1].

Забележка[1]: В действителност само локалните указатели имат неопределена стойност. Глобалните указатели (статични и външни) и локалните статични указатели се инициализират автоматично с NULL. Стойността NULL е разгледана по-надолу в тази глава.

За да му се присвои адрес на променлива, се използва операцията за извличане на адрес на променлива **&**:

&име-променлива

Тази конструкция връща като резултат адреса на променливата след символа &. Фиг.64 описва нагледно какво се случва, когато се извлича адрес на многобайтова променлива (в примера се приема, че тип int заема 4 байта).



Фиг. 64 Извличане на адрес на променлива

Когато се извлича адресът на многобайтова променлива, всъщност се извлича адресът на най-младшия байт на паметта, заемана от променливата.

Имайки информация за типа на обекта, т.е. . колко байта заема той в паметта, и адреса на най-младшия байт за многобайтови променливи, указателят може да осъществи безпроблемно достъп до сочения от него обект.

	Операцията за извличане на адрес & не може да се прилага върху register-променливи.

Достъпът до променливата, сочена от указател, става чрез операцията за косвен достъп ***** (нарича се още дереференциране на указател)**:**

***име-указател**

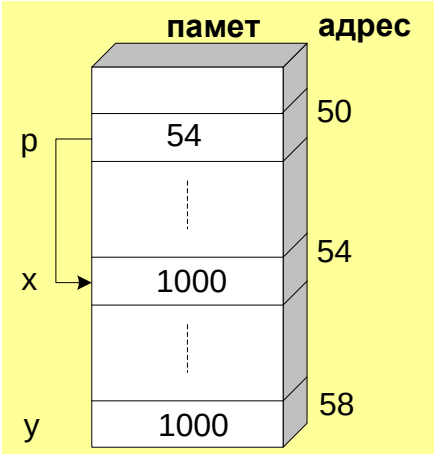
Тази конструкция е еквивалентна на директното използване на името на променливата, сочена от указателя , т.е. .

***име-указател ≈ име-променлива**

Пример:

```
int x, y;       /* Дефиниране на променливи x и y от тип int            */
int * p;        /* Дефиниране на указател към променлива от тип int     */

p   = &x;       /* Адресът на x  се присвоява на указателя p             */
*p = 1000;      /* Същото като x = 1000;                                 */

y = *p;         /* Същото като y = x;                                    */
```

Схематично това изглежда така:



Фиг. 65 Представяне на указателя p в паметта

Компилаторът заделя място някъде в паметта на компютъра за двете променливи x и y, и за указателя p (адресите в паметта са произволно избрани в примера). След изпълнение на конструкцията p = &x; в указателя се записва адресът на променливата x (в случая адресът е 54), т.е. . p вече сочи към x както е показано със стрелката. След това достъпът до променливата x се осъществява посредством указател (косвен достъп), а променливата y се достъпва директно (директен достъп).

Пример 17-1:

```
#include <stdio.h>

int main(void)
{
        /* Дефиниране на променлива от тип char */
        char   ch;
        /* Дефиниране на указател от тип char*  */
        char * p;

        /* Присвояване на адреса на ch на p     */
        p = &ch;

        /* Присвояване на символ на ch косвено  */
        *p = 'C';

        /* Отпечатване съобщение на екрана      */
        printf("I like %c\n", *p);

        return 0;
}
```

	Винаги проверявайте указател дали сочи към валиден адрес преди да го използвате.

Внимавайте, когато дефинирате няколко указателя на един ред. Например дефиницията

```
int* p1, p2, p3;
```

може да Ви остави с впечатлението, че се дефинират три указателя. Това обаче не е вярно, тази дефиниция е еквивалентна на следното:

```
int* p1; /* указател към обект тип int */
int  p2; /* обект тип int    */
int  p3; /* обект тип int    */
```

Звездичката не принадлежи към името на типа нито към името на променливата, а просто указва на компилатора, че променливата след нея е указател. Затова, ако искате да дефинирате няколко указателя наведнъж, е необходимо пред всяко име да добавите звездичка, т.е. .

```
int *p1, *p2, *p3;
```

17.2 Указател към указател

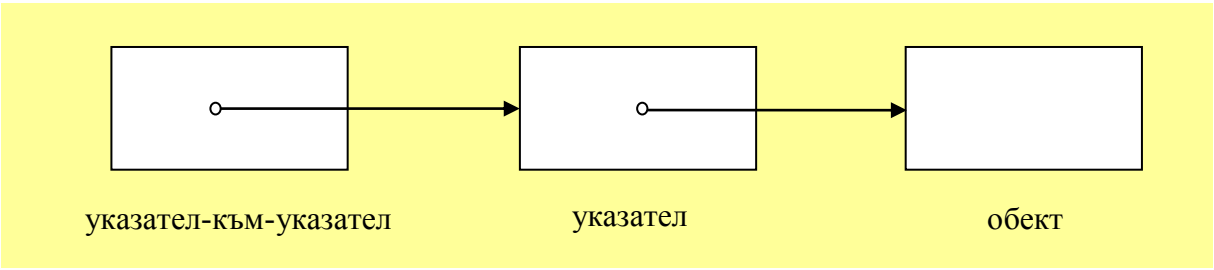
Указателят също е променлива (чийто тип е тип*), която се съхранява някъде в паметта на компютъра на определен адрес. Този адрес може да се присвои на друг указател. Дефинирането на указател-към-указател изглежда така:

тип **име-указател-към-указател;

където

тип

Тип на обекта, към който сочи указателят, сочен от указателя-към-указател (Фиг. 66).



Фиг. 66 Указател-към-указател

Горната дефиниция може да се разгледа по следния начин:

тип* *име-променлива;

където

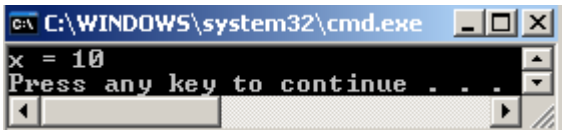
име-променлива се явява указател към обект, чийто тип е тип*, т.е. . име-променлива е указател към друг указател.

Достъпът до обект чрез указател-към-указател изисква двойно дереференциране, т.е.

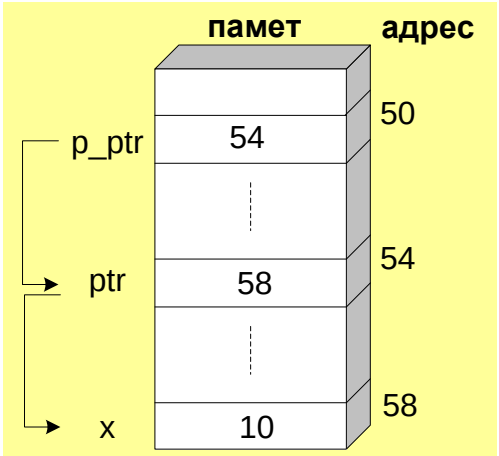
****указател-към-указател**

Пример 17-2:

```
#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.67 показва нагледно връзката между, p_ptr, ptr и x.



Фиг. 67 Указател-към-указател

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

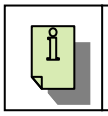
присвоява на променливата x , т.е. .

(**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.

17.3 Връзка между указатели и масиви

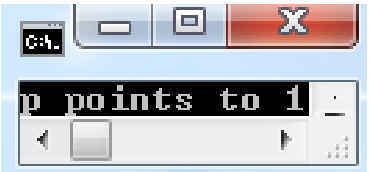
Масивите и указателите в C са тясно свързани. Името на масив, в действителност, представлява адресът на първия елемент на масива, т.е. . името на масив е указател към първия елемент на масива (погледнете Табл.42 от 14.2 Обичайни унарни преобразувания). Ако имаме масив a, то използването на a е еквивалентно на &a[0]



Името на масив е указател към първия елемент на масива.

Пример 17-3:

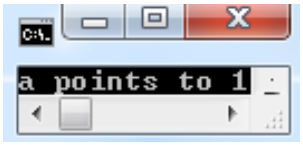
```
#include <stdio.h>

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

	p = a; /* Същото като p = &a[0] */
	printf("p points to %d\n", *p);

	return 0;
}
```



Все пак, има една важна разлика между обикновен указател и име на масив и тя е, че име на масив е константен указател и не може да се променя да сочи към друго освен началото на масива. Компилаторът не заделя памет за името на масива, а просто го асоциира с адреса на първия елемент на масива. След като име на масив е също и указател, то операцията дереференциране на указател също може да се приложи към него. Резултатът е първият елемент на масива.

Пример 17-4:

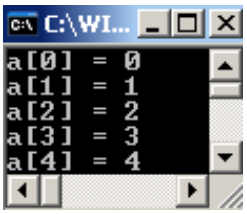
```
#include <stdio.h>

int main(void)
{
	int a[4];

	*a = 1; /* Същото като a[0] = 1 */
	printf("a points to %d\n", *a); /* *a ≈ a[0] */

	/* a = &a[2] */ /*!!!Грешка, а не може да се променя */

	return 0;
}
```



Ако указател сочи към масив, той може да се индексира по същия начин както и името на масива.

Пример 17-5:

```
#include <stdio.h>

int main(void)
{
	int    a[5];  /* Дефиниране на масив от 10 int елемента */
	int     *p;  /* Дефиниране на указател към int         */
	int      i;

	p = a;       /* Адресът на елемента a[0]  се присвоява на p  */

	/* Инициализиране на масива */
	for (i = 0 ; i < 5 ; i++)
	{
		p[i] = i;
	}

	/* Отпечатване на масива */
	for (i = 0 ; i < 5 ; i++)
	{
		printf("a[%d] = %d\n",i, p[i]);
	}

	return 0;
}
```



Всяка конструкция от вида a[i] се преобразува от компилатора в *(a+i), т. е. индексирането на масив се заменя с указателна аритметика. Указателната аритметика е разгледана по-надолу в тази глава.

Спомнете си, че в глава 10 Константи се запознахте с низовите константи и че низовите константи по същество са масиви от символи. Низовите константи могат да се използват да инициализират също и символен указател по следния начин:

char* име-указател;
име-указател = "низ";

Когато срещне подобна конструкция, компилаторът разполага низа в паметта и връща адреса на първия символ, т.е. . на указателя се присвоява адреса на първия символ на низа.

Пример 17-6:

```
#include <stdio.h>

int main(void)
{
	char* p_str;

	p_str = "ANSI C. Beginner's course.";
	printf("%s\n", p_str);

	return 0;
}
```

Като повечето типове, указателите също могат да бъдат елементи на масив. Обобщената конструкция на дефиниране на масив от указатели е:

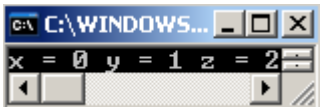
тип * име-масив[размер];

Забележете, че име на масив от указатели се явява указател-към-указател.

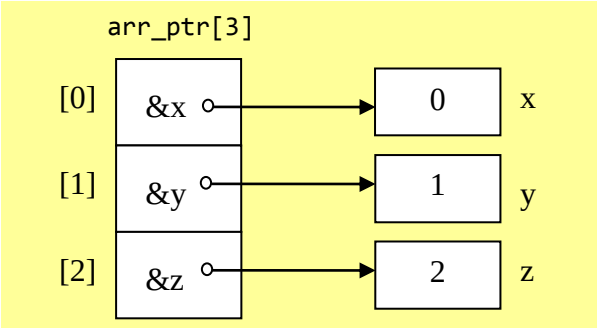
Пример 17-7:

```
#include <stdio.h>

int main(void)
{
	int x, y, z;
	int* arr_ptr[3];
	int i;

	/* Присвояване на адреси на променливи */
```

```
	arr_ptr[0] = &x;
	arr_ptr[1] = &y;
	arr_ptr[2] = &z;

	/* Косвено присвояване на стойности на обектите, сочени от указателите */
	for(i = 0 ; i < 3 ; i++)
	{
		*arr_ptr[i] = i;
	}

	/* Отпечатване на стойностите на обектите */
	printf("x = %d y = %d z = %d\n", x, y, z);

	return 0;
}
```



Фиг.68 илюстрира Пример 17-7.



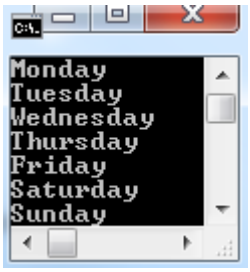
Фиг. 68 Илюстрация към Пример 17-7

В 16 Масиви Ви показах как да създадете таблица от низове. Сега ще Ви покажа по-ефективен начин да направите това с помощта на указатели. За целта просто дефинирайте масив от char-указатели и ги инициализирайте с низови константи.

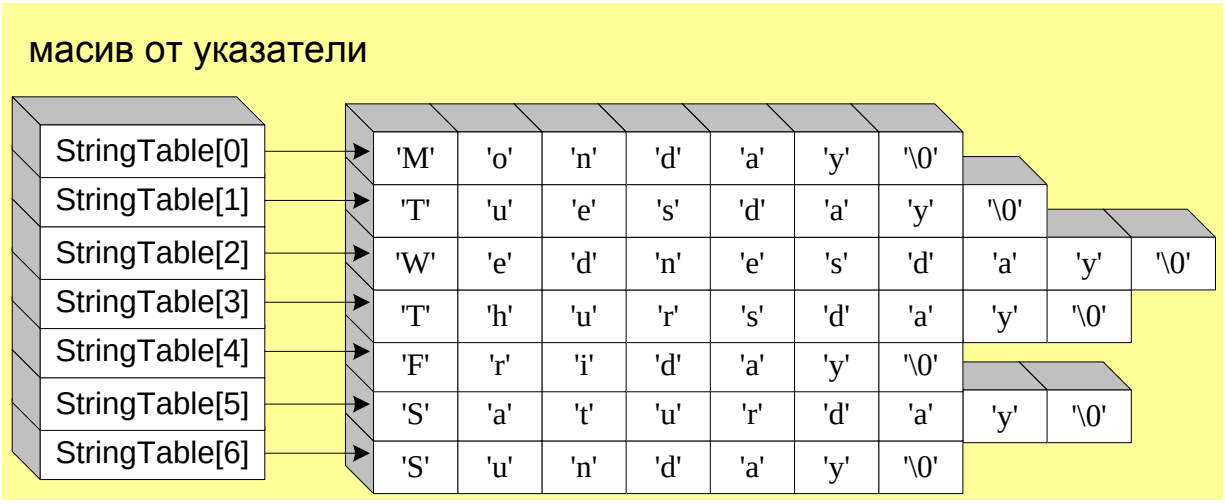
Пример 17-8:

```
#include <stdio.h>

#define WEEK_DAYS	7

int main(void)
{
	char *StringTable[WEEK_DAYS] =
	{
		"Monday",
		"Tuesday",
		"Wednesday",
		"Thursday",
		"Friday",
		"Saturday",
		"Sunday"
	};
```

```
        int index;

        /* Отпечатване на всеки низ от таблицата */
        for(index = 0 ; index < WEEK_DAYS ; index++)
        {
                printf("%s\n", StringTable[index]);
        }

        return 0;
}
```



Фиг.69 показва как изглежда низовата таблица в паметта.



Фиг. 69 Низова таблица

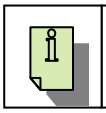
Ако сравните Фиг.63 и Фиг.69, ще забележите, че реализирането на низова таблица под формата на масив от char-указатели използва паметта по-ефективно. Освен това, ако трябва да сортирате низовете в някакъв ред, е необходимо просто да сортирате елементите-указатели на масива, вместо да разменяте местата на целите низове.

17.4 Обобщени указатели

C89 въвежда т.нар. обобщени указатели. Това са указатели към тип void, затова се наричат още void-указатели.

void *име-указател;

Обобщените указатели могат да сочат към променливи от всякакви типове.



Обобщените указатели могат да сочат към променливи от всякакви типове.

Можете да присвоявате адресите на променливите директно на void-указателя. Компилаторът преобразува автоматично адреса на променливата във void-указател.

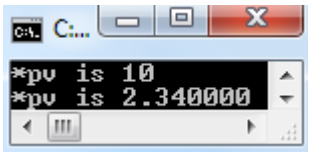
Достъпът до обект чрез void-указател обаче изисква явно преобразуване на void-указателя в указател към типа на обекта. Ако се замислите, ще откриете причината за това. След като един void-указател може да сочи към всякакви типове, то този указател няма никаква информация за обекта, към който сочи, освен за адреса, на който обектът е разположен. За да осъществи достъп до него, указателят трябва да знае колко байта трябва да достъпи. Преобразувайки го в указател към типа на обекта, void-указателят получава тази информация. За целта се използва операцията за преобразуване на типове (вижте отново 14.5 Явно преобразуване на типове):

(тип*)име-void-указател

Тази конструкция принуждава компилатора да преобразува **име-void-указател** в указател към посочения в скобите тип. Казано с прости думи, това преобразуване инструктира компилатора да третира void-указателя все едно е указател към обект от указания тип.

Пример 17-9:

```
#include <stdio.h>

int main(void)
{
   void *pv;
   int i = 10;
   double d = 2.34;

   pv = &i; /* автоматично преобразуване на тип int*  в тип void* */
   printf("*pv is %d\n", *((int*)pv)); /* явно преобразуване на тип void* в тип
                                          int* */
   pv = &d; /* автоматично преобразуване на тип double*  в тип void* */
   printf("*pv is %f\n", *((double*)pv)); /* явно преобразуване на тип void* в тип
                                             double* */

   return 0;
}
```



Конструкцията (int*)pv указва на компилатора, че указателят pv трябва да се третира все едно е указател към обект от тип int, докато конструкцията (double*)pv указва на компилатора, че указателят pv трябва да се третира все едно е указател към обект от тип double.

17.5 Операции с указатели

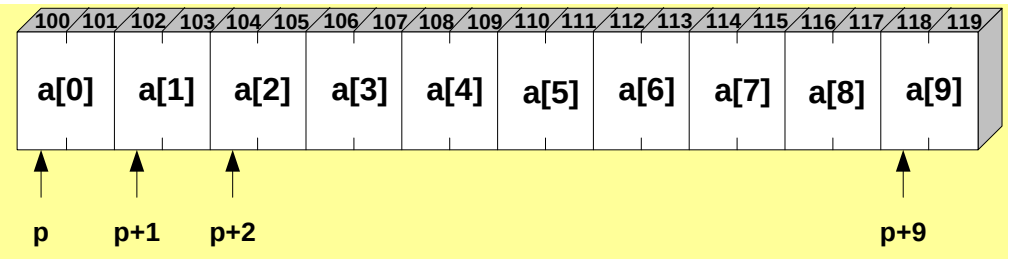
17.5.1 Събиране и изваждане на указател с цяло число

За да разберете по-добре как действа указателната аритметика, нека дефинираме масив от int елементи и указател, който сочи към началото на този масив.

Пример 17-10:

```
#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;  /* Преместване на указателя към следващия елемент на масива */

      /*...*/

      return 0;
}
```

На Фиг.70 е показано разположението на масива **a** в паметта. Приема се, че int променливите са с размер два байта.



Фиг. 70 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]**, а не към следващия байт в паметта. 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 указателят се прехвърля на адрес 102, който е адресът на следващия елемент на масива.

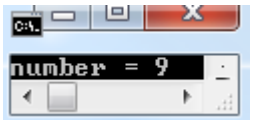
След като се запознахте с указателната аритметика, би трябвало вече да може да си обясните защо израз от вида a[i] е еквивалентен на *(a + i), където a може да бъде както име на масив, така и нормален указател.

17.5.2 Изваждане на указатели от един и същ тип

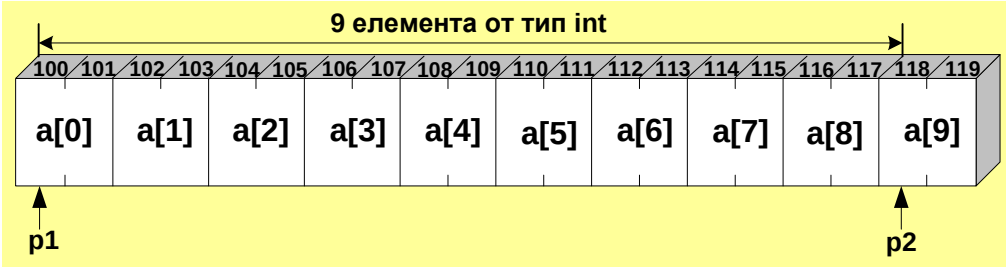
Изваждането на указатели от един и същ тип дава като резултат броя на елементите между двата указателя. Изваждането на указатели има смисъл само ако сочат към общ обект например масив.

Пример 17-11:

```
#include <stdio.h>

int main(void)
{
      int a[10];            /* Дефиниране на масив a с 10 елемента от тип int  */
      int *p1, *p2, number; /* Дефиниране на два int указателя                  */

      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.71 показва нагледно връзката между указателите p1, p2 и масива a.



Фиг. 71 Изваждане на указатели

```
p2 - p1 = (p1 + 9) - p1 = p1 + 9 - p1 = 9
```

17.5.3 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.

Пример 17-12:

```
#include <stdio.h>

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

	int *p1, *p2; /* Дефиниране на указатели p1 и p2 */

	p1 = a;     /* p1 сочи към първия елемент    */
	p2 = &a[4]; /* p2 сочи към последния елемент */

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

	return 0;
}
```

17.5.4 Сравнение на указател с цяло число

Единственото цяло число, с което указател може да се сравнява и което може директно да му се присвоява, е 0. 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:

```
#define NULL  ((void*)0)
```

или

```
#define    NULL   (0)
```

Указатели, които имат стойност NULL се наричат нулеви указатели и се счита, че не сочат към нищо.

Тъй като локалните указатели сочат към произволен адрес, след създаването си е добра практика те да се инициализират с NULL. Това позволява указателят да бъде проверен дали сочи към валиден адрес чрез проверка за NULL стойност преди да бъде използван.

Пример 17-13:

```
#include <stdio.h> /* Включва <stddef.h> */

int main(void)
{

   int *p; /* Локален указател, инициализира се с произволна стойност */

   p = NULL; /* NULL се заменя с (0) или (void*)0 */

   /*...*/

   /* Проверка за валиден указател */
   if (NULL == p) /* NULL се заменя с (0) или (void*)0 */
   {
         printf("Inavlid pointer\n");
   }
   else
   {
         /*...*/
   }

   return 0;
}
```

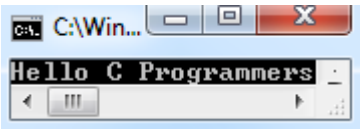
17.5.5 Операциите ++/-- и указатели

Операциите инкрементиране (++) и декрементиране (--) също могат да се прилагат и към указатели. Ефектът от тях е увеличаване/намаляване на указателя с 1, т.е. . ако p е указател, конструкциите p++ и ++p са еквивалентни на p = p + 1. Следващият пример отпечатва низ, сочен от указател, символ по символ.

Пример 17-14:

```
#include <stdio.h>

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

      while(*p != '\0') /* Проверка за достигане на края на низа */
      {
```

```
            putchar(*p); /* Отпечатване на  текущия символ */
            p++; /* Увеличаване на указателя към следващия символ */
      }

      return 0;
}
```



Опитните С програмисти биха написали следните две конструкции:

```
putchar(*p); /* Отпечатване на  текущия символ */
p++; /* Увеличаване на указателя към следващия символ */
```

по следния начин:

```
putchar(*p++); /* Отпечатване на  текущия символ и увеличаване на указателя към
                  следващия символ */
```

Конструкцията *p++ (или *p--) често е объркваща, затова ще я разгледам подробно. От таблицата с приоритети може да видите, че постфиксната форма на операциите ++ и -- е с по-висок приоритет от операцията дереференциране на указател *. Поради тази причина начинаещите програмисти често допускат, че указателят първо се увеличава (или намалява), след което се дереференцира, т.е. . че се извършват следните две операции: 1. **p = p + 1** и 2. ***p**. На практика обаче се извършва точно обратното – указателят първо се дереференцира, след което се увеличава (или намалява): 1. ***p** и 2. **p = p + 1**. Причината за това не винаги е очевидна за един начинаещ С програмист и се дължи на спецификата на работа на постфиксните операции. Спомнете си, че при постфиксната форма първо се извлича стойността на операнда и тази стойност ще се използва, ако операцията е част от друг израз, след което операндът се увеличава (или намалява) с 1. В нашия случай операндът е указателят p. Текущата стойност на p, която се явява адреса на текущия символ, се извлича и операцията дереференциране се прилага върху нея, след което p се увеличава (или намалява). За да се убедите в това, може да използвате скоби по следния начин: *(p++). Ще видите, че резултатът е същият. Изводът е, че операцията ++ (или --) наистина се прилага първа, тъй като е с по-висок приоритет.

За Ваше улеснение Табл.44 показва всички комбинации на операциите * и ++/--, приложени едновременно към указател, с кратко описание.

*p++ ≈ *(p++) *p-- ≈ *(p--)	Първо се извлича стойността на указателя. Операцията * се прилага върху тази стойност и след това указателят се увеличава/намалява.
(*p)++	Първо се извлича обектът, сочен от указателя, след което стойността на обекта се увеличава/намалява.

(*p)--	
***++p ≈ *(++p)** ***--p ≈ *(--p)**	Първо указателят се увеличава/намалява, след това се извлича обектът, сочен от указателя.
++*p ≈ ++(*p) **--*p ≈ --(*p)**	Първо се извлича обектът, сочен от указателя, след което стойността на обекта се увеличава/намалява.

Табл. 44 Операциите ++/-- и указатели

17.6 Квалификаторът const и указатели

Квалификаторът const може да се прилага както към обект, сочен от указател, така и към самия указател или едновременно към обекта и указателя. Обобщената форма на дефиниране на указател и квалификатора const изглежда така:

квалификатор1$_{опц}$ тип * квалификатор2$_{опц}$ име-указател;

По-долу са показани всички възможни комбинации.

- **const тип * име-указател;**

Когато const се приложи на мястото на квалификатор1, той влияе само на обекта. Това означава, че указателят не може да променя обекта. Самият указател може да се променя обаче. Такава дефиниция може да се интерпретира като **указател към константна-променлива.**

Пример 17-15:

```
#include <stdio.h>

int main(void)
{
  int x, y;
  const int *px;

  x = 10;
  px = &x;
  printf("x = %d\n", *px);

  /* *px = 20; */ /* !!!Грешка, обектът не може да се променя от указателя */

  px = &y; /* Позволено, указателят може да сочи към други обекти */

  return 0;
}
```

- **тип * const име-указател = &име-променлива;**

Когато const се приложи на мястото на квалификатор2, той влияе само на

указателя. Това означава, че указателят може да променя обекта. Самият указател не може да се променя обаче. Такава дефиниция може да се интерпретира като **константен-указател към променлива**. Обърнете внимание, че в този случай указателят трябва да се инициализира с адрес на променлива по време на дефинирането си.

Пример 17-16:

```
#include <stdio.h>

int main(void)
{
  int x, y;
  int * const px = &x;

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

  *px = 20; /* Позволено, обектът може да се променя от указателя */
  printf("x = %d\n", x);

  /* px = &y; */ /* !!!Грешка, указателят не може да сочи към други обекти */

  return 0;
}
```

- **const тип * const име-указател = &име-променлива;**

Когато const се приложи на мястото на квалификатор1 и квалификатор2, той влияе и на обекта, и на указателя. Това означава, че указателят не може да променя обекта и самият указател не може да се променя също. Такава дефиниция може да се интерпретира като **константен-указател към константна-променлива**. Обърнете внимание, че в този случай указателят трябва да се инициализира с адрес на променлива по време на дефинирането си.

Пример 17-17:

```
#include <stdio.h>

int main(void)
{
  int x, y;
  const int * const px = &x;

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

  /* *px = 20; */ /* !!!Грешка, обектът не може да се променя от указателя */

  /* px = &y; */ /* !!!Грешка, указателят не може да сочи към други обекти */

  return 0;
}
```

17.7 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. Общата форма на дефиниране на указател към функция е:

тип (*име-указател)(списък-параметри);

където

тип
Дефинира типа на резултата, връщан от функцията.

списък-параметри
Дефинира списъка с параметри на функцията.

За да присвоите адреса на функция на указател към нея, използвайте една от следните две конструкции:

име-указател = име-функция;

или

име-указател = &име-функция; [1]

Забележка[1]: Името на функция автоматично се преобразува в указател към функцията затова операцията за извличане на адрес & може да се изпусне (погледнете Табл.42 от 14.2 Обичайни унарни преобразувания).

За да извикате функция чрез указател към нея, използвайте една от следните две конструкции

(*име-указател)(списък-аргументи)

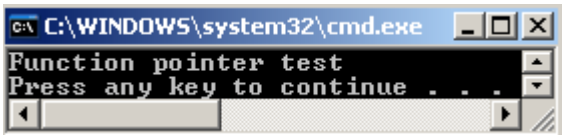
или

име-указател(списък-аргументи)

Двете конструкции за извикване на функция чрез указател са напълно еквивалентни.

Пример 17-18:

```
#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 foo() */
	(*p)();			/* Извикване на функцията чрез указателя p	*/

	return 0;
}

void foo(void)
{
	printf("Function pointer test\n");
}
```

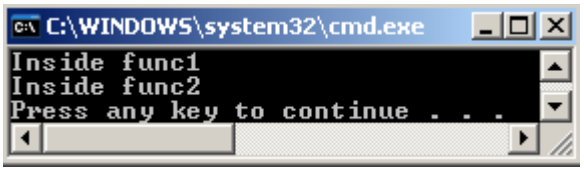


Указателите към функции позволяват предаването на функции като аргументи на други функции. За целта съответният параметър трябва да се декларира като указател към функция по показния по-горе синтаксис.

Пример 17-19:

```
#include <stdio.h>

void test_foo(void (*pf)(void));
void func1(void);
void func2(void);

int main(void)
{
	test_foo(func1);

	test_foo(func2);

	return 0;
}

void test_foo(void (*pf)(void))
{
	pf(); /* Извикване на функцията, сочена от pf */
}

void func1(void)
{
	printf("Inside func1\n");
}

void func2(void)
{
```

```
	printf("Inside func2\n");
}
```



В горния пример функцията test_foo() има параметър pf, който е указател към функция, която не приема аргументи и не връща стойност, т.е. . **void (*pf)(void)**.

Може да използвате typedef и да опростите декларацията на указател към функция.

Пример 17-20:

```
#include <stdio.h>

typedef void (*PointerToFunction)(void);

void test_foo(PointerToFunction pf);
void func1(void);
void func2(void);

int main(void)
{
	test_foo(func1);
	test_foo(func2);

	return 0;
}

void test_foo(PointerToFunction pf)
{
	pf(); /* Извикване на функцията, сочена от pf */
}

void func1(void)
{
	printf("Inside func1\n");
}

void func2(void)
{
	printf("Inside func2\n");
}
```

Конструкцията **typedef void (*PointerToFunction)(void)** декларира тип PointerToFunction, който се интерпретира като указател към функция, която не приема аргументи и не връща стойност. Този тип може да се използва за деклариране на параметър по обичайния начин, както е показано в горния пример.

17.8 Преобразувания свързани с указатели

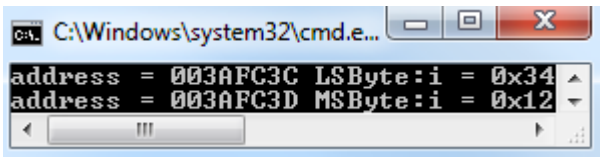
Указател към един обект може да се преобразува в указател към друг обект. Това може да се каже и по друг начин: адресът на обект от един тип може да се преобразува в указател към обект от друг тип.

Void-указател може директно да се присвоява на указател към обект и обратно. Явно преобразуване в случая не се изисква. За достъп до обекта обаче void-указателят трябва явно да се преобразува в указател към типа на обекта (вижте отново Пример 17-9).

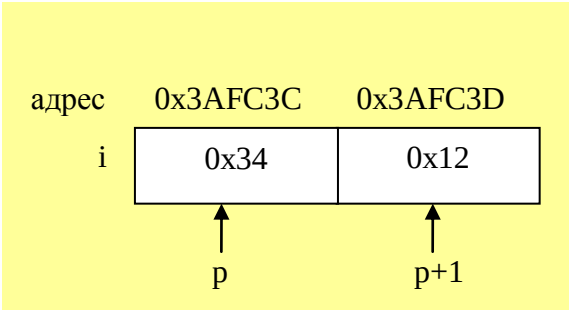
Преобразуването на указател от един тип в указател към друг тип винаги изисква явно преобразуване.

Пример 17-21:

```
#include <stdio.h>

int main(void)
{
   unsigned short i  = 0x1234;
   char* p;

   p = (char*)&i; /* Преобразуване на тип short* в тип char* */
   printf("address = %p LSByte:i = %#x\n", p, *p);
   printf("address = %p MSByte:i = %#x\n", p+1, *(p+1));

   return 0;
}
```



Фиг.72 илюстрира Пример 17-21. На char-указателя p се присвоява адреса на unsigned short променливата i. За целта адресът на променливата се преобразува в указател към char изрично.



Фиг. 72 Илюстриране на Пример 17-21

Първият printf() отпечатва адреса, към който сочи указателят и съхранената

на него стойност, а вторият printf() отпечатва следващия адрес и стойността, която се съхранява там (при Вас точните стойности на адресите може да се различават от тези, показани в примера). Обърнете внимание, че форматният спецификатор за отпечатване на стойностите на указатели (които се явяват адреси) е **%p**. От показаните резултати може да се направи изводът, че компютърът, на който се изпълнява кодът, е Little-endian, т.е. . младшият байт на променливата се разполага на младшия адрес в паметта, заделена за нея.

Може да преобразувате и цели числа в указатели.

Не всички преобразувания, свързани с указатели, обаче са безопасни. Указателите позволяват да се наруши правилото за подравнен достъп до променливите. Преобразуването на указател към променлива тип S с модул на подравняване m_S в указател към променлива тип D с модул на подравняване m_D е:

- безопасно ако $m_S \geq m_D$

  „Безопасно“ означава, че резултатният указател ще работи както се очаква, ако се използва за извличане или съхраняване на обект от тип D.

Пример 17-22:

```
#include <stdio.h>

int main(void)
{
   unsigned  ui; /* Обект тип S */
   unsigned short * p_ushort; /* Указател към обект тип D */

   p_ushort = (unsigned short *)&ui; /* преобразуване на S* в D* */
   *p_ushort     = 0x5678;
   *(p_ushort+1) = 0x1234;
   printf("ui = %#x\n", ui);

   return 0;
}
```

- опасно ако $m_S < m_D$

  „Опасно“ означава, че използването на резултатния указател за извличане или съхраняване на обект от тип D може да причини грешка, спирайки изпълнението на програмата, ако компютърната архитектура неподдържа неподравнен достъп до паметта.

Въпроси за самопроверка

1. Покажете как се дефинира указател?

2. Дефинирайте указател pd, който сочи към променливи от тип double.

3. Към обекти от кои типове може да сочи един указател?

4. Кой е операторът за извличане на адрес на променлива и как се използва?

5. Как се осъществява косвен достъп до променлива посредством указател?

6. Напишете програма, която дефинира променлива d от тип double и указател pd към променливи от тип double. Задайте косвено стойност на d и отпечатайте стойността на екрана директно.

7. Кои операции могат да се прилагат към указателите?

8. Дефинирайте указател към функция, която приема един аргумент тип int и не връща стойност.

9. Покажете как може да опростите горната дефиниция с помощта на typedef.

10. Обяснете връзката между указатели и масиви.

18 Структури

18.1 Въведение

Структурата е средство, чрез което променливи от различен тип, но свързани логически, могат да се обединят в едно цяло. Променливите, принадлежащи на една структура, се наричат членове. Декларирането на структура става по следния начин:

```
struct  име-структура
{
        тип   име-променлива1;
        тип   име-променлива2;
        . . .
        тип   име-променливаN;
};
```

Членовете на една структура могат да бъдат:

- променливи от базови типове;
- масиви;
- указатели;
- променливи от други структури;
- променливи от union-тип (union-типовете са разгледани в 19 Обединения);
- битови полета (битовите полета са разгледани са в 20 Битови полета);
- променливи от enum-тип (enum-типовете са разгледани в 21 Изброявания).

Декларацията на структура представлява създаване на потребителски тип данни, но не води до заделяне на памет. Това е просто един шаблон, който показва структурата на потребителския тип данни. Вземете за пример тип int, който е вграден в езика С. Компилаторът заделя памет едва когато потребителят дефинира променлива от този тип. Аналогично е и със структурите, компилаторът заделя памет едва когато потребителят дефинира променлива от структурен тип. Ще наричаме такива променливи структурни-променливи за по-кратко.

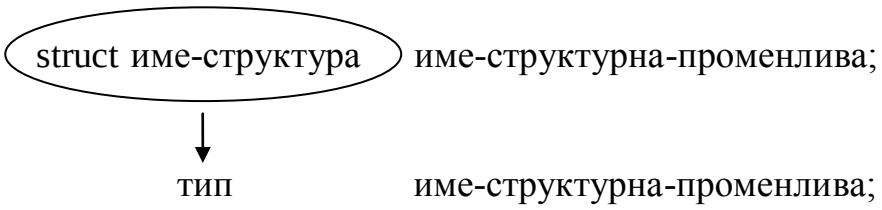
Декларацията на една структура е видима от точката на деклариране до

края на блока или файла, в който се намира декларацията.

Дефинирането на структурна-променлива изглежда така:

struct име-структура име-структурна-променлива;

Представете си тази форма по следния начин:



Цялата конструкция **struct име-структура** представлява името на типа данни. След като срещне такава дефиниция, компилаторът заделя памет за всички членове на структурата. Достъпът до тези членове става по следния начин:

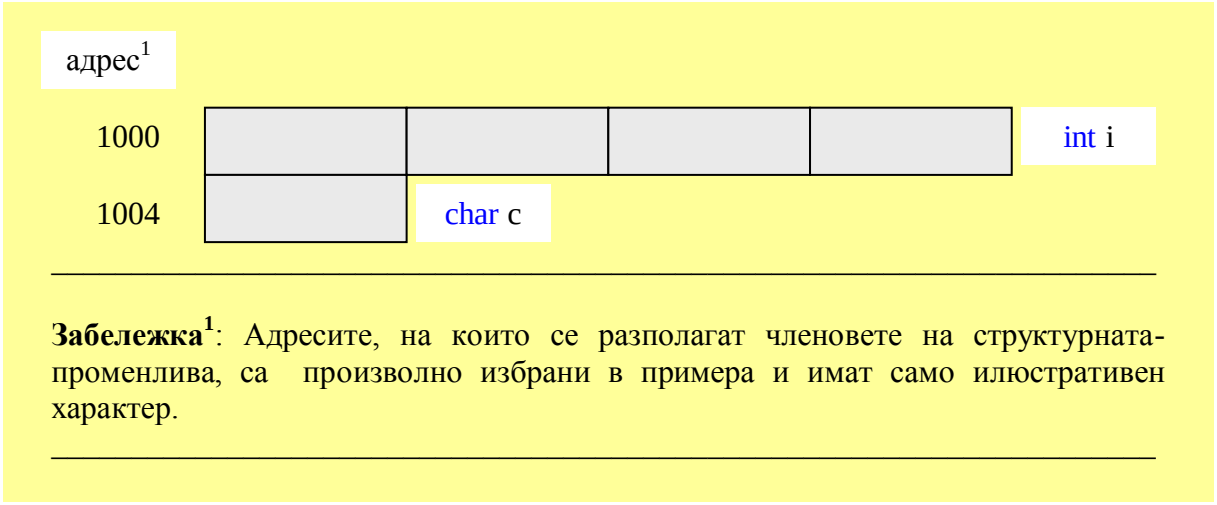
име-структурна-променлива . име-променлива

т.е. . името на структурната-променлива е последвано от точка, последвана от името на член на структурата. Членовете на структурната-променлива могат да се използват навсякъде където обикновени променливи от този тип могат да се използват.

Пример 18-1:

```
#include <stdio.h>

/* Декларация на структура */
struct mystruct
{
      int   i;
      char ch;
};

int main(void)
{
       /* Дефиниране на структурна-променлива от тип mystruct */
       struct mystruct   s;

       /* Присвояване на стойности на членовете на s          */
       s.ch = 'C';
       s.i  = 1;

       /* Отпечатване съобщение на екрана                     */
       printf("%c is number %d\n", s.ch, s.i);

       return 0;
}
```

Ако приемем, че тип int заема 4 байта в паметта, схематично структурната променлива s може да се представи в паметта така:



Забележка[1]: Адресите, на които се разполагат членовете на структурната-променлива, са произволно избрани в примера и имат само илюстративен характер.

Фиг. 73 Структурна-променлива s

Структурните-променливи могат да се дефинират и по време на деклариране на самата структура. В този случай името на структурата може да се пропусне. Обърнете внимание, че ако изпуснете името на структурата няма как по-късно в кода да дефинирате други структурни-променливи от този тип.

Пример 18-2:

```
#include <stdio.h>

int main(void)
{
	/* Декларация на структура */
	struct
	{
		int   i;
		char ch;
	}s; /* Дефиниране на структурна променлива*/

	/* Присвояване на стойности на членовете на s */
	s.ch = 'C';
	s.i  = 1;

	/* Отпечатване съобщение на екрана            */
	printf("%c is number %d\n", s.ch, s.i);

	return 0;
}
```

18.2 Инициализиране на структурни-променливи

Като всяка променлива, структурните-променливи също могат да се инициализират по време на дефинирането си. За целта е необходимо да се осигури инициализатор за всеки член на структурата. Инициализаторите трябва да са константи, разделени със запетая един от друг и затворени във фигурни скоби.

struct име-структура име-структурна-променлива =
{
инициализатор-член1,
инициализатор-член2,
…
инициализатор-членN,
};

Пример 18-3:

```
#include <stdio.h>

/* Декларация на структура */
struct mystruct
{
      int   i;
      char ch;
};

int main(void)
{
       struct mystruct   s = {1, 'C'}; /* Дефиниране и инициализиране на
                                          структурна-променлива */;
       /* Отпечатване съобщение на екрана   */
       printf("%c is number %d\n", s.ch, s.i);

       return 0;
}
```

Ако инициализаторите са по-малко от членовете, членовете, за които недостигат инициализатори, се инициализират с 0 или NULL (за указатели).

18.3 Структури и операцията sizeof

Размерът на една структура (броят на байтовете, заемани от нейните членове в паметта) трябва да се определя с операцията sizeof. Без да навлизам в детайли, само ще Ви припомня, че различните компютри имат различни изисквания как да се разполагат многобайтовите променливи в паметта. Едни компютри изискват многобайтовите променливи да се разполагат на определен адрес, най-често кратен на размера на

променливата. Такива променливи се наричат подравнени в паметта. За такива компютри достъпът до неподравнени променливи би довел до генериране на изключение. Други компютри нямат изискване за подравняване на многобайтовите променливи, но при тях достъпът до неподравнени променливи изисква повече машинни инструкции, отколкото ако са подравнени.

Поради гореописаните изисквания, при разполагане на членовете на структурна-променлива в паметта, могат да се появят „дупки“ (уплътняващи байтове) между два съседни члена и след последния член. Това означава, че действителният размер на структурната-променлива може да е по-голям от сумарния размер на всички членове (вижте отново 9.7 Подравняване на променливите в паметта).

Винаги използвайте операцията sizeof, за да определяте размера на структура.

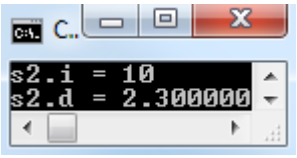
18.4 Присвояване на структурни-променливи

Структурните-променливи могат да се присвояват една на друга. Това е възможно обаче само ако структурните-променливи са от един и същ тип.

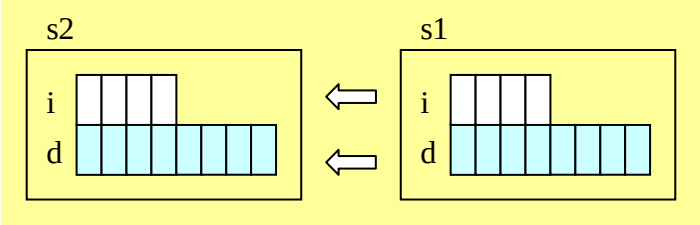
Пример 18-4:

```
#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 различни типове */
```

```
        printf("s2.i = %d\n",  s2.i);
        printf("s2.d = %lf\n", s2.d);

        return 0;
}
```



Фиг.74 показва присвояването на членовете на s1 на s2. Приема се, че тип int е 4 байта, а double – 8 байта.



Фиг. 74 Присвояване на структури

Горният пример показва факта, че дори две структури да са абсолютно идентични, те все пак представляват различни типове.

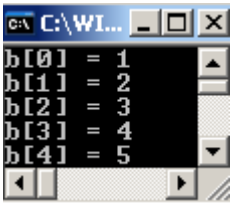
	Дори две структури да са абсолютно идентични, те все пак представляват различни типове.

Присвояването на структурни-променливи позволява да заобиколите ограничението да копирате един масив в друг директно. Просто пакетирайте масива в структура както е показано в следващия пример.

Пример 18-5:

```
#include <stdio.h>

#define SIZE  5

struct Array
{
        int ar[SIZE];
};

int main(void)
{
        struct Array a = { {1,2,3,4,5} };
        struct Array b;
        int i;

        b = a; /* Копиране на структурните-променливи (масивите) */

        /* Отпечатване на елементите на масива b */
        for(i = 0 ; i < SIZE ; i++)
        {
```

```
            printf("b[%d] = %d\n", i, b.ar[i]);
      }

      return 0;
}
```



18.5 Указатели към структури

Членовете на една структура могат да се достигат и чрез указател. Обобщената форма на дефиниране на указател към структура изглежда така:

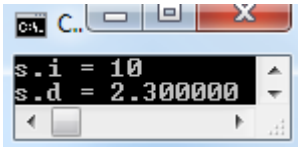
struct име-структура *име-указател;

Достъпът до членовете на структура чрез указател става по следния начин:

име-указател -> име-член

Пример 18-6:

```
#include <stdio.h>

struct mystruct
{
      int i;
      double d;
};

int main(void)
{
      struct mystruct s;   /* Дефиниране на структурна-променлива           */
      struct mystruct *ps; /* Дефиниране на указател към структура mystruct */

      ps = &s; /* Присвояване на адреса на структурната-променлива s */
      ps->i = 10;
      ps->d = 2.3;
      printf("s.i = %d\n", ps->i);
      printf("s.d = %f\n", ps->d);

      return 0;
}
```



18.6 Масиви от структури

Структурите също могат да бъдат елементи на масиви. Общата форма на дефиниране на масив от структури е:

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:

```
struct име-структура име-масив[размер] =
{
        {списък-инициализатори-структура1},
        {списък-инициализатори-структура2},
        …
        {списък-инициализатори-структураN},
};
```

където

списък-инициализатори-структура1, …
Състои се от **инициализатор-член1, … , инициализатор-членN**

Следващият пример демонстрира употребата на масив от структури. Преди да разгледаме примера ще Ви запозная с още една библиотечна функция, с чиято помощ може да четете низове от клавиатурата, включително и интервалите между отделните думи. Това е функцията gets(). Общата форма на извикване на тази функция е:

gets(име-буфер);

където

име-буфер

Име на символен масив, в който се съхранява прочетеният низ.

Функцията gets() чете въведените символи до прочитане на символ за нов ред (който се въвежда при натискане на ENTER) и заменя символа за нов ред с нулев символ.

Пример 18-7:

```
#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");
   }
}
```

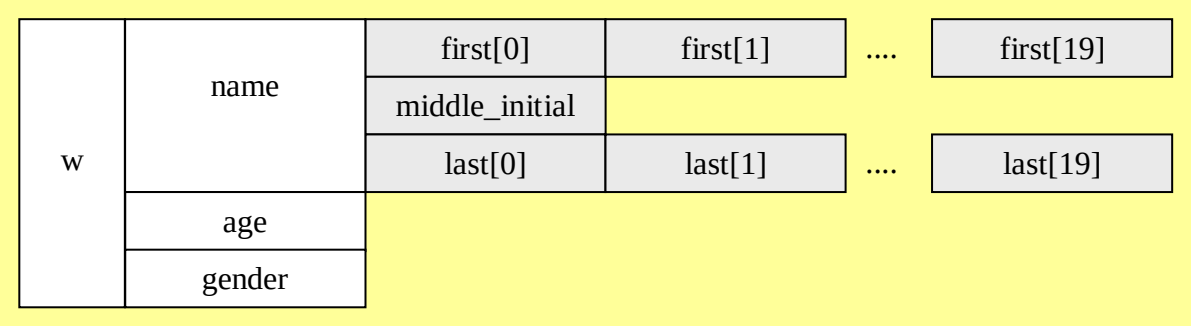
18.7 Вложени структури

Членовете на една структура могат да бъдат променливи от друга структура.

Пример 18-8:

```
#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.75 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      */
};
```

Въпроси за самопроверка

1. Обяснете какво представлява структурата?

2. Покажете как се декларира структура?

3. Как се достъпват членовете на структура?

4. Какъв тип могат да бъдат членовете на една структура?

5. Декларирайте структура с име mystr, която има членове d от тип double и i от тип int. Дефинирайте структурна-променлива с име s. Задайте стойности на членовете на структурата и ги отпечатайте на екрана.

6. Как може да опростите декларацията на структурата от т.5 с помощта на typedef?

7. Може ли една структура да съдържа член от себе си?

19 Обединения

19.1 Въведение

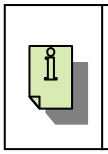
Обединението, подобно на структурата, е потребителски тип данни, в който обаче всички членове използват една и съща памет, т.е. . обединението е средство, чрез което могат да се създадат променливи, които да поделят обща памет. Разбира се, в даден момент може да се използва само една от тези променливи. Декларирането на обединение става по следния начин:

```
union  име-обединение
{
      тип   име-променлива1;
      тип   име-променлива2;
      . . .
      тип   име-променливаN;
};
```

Членовете на едно обединение могат да бъдат:

- променливи от базови типове;
- масиви;
- указатели;
- структурни-променливи;
- променливи от друго обединение;
- битови полета (битовите полета са разгледани са в 20 Битови полета);
- променливи от enum-тип (enum-типовете са разгледани в 21 Изброявания).

Декларацията на обединението представлява създаване на потребителски тип данни, но не води до заделяне на памет. Това е просто един шаблон, който показва структурата на потребителския тип данни. Компилаторът заделя памет (достатъчна да побере променливата с най-голям размер), едва когато потребителят дефинира променлива от този тип. Променливите от този тип ще наричаме union-променливи за по-кратко.



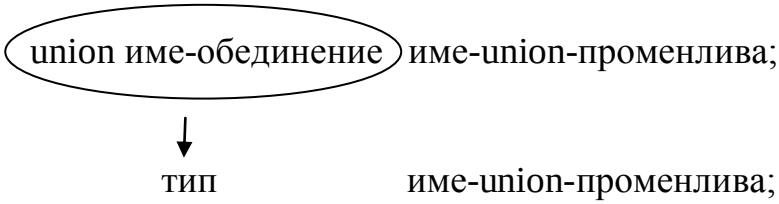
Обединението е средство, чрез което две или повече променливи могат да използват обща памет. Компилаторът заделя памет достатъчна да побере променливата с най-голям размер.

Декларацията на едно обединение е видима от точката на деклариране до края на блока или файла, в който се намира декларацията.

Дефинирането на union-променлива изглежда така:

union име-обединение име-union-променлива;

Представете си тази форма по следния начин:



Цялата конструкция **union име-обединение** представлява името на типа данни. Когато срещне такава дефиниция, компилаторът заделя памет, достатъчна да побере променливата с най-голям размер. Достъпът до членовете на обединението става по следния начин:

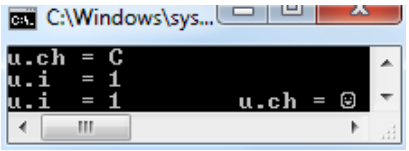
име-union-променлива . име-променлива

т.е. . името на union-променливата е последвано от точка, последвана от името на член на обединението.

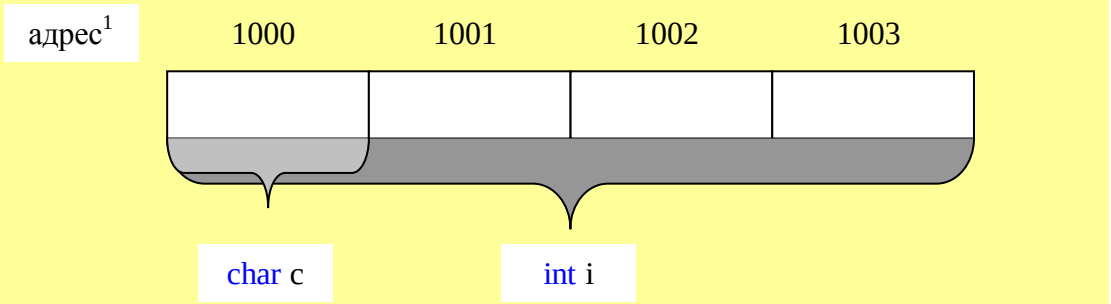
Пример 19-1:

```
#include <stdio.h>

/* Декларация на обединение */
union myunion
{
      int  i;
      char ch;
};

int main(void)
{
       /* Дефиниране на union-променлива от тип myunion */
       union myunion   u;

       /* Присвояване на стойност u.ch                  */
       u.ch = 'C';

       /* Отпечатване  u.ch на екрана                   */
       printf("u.ch = %c\n", u.ch);
```

```
        /* Присвояване на стойност u.i                    */
        u.i  = 1;

        /* Отпечатване  u.i на екрана                     */
        printf("u.i  = %d\n", u.i);

        /* !!!Внимание. Отпечатване на u.i и u.ch едновременно на екрана */
        printf("u.i  = %d\tu.ch = %c\n", u.i, u.ch);

        return 0;
}
```



Компилаторът заделя памет, така че да побере променливата i. Ако допуснем, че тип int използва 4 байта, то схематично това изглежда така:



Забележка[1]: Адресите, на които се разполагат членовете на union-променливата, са произволно избрани в примера и имат само илюстративен характер.

Фиг. 76 Union-променлива u

Променливата ch, която е от тип char, заема младшият байт на променливата i. Това показва, че когато съхраните стойност например в променливата i, тази стойност използва също и паметта, в която се съхранява променливата ch и обратно.

Union-променливите могат да се дефинират и по време на декларирането на самото обединение. В този случай името на обединението може да се пропусне. Обърнете внимание, че ако изпуснете името на обединението няма как по-късно в кода да дефинирате други union-променливи от този тип.

19.2 Инициализиране на union-променливи

Като всяка променлива union-променливите също могат да се инициализират по време на дефинирането си. Обърнете внимание, че само първият член на обединението може да се инициализира. Ето защо типът на инициализатора трябва да бъде съвместим с типа на първия член и може да бъде само константа.

Пример 19-2:

```
#include <stdio.h>

/* Декларация на обединение */
union myunion
{
      int  i;
      char ch;
}u = {1}; /* Инициализира се члена i */

int main(void)
{
      /* Отпечатване  u.i на екрана */
      printf("u.i  = %d\n", u.i);

      return 0;
}
```

19.3 Обединение и операцията sizeof

Действителният размер на union-променлива трябва да се определя с операцията sizeof. Причината е, че компилаторът може да вмъкне уплътняващи байтове след последния член на обединението.

	**Винаги използвайте операцията sizeof, за да определяте размера на обединение.**

19.4 Присвояване на union-променливи

Union-променливите могат да се присвояват една на друга. Това е възможно обаче само ако са от един и същ тип.

19.5 Указатели и обединения

Членовете на едно обединение могат да се достигат и чрез указател. Обобщената форма на дефиниране на указател към обединение изглежда така:

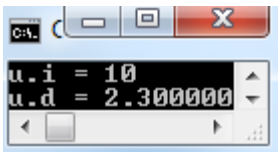
union име-обединение *име-указател;

Достъпът до членовете на обединение чрез указател става по следния начин:

име-указател -> име-член

Пример 19-3:

```
#include <stdio.h>

struct myunion
{
	int i;
	double d;
};

int main(void)
{
	struct myunion u;   /* Дефиниране на union-променлива                */
	struct myunion *pu; /* Дефиниране на указател към обединение myunion */

	pu = &u; /* Присвояване на адреса на union-променлива u */
	pu->i = 10;
	printf("u.i = %d\n", pu->i);
	pu->d = 2.3;
	printf("u.d = %f\n", pu->d);

	return 0;
}
```



19.6 Масиви от обединения

Обединенията също могат да бъдат елементи на масиви. Общата форма на дефиниране на масив от обединение е:

union име-обединение име-масив[размер];

тип на елементите

Всички правила за работа с масиви и обединения са валидни и при работа с масиви от обединения.

Въпроси за самопроверка

1. Обяснете какво представлява обединението?

2. Покажете как се дефинира обединение?

3. Какъв тип могат да бъдат членовете на едно обединение?

4. Как се достъпват членовете на обединение?

5. Декларирайте обединение с име myunion, което има членове d от тип double и i от тип int. Дефинирайте union-променлива с име u. Задайте стойности на членовете на обединението и ги отпечатайте на екрана.

6. Как може да опростите декларацията на обединението от т.5 с помощта на typedef?

20 Битови полета

20.1 Въведение

Битовото поле е целочислена променлива, която се състои от определен брой битове. Битовото поле може да бъде само член на структура или обединение. Дефинирането на битово поле изглежда така:

тип име-битово-поле: широчина;

където

тип

Може да бъде int[1], signed int или unsigned int

Забележка[1]: В контекста 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 константен 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.

Пример 20-1:

```
struct BitFieldsDemo
{
      unsigned b0 : 3;
      signed   b1 : 3;
      unsigned b2 : 1
};
```

Битовите полета могат да се използват навсякъде, където целочислени променливи могат да се използват.

За достъп до битовите полета се използват същите конструкции както и за достъп до обичайните членове на структура или обединение:

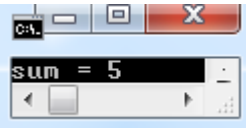
име-структурна-променлива . име-битово-поле

име-указател-към-структура -> име-битово-поле

Пример 20-2:

```
/* Деклариране на структура с битови полета */
struct BitFieldsDemo
```

```
{
	unsigned b0 : 3;
	unsigned b1 : 8;
};

int main(void)
{
	struct BitFieldsDemo	b, *p;
	int			sum;

	p = &b;

	b.b0	= 1;
	p->b1	= 4;

	sum = b.b0 + b.b1;

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

	return 0;
}
```



Когато едно битово поле е част от израз, то се преобразува в тип int или unsigned int съгласно обичайните унарни преобразувания (погледнете Табл.42 от 14.2 Обичайни унарни преобразувания).

Една структура може да съдържа битови полета и обичайни членове едновременно.

Пример 20-3:

```
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. Това води до минимизиране на броя байтове заемани от структурната-променлива.

20.2 Битови полета със специално предназначение

Преди да разгледам битовите полета със специално предназначение, ще отбележа, че C89 не дефинира размера на сегмента за съхранение (**storage unit**) на битовите полета, нито дали пакетирането на битовете започва от

младшия, или старшия бит на този сегмент. Типично размерът на сегмента съвпада с размера на машинната дума, т.е. . 8-, 16- 32- и т.н. бита, но това не е задължително.

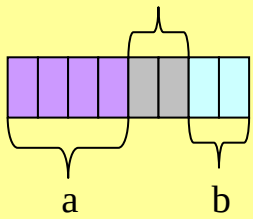
C89 не дефинира размера на сегмента за съхранение (storage unit) на битовите полета, нито дали пакетирането на битовете започва от младшия или старшия бит на сегмента.

С стандартът позволява дефинирането на неименувани (безименни) битови полета. Неименуваните битови полета с широчина различна от 0 играят ролята на уплътняващи битове и не могат да се достъпват от кода.

Пример 20-4:

```
struct S
{
      unsigned a : 4;
      unsigned   : 2; /* уплътняващи битове */
      unsigned b : 2;
};
```

Уплътняващи битове



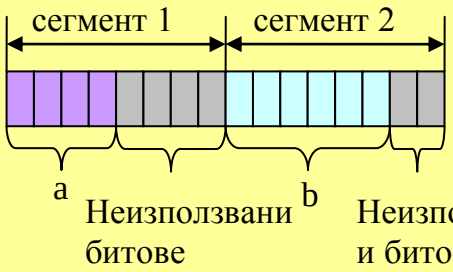
Приема се, че сегментът е 8 бита и битовите полета се съхраняват от ляво надясно (от старшия към младшия бит).

Фиг. 77 Уплътняващи битове

Неименувано битово поле с дължина 0 бита принуждава компилатора да спре пакетирането на битовите полета в текущия сегмент за съхранение, в който е пакетирал предходните битови полета, и да започне пакетирането от началото на следващия сегмент.

Пример 20-5:

```
struct S
{
      unsigned a : 4;
      unsigned   : 0;
      unsigned b : 6;
};
```



Приема се, че сегментът е 8 бита и битовите полета се съхраняват от ляво надясно (от старшия към младшия бит).

Фиг. 78 Нулево битово поле

	Използвайте битовите полета само за създаване на целочислени променливи с определена дължина, без да разчитате на определено вътрешно разполагане на битовите полета в паметта. Код, който разчита на конкретно разполагане на битовите полета в паметта е непреносим на друга компютърна платформа.

20.3 Инициализиране на битови полета

Битовите полета могат да се инициализират по същия начин както и обичайните полета на структурата.

Пример 20-6:

```
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 съдържа неименувани битови полета, инициализатори за тях не се изискват. Ако се опитате да добавите инициализатор за неименувано поле, компилаторът ще генерира грешка.

Пример 20-7:

```
struct S
{
	unsigned a : 4;
	unsigned   : 4;
	unsigned b : 6;
};
```

```
int main(void)
{
	struct S  bitfields = {3, 10}; /* bf.a = 3, bf.b = 10 */
	/*...*/
	return 0;
}
```

20.4 Ограничения на битовите полета

Битовите полета имат следните ограничения:

- Битово поле не може да съществува самостоятелно извън тялото на структура или обединение.

Пример 20-8:

```
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 20-9:

```
 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 20-10:

```
int main(void)
{
	int:5  *p; /* !!!Грешка, това е невалидна конструкция в C */

	return 0;
}
```

- Не може да се дефинира масив от битови полета

Пример 20-11:

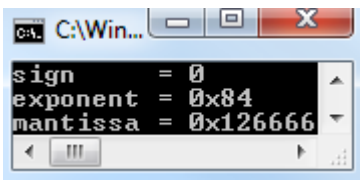
```
int main(void)
{
      int:5  arr[10]; /* !!!Грешка, това е невалидна конструкция в C */

      return 0;
}
```

20.5 Употреба на битовите полета

Както вече разбрахте, C89 дава голяма свобода на компилаторите по отношение на битовите полета. Тяхната употреба обикновено е привързана към конкретна компютърна архитектура. По тази причина битовите полета имат много ниска преносимост на други компютърни архитектури. Следващият пример отпечатва стойностите на съставните части на число с плаваща запетая тип float по стандарта IEEE 754 в средата на Microsoft Visual Studio C++ 2010.

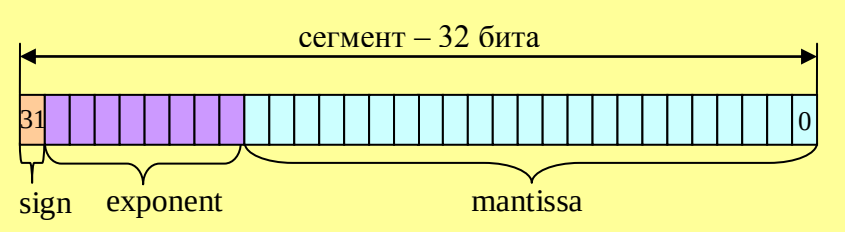
Пример 20-11:

```
#include <stdio.h>

struct IEEE754_SP {
    unsigned int mantissa : 23;
    unsigned int exponent :  8;
    unsigned int sign     :  1;
} *fp;

int main(void)
{
    float f = 36.6f;

    fp = (struct IEEE754_SP *)&f;

    printf("sign     = %d\n" , fp->sign ? 1 : 0);
    printf("exponent = %#x\n", fp->exponent    );
    printf("mantissa = %#x\n", fp->mantissa    );

    return 0;
}
```



Обръщам Ви внимание, че горният пример може даде други резултати, ако се изпълни на други компютърни платформи или дори да не се компилира,

ако широчината на тип int е 16 бита, тъй като широчината на битовото поле **mantissa** я надхвърля.



Фиг. 79 Разположение на битовите полета на структурата IEEE754_SP

Все пак, може да използвате битови полета просто да дефинирате целочислени променливи с определена дължина, без да се интересувате от конкретно разположение на битовете в паметта. Например битовите полета често се използват за дефиниране на флагове. Флагът е променлива, която има само две стойности (0 и 1) и служи за указване дали дадено събитие е възникнало, или не. Обикновено флаг със стойност 0 означава, че събитието не е възникнало. Следващият пример ще се работи еднакво за всички компютърни платформи.

Пример 20-12:

```
#include <stdio.h>

#define TRUE  1
#define FALSE 0

/*Деклариране на структура от флагове */
struct
{
	unsigned b0: 1;
	unsigned b1: 1;
	unsigned b2: 1;
	unsigned b3: 1;
	unsigned b4: 1;
	unsigned b5: 1;
	unsigned b6: 1;
	unsigned b7: 1;
} flags;

int main(void)
{
	/*...*/
	flags.b0 = FALSE;
	/*...*/

	if (TRUE == flags.b0)/* Проверка на флаг b0 */
	{
		/*...*/
	}

	/*...*/

	return 0;
}
```

Въпроси за самопроверка

1. Покажете как се дефинира битово поле?
2. Какво представлява битово поле без име и широчина различна от 0?
3. Какво представлява битово поле без име и широчина равна на 0?
4. Какви ограничения налага стандартът C на битовите полета?

21 Изброявания

Изброяването е средство, чрез което могат да се създават именувани целочислени константи. Декларирането на изброяване става по следния начин:

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

Декларацията на едно изброяване е видима от точката на деклариране до края на блока или файла, в който се намира декларацията.

По-подразбиране **име-константа1** се инициализира автоматично с 0, **име-константа2** с 1 и т.н., т.е. . всяка следваща константа от списъка е с единица по-голяма от предходната. Ще наричаме тези константи enum-константи.

Enum-константите могат изрично да бъдат инициализирани от потребителя със следната форма:

```
enum име-изброяване
{
        име-константа1  = инициализатор1,
        име-константа2  = инициализатор2,
        ...
        име-константаN = инициализаторN
};
```

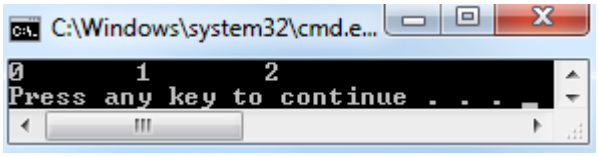
където

инициализатор1, ... , инициализаторN
Може да бъде всеки целочислен константен израз тип int.

Важно е да запомните, че всяка константа, която не е изрично инициализирана, е с единица по-голяма от предходната. Типът на enum-константите технически съвпада с тип int, т.е. . инициализаторът може да има стойности каквито тип int може да съдържа за съответната система. Друго, което трябва да разберете, е, че горните две конструкции не водят до заделяне на памет. Това са конструкции, чрез които, както казах, може да зададете имена на целочислени константи и да използвате тези имена в програмата вместо числовите стойности.

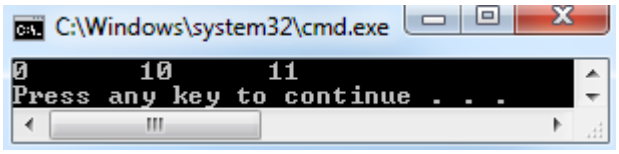
Пример 21-1:

```
#include <stdio.h>

/* Декларация на enum-константи */
enum myenum {CAR, TRUCK, TRACTOR};

int main(void)
{
	/* Отпечатване на enum-константите на екрана */
	printf("%d\t%d\t%d\n",CAR,TRUCK,TRACTOR);

	return 0;
}
```



Пример 21-2:

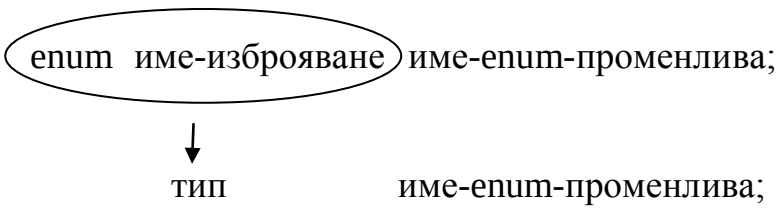
```
#include <stdio.h>

/* Декларация на enum-константи */
enum myenum {CAR, TRUCK = 10, TRACTOR};

int main(void)
{
	/* Отпечатване на enum-константите на екрана */
	printf("%d\t%d\t%d\n",CAR,TRUCK,TRACTOR);

	return 0;
}
```



Също така може да дефинирате и променливи от тип enum. Ще наричаме такива променливи enum-променливи за по-кратко.

enum име-изброяване име-enum-променлива;

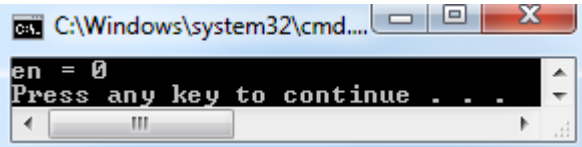
Представете си тази форма по следния начин:



Цялата конструкция **enum име-изброяване** представлява името на типа данни. След като срещне такава дефиниция, компилаторът заделя памет за enum-променливата. На enum-променливата могат да се присвояват стойности от списъка с enum-константи.

Пример 21-3:

```
#include <stdio.h>

/* Декларация на enum-константи */
enum myenum {CAR, TRUCK, TRACTOR};

int main(void)
{
	/* Дефиниране на променлива */
	enum myenum  en;

	/* Задаване на стойност на en*/
	en = CAR;

	/* Отпечатване на en         */
	printf("en = %d\n", en);

	return 0;
}
```



Enum-променливите могат да се дефинират и по време на декларирането на самото изброяване. В този случай името на изброяването може да се пропусне. Обърнете внимание, че ако изпуснете името на изброяването, няма как по-късно в кода да дефинирате други enum-променливи от този тип.

По същество enum-променливите са целочислени променливи. Действителният тип, който стои зад един enum-тип, зависи от имплементацията на компилатора. Компилаторът може да избере най-подходящия от целочислените типове char (signed, unsigned), short (signed, unsigned) или int в зависимост от стойностите на enum-константите. Когато enum-променлива е част от израз, тя се преобразува в тип int съгласно обичайните унарни преобразувания (погледнете Табл.42 от 14.2 Обичайни унарни преобразувания).

Можете да използвате enum-променливи навсякъде, където можете да използвате и целочислени променливи (в израз, като параметър на функция, като аргумент на функция и т.н.). Може да дефинирате указатели към enum-променливи, масиви от enum-променливи и т.н.

Въпроси за самопроверка

1. Обяснете какво представлява изброяването?

2. Покажете как се декларира изброяване?

3. Ако е дадено изброяването enum rgb {RED, GREEN, BLUE};, какви стойности компилаторът ще присвои на enum-константите RED, GREEN и BLUE?

22 Функции

22.1 Въведение

Функциите са основните градивни елементи на една C програма. Една функция може да се разглежда като парче код, което започва да се изпълнява, когато функцията се извика. Тук е показан още веднъж обобщеният изглед на една C програма.

```
/* Прототипи на функции */
тип-резултат     func1(списък-параметри);
тип-резултат     func2(списък-параметри);
...
тип-резултат     funcN(списък-параметри);

/* Входна точка на програмата */
int main(void)
{
        func1(списък-аргументи);
        func2(списък-аргументи);
        ...
        funcN(списък-аргументи);
        ...
        return 0;
}

/* Дефиниране на функции */
тип-резултат     func1(списък-параметри)
{
        /* Може да извиква други функции */
}

тип-резултат     func2(списък-параметри)
{
        /* Може да извиква други функции */
}
…
тип-резултат     funcN(списък-параметри)
{
        /* Може да извиква други функции */
}
```

Една функция може да извиква друга функция, която от своя страна извиква трета функция, и т.н. Както казах в началото на книгата, програмата започва с извикването на функцията main.

Една функция се дефинира по следния начин:

```
тип-резултат      име-функция (списък-параметри)
{
        /**************************************
        * Между тези две фигурни скоби        *
        * се разполага кодът, който се изпълнява *
        * от функцията                        *
        **************************************/
}
```

Частта на функцията, означена като **тип-резултат име-функция (списък-параметри),** се нарича още заглавие на функцията. Както се вижда то 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 (базов тип, структурен-тип, union-тип, enum-тип, указателен тип) с изключение на масиви, т.е. . една функция не може да връща цял масив, като резултат от изпълнението си. Върнатата 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.

Една функция връща стойност с помощта на оператора return. Той има следния синтаксис:

```
/* За функции, които не връщат стойност */
return;

/* За функции, които връщат стойност */
return      връщана-стойност;
```

връщана-стойност може да бъде константа, променлива или израз. Типът на **връщана-стойност** трябва да е съвместим с дефинирания в заглавието на функцията тип-резултат. Под "съвместим" се има предвид същия тип или тип, който може да се преобразува към типа, указан с **тип-резултат.** Правилата за преобразуване са същите като при присвояване.

Операторът return може да се появи навсякъде в една функция. Той води до прекратяване на изпълнението на кода на функцията и програмата продължава с изпълнението на оператора след извикването на функцията. Една функция може да има повече от един оператор return. За функции, които не връщат стойност, затварящата фигурна скоба на тялото на

функцията действа също като return и той може да се изпусне в края на функцията.

- име-функция

Името на функцията се използва за нейното извикване. Образуването на името следва същите правила като при променливите. Извикването на функция става така:

име-функция(списък-аргументи);

където

списък-аргументи
Представлява стойности, разделени със запетаи, които се подават на функцията. Ако функцията не приема стойности, списък-аргументи е празен.

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 аргументите, подавани на функцията при нейното извикване, т.е. .

тип име-параметър1, тип име-параметър2, ... , тип име-параметърN

където

тип
Може да бъде всеки един от типовете данни в C (базов тип, структурен-тип, union-тип, enum-тип, указателен тип, масиви).

Ако функцията не приема аргументи, списък-параметри се заменя с ключовата дума void.

Обърнете внимание на следните термини: **аргумент** и **параметър**. Аргументът може да бъде константа, променлива или всеки допустим израз, който се подава на функцията при нейното извикване. Параметърът е променливата, която приема този аргумент. Ето защо аргументът и съответният параметър трябва да са от съвместими типове. Под „съвместим“ се има предвид същият тип или тип, който може да се преобразува към типа на параметъра. Правилата за преобразуване са

същите като при присвояване.

Частта на функцията, затворена между { и }, се нарича тяло на функцията. Тялото на функцията съдържа операторите, които изграждат кода, изпълняван от тази функция.

Всяка функция е добре да има прототип. Прототипът съвпада по форма със заглавието на една функция, но завършва с точка и запетая, т.е. :

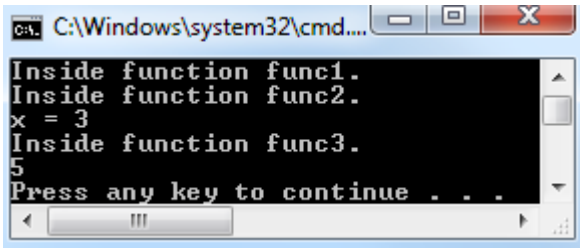
тип-резултат име-функция(списък-параметри);

Прототипите не са задължителни в C89, но няма да видите професионално написана програма без тях. Те дават информация на компилатора за типа и броя на параметрите, а също и за типа на връщаната стойност. Въз основа на тази информация компилаторът определя дали функцията се извиква с правилния тип, и/или брой аргументи, а също дали връщаната стойност е от обявения с **тип-резултат** тип. В случай на неправилно използване на функцията, компилаторът ще генерира съобщение за грешка. Прототипът трябва да се разполага в кода преди извикването на функцията. Обикновено прототипите се поставят в началото на файла. Функцията main е единствената функция, която не се нуждае от прототип, защото е определена от стандарта.

Пример 22-1:

```
#include <stdio.h>

/* Прототипи на функции */
void func1(void);
void func2(int x);
int  func3(int x, int y);

int main(void)
{
	int a;

	/* Извикване на функции */
	func1();
	/*func1(3);*/	/* !!!Грешка, func1(), няма параметри */

	func2(3);
	/*func2(); */	/* !!!Грешка, func2() изисква един аргумент от тип int */

	a = func3(2,3);
	/*a = func3(2);*/  /* !!!Грешка, func3() изисква два аргумента, а не един */

	/* Отпечатване на резултата от func3() на екрана    */
	printf("%d\n", a);

	return 0;
}

/* Дефиниране на функцията func1 */
void func1(void)
```

```
{
	printf("Inside function func1.\n");

	return;
}

/* Дефиниране на функцията func2 */
void func2(int x)
{
	printf("Inside function func2.\n");
	printf("x = %d\n", x);

	return;
}

/* Дефиниране на функцията func3 */
int func3(int x, int y)
{
	int temp;

	printf("Inside function func3.\n");
	temp = x + y;

	return temp;
}
```



Винаги използвайте прототипи. Това ще позволи на компилатора да хване грешки, свързани с неправилно използване на функцията, като подаване на грешен тип и/или брой аргументи.

22.2 Механизъм 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, например структурна-променлива с много членове. Вместо тези данни да се копират в параметъра, може да се подаде само адреса на структурната променлива. Това значително увеличава производителността на програмата и избягва проблеми с препълване на стека.

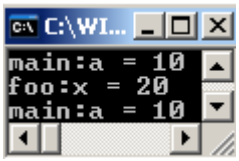
Независимо как се подава аргументът действията, които се извършват, са аналогични на действията при присвояване. Ако типът на аргумента е различен от този на параметъра, компилаторът ще се опита да го преобразува. Ако преобразуването е невъзможно, ще се генерира грешка при компилиране.

22.2.1 Подаване на аргументи чрез копие

Следващият пример демонстрира факта, че изменението на параметъра в тялото на функцията не влияе по никакъв начин на аргумента.

Пример 22-2:

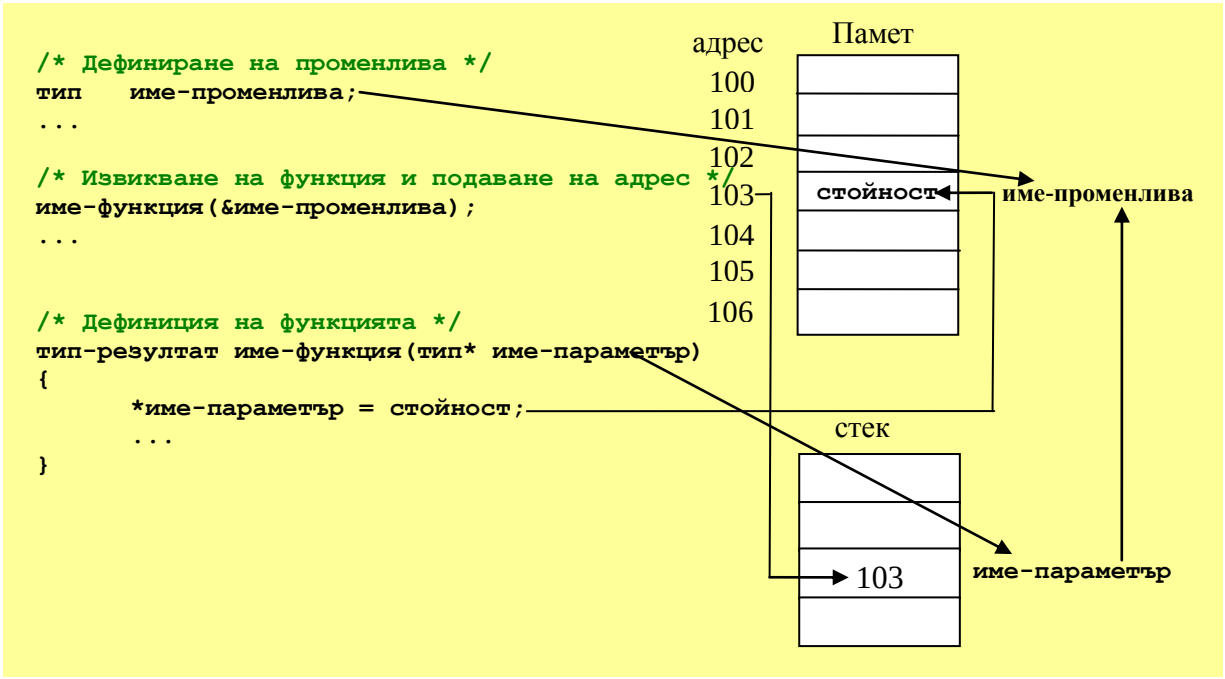
```
#include <stdio.h>

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;
	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.

22.2.2 Подаване на аргументи чрез адрес

За да се подаде аргумент чрез адрес, е необходимо параметърът да бъде дефиниран като указател към съответния тип. При извикване на функцията се подава адреса на аргумента, а не неговата стойност. Преди влизане в тялото на функцията адресът 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.81 илюстрира това.

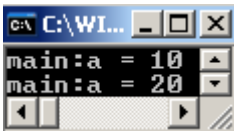


Фиг. 81 Подаване на аргумент чрез указател

Пример 22-3:

```
#include <stdio.h>

void foo(int* p);

int main(void)
{
	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;
}
```

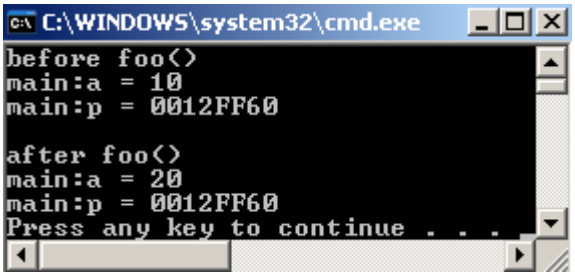
```
void foo(int* p)
{
	*p = 20; /* Аргументът, сочен от указателя p, се променя */
}
```



	Технически погледнато един указател-аргумент също се подава чрез копие на указателя-параметър, т.е. адресът, 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 22-4:

```
#include <stdio.h>

void foo(int* ptr);

int main(void)
{
	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;

	/* "разскачване" на ptr от сочения от него обект */
	ptr = NULL;
}
```

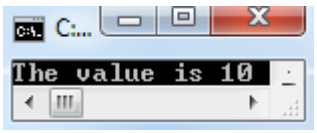


Обърнете внимание, че когато стойността на указател се отпечатва на екрана форматният спецификатор е **%p**.

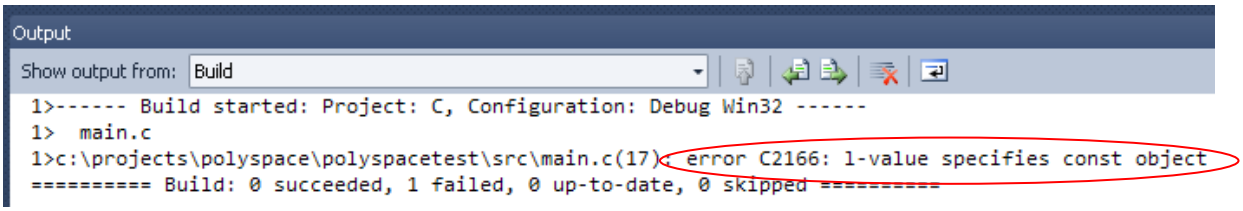
Следващият пример показва как да защитим обекта, сочен от указателя-параметър, от изменение в тялото на функцията.

Пример 22-5:

```
#include <stdio.h>

void print_value(const int* ptr);

int main(void)
{
	int x = 10;

	print_value(&x);

	return 0;
}

void print_value(const int* ptr)
{
	printf("The value is %d\n", *ptr);

	/* *ptr = 20; */ /*!!!Грешка, обектът не може да се променя от указателя */
}
```



В случаи, когато обект се подава на функция по адрес и тази функция не трябва да го изменя, е препоръчително параметърът да се декларира като указател към константна-променлива. Тогава всеки случаен опит да се промени обектът от функцията ще бъде засечен от компилатора при компилиране на кода. За да се убедите в това, махнете коментара от израза *ptr = 20 и компилирайте. Ще получите следното съобщение за грешка:

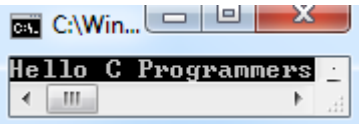


	Ако обект се подава на функция по адрес и тази функция не трябва да го изменя, е препоръчително параметърът да се декларира като указател към константен-обект.

Следващият пример е малко по-практичен. Той демонстрира копиране на един низ в друг. Забележете, че copc-указателят е деклариран като указател към константен-обект, за да защитим обекта, към който сочи, от случайно изменение във функцията.

Пример 22-6:

```
#include <stdio.h>

void copy_strings(char* destination, const char* source);

int main(void)
{
	char a[] = "Hello C Programmers\n";
	char b[sizeof(a)];

	copy_strings(b,a); /* Копиране на  низа a  в буфера b */
	printf("%s", b);

	return 0;
}

void copy_strings(char* destination, const char* source)
{
	while(*source != '\0') /* Проверка за край на низа */
	{
		*destination++ = *source++; /* Копиране на текущия символ и
					       увеличаване на указателите */
	}

	*destination = '\0'; /* Добавяне на нулев символ */
}
```



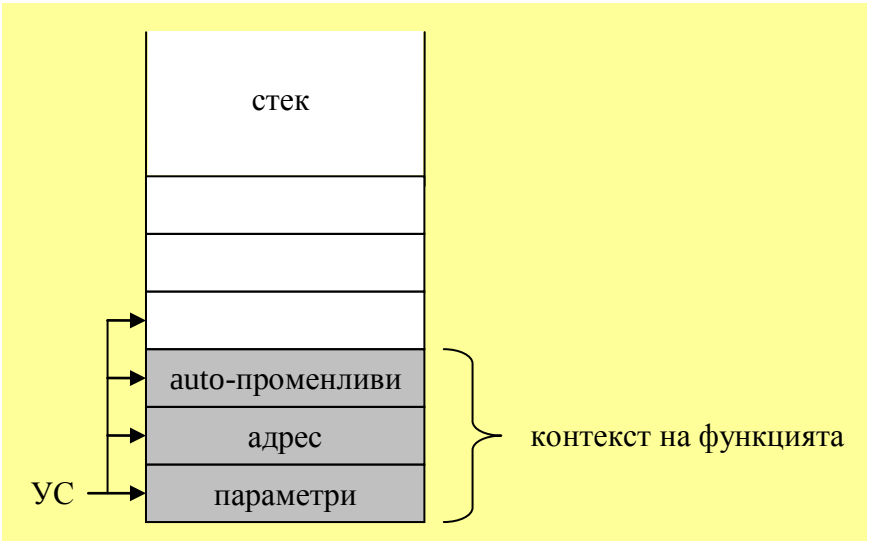
Сега след като се запознахте с указателите и подаването на аргументи на функция чрез адрес, би трябвало да разбирате как работи функцията scanf() (12 Четене на данни от клавиатурата). Функцията scanf() приема адресите на своите аргументи и съхранява в тях прочетените от клавиатурата данни.

22.3 Стек

В тази точка ще разгледам какво точно е стек и обобщения принцип на работа на стека. Когато програмирате на език от по-високо ниво, какъвто е езикът 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

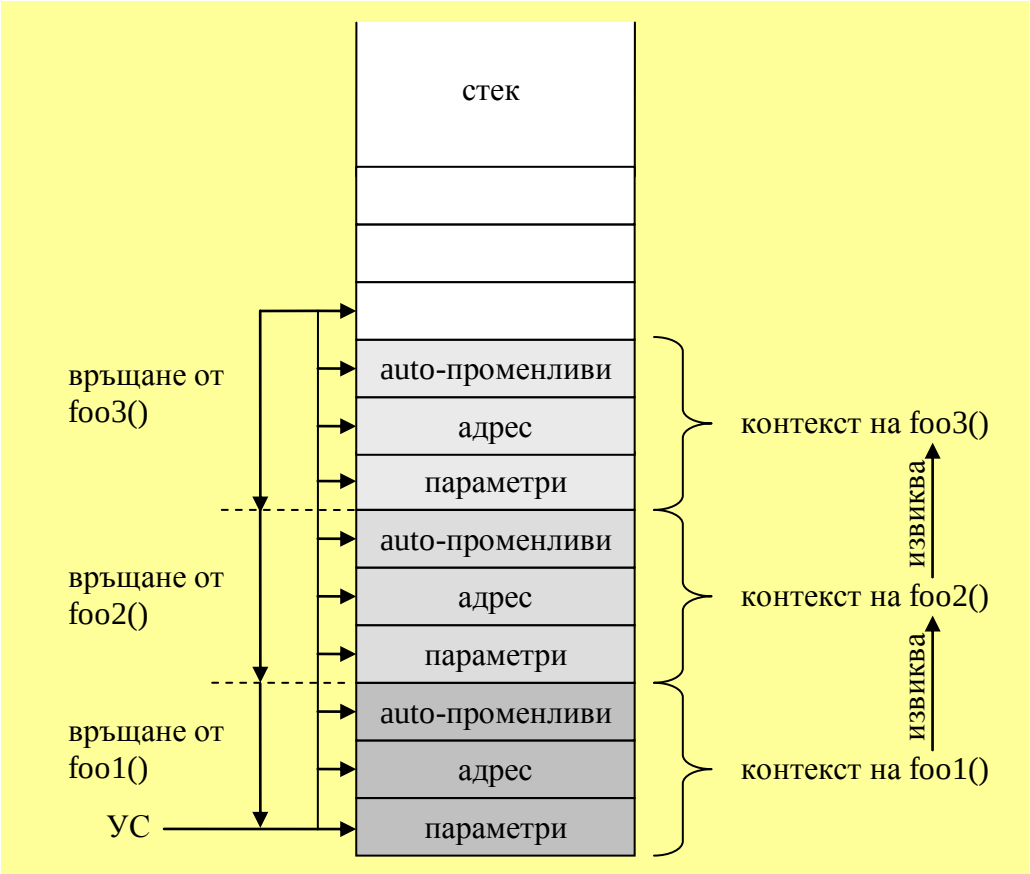
поредната свободна клетка от него. Когато извикате една функция, компилаторът заделя памет за параметрите (ако има такива) в стека и УС се увеличава така, че да сочи към следващата свободна клетка. Компилаторът също генерира асемблерен код, който копира аргументите в съответните параметри, след което се прави преход към тялото на функцията (с помощта на специална асемблерна инструкция). Спомнете си, че в глава 5 Архитектура на компютъра Ви описах обобщения принцип на извличане и изпълнение на инструкциите от един процесор и по-точно, че адресът на всяка следваща инструкция се съхранява в специален регистър, наречен **Брояч на инструкциите**. Адресът, съхранен в този регистър, също се запомня в стека. Това е необходимо, за да може след връщане от функцията програмата да продължи с изпълнение на инструкцията, която се намира след точката на извикване на функцията. 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 паметта за локалните auto-променливи на функцията се освобождава, адресът на връщане се записва отново в Брояча на инструкциите и 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. 82).



Фиг. 82 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. 83).



Фиг. 83 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.

22.4 Указатели и функции

В предишната точка разгледахме, че указателите могат да се подават на функции като аргументи. Затова в тази точка ще разгледам само особеностите свързани с връщане на указател от функция. Общата форма на дефиниране на функция, връщаща указател, е следната:

```
тип*   име-функция(списък-параметри)
{
      …
      return  указател;
}
```

Обърнете внимание, че това, което връща функцията, е адрес. Спомнете си, че паметта за локалните auto-променливи се освобождава при завършване на функцията. Това означава, че ако функцията върне адрес на такава променлива, то този адрес ще бъде невалиден след излизане от функцията. Компилаторът може да генерира предупреждение, но ще компилира кода без проблем.

Пример:

```
int* foo(void)
{
  int x = 10;
  /*...*/
  return &x;/*!!!Внимание, адресът ще бъде невалиден след излизане от функцията */
}
```

Ето защо, 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 (глобална или локална статична променлива).

Пример:

```
int* foo(void)
{
  static int x = 10;
  /*...*/
  return &x;/* Позволено, променливата съществува и след излизане от функцията */
}
```

22.5 Масиви и функции

Масивите също могат да се подават като аргументи на функция. За целта името на масива служи като аргумент. Спомнете си, че името на масив е

указател. Тогава е логично и параметърът, 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 адресът на първия елемент на масива, а не целият масив.

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);**

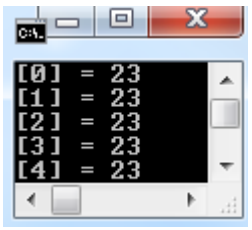
В първата декларация, въпреки че е посочен и размера на масива, той се игнорира от компилатора. Ето защо размерът може да се изпусне както е показано във втората декларация. В крайна сметка тези две декларации се трансформират до третата декларация. Тя най-точно отразява механизма на предаване на масив на функция.

След като функцията има адреса на масива, тя може да го променя.

Пример 22-7:

```
#include <stdio.h>

#define SIZE 5

void init_array(int p[], int initializator);
void print_array(int p[]);

int main(void)
{
   int arr[SIZE]; /* Същото като int arr[5] */

   init_array(arr, 23); /* Инициализиране на всички елементи на масива с 23 */
   print_array(arr); /* Отпечатване на елементите на масива */

   return 0;
}

void init_array(int p[], int initializator)
{/* същото като void print_array(int *p, int initializator) */

      int index;

      for (index = 0 ; index < SIZE ; index++)
      {
            p[index] = initializator;
      }
}
```

```
void print_array(int p[])
{/* същото като void print_array(int *p) */
	int index;

	for (index = 0 ; index < SIZE ; index++)
	{
		printf("[%d] = %d\n", index, p[index]);
	}
}
```



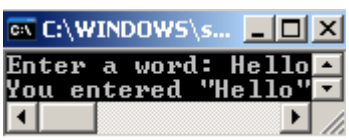
Също както една функция не може да приеме цял масив като аргумент, така функцията не може да върне цял масив като резултат. Тъй като име на масив е указател, то конструкция от вида

return a;

където a е име на масив, в действителност връща адреса на първия елемент на масива. Това означава, че типът на връщания резултат в дефиницията на функцията трябва да бъде указател (вижте 22.4 Указатели и функции).

Пример 22-8:

```
#include <stdio.h>

char* read_word(void);

int main(void)
{
	printf("You entered \"%s\"\n", read_word());

	return 0;
}

char* read_word(void)
{
	static char word[20]; /* Създаване на буфер, в който се съхранява
				 въведената дума */

	printf("Enter a word: "); /* Извеждане на подсказващо съобщение */
	scanf("%s", word);	  /* Четене на въведената дума	       */

	return word; /* Връщане на адреса на първия символ на думата */
}
```



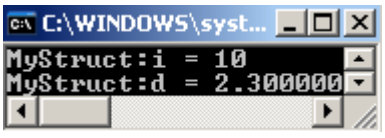
22.6 Структури и функции

Структурните-променливи могат да се подават като аргументи на функции. По подразбиране те се предават по стойност. Това означава, че всяка промяна на параметъра няма да се отрази на аргумента. Общата форма на дефиниране на функция, която приема структурна-променлива за аргумент, е следната:

```
тип   име-функция(struct име-структура   име-параметър)
{
      …
}
```

Пример 22-9:

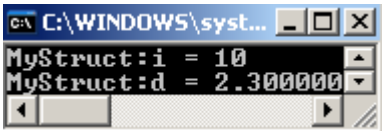
```
#include <stdio.h>

struct MyStruct
{
      int i;
      double d;
};

void print_structure(struct MyStruct s);

int main(void)
{
      struct MyStruct  ms;

      ms.i = 10;
      ms.d = 2.3;
      print_structure(ms);

      return 0;
}

void print_structure(struct MyStruct s)
{
      printf("MyStruct:i = %d\n",  s.i);
      printf("MyStruct:d = %f\n", s.d);
}
```



Когато структурна-променлива се предава на функция по стойност, трябва да се има предвид, че ако структурата съдържа много членове, то времето за копиране на тези членове в членовете на параметъра може да окаже влияние на производителността на програмата. Също така, може да възникне и препълване на стека. По тези причини структурните-променливи се предават на функция най-често по адрес.

Пример 22-10:

```
#include <stdio.h>

struct MyStruct
{
	int i;
	double d;
};

void print_structure(struct MyStruct *ps);

int main(void)
{
	struct MyStruct  ms;

	ms.i = 10;
	ms.d = 2.3;
	print_structure(&ms);

	return 0;
}

void print_structure(struct MyStruct *ps)
{
	printf("MyStruct:i = %d\n",  ps->i);
	printf("MyStruct:d = %lf\n", ps->d);
}
```



Обърнете внимание, че функцията може да променя членовете на структурната променлива ms. Ако искате да се защитите от подобна възможност, използвайте ключовата дума const, т.е. . const struct MyStruct *ps.

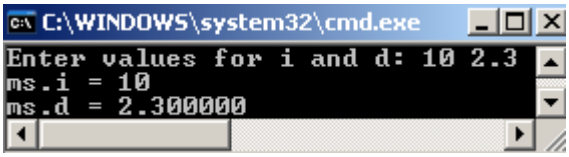
Структурните-променливи също така могат да се връщат като резултат от функции. Общата форма на дефиниране на функция, която връща структурна-променлива, е:

struct име-структура име-функция(списък-параметри)
{
	…
	return структурна-променлива;
}

Пример 22-11:

```
#include <stdio.h>

struct MyStruct
```

```
{
	int i;
	double d;
};

struct MyStruct init_structure(void);

int main(void)
{
	struct MyStruct  ms;

	ms = init_structure();
	printf("ms.i = %d\n", ms.i);
	printf("ms.d = %f\n", ms.d);

	return 0;
}

struct MyStruct init_structure(void)
{
	struct MyStruct ms;

	printf("Enter values for i and d: ");
	scanf("%d%lf", &ms.i, &ms.d);

	return ms;
}
```



Ако една функция връща указател към структура, то обърнете внимание, че обектът, чийто адрес се връща, трябва да бъде със статична продължителност на живот (вижте 22.4 Указатели и функции).

22.7 Клас памет на функциите

Ключовите думи extern и static също могат да се прилагат и към 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 22-12:

```
#include <stdio.h>

static void foo(void);

int main(void)
{
```

```
        foo();

        return 0;
}

static void foo(void)
{
        printf("foo() is not visible otside the file.\n");
}
```

Функцията foo() може да се използва само във файла, в който е дефинирана.

Ако една функция ще се извиква от различни файлове, то всеки файл трябва да включва и прототипа на функцията преди точката, в която функцията се извиква. Ключовата дума extern не е задължителна, т.е. . следните две форми на прототипи са еквивалентни.

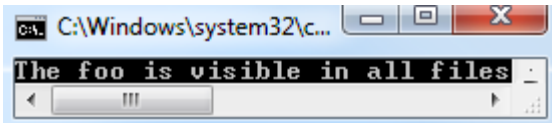
тип-резултат име-функция(списък-параметри);

extern тип-резултат име-функция(списък-параметри);

Пример 22-13:

```
/*сорс-файл1*/
extern void foo(void);

int main(void)
{
   foo();

   return 0;
}
```

```
/*сорс-файл2*/
#include <stdio.h>

void foo(void)
{
    printf("The foo is visible in all files\n");
}
```



Функцията foo() е дефинирана в сорс-файл2, но се извиква в сорс-файл1. Ето защо сорс-файл1 съдържа и прототипа на foo().

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

Въпроси за самопроверка

1. Как се дефинира функция?

2. Кои са съставните компоненти на една функция.

3. За какво служи прототипа?

4. Напишете функция, която не връща стойност, но има един параметър от тип double. Нека функцията отпечатва стойността на параметъра на екрана.

5. Какви са механизмите на подаване на аргументи на функция?

6. Какъв тип може да бъде **тип-резултат** на функцията?

7. Какъв тип могат да бъдат параметрите на функцията?

23 Предпроцесор

23.1 Въведение

Предпроцесорът, както показва неговото име, анализира кода преди неговата компилация. Предпроцесорът не прави синтактичен анализ на кода, а най-общо казано преобразува текста на кода в текст готов за компилация. За целта той използва предпроцесорни команди, наречени директиви.

23.2 Директива #include

Синтаксисът на тази директива е:

#include <име-файл>
#include "име-файл"

Резултатът от тази директива е, че съдържанието на посочения в директивата файл се включва в текущия файл по-същия начин както ако директно въведете това съдържание във файла. Обикновено скобите < > са указание за предпроцесора да търси файла в директорията, където е инсталиран компилатора. Тази форма се използва предимно за добавяне на библиотечни хедър-файлове. Формата с " " е указание за предпроцесора да търси файла първо в директория на проекта, и след това в директорията на компилатора. Тази форма се използва предимно за добавяне на потребителски хедър-файлове. Допълнително и двете форми позволяват да се зададе директно пътя към файла.

Всичко казано по-горе може да се илюстрира със следния пример:

Нека във файл с име myheader.h добавим следния код.

```
/* Декларация на enum-константи */
enum myenum {CAR, TRUCK = 70, TRACTOR};
```

Съхранете файла в директорията C:\PROJECTS\MVC++\C_training. Включете този файл в кода както е показано по-долу.

main.c

```
/* Добавяне на потребителски хедър-файлове */
#include "myheader.h"

/* Може директно да зададете пътя към файла           */
/* #include "C:\PROJECTS\MVC++\C_training\myheader.h" */

int main(void)
```

```
{
	/* Дефиниране на променлива */
	enum myenum en;

	/* Присвояване на стойност на променливата*/
	en = CAR;

	return 0;
}
```

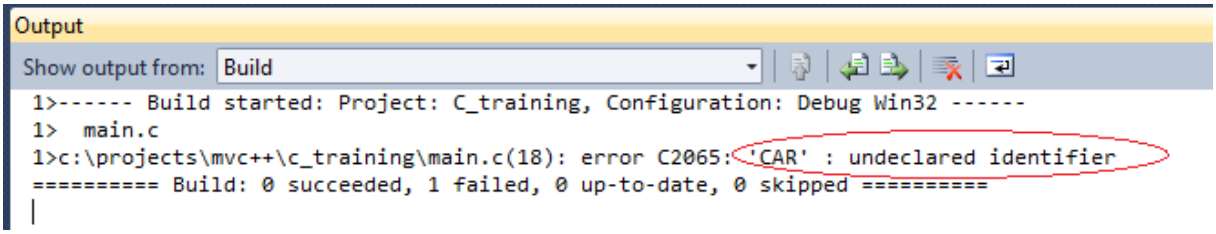
След като предпроцесорът обработи файла се получава следното[1]:

main.i

```
enum myenum {CAR, TRUCK = 70, TRACTOR};

int main(void)
{
	enum myenum en;

	en = CAR;

	return 0;
}
```

Забележка[1]: Предпроцесорът премахва и коментарите от кода.

Както се вижда, съдържанието на myheader.h се добавя във файла main.c. Може да пробвате да закоментирате реда `#include "myheader.h"` и да компилирате програмата. Компилаторът ще генерира следната грешка:



И това е логично, защото компилаторът няма информация за константата CAR и не знае какво е това.

23.3 Директива #define

Синтаксисът на тази директива е:

#define име-макрос заменящ-текст

Резултатът от тази директива е, че където в кода предпроцесорът срещне **име-макрос** то се заменя със **заменящ-текст**. Между **име-макрос** и **заменящ-текст** е необходимо да има поне един интервал.

Пример 23-1:

```
/* Добавяне на библиотечни хедър-файлове */
#include <stdio.h>

/* Дефиниране на макроси */
#define CAR         1
#define TRUCK       2
#define TRACTOR     3
#define message     "Hello World\n"

int main(void)
{
       /* Дефиниране на променливи */
       int i, j, k;

       /* Присвояване на стойност на променливите*/
       i = CAR;         /* Предпроцесорът заменя CAR с 1     */
       j = TRUCK;       /* Предпроцесорът заменя TRUCK с 2   */
       k = TRACTOR;     /* Предпроцесорът заменя TRACTOR с 3 */

       /* Отпечатване на съобщение на дисплея    */
       printf(message); /* Предпроцесорът заменя message с "Hello World\n" */

       /*Отпечатване на стойностите на екрана    */
       printf("%d\t%d\t%d\n", i, j, k);

       return 0;
}
```

Задаването на **име-макрос** следва същите правила както при задаване на имена на променливи, а **заменящ-текст** може да бъде произволен текст, стига след замяната на **име-макрос** в кода да се получава синтактически правилна конструкция както в горния пример. Горният пример показва още, че с #define също могат да се дефинират константи както с enum, с тази разлика, че с #define могат да се дефинират всякакви видове константи, а не само целочислени.

	**Избягвайте директното използване на константи в кода. Такива константи се наричат „магически числа“, тъй като значението им в програмата е неясно. Може да използвате директивата #define , за да дадете смислено име на всяка константа. Това подобрява разбираемостта на програмата.**

	**Прието е имената на макросите да се изписват само с главни букви. Това ги отличава от имена на променливи.**

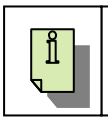
Директивата #define има и втора форма, която има и параметри. Синтаксисът на тази форма е:

#define име-макрос(списък-параметри) заместващ-текст

където

списък-параметри

Състои се от имена на параметри, разделени със запетая.



Макрос с параметри се нарича още макро-функция.

Параметрите от **списък-параметри** могат да се използват в **заместващ-текст**. При извикване на макроса в кода, в скобите след името се подават аргументи. Всяка поява на параметър в заместващия текст се заменя със съответния аргумент.

Пример 23-2:

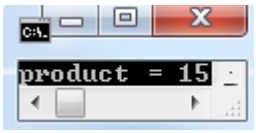
```
#include <stdio.h>

/* Дефиниране на макро-функция */
#define ADD(a,b)    a+b

int main(void)
{
       int sum;

       /* извикване на макро-функцията ADD с аргументи 3 и 4 */
       sum  = ADD(3,4); /* макросът ADD(3,4) се заменя с израза 3+4 */
       printf("sum = %d\n", sum);

       return 0;
}
```

Аргументите на макроса могат да бъдат и изрази. В този случай обаче е възможно действителният резултат да се различава от очакванията. Например разгледайте следния код:

Пример 23-3:

```
#include <stdio.h>

/* Дефиниране на макро-функция */
#define MUL(a,b)    a*b

int main(void)
{
```

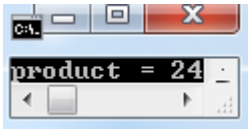
```
    int product;

    /* извикване на макро-функцията MUL с аргументи 3+3 и 4 */
    product  = MUL(3+3,4); /* макросът MUL(3+3,4) се заменя с израза 3+3*4 */
    printf("product = %d\n", product);

    return 0;
}
```



Резултатът от предходната програма е 15, а вероятно сте очаквали 24. За да се избегне този проблем, се препоръчва всички параметри в заместващия текст да се обграждат със скоби, включително и целия заместващ текст.

Пример 23-4:

```
#include <stdio.h>

/* Дефиниране на макро-функция */
#define MUL(a,b)   ((a)*(b))

int main(void)
{
   int product;

   /* извикване на макро-функцията MUL с аргументи 3+3 и 4 */
   product  = MUL(3+3,4); /* макросът MUL(3+3,4) се заменя с израза ((3+3)*(4)) */
   printf("product = %d\n", product);

   return 0;
}
```



Винаги обграждайте всички параметри в заместващия текст на макро-функция в скоби, включително и целия заместващ текст.

Препоръчително е името на макро-функциите да подсказва, че това е макро-функция, а не обикновена функция. Например може да използвате само главни букви или префикс m: mMacroFunctionName.

Декларацията на макрос може да се поставя навсякъде във файл например във функция, в блок с код, извън функция и т.н. Тя е видима от точката на деклариране до края на файла. За предпочитане е обаче декларациите на макросите да се поставят в началото на файла.

23.4 Директива #undef

Директивата #undef отдефинира макрос. Тя има следния синтаксис:

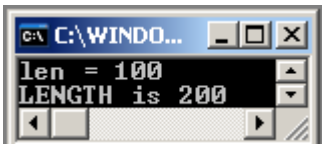
#undef име-макрос

Ако име-макрос съществува, след изпълнението на тази директива то ще бъде невалидно. Ако макрос с такова име не съществува, директивата просто се игнорира.

Директивата #undef позволява един макрос да бъде предефиниран с друго тяло.

Пример 23-5:

```
#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;
}
```



23.5 Директиви #ifdef-#else-#endif

Синтаксисът на тези директиви е:

```
#ifdef име-макрос
    програмен-текст
#else
    програмен-текст
#endif
```

Тези директиви работят по следния начин:

1. Предпроцесорът проверява дали **име-макрос** е дефинирано с

директивата #define.

2. Ако **име-макрос** е дефинирано с директивата #define, програмният текст, заключен между #ifdef и #else, се подава на компилатора, а програмният текст, заключен между #else и #endif, не се компилира.

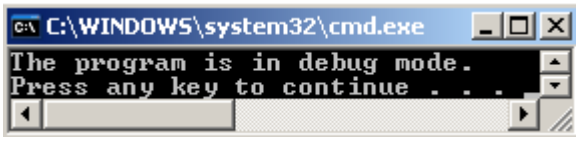
3. Ако **име-макрос** не е дефинирано с директивата #define или е било отдефинирано с #undef, програмният текст, заключен между #ifdef и #else, не се компилира, а програмният текст, заключен между #else и #endif, се подава на компилатора.

Директивата #else не е задължителна и може да се пропусне. Следната конструкция е напълно валидна:

```
#ifdef име-макрос
    програмен-текст
#endif
```

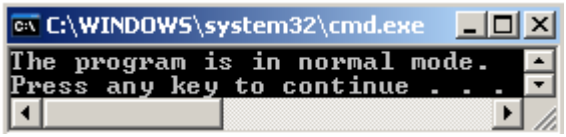
Пример 23-6:

```
#include <stdio.h>

#define DEBUG

int main(void)
{

#ifdef DEBUG
	printf("The program is in debug mode.\n");
#else
	printf("The program is in normal mode.\n");
#endif

	return 0;
}
```



Закоментирайте директивата #define както е показано по-долу, компилирайте и стартирайте програмата отново.

```
/* #define DEBUG */
```



23.6 Директиви #ifndef-#else-#endif

Синтаксисът на тези директиви е:

```
#ifndef име-макрос
    програмен-текст
#else
    програмен-текст
#endif
```

Тези директиви работят по следния начин:

1. Предпроцесорът проверява дали **име-макрос** е дефинирано с директивата #define.

2. Ако **име-макрос** не е дефинирано с директивата #define или е било отдефинирано с #undef, програмният текст, заключен между #ifndef и #else, се подава на компилатора, а програмният текст, заключен между #else и #endif, не се компилира.

3. Ако **име-макрос** е дефинирано с директивата #define, програмният текст, заключен между #ifndef и #else, не се компилира, а програмният текст, заключен между #else и #endif, се подава на компилатора.

Директивата #else не е задължителна и може да се пропусне. Следната конструкция е напълно валидна:

```
#ifndef име-макрос
    програмен-текст
#endif
```

23.7 Директиви #if-#else-#endif

Синтаксисът на тези директиви е:

```
#if управляващ-израз
    програмен-текст
#else
    програмен-текст
#endif
```

където

управляващ-израз

Трябва се изчислява до константна целочислена аритметична стойност.

Тези директиви работят по следния начин:

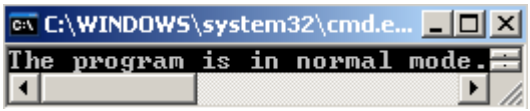
1. Изчислява се **управляващ-израз** на директивата #if.

2. Ако резултатът е ненулева стойност, програмният текст, заключен между #if и #esle, се подава на компилатора, а програмният текст, заключен между #else и #endif, не се компилира.

3. Ако резултатът е нула, програмният текст, заключен между #if и #else не се компилира, а програмният текст, заключен между #else и #endif, се подава на компилатора.

Пример 23-7:

```
#include <stdio.h>

#define    DEBUG    0

int main(void)
{

#if DEBUG /* Същото като #if DEBUG != 0 */
      printf("The program is in debug mode.\n");
#else
      printf("The program is in normal mode.\n");
#endif

      return 0;
}
```



Директивата #else не е задължителна и може да се пропусне. Следната конструкция е напълно валидна:

```
#if управляващ-израз
    програмен-текст
#endif
```

23.8 Предпроцесорен оператор defined

Предпроцесорният оператор defined е алтернатива на директивата #ifdef. Той също може да се използва, за да се определи дали дадено **име-макрос**

е дефинирано с директивата #define.

defined име-макрос

или

defined (име-макрос)

Ако **име-макрос** е дефинирано с #define, резултатът е 1, а ако не е дефинирано или е било отдефинирано с #undef, резултатът е 0.

Логическата операция ! също може да се използва в комбинация с defined.

!defined име-макрос

или

!defined (име-макрос)

Ако **име-макрос** не е дефинирано с #define или е било отдефинирано с #undef, резултатът е 1, а ако е дефинирано, резултатът е 0.

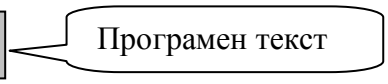
Предпроцесорният оператор defined може да се използва в управляващия израз на директивата #if.

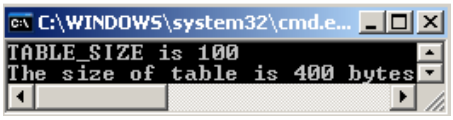
Следните двойки конструкции са еквивалентни:

```
#if defined (име-макрос)  ≈  #ifdef име-макрос

#if !defined (име-макрос) ≈  #ifndef име-макрос
```

Пример 23-8:

```
#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;
}
```





Предимството на оператора defined пред директивата #ifdef е, че с негова помощ могат да се проверят няколко макро-имена наведнъж. За целта резултатите от няколко оператора defined могат да се комбинират с логическите операции && и ||.

Пример 23-9:

```
#include <stdio.h>

#define MACRO1
#define MACRO2

#if defined (MACRO1) && defined (MACRO2)
#define TABLE_SIZE 100
#else
#define TABLE_SIZE 200
#endif

int table[TABLE_SIZE];

int main(void)
{
	/*...*/

	return 0;
}
```

Пример 23-10:

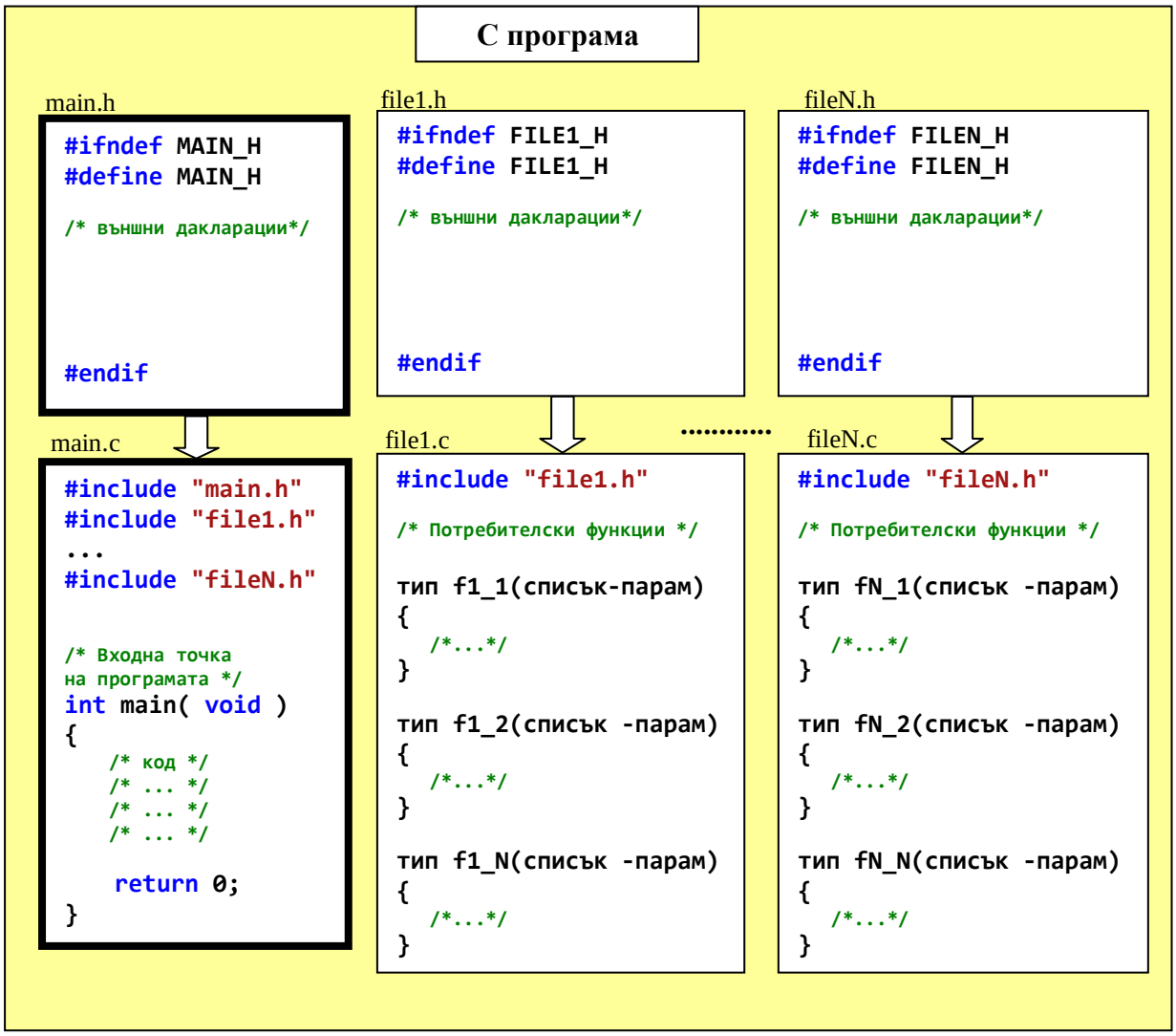
```
#include <stdio.h>

#define MACRO1 /* или #define MACRO2 */

#if defined (MACRO1) || defined (MACRO2)
#define TABLE_SIZE 100
#else
#define TABLE_SIZE 200
#endif

int table[TABLE_SIZE];

int main(void)
{
	/*...*/

	return 0;
}
```

Въпроси за самопроверка

1. Каква е ролята на предпроцесора ?
2. Какво прави конструкцията #include "file.h"
3. Покажете как се дефинира макрос?
4. Опишете как работят директивите #ifdef-#else-#endif?
5. Опишете как работят директивите #ifndef-#else-#endif?
6. Опишете как работят директивите #if-#else-#endif?
7. Опишете как работи предпроцесорният оператор defined?

24 Организация на многофайлова С програма

Повечето примери, разгледани досега, се състояха от един сорс-файл. Само малки и тривиални програми се състоят от един сорс-файл. На практика повечето С програми се състоят от няколко сорс-файла. Фиг.84 показва типичната организация на С програма, състояща се от няколко сорс-файла.



Фиг. 84 Типична организация на С програма

За всеки сорс-файл се създава хедър-файл със същото име, който се включва в сорс-файла с директивата #include. В този хедър-файл се добавят външни декларации. Под външни декларации се има предвид декларации на глобални променливи, макроси, прототипи на функции, потребителски типове и т.н., които ще се използват и от други сорс-файлове. Всички декларации на променливи, макроси и функции и т.н., които се използват само в сорс-файла, се добавят в самия сорс-файл. Всеки сорс-файл, който използва функции, глобални променливи, макроси и т.н., дефинирани в друг сорс-файл, включва съответния хедър-файл с директивата #include.

Обърнете внимание как съдържанието на хедър-файловете е заградено в конструкция от вида:

```
#ifndef   FILENAME_H
#define   FILENAME_H
/* Съдържанието на хедър-файла се поставя  тук  */
#endif
```

Предназначението на тази конструкция е да предпази хедър-файла от многократно включване в един и същ сорс-файл. При първото му включване в сорс-файла с директивата #include, макросът FILENAME_H се дефинира и съдържанието се подава на компилатора. Ако по-късно същият хедър-файл е включен отново с директивата #include (косвено или директно), неговото съдържание ще бъде игнорирано от компилатора, тъй като макросът FILENAME_H е бил дефиниран по-рано. Ако пропуснете да защитите хедър-файл по този начин и го добавите повече от веднъж в един и същ сорс-файл, компилаторът може да генерира грешка, ако срещне декларации, които не е позволено да се появяват повече от веднъж в същия сорс-файл. Такива са например декларациите на структура, обединение, изброяване и т.н.

Пример 24-1:

color.h

main.h | color.h | color.c | main.c

(Global Scope)

```
1  #ifndef COLOR_H
2  #define COLOR_H
3
4  typedef enum  {RED_MAX = 255, GREEN_MAX = 255, BLUE_MAX = 255}rgb;
5
6  extern int set_color(rgb r, rgb g, rgb b);
7
8  #endif
9
```

color.c

main.h | color.h | color.c | main.c

(Global Scope)

```
1  #include "color.h"
2
3  int set_color(rgb r, rgb g, rgb b)
4  {
5      int  ResColor;
6
7      ResColor = r + g + b;
8
9      return ResColor;
10 }
11
```

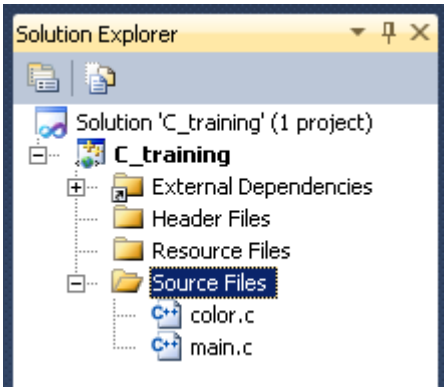
main.h

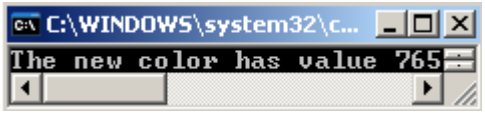
```
main.h*  color.h  color.c  main.c
(Global Scope)
1  #ifndef MAIN_H
2  #define MAIN_H
3
4  /*...*/
5
6  #endif
7
```

main.c

```
main.h  color.h  color.c  main.c
(Global Scope)
1  #include <stdio.h>
2  #include "main.h"
3  #include "color.h"
4
5  int main( void )
6  {
7      printf("The new color has value %d\n", set_color(RED_MAX, GREEN_MAX, BLUE_MAX));
8
9      return 0;
10 }
11
```

Добавяне на color.c заедно с main.c към проекта:





Файловете main.c, main.h (в случая този файл е празен), color.c и color.h са съхранени в C:\projects\MVC++\C_training. Файлът color.c дефинира функция get_color(), която приема три аргумента тип rgb. Типът rgb е деклариран в хедър-файла color.h, заедно с прототипа на функцията. Функцията get_color() се извиква в main.c с аргументи enum-константите, декларирани също в color.h. Ето защо main.c включва и хедър-файла color.h.

Освен ако е много кратка, винаги разбивайте програмата на модули. Всеки модул може да се състои от един или повече логически свързани сорс-файлове. Това позволява всеки модул да се разработва, тества и документира поотделно и от различни програмисти. Това позволява също и преизползваемост на модулите в други проекти без или с малки промени.

Следващите фигури описват примерна организация на .c и .h файлове.

```
/* 1. Описание на файла: име, автор, версия и т.н.   */
...
/* 2. Системни хедър-файлове                         */
#include <SystemHeader.h>
...
/* 3. Потребителски хедър-файлове                    */
#include "UserHeader.h"
...
/* 4. Дефиниции на вътрешни макроси                  */
#define  име-макрос заменящ-текст
...
/* 5. Декларации на вътрешни потребителски типове    */
struct  име-структура  {членове};
...
union   име-обединение {членове};
...
enum    име-обединение {членове};
...
typedef тип ново-име-на-тип;
...
/* 6. Дефиниции на глобални външни променливи        */
тип име-променлива;
...
/* 7. Дефиниции на глобални статични променливи      */
static тип име-променлива;
...
/* 8. Прототипи на статични функции                  */
static тип име-функция(списък-параметри);
...
/* 9. Дефиниции на глобални и статични функции       */
тип име-функция(списък-параметри)
{

}
...
static тип име-функция(списък-параметри)
{

}
...
```

Фиг. 85 Примерна структура на .c файл

```
/* 1. Описание на файла: име, автор, версия и т.н.   */
...
/* 2. Системни хедър-файлове                         */
#include <SystemHeader.h>
...
/* 3. Потребителски хедър-файлове                    */
#include "UserHeader.h"
...
/* 4. Дефиниции на външни макроси                    */
#define  име-макрос заменящ-текст
...
/* 5. Декларации на външни потребителски типове      */
struct  име-структура  {членове};
...
union   име-обединение {членове};
...
enum    име-обединение {членове};
...
typedef тип ново-име-на-тип;
...
/* 6. Декларации на глобални външни променливи       */
extern тип име-променлива;
...
/* 7. Прототипи на външни функции                    */
extern тип име-функция(списък-параметри);
...
```

Фиг. 86 Примерна структура на .h файл

25 Сглобяване на всичко заедно

В тази глава ще Ви представя една съвсем проста и груба имитация на електронна кодова брава. Целта е да добиете по-ясна представа за практическата организация на една C програма. Фиг.87 представя примерен вид на нашата електронна кодова брава. Тя се състои от течно кристален дисплей (LCD) и клавиатура. Ще използваме монитора за имитация на LCD дисплея.

Фиг. 87 Електронна кодова брава

Програмата се състои от следните модули:

- lcd

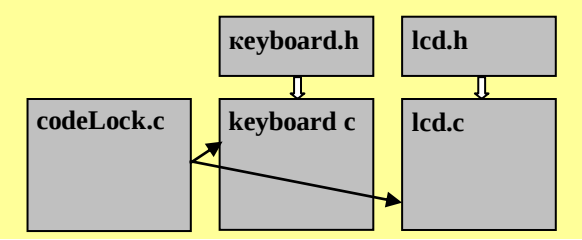
Съдържа средства за управление на LCD дисплей. Модул lcd се състои от един copc-файл lcd.c.

- keyboard

Съдържа средства за управление на клавиатурата. Модул keyboard се състои от един copc-файл keyboard.c.

- codeLock

Използва функциите, дефинирани в горните два модула и имплементира логиката на електронната кодова брава. Модул codeLock се състои от един copc-файл codeLock.c. Фиг.88 показва структурата на програмата.



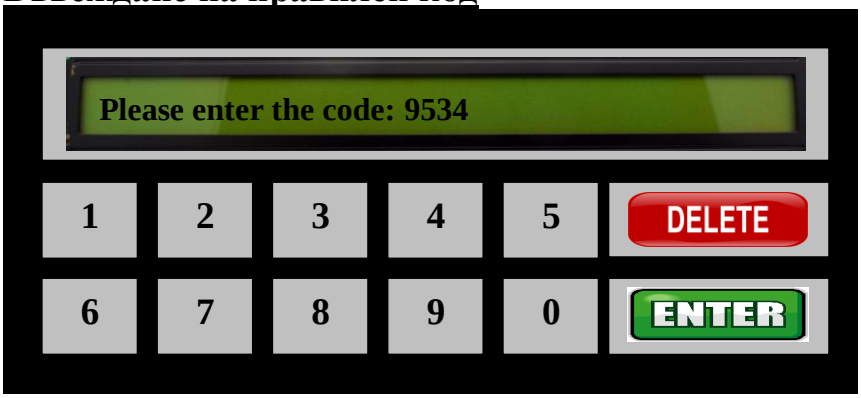
Фиг. 88 Структура на програмата

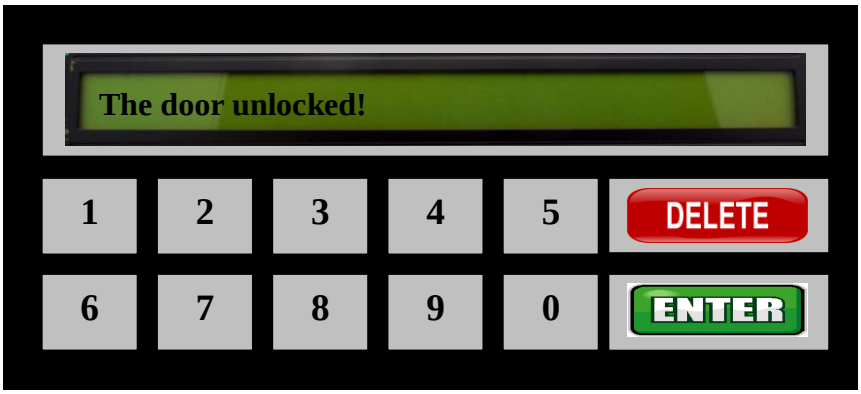
Програмата симулира следното поведение:

Включване на устройството

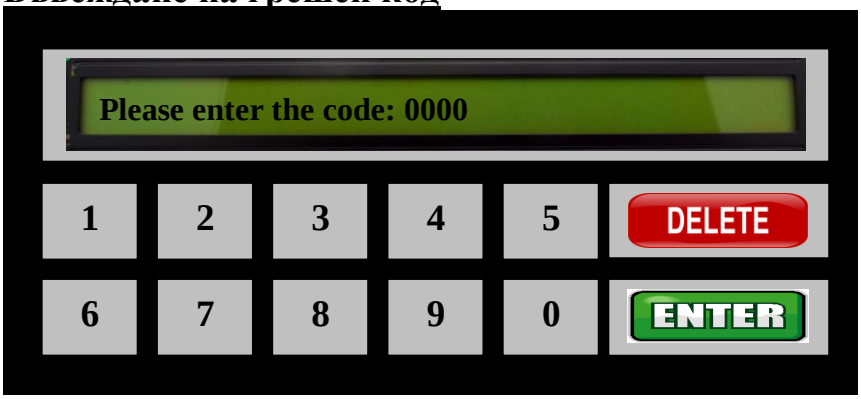
Работно състояние на устройството

Въвеждане на правилен код

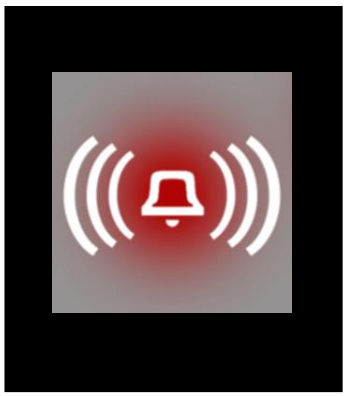




Въвеждане на грешен код



Въвеждане на 3 поредни грешни кода



Следва кода на програмата с описание в коментарите.

CodeLock.c

```
/* 1. Описание на файла: име, автор, версия и т.н.   */
/*Име: main                                          */
/*Автор: Д. Петков                                   */
/*Версия: 1.0                                        */
/*Описание: Кодова брава                             */

/* 2. Системни хедър-файлове                         */
#include <stdio.h>

/* 3. Потребителски хедър-файлове                    */
#include "stdtypes.h"
#include "lcd.h"
#include "Keyboard.h"

/* 4. Дефиниции на вътрешни макроси                  */
#define DELAY        1000000000
#define ALARM        "\a\a\a\a\a\a\a\a\a"
#define ATTEMPTS     3
#define CODE         9534

/* 5. Декларации на вътрешни потребителски типове    */

/* 6. Дефиниции на глобални външни променливи        */

/* 7. Дефиниции на глобални статични променливи      */

/* 8. Прототипи на статични функции                  */
static void    delay(uint32 u32delay);
static boolean validateCode(const uint32 u32code);
static void    startAlarm(void);

/* 9.Дефиниции на глобални и статични функции       */

/*****************************************************/
/* Име: main                                         */
/* Описание: Входна точка на програмата              */
/* Параметри:                                         */
/* Резултат:                                         */
/*****************************************************/
int main(void)
{
  uint32  u32Code;             /* Съхранява прочетения от клавиатурата код    */
  boolean codeStatus;          /* Съдържа статуса на въведения код            */
  uint8   attemptsCounter = 0; /* Брояч на опитите на въвеждане на грешен код */

  initLcd();      /* Конфигуриране на дисплея      */
  delay(DELAY);   /* Изчакване                     */
  clearLcd();     /* Изчистване на дисплея         */
  initKeyboard(); /* Конфигуриране на клавиатурата */
  delay(DELAY);   /* Изчакване                     */
  clearLcd();     /* Изчистване на дисплея         */

  while(ATTEMPTS != attemptsCounter) /* Проверка на броя грешно въведени кодове */
  {
      sendMessageToLcd("Please enter the code: "); /* Подканване за въвеждане на
                                                      кода */
      u32Code = readKeyboard();           /* Прочитане на кода               */
      clearLcd();                         /* Изчистване на дисплея           */
      codeStatus = validateCode(u32Code); /* Проверка на въведения код       */

      if(STD_TRUE == codeStatus) /* Проверка за валидност на кода */
```

```
        {
                sendMessageToLcd("The door unlocked!"); /* Кодът е приет, вратата е
                                                           отключена */
                delay(DELAY);         /* Изчакване                    */
                clearLcd();           /* Изчистване на дисплея        */
                attemptsCounter = 0; /* Нулиране на брояча на опитите */
        }
        else
        {
                sendMessageToLcd("Wrong code!"); /* Кодът е отхвърлен */
                delay(DELAY);       /* Изчакване                     */
                clearLcd();         /* Изчистване на дисплея         */
                attemptsCounter++; /* Увеличаване на брояча на опитите */
        }
    }

    sendMessageToLcd("I'm calling the police!!!\n"); /* Предупредително съобщение */
    startAlarm(); /* Включване на алармата */

    return 0;
}

/*****************************************************/
/* Име: delay                                        */
/* Описание: осигурява забавяне между съобщенията    */
/* Параметри: uint32 u32delay                         */
/* Резултат: void                                    */
/*****************************************************/
static void delay(uint32 u32delay)
{
    uint32 i32;

    for(i32 = 0 ; i32 < u32delay; i32++);
}

/*****************************************************/
/* Име: validateCode                                 */
/* Описание: Проверка на въведения код               */
/* Параметри: uint32 u32code                          */
/* Резултат: Boolean                                 */
/*           TRUE  - кодът е верен                   */
/*           FALSE - кодът е грешен                  */
/*****************************************************/
static boolean validateCode(const uint32 u32code)
{
        return (CODE == u32code) ? STD_TRUE : STD_FALSE;
}

/*****************************************************/
/* Име: startAlarm                                   */
/* Описание: Включване на алармата                   */
/* Параметри: void                                   */
/* Резултат: void                                    */
/*****************************************************/
static void startAlarm(void)
{
        printf(ALARM); /* Звукова сигнализация */
}
```

lcd.h

```
#ifndef LCD_H
#define LCD_H
```

```
/* 1. Описание на файла: име, автор, версия и т.н.  */

/* 2. Системни хедър-файлове                        */

/* 3. Потребителски хедър-файлове                   */

/* 4. Дефиниции на външни макроси                   */

/* 5. Декларации на външни потребителски типове     */

/* 6. Декларации на глобални външни променливи      */

/* 7. Прототипи на външни функции                   */
extern void initLcd(void);
extern void sendMessageToLcd(const char* message);
extern void clearLcd(void);

#endif
```

lcd.c

```
/* 1. Описание на файла: име, автор, версия и т.н.  */
/*Име: lcd                                          */
/*Автор: Д. Петков                                  */
/*Версия: 1.0                                       */
/*Описание: Функции за управление на LCD дисплея    */

/* 2. Системни хедър-файлове                        */
#include <stdio.h>

/* 3. Потребителски хедър-файлове                   */
#include "stdtypes.h"
#include "lcd.h"

/* 4. Дефиниции на вътрешни макроси                 */

/* 5. Декларации на вътрешни потребителски типове   */

/* 6. Дефиниции на глобални външни променливи       */

/* 7. Дефиниции на глобални статични променливи     */

/* 8. Прототипи на статични функции                 */

/* 9. Дефиниции на глобални и статични функции      */

/****************************************************/
/* Име: initLcd                                     */
/* Описание: Конфигуриране на дисплея               */
/* Параметри: void                                   */
/* Резултат: void                                   */
/****************************************************/
void initLcd(void)
{
      printf("LCD is initializing...");
}

/****************************************************/
/* Име: sendMessageToLcd                            */
/* Описание: Изпращане на съобщение към дисплея     */
/* Параметри: const char* message                    */
/* Резултат: void                                   */
/****************************************************/
```

```
void sendMessageToLcd(const char* message)
{
	printf("%s", message);
}

/*************************************************/
/* Име: clearLcd                                 */
/* Описание: Изчистване на дисплея               */
/* Параметри: void                                */
/* Резултат: void                                */
/*************************************************/
void clearLcd(void)
{
	system("cls"); /* Изчиства конзолния прозорец */
}
```

keyboard.h

```
#ifndef KEYBOARD_H
#define CODELOCK_H

/* 1. Описание на файла: име, автор, версия и т.н.  */

/* 2. Системни хедър-файлове                        */

/* 3. Потребителски хедър-файлове                   */

/* 4. Дефиниции на външни макроси                   */

/* 5. Декларации на външни потребителски типове     */

/* 6. Декларации на глобални външни променливи      */

/* 7. Прототипи на външни функции                   */
extern void   initKeyboard(void);
extern uint32 readKeyboard(void);

#endif
```

keyboard.c

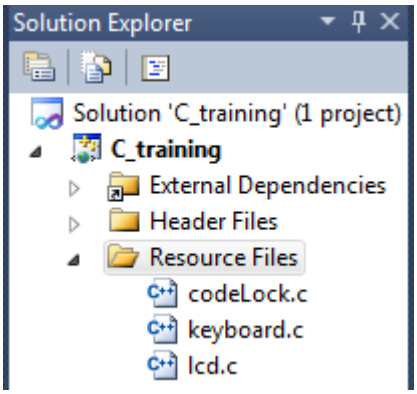
```
/* 1. Описание на файла: име, автор, версия и т.н.  */
/*Име: keyboard                                     */
/*Автор: Д. Петков                                  */
/*Версия: 1.0                                       */
/*Описание: Функции за управление на клавиатурата   */

/* 2. Системни хедър-файлове                        */
#include <stdio.h>

/* 3. Потребителски хедър-файлове                   */
#include "stdtypes.h"
#include "Keyboard.h"

/* 4. Дефиниции на вътрешни макроси                 */

/* 5. Декларации на вътрешни потребителски типове   */

/* 6. Дефиниции на глобални външни променливи       */

/* 7. Дефиниции на глобални статични променливи     */
/* 8. Прототипи на статични функции                 */
```

```
/* 9. Дефиниции на глобални и статични функции      */

/****************************************************/
/* Име: initKeyboard                                */
/* Описание: Конфигуриране на клавиатурата          */
/* Параметри: void                                  */
/* Резултат: void                                   */
/****************************************************/
void initKeyboard(void)
{
	printf("KBD is initializing...");
}

/****************************************************/
/* Име: readKeyboard                                */
/* Описание: Прочитане на въведения код             */
/* Параметри: void                                  */
/* Резултат: void                                   */
/****************************************************/
uint32 readKeyboard(void)
{
	uint32  u32Code;

	scanf("%ul", &u32Code);

	return u32Code;
}
```

stdtypes.h

```
#ifndef STDTYPES_H
#define STDTYPES_H

#define STD_FALSE   0
#define STD_TRUE    1

typedef      unsigned char    boolean;

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;

#endif
```

Съхранете тези файлове в директорията C:\projects\MVC++\C_training и добавете .c файловете към проекта както е показано на Фиг.89. Компилирайте и стартирайте програмата (Ctr+F5).



Фиг. 89 Проект-електронна кодова брава

Част III: C99

26 Общи сведения

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 изпълним код;
- добавяне на нови целочислени типове;
- добавяне на булев тип;
- назначени инициализатори (**designated initializer**);
- съставни литерали (**compound literal**);
- масиви с променлива дължина (**VLA-масиви**);
- вложени-функции (**inline-функции**);
- добавяне на нови хедър-файлове и библиотечни функции;
- други.

Тази част разглежда по-важните и интересни изменения в C99. Подробно описание на всички нововъведения ще намерите в книгата **ANSI C. Пълен справочник**.

Не забравяйте да компилирате следващите примери със средата на Pelles (вижте 7.3 Pelles C for Windows).

27 Ключови думи в C99

C99 дефинира 37 ключови думи (Табл.45).

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. 45 Ключови думи в C99

Горният индекс C99 показва ключовите думи добавени в C99.

28 Нововъведения в C99

28.1 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 28-1:

```
//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;
}
```

28.2 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 28-2:

```
#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 28-3:

```
#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;
}
```

Област на видимост на x

28.3 Нови целочислени типове

Към целочислените типове в C89, C99 добавя два нови: **unsigned 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.46 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. 46 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.

28.4 Булев тип

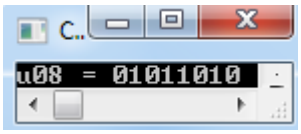
C99 въвежда беззнаков целочислен тип **_Bool**, който може да съдържа само две стойности: 0 и 1. За удобство хедър-файла <stdbool.h> дефинира макроса **bool**, който е синоним на **_Bool** и макросите **false** и **true**, които имат стойност 0 и 1 съответно.

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 преобразува в тип _Bool, резултатът е 0, ако стойността на указателя е NULL, иначе резултатът е 1.

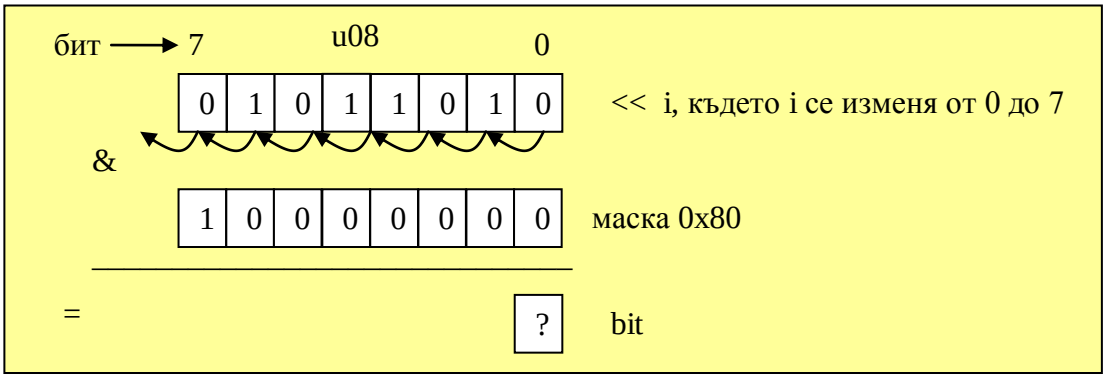
Следващият пример отпечатва на екрана битовото представяне на 8-битова променлива.

Пример 28-4:

```
#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.90 илюстрира Пример 28-4. За простота на илюстрацията не са отчетени автоматичните преобразувания, дължащи се на целочисленото повишение на типовете.



Фиг. 90 Илюстрация към Пример 28-4

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.

28.5 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.

28.5.1 Назначен инициализатор на масив

Назначеният инициализатор на масив има следния синтаксис:

[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.

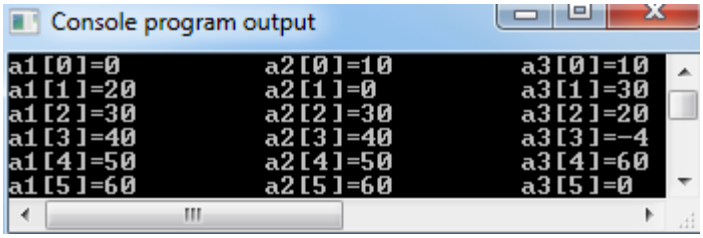
```
                                        /* индекс -> 0   1   2   3   4  5 */
int a3[6] = {10, 30, 40, 50, 60, [2] = 20, -4}; /* 10, 30, 20, -4, 60, 0 */
```

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 28-5:

```
#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;
}
```



28.5.2 Назначен инициализатор на структурна-променлива

Назначеният инициализатор на структурна-променлива има следния синтаксис:

.име-член = инициализатор

където

име-член
: Име на член на структурата.

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:

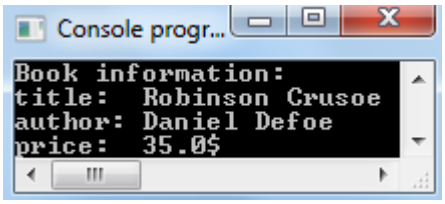
```
struct book {char title[40]; char author[40]; double price;};

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 28-6:

```
#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;
}
```



28.5.3 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  */
```

28.6 Съставни литерали

Съставните литерали Ви позволяват да дефинирате безименни инициализирани обекти (променливи) от всеки обектен тип, използвайки следния синтаксис:

(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 литерал-структура от тип struct ms и
                                  инициализира 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 28-7:

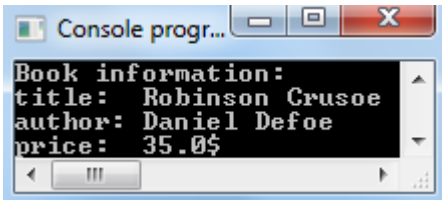
```
#include <stdio.h>

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 28-8:

```
#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);
}
```

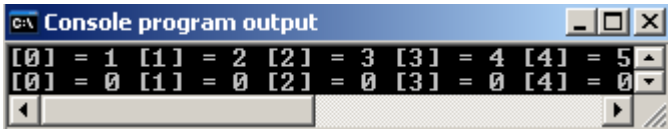


Може да прилагате квалификатора const към типа на съставния литерал.

Пример: `(const int []){1,2,3,4,5};`

Пример 28-9:

```
#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');
}
```



28.7 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-масивите имат следните ограничения:

- Могат да бъдат само локални auto-масиви. Глобалните и статичните масиви не могат да бъдат VLA-масиви.

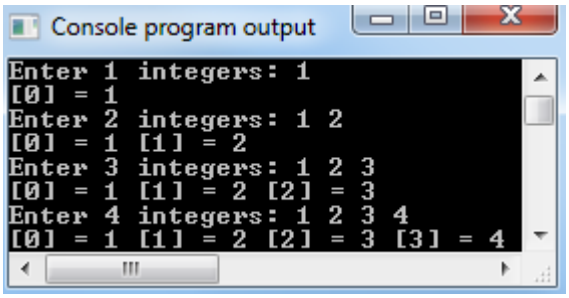
- Членове на структури и обединения не могат да бъдат VLA-масиви.

- 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-масивът не може да променя дължината си.

Пример 28-10:

```
#include <stdio.h>

void foo(int n);
void print_array(const int* p, int size);

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');
}
```



VLA-масивите могат да се използват и като параметри на функции. 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 */
```

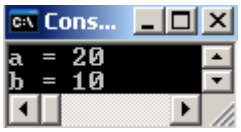
28.8 Вложени функции

Извикването на обикновена функция принуждава компютъра да запомни адреса на следващата инструкция, 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 дефиницията и се добави спецификаторът **inline**. Ключовата дума inline е заявка към компилатора да вмъкне кода на функцията в мястото на извикването 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 в мястото на извикване, е необходимо дефиницията на функцията да бъде разположена преди нейното извикване.

Пример 28-11:

```
#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;
}
```

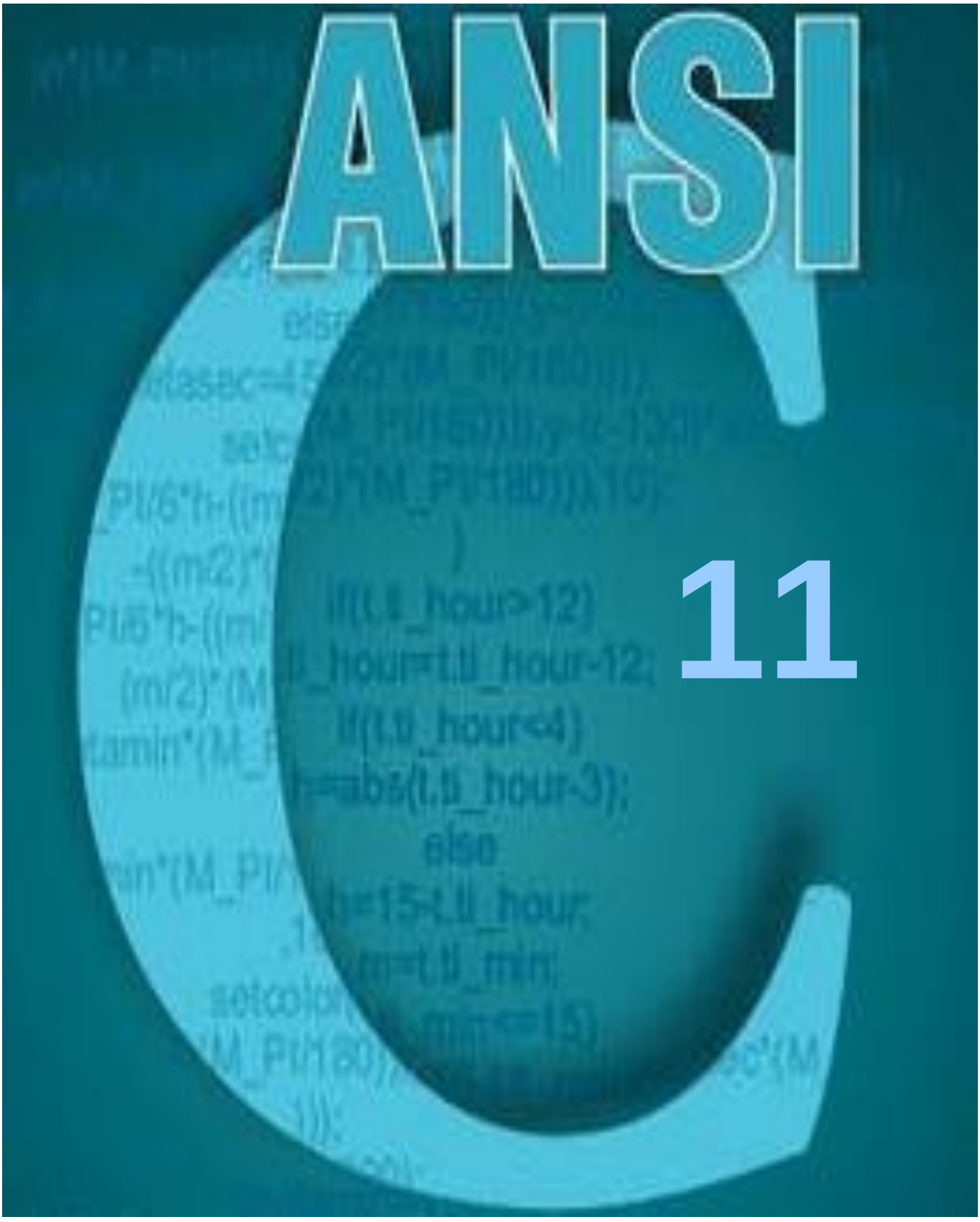


Ако компилаторът приеме заявката, тялото на функцията swapf() ще бъде вмъкнато в мястото на извикването както е показано по-долу.

```
#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 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;
}
```

Ако една вложена функция ще се извиква от различни сорс-файлове, всеки сорс-файл трябва да вижда дефиницията, за да може да вмъкне кода. За целта дефиницията на вложената функция се поставя в хедър-файл, който се добавя във всеки сорс-файл, който я използва.

Част IV: C11



29 Общи сведения

C11 е текущият 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 твърде нов по време на писане на тази книга. Едва няколко компилатора го поддържат.

C11 включва следните промени по отношение на C99:

- 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**);
- други.

Тази част прави преглед на C11. Спецификацията на стандарта C11 (на английски език) може да бъде изтеглена от показания линк:

http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf

За съжаление, повечето подобни спецификации се пишат на академичен език. Спецификацията на стандарта C11 не е изключение. Описанията на нововъведенията в нея са доста кратки и неясни, а по време на писане на тази книга има само няколко книги от чуждестранни автори, които разглеждат съвсем бегло C11. Не се тревожете, ако обясненията в следващите няколко точки не са Ви съвсем ясни. Вероятно ще минат няколко години, докато производителите на C компилатори за микроконтролери добавят поддръжка на C11. Бъдещето ще покаже до каква степен този стандарт ще бъде възприет.

Не забравяйте да компилирате следващите примери със средата на Pelles (вижте 7.3 Pelles C for Windows).

30 Ключови думи в C11

C11 дефинира 44 ключови думи (Табл.47).

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. 47 Ключови думи в C11

Горният индекс C11 показва ключовите думи добавени в C11.

31 Преглед на C11

31.1 Спецификатор _Atomic(тип)

Спецификаторът _Atomic се използва за дефиниране на атомични (неделими) типове. Той има следния синтаксис:

_Atomic(тип)

където

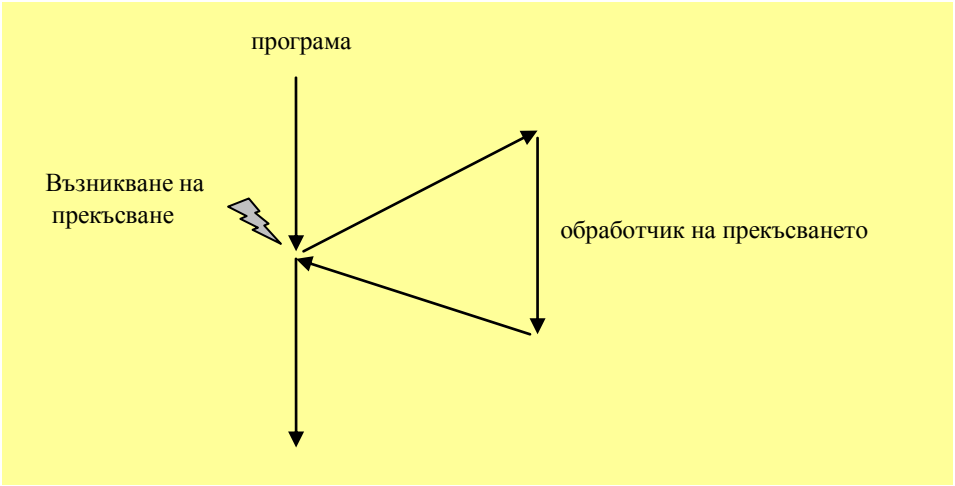
тип

Може да бъде всеки тип с изключение на масив, функция, атомичен тип или квалифициран тип.

Пример: `_Atomic(int) counter;`

За да разберете какво се има предвид под "неделим тип", ще Ви припомня как един 8-битов компютър например осъществява достъп (запис или четене) до една многобайтова променлива в паметта. В глава 9.7 Подравняване на променливите в паметта Ви обясних, че за трансфер на многобайтови данни, като тип short, int, long, float и double, на 8-битовия компютър са необходими толкова на брой операции (машинни инструкции), колкото е размерът в байтове на данните. Ако по средата на достъпа до такава променлива възникне прекъсване[1], то променливата ще бъде записана/прочетена частично.

Забележка[1]: Прекъсването е събитие, което принуждава процесорът да спре текущото изпълнение на програмата и да обработи това събитие. За целта се изпълнява подпрограма, която се нарича обработчик на прекъсването. След завършване на подпрограмата за обработката на прекъсването, процесорът продължава изпълнението на програмата от точката, в която е възникнало то.



Фиг. 91 Обработка на прекъсване

Източниците на прекъсване могат да бъдат най-различни например кликане с мишката, натискане на клавиш на клавиатурата и т.н. Всеки източник на прекъсване има и свой обработчик.

Подобна ситуация може да доведе до логическа грешка, ако преди възникването на прекъсването програмата например е записвала някаква стойност в променливата, а подпрограмата за обработка на прекъсването чете стойността на същата тази променлива. Ако дефинирате такава променлива със спецификатора _Atomic(), това гарантира, че прекъсването няма да бъде обработено, докато записа в променливата не завърши, т.е. . достъпът до такава променлива е неделим. Ако Ви е по-лесно, може да си представите, че достъпът до неделима променлива става с една операция.

Обобщено можем да кажем, че един тип се явява делим, ако е по-голям от машинната дума на компютъра. За достъп до променлива от такъв тип са необходими повече от една машинна инструкция.

31.2 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.

31.3 Оператор _Alignof

Операторът _Alignof връща модула на подравняване на своя операнд. Той има следния синтаксис:

_Alignof(тип)

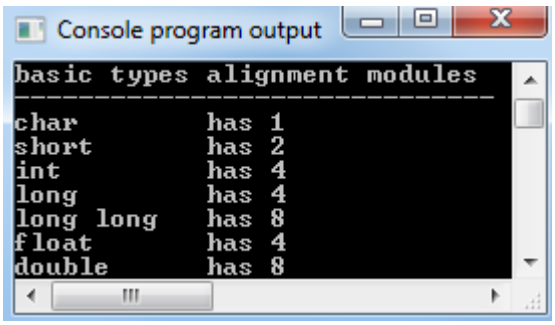
Ако сте забравили какво е подравняване, вижте отново 9.7 Подравняване на променливите в паметта. Модулът на подравняване на една променлива определя кратността на адресите, на които обектът може да се разполага и винаги е цяло неотрицателно число 2^n, където n = 0, 1, 2, 3 и т.н. Например

променлива с модул на подравняване 4 означава, че тази променлива може да се разполага само на адреси кратни на 4.

Следващият пример показва модулите на подравняване на базовите типове.

Пример 31-1:

```
#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;
}
```



31.4 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:

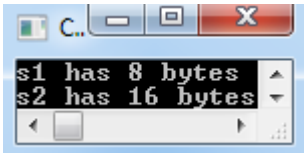
```
_Alignas(3) short x; /* Неправилно */
_Alignas(4) short y; /* Правилно   */
```

Следващият пример демонстрира влиянието на _Alignas върху размера памет, заеман от структура.

Пример 31-2:

```
/* 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.

31.5 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.

31.6 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
```

31.7 Многопоточност

31.7.1 Въведение

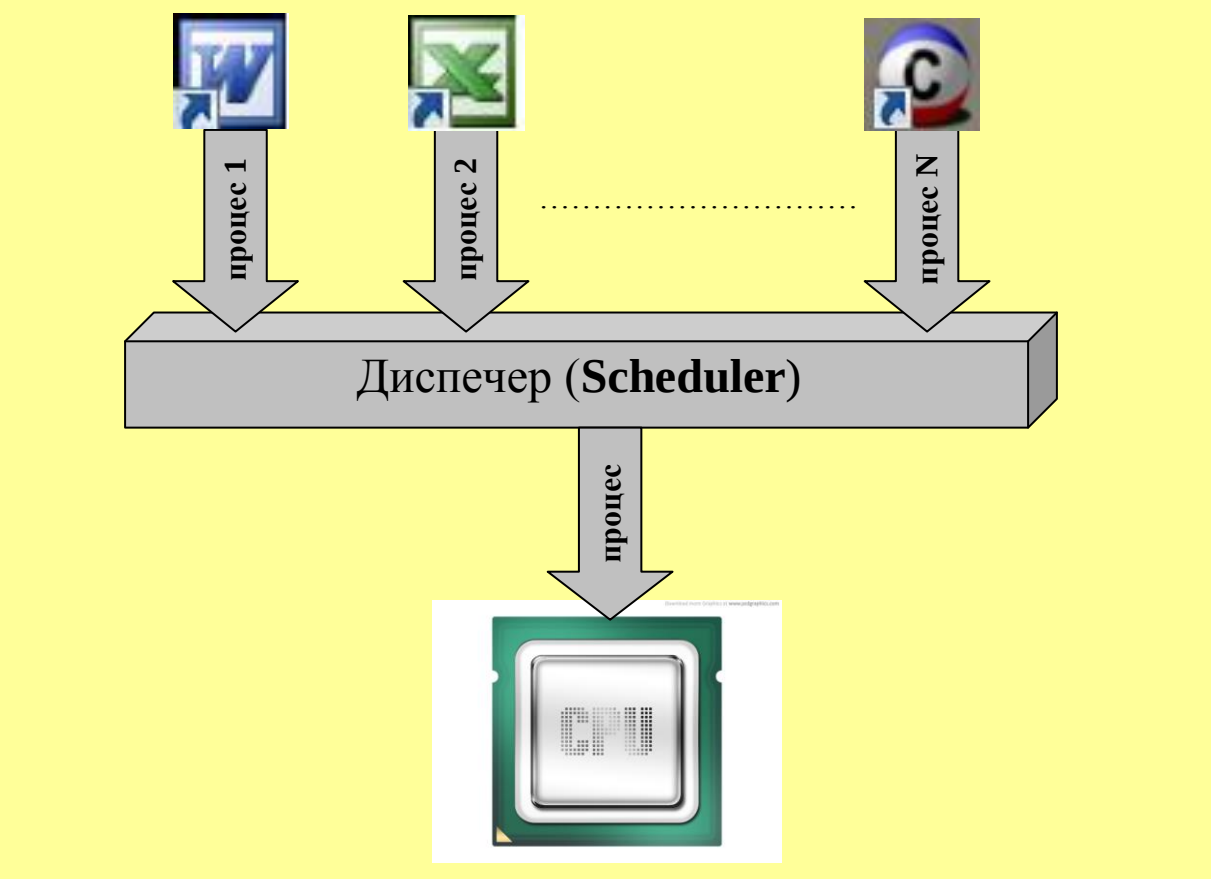
Поддръжката на многопоточно програмиране (**multithreaded programming**) е може би най-интересното и съществено въведение в C11. Хедър-файлът <thread.h> декларира макроси, типове, enum-константи и функции за поддръжка на многопоточно програмиране. Преди да разгледаме как се създават многопоточни програми в C11, ще Ви запозная с някои базови понятия и принципи. Пълното описание на многопоточността излиза извън рамките на тази книга. Ако решите да навлезете по-дълбоко в тази материя, може да потърсите някоя книга по операционни системи.

31.7.2 Многозадачност

Многозадачност (**multitasking**) е способност да извършваме много задачи едновременно. Следващите две забавни картинки описват това нагледно.

Фиг. 92 Примери за многозадачност

В компютърната техника многозадачност означава изпълнение на няколко програми едновременно. Когато стартирате една програма за изпълнение, Вие всъщност стартирате процес (**process**). В еднопроцесорните компютърни системи само един процес може да се изпълнява в даден момент от времето. Многозадачността се постига чрез превключване на процесора тук към един процес, ту към друг. Това превключване става толкова бързо, че за потребителя изглежда, че процесите (програмите) се изпълняват едновременно (Фиг.93).



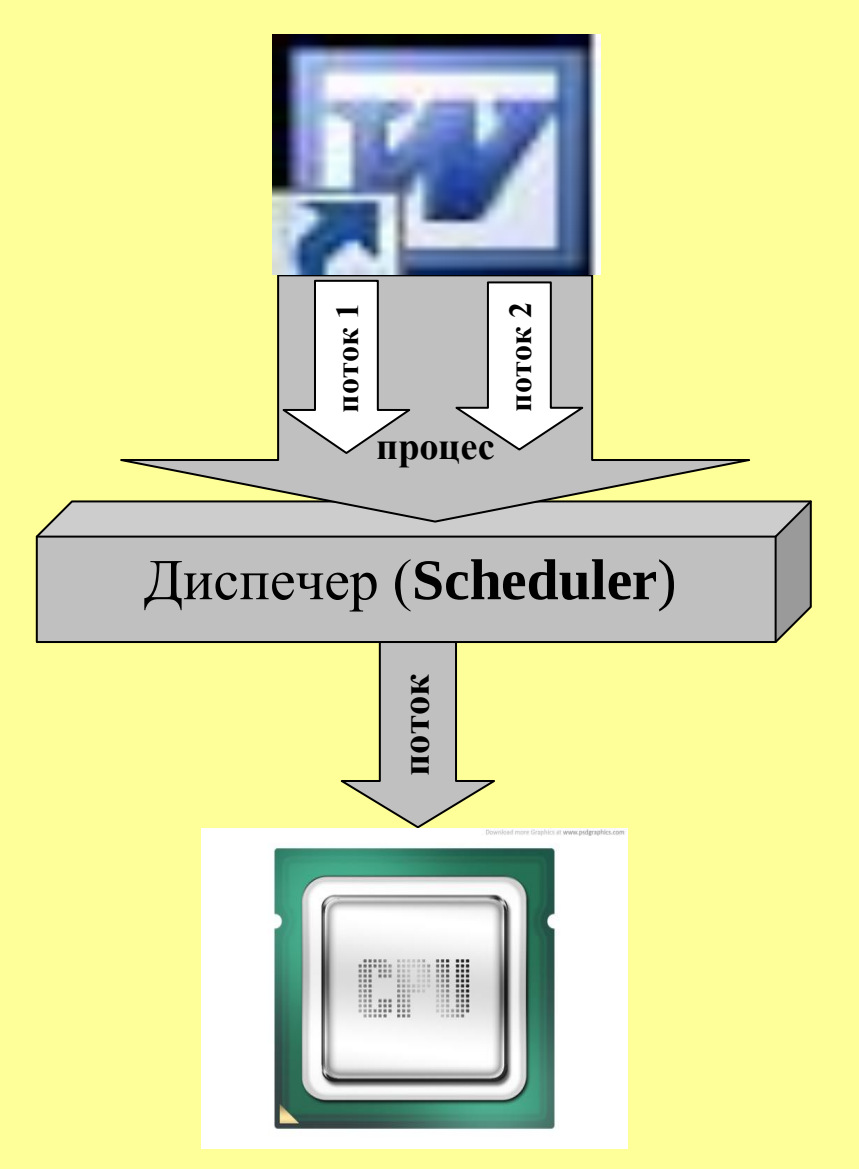
Фиг. 93 Многозадачност в компютърните системи

Диспечерът е част от операционната система, която се грижи за превключване на процесора към изпълнение на даден процес. Истински паралелизъм се получава, ако стартирате няколко програми на многопроцесорна (многоядрена) компютърна система.

31.7.3 Многопоточност

Многопоточността (**multithreading**) разширява идеята на многозадачността и позволява една програма да се състои от една или повече части, които могат да се изпълняват едновременно. Всяка част на програмата се нарича поток (**thread**). При многопоточните програми диспечерът на операционната система се грижи не само за превключване

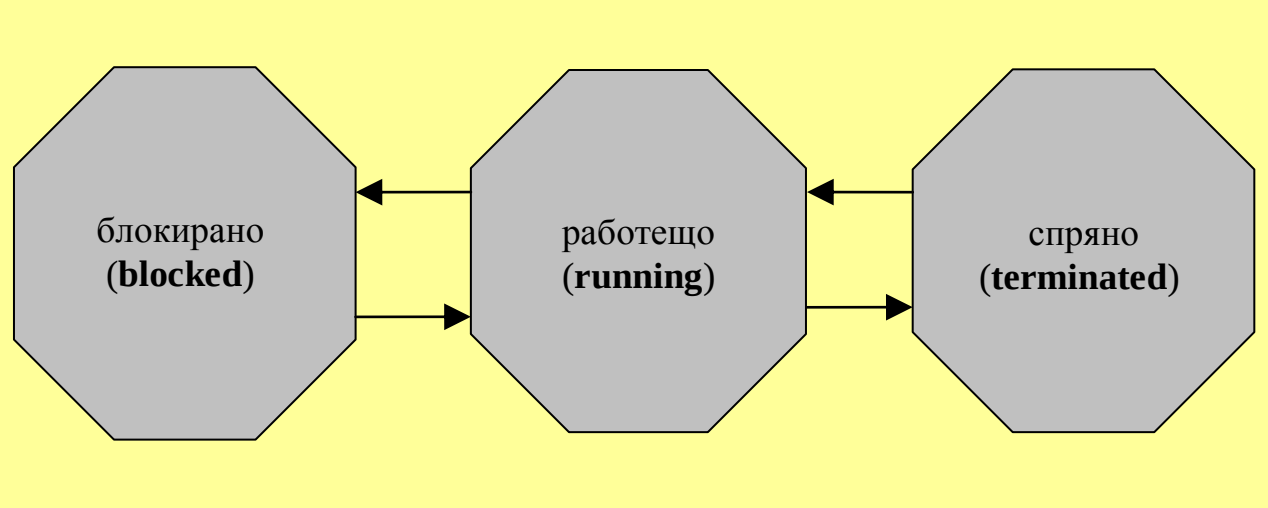
на процесора към изпълнение на даден процес, но и към даден поток на процеса (Фиг.94). Например може да принтирате страници в Word (поток 1) докато продължавате да въвеждате текст (поток 2).



Фиг. 94 Многопоточност в компютърните системи

Много програмни езици съдържат средства за управление на многопоточността. С появата на С11 езикът С също въвежда такива средства.

Всички примери досега бяха еднопоточни програми. Те се състояха от един главен поток, който изпълнява функцията main(). При многоточните програми главният поток (**main thread**) създава други потоци, които от своя страна също могат да създадат потоци и т.н. Един поток може да се намира в едно от няколко възможни състояния. Тези състояния зависят от имплементацията. Фиг.95 показва типични състояния на поток.



Фиг. 95 Състояния на поток

- работещо състояние (**running**). Работещ поток означава, че процесорът изпълнява неговия код;
- блокирано състояние (**blocked**). Изпълнението на всеки работещ поток може да се блокира до възникване на определено събитие;
- спряно състояние (**terminated**). Изпълнението на потока спира и всички използвани от него ресурси се освобождават.

Потоците могат да бъдат напълно независими един от друг, но могат и да използват общи ресурси като например общи променливи. Достъпът до такива общи ресурси трябва да бъде синхронизиран. Например, докато единият поток достъпва ресурса другият трябва да чака и обратно. Едновременният достъп на потоци до общ ресурс може да причини логическа грешка в програмата например, ако единият поток записва данни в една променлива, докато другият я чете в същия момент. Програмният код, който осъществява достъп до общ ресурс, се нарича **критична секция**.

31.7.4 Синхронизация на потоци

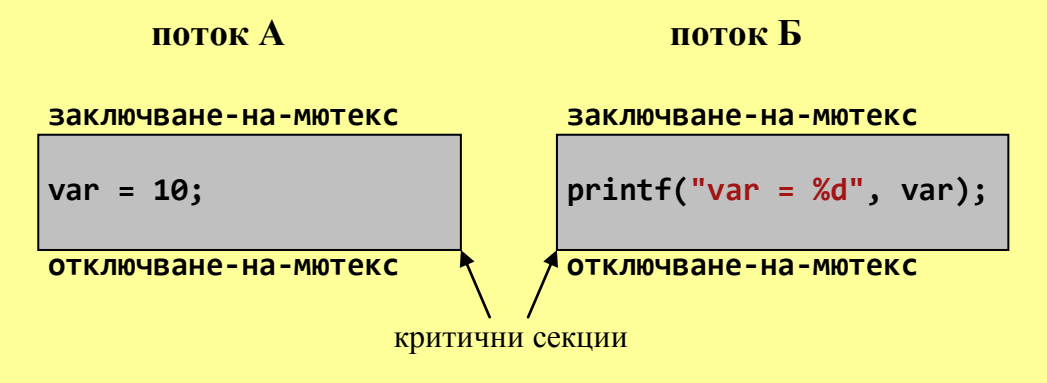
31.7.4.1 Мютекс

Мютексът (**mutex – mutual exclusion**) е едно от основните средства, използвани за синхронизация на потоци. Типичните действия свързани с мютексите са:

- създаване на мютекс;
- унищожаване на мютекс;

- заключване на мютекс;
- отключване на мютекс.

Мютексът се използва по следния начин. Когато общ ресурс трябва да бъде защитен от едновременен достъп на няколко потока, критичната секция може да бъде предхождана със заключване на мютекс и да завършва с отключването му. Когато един поток заключи даден мютекс всеки друг поток, който се опита да заключи същия мютекс, ще влезе в блокирано състояние и ще остане така, докато първият поток не отключи мютекса. Например, ако **поток А** записва данни в целочислена променлива var, а **поток Б** чете тази променлива и я изобразява на екрана, то достъпа до нея трябва да бъде осъществен по следния начин:

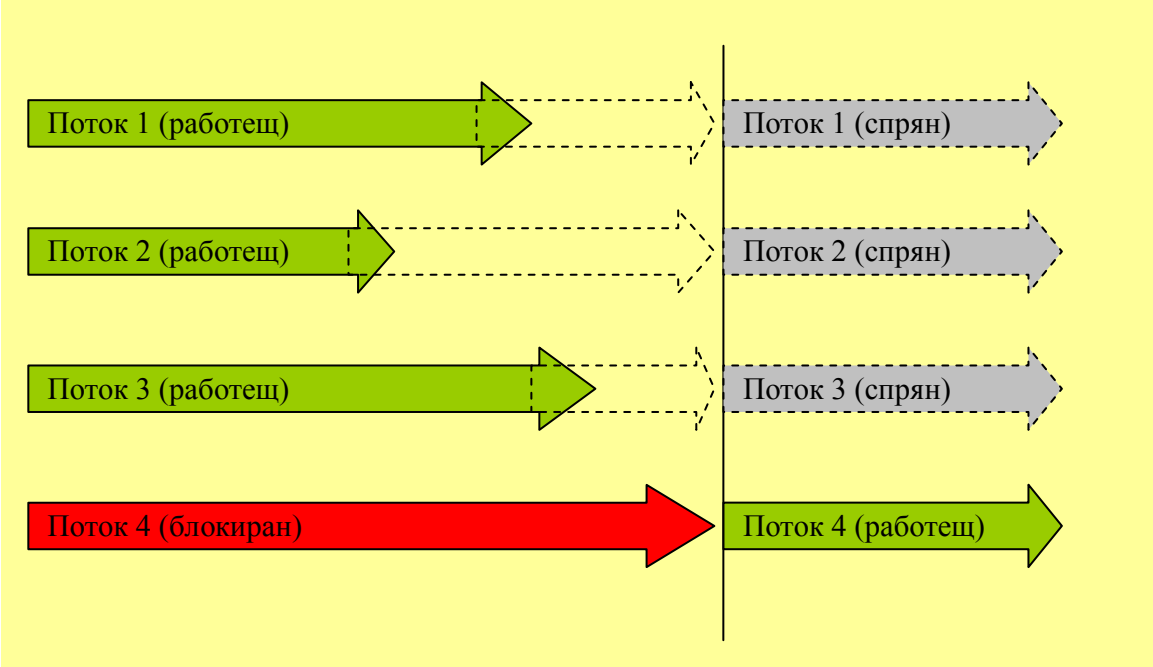


Фиг. 96 Използване на мютекс

Този механизъм гарантира, че потокът, който първи заключи мютекса, има достъп до общия ресурс, без опасност от едновременна намеса на друг поток. Това, което е важно да разберете, е, че всеки поток, който има достъп до общ ресурс, трябва да загражда кода, осъществяващ достъпа, със заключване и отключване на мютекса, в противен случай няма какво да го спре от достъп до ресурса, докато друг поток също го използва. Мютексът може да бъде отключен само от потока, който го е заключил, т.е. . ако поток А първи е заключил мютекса, поток Б не може да го отключи. Това може а стане само когато поток Б получи достъп до него и го заключи преди това.

31.7.4.2 Присъединяване на потоци

Присъединяването на потоци е процес, при който един поток изчаква друг поток (или потоци) да завърши изпълнението си преди да продължи да изпълнява кода си. Изчакващият поток влиза в блокирано състояние, докато чака (Фиг.97).



Фиг. 97 Присъединяване на потоци

Поток 4 влиза в блокирано състояние, докато потоците 1, 2 и 3 завършат изпълнение, след което продължава изпълнението си.

31.7.5 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.

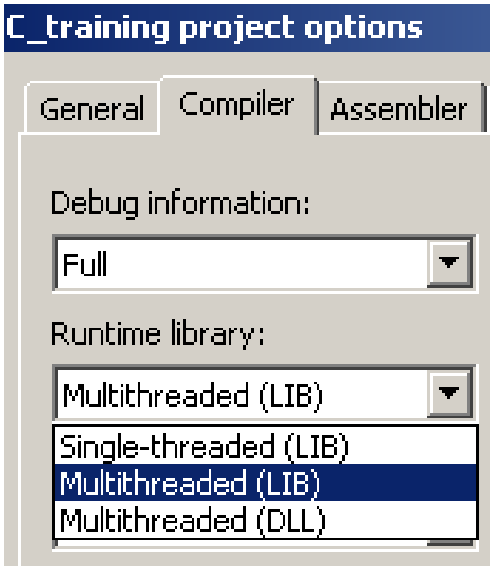
Функцията **thrd_exit()** завършва изпълнението на потока, от който се

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.

31.7.6 Примери

Преди да компилирате следващите примери в средата на Pelles, трябва да изберете следната настройка от **Project\Project options…**:



Фиг. 98 Поддръжка на многопоточност в средата на Pelles

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 31-3:

```
#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.

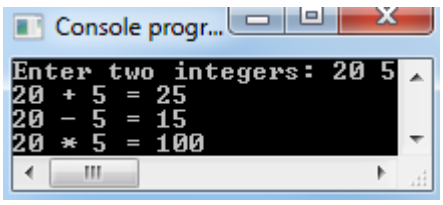
Присъединяване на един поток към друг

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 31-4:

```
#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
}
```



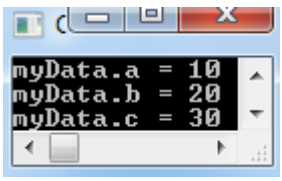
Използване на мютекс

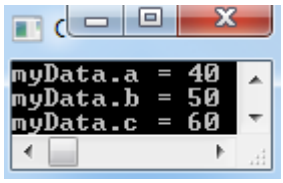
Разгледайте следния пример, който използва два потока, които записват данни в една и съща структурна променлива. Единият поток записва стойности 10, 20 и 30, а другият 40, 50, 60. Главният поток отпечатва данните, които последният поток е записал в променливата.

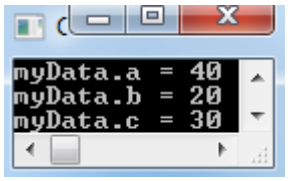
Пример 31-5:

```
#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.

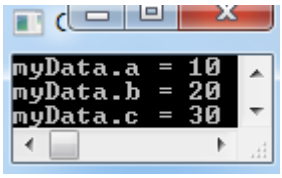
Пример 31-6:

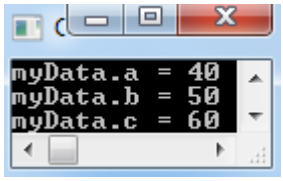
```
#include <stdio.h>
#include <threads.h>

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);
}
```

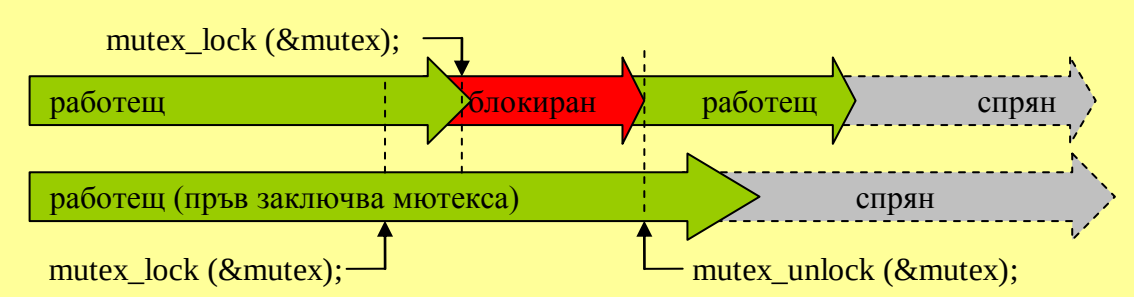
Критична секция

Критична секция





Потокът, който пръв заключи мютекса придобива правото да записва единствен в променливата. Фиг.99 илюстрира Пример 31-6.



Фиг. 99 Илюстрация към Пример 31-6

Част V: Стандартна Библиотека

32 Въведение

Езикът C предоставя голям набор от стандартни функции, наречени библиотечни, които можете да използвате наготово. Всяка библиотечна функция е декларирана в хедър-файл, чието съдържание се прави налично в програмата чрез #include предпроцесорната директива.

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

Табл.48 изброява всички хедър-файлове с кратко описание.

№	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 C 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>**	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. 48 Стандартни хедър-файлове

Следващите глави разглеждат някои по-интересни библиотечни функции. Пълно описание на стандартната библиотека ще намерите в книгата **ANSI C. Пълен справочник**.

33 Функции за вход-изход

33.1 putchar

```
#include <stdio.h>
int putchar(int c);
```

Функцията putchar() извежда символа c (преобразуван в тип unsigned char) на екрана.

Функцията putchar() връща изведения символ при успех. При възникване на грешка, putchar() връща макроса **EOF** (**End Of File**).

Пример 32-1:

```
#include <stdio.h>

int main(void)
{
  char c;

  for (c = 'A' ; c <= 'Z' ; c++)
  {
        putchar(c);
  }

  putchar('\n'); /* Извеждане на символ за нов ред */

  return 0;
}
```

33.2 getchar

```
#include <stdio.h>
int getchar(void);
```

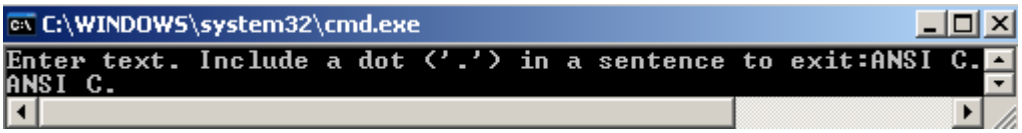
Функцията getchar() чете поредния символ (unsigned char) от клавиатурата.

Функцията getchar() връща символа при успех. Ако възникне грешка, getchar() връща макроса EOF.

Пример 33-2:

```
#include <stdio.h>

int main(void)
{
```

```
  char c;

  printf ("Enter text. Include a dot ('.') in a sentence to exit:");

  do
  {
    c=getchar(); /* Прочита символ от клавиатурата */
    putchar (c); /* Извежда символа на екрана      */
  } while (c != '.');

  putchar('\n');

  return 0;
}
```



33.3 puts

```
#include <stdio.h>
int puts(const char *str);
```

Функцията puts() извежда низа str на екрана и автоматично добавя символ за нов ред. Нулевият символ '\0' не се извежда.

Функцията puts() връща неотрицателна стойност при успех. При възникване на грешка puts() връща макроса EOF.

Пример 32-3:

```
#include <stdio.h>

int main(void)
{
  char string [] = "Hello world!";
  puts(string);

  return 0;
}
```

33.4 gets

```
#include <stdio.h>
char* gets(char *str);
```

Функцията gets() чете символи от клавиатурата и ги съхранява в буфера str. Четенето и записването на символи в буфера спира при прочитане на символ за нов ред. Символът за нов ред не се копира в буфера. Нулевият символ '\0' автоматично се добавя в края на съхранените в буфера символи.

Функцията gets() връща str при успех. Ако възникне грешка при четенето, gets() връща нулев указател.

Пример 32-4:

```
#include <stdio.h>

int main(void)
{
  char string [256];

  printf ("Insert your full address: ");
  gets (string);
  printf ("Your address is: %s\n",string);

  return 0;
}
```

Забележка: Функцията gets() е премахната в C11.

---

34 Математически функции

34.1 abs

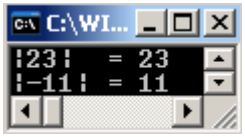
```
#include <stdlib.h>
int abs(int x);
```

Функцията abs() връща абсолютната стойност на цели числа тип int,т.е. .

$$|x| = \begin{cases} x, \text{ако } x > 0 \\ -x, \text{ако } x < 0 \end{cases}$$

Пример 34-1:

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

int main(void)
{
  printf ("|23|  = %d\n",abs(23));
  printf ("|-11| = %d\n",abs(-11));

  return 0;
}
```



34.2 fabs

```
#include <math.h>
double fabs(double x);
```

Връща абсолютната стойност на числа с плаваща запетая.

Пример 34-2:

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

int main(void)
{
  printf("|4.5|  = %.1f\n", fabs(4.5));
  printf("|-4.5| = %.1f\n", fabs(-4.5));

  return 0;
}
```

34.3 pow

```
#include <math.h>
double pow(double x, double y);
```

Връща x повдигнато на степен y, т.е. . x^y.

Пример 34-3:

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

int main(void)
{
    printf("2^5 = %f\n", pow(2, 5));

    return 0;
}
```

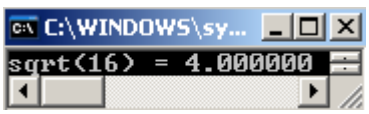
34.4 sqrt

```
#include <math.h>
double sqrt(double x);
```

Връща корен квадратен от числото x, т.е. . √x.

Пример 34-4:

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

int main(void)
{
  printf("sqrt(16) = %f\n", sqrt(16.0));

  return 0;
}
```



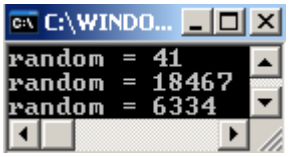
34.5 rand

```
#include <stdlib.h>
int rand(void);
```

Функцията rand() връща псевдо-случайно число в диапазона 0 ÷ RAND_MAX. RAND_MAX е макрос, който указва максималното възможно число, което може да се върне от rand().

Пример 34-5:

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

int main(void)
{
  printf ("random = %d\n", rand());
  printf ("random = %d\n", rand());
  printf ("random = %d\n", rand());

  return 0;
}
```



34.6 srand

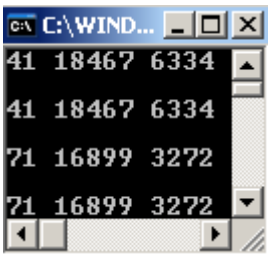
```
#include <stdlib.h>
void srand(unsigned int seed);
```

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(), ефектът е все едно srand() е била извикана с аргумент 1.

Пример 34-6:

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

int main(void)
{
  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;
}
```



35 Функции за обработка на символи

```
#include <ctype.h>

int isalnum (int character);
int isalpha (int character);
int iscntrl (int character);
int isdigit (int character);
int isgraph (int character);
int islower (int character);
int isprint (int character);
int ispunct (int character);
int isspace (int character);
int isupper (int character);
int isxdigit(int character);
```

Тези функции приемат за аргумент символ и връщат стойност, различна от 0, ако дадено условие е изпълнено, в противен случай връщат 0.

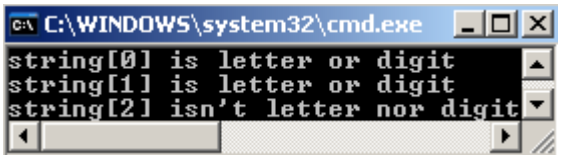
Условията, които всяка функция проверява дали са изпълнени са:

isalnum: символът е буква A÷Z или a÷z или цифра 0÷9
isalpha: символът е буква A÷Z или a÷z
iscntrl: управляващ символ (0x00÷0x1F и 0x7F в ASCII кодовата таблица)
isdigit: символът е цифра 0÷9
isgraph: графичен символ (0x2÷0x7E в ASCII кодовата таблица)
islower: символът е малка буква a÷z
isprint: печатен символ (0x20 - 0x7E в ASCII кодовата таблица)
ispunct: пунктуационен символ (всеки печатен символ без интервал, a÷z, A÷Z и 0÷9)
isspace: бели-символи (интервал ' ', хор. табулация '\t', вер. табулация '\v', връщане на курсора '\r', нов ред '\n', нова страница '\f')
isupper: символът е голяма буква A÷Z
isxdigit: шестнайсетичен символ (0÷9, A÷F, или a÷f)

Пример 35-1:

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

int main(void)
{
	char string[4] = "a1#";
	int  i;

	for(i = 0; i < 3 ; 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;
}
```


C:\WINDOWS\system32\cmd.exe
string[0] is letter or digit
string[1] is letter or digit
string[2] isn't letter nor digit

36 Функции за обработка на низове

36.1 strlen

```
#include <string.h>
size_t strlen(const char *str);
```

Функцията strlen() връща броя на символите на низа str. Нулевият символ не се отчита. Типът size_t е имплементационно-дефиниран беззнаков целочислен тип (примерно typedef unsigned long size_t).

Пример 36-1:

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

int main(void)
{
  char str[] = "Programming language C.";

  printf("The length of str is %d characters.\n", strlen(str));

  return 0;
}
```

36.2 strcpy

```
#include <string.h>
char *strcpy(char *str1 , const char *str2);
```

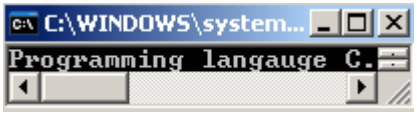
Функцията strcpy() копира низа str2 в низа str1 (включително и нулевия символ). Ако се получи застъпване, поведението е недефинирано. str1 трябва да е достатъчно голям[1], за да побере символите на str2, включително и нулевия символ.

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 36-2:

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

int main(void)
{
  char str[40];

  strcpy(str, "Programming langauge C.");
  printf("%s\n", str);

  return 0;
}
```



36.3 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.

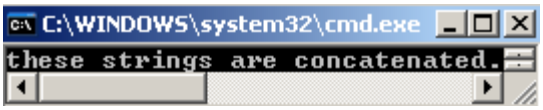
Забележка[1]: Големината на буфера, в който се съхраняват символите на низа, сочен от str1, трябва да има големина поне (strlen(str1) + strlen(str2) + 1) байта, за да се гарантира, че ще може да побере всички символи на низа str2 плюс нулев символ.

Функцията strcat() връща str1.

Пример 36-3:

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

int main(void)
{
  char str[80];

  strcpy (str,"these ");
  strcat (str,"strings ");
  strcat (str,"are ");
  strcat (str,"concatenated.");
  puts (str);
```

```
  return 0;
}
```



36.4 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 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 символът в str2 е нулев символ), < 0 ако str1 е по-малък от str2 (символът в str1 има по-малка числова стойност от символа в str2 или символът в str1 е нулев символ).

Пример 36-4:

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

int main(void)
{
  printf ("%d\n", strcmp("abcd", "abcd")); /*   0 */
  printf ("%d\n", strcmp("bcde", "abcd")); /* > 0 */
  printf ("%d\n", strcmp("abcd", "bcde")); /* < 0 */

  return 0;
}
```

36.5 strstr

```
#include <string.h>
char *strstr(const char *str1 , const char *str2 );
```

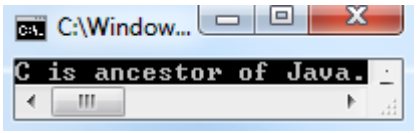
Функцията strstr() претърсва низа str1 за първата поява на низа str2 в него.

Нулевият символ на str2 не се взима под внимание.

Функцията strstr() връща указател към първия символ на намерения в str1 подниз, съвпадащ с низа str2. Ако няма съвпадение, strstr() връща нулев указател. Ако str2 е нулев низ, strstr() връща str1.

Пример 36-5:

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

int main(void)
{
  char str[] = "Programming language Java";
  char *p_substr;

  p_substr = strstr (str, "Java");

  printf("C is ancestor of %s.\n", p_substr);

  return 0;
}
```



37 Функции за динамично управление на паметта

37.1 malloc

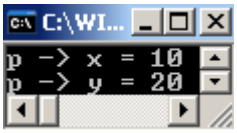
```
#include <stdlib.h>
void * malloc( size_t  size );
```

Функцията malloc() заделя памет, чийто размер е указан с **size** в байтове. Заделената памет е неинициализирана.

При успех malloc() връща указател към първия байт от динамично заделената памет. При неуспех malloc() връща нулев указател.

Пример 37-1:

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

typedef struct
{
      int x;
      int y;
}TCoord;

int main(void)
{
  TCoord* p; /* Дефиниране на указател към структура TCoord */

  /* Динамично заделяне на памет за структурна-променлива от тип Tcoord */
  p = (TCoord*)malloc(sizeof(TCoord));

  /* Проверка за валиден указател */
  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); /* Освобождаване на динамичната памет */
  }

  return 0;
}
```



Нека разгледаме конструкцията p = (TCoord*)malloc(sizeof(TCoord)); подробно. Изразът sizeof(TCoord) връща броя байтове, заемани от обект от тип TCoord. Ако има достатъчно свободна памет, функцията malloc() ще задели памет за тях и ще върне указател към първия байт. Този адрес се присвоява на указателя p, който е указател към тим TCoord. От тук нататък динамично заделеният обект може да се манипулира чрез указателя по

същия начин, както ако беше нормален обект от същия тип. Паметта, използвана за динамично дефиниране на обекти, се нарича хийп (**heap**).

Паметта за динамично заделените обекти трябва да се освобождава ръчно от потребителя с помощта на функцията free().

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. Ефектът от неосвободена памет се нарича "загуба на памет" ("memory leakage").

37.2 free

```
#include <stdlib.h>
void free(void *ptr);
```

Функцията free() освобождава динамично заделената памет (с помощта на функцията malloc()), към която сочи указателят ptr. Ако ptr е нулев указател, никакви действия не се извършват. Ако ptr не сочи към динамично заделена памет или тази памет вече е била освободена от предишно извикване на free(), поведението е недефинирано.

Пример 37-2:

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

int main(void)
{
  double* pd;

  /* Динамично заделяне на памет за обект тип double */
  pd = (double*)malloc(sizeof(double));

  *pd = 3.14; /* Присвояване на стойност на обекта */

  printf("pd -> %f\n", *pd);

  free(pd); /* Освобождаване на паметта, сочена от pd */

  return 0;
}
```

38 Функции за преобразуване

38.1 atoi

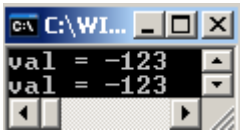
```
#include <stdlib.h>
int atoi(const char *str);
```

Функцията atoi() преобразува символен низ, представляващ десетична числова стойност тип int, в числовия му еквивалент. Например низът "-123" ще се преобразува в числото -123. Всички водещи бели-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 низът " \n\v\t-123av$%" ще се преобразува в числото -123.

При успех функцията atoi() връща преобразуваното число. Ако низът не може да се преобразува в число, atoi() връща 0.

Пример 38-1:

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

int main(void)
{
  int val;

  val = atoi("-123");
  printf ("val = %d\n",val);

  val = atoi(" \n\t\v-123av$%");
  printf ("val = %d\n",val);

  return 0;

}
```



38.2 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 низът "-123000000" ще се преобразува в числото -123000000. Всички водещи бели-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 низът " \n\v\t-123000000av$%456" ще се преобразува в числото -123000000.

При успех функцията atol() връща преобразуваното число. Ако низа не може да се преобразува в число, atol() връща 0.

Пример 38-2:

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

int main(void)
{
  long val;

  val = atol("-123000000");
  printf ("val = %ld\n",val);

  val = atol(" \n\t\v-123000000av$%456");
  printf ("val = %ld\n",val);

  return 0;
}
```

38.3 atof

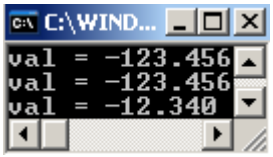
```
#include <stdlib.h>
double atof(const char *str);
```

Функцията atof() преобразува символен низ, представляващ числова стойност тип double, в числовия му еквивалент. Например низът "-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 низът " \n\t\v -12.34av$%" ще се преобразува в числото -12.34.

При успех функцията atof() връща преобразуваното число. Ако низът не може да се преобразува в число, atof() връща 0.

Пример 38-3:

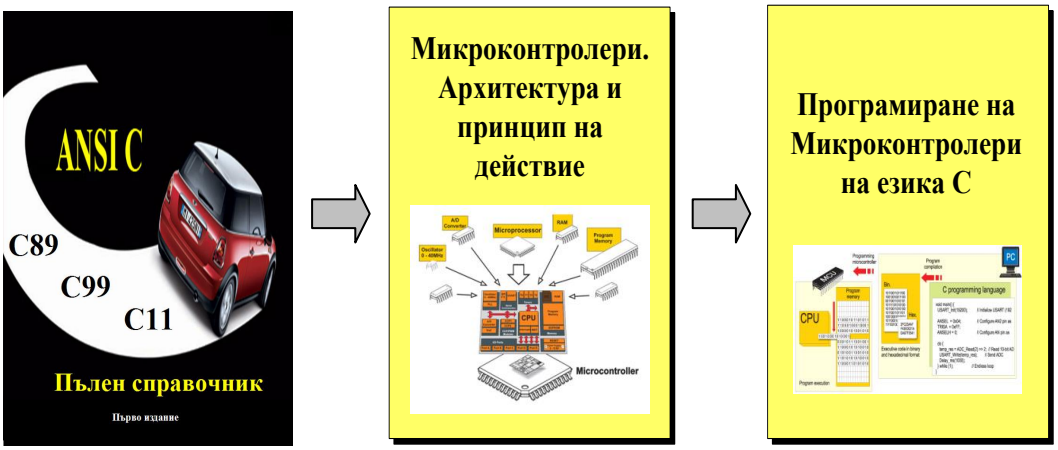
```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  double val;

  val = atof("-123.456");
  printf ("val = %.3f\n",val);

  val = atof("-1.23456e2");
  printf ("val = %.3f\n",val);

  val = atof(" \n\t\v-12.34av$%");
  printf ("val = %.3f\n",val);

  return 0;
}
```



Заключение

Ето че завършихте курса по програмиране на С за начинаещи. Може да си отдъхнете малко, но не се отпускайте  . Всичко, от което се нуждаете, е повече практика, за да подобрите своите умения. Препоръчвам Ви да направите един бърз преглед на целия материал дотук.

Имате няколко възможности как да продължите напред. Може да продължите с изучаване на детайлите на езика С, след което да преминете към изучаване на микроконтролерите и тяхното програмиране:



Ако се чувствате уверени, че разбирате добре нещата, описани в тази книга, може да преминете директно към изучаване на микроконтролерите и тяхното програмиране и да използвате книгата **ANSI C. Пълен справочник,** когато искате да си изясните детайли свързани с езика С:

Един последен съвет в заключение: Всички коментари в тази книга са на български език единствено за да служат като пояснения към основния текст. Ако решите да се занимавате професионално с програмиране, ще трябва да използвате коментари на английски език. Това е задължително особено в международните софтуерни компании.

Отговори

Част I Компютри и програмиране

1 Въведение в компютрите

1. **Кои са основните компоненти на един компютър и как си взаимодействат те?**

   Двата основни компонента на един компютър са: **хардуер** и **софтуер**. Хардуерът включва всички електронни елементи, от които е изграден компютърът. Софтуерът е програма, която "казва" на хардуера как да обработва информацията.

2. **Какъв е форматът на информацията, който съвременните компютри обработват?**

   Съвременните компютри са цифрови. Независимо каква информация искаме да обработим, тя трябва да бъде подадена на компютъра в цифров (двоичен) вид.

3. **От какви стойности се състои информацията, обработвана от съвременните компютри?**

   Цифровата информация е съвкупност от комбинации само от две стойности: нула (0) и единица (1).

4. **Кое е най-малкото количество информация в контекста на съвременните компютри?**

   Най-малкото количество информация, което може да се предаде в цифров вид, е 0 или 1 и се нарича **бит**.

5. **Какво е шина в контекста на компютрите?**

   Група проводници за предаване на битове в паралел формират шина.

6. **Какво е байт?**

   Съвкупност от 8 бита се нарича байт.

7. **Ако имате информация от 10 байта, на колко бита се равнява тя?**

   10 байта = 10 x 8 = 80 бита

8. **Какво е MSB и LSB?**

Всеки бит в един байт си има номер. Най-десният бит има номер 0 и се нарича най-младши бит (**LSB –Least Significant Bit**) или само младши бит. Най-левият бит има номер 7 и се нарича най-старши бит (**MSB – Most Significant Bit**) или само старши бит.

2 Бройни системи

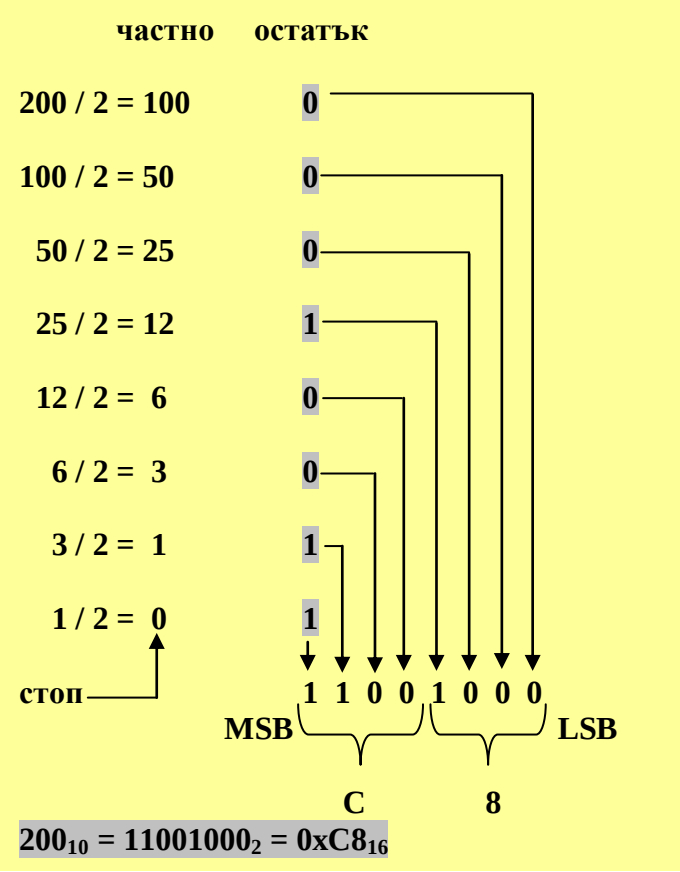
1. **Как се наричат трите бройни системи, които се използват в компютърната техника?**

   Двоична, десетична и шестнайсетична бройни системи.

2. **Какво представлява основата (базата) на една бройна система?**

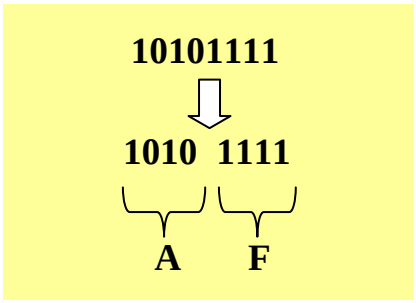
   Броят на символите, използвани от една бройна система, се нарича основа или база **b** (**radix** или **base**) на бройната система.

3. **Преобразувайте десетичното число 200 в двоична и шестнайсетична бройна система.**



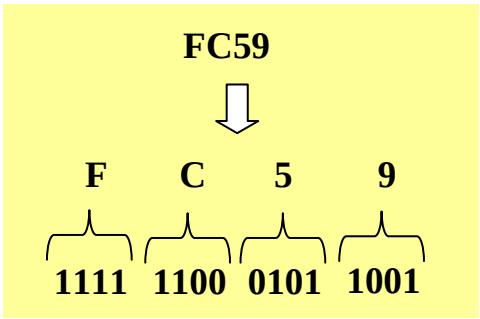
4. **Преобразувайте двоичното число 10101111 в десетична и шестнайсетична бройна система.**

   $10101111_2 = 1*2^7 + 0*2^6 + 1*2^5 + 0*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0 = 175_{10}$



10101111_2 = $0xAF_{16}$

5. **Преобразувайте шестнайсетичното число FC59 в десетична и двоична бройна система.**



0xFC59 = 1111110001011001_2 = F*16^3 + C*16^2 + 5*16^1 + 9*16^0 = 15*16^3 + 12*16^2 + 5*16^1 + 9*16^0 = 61440 + 3072 + 80 + 9 = 64601_{10}

3 Представяне на знакови цели числа в двоична бройна система.

1. **Представете следните числа в допълнителен код:** 0, -1, -120, -127

0 -> 00000000

-1-> 00000001 (+1) -> 11111110 (-1 в обратен код)
+ 1

11111111 (-1 в допълнителен код)

-120 -> 01111000 (+120) -> 10000111 (-120 в обратен код)
+ 1

10001000 (-120 в допълнителен код)

-127 -> 01111111 (+127) -> 10000000 (-127 в обратен код)
+ 1

10000001 (-127 в допълнителен код)

Част II C89(C90)

4 Въведение в C програмирането

1. **Как се казва главната функция, която задължително трябва да присъства във всяка C програма?**

   Всяка C програма започва изпълнението си от функция, наречена main().

2. **Ако имате функция с име func, която не приема и не връща стойност, покажете как се извиква в кода.**

   func();

3. **Обяснете как протича изпълнението на програмата при извикване на една функция.**

   Когато една функция се извика, програмата преминава към изпълнение на операторите, намиращи се в нейното тяло.

4. **Обяснете как протича изпълнението на програмата при завършване на изпълнението на една функция.**

   При завършване на изпълнението на една функция, програмата преминава към изпълнение на първия оператор, разположен след точката на извикване на функцията.

5. **Как се създава коментар и за какво служи?**

   Коментарът започва с последователността /* и завършва с */. Коментарите се използват за вмъкване на всякаква пояснителна информация в кода.

6. **Довършете следното твърдение: Всеки C оператор трябва да завършва с...**

   ...точка и запетая.

5 Променливи и типове данни в C

1. **Какво е променлива?**

   Променлива, това е област от паметта на компютъра, която се използва за съхранение на данни.

2. **Напишете общата форма за дефиниране на променлива**

   тип име-променлива;

3. **Каква информация носи типа на една променлива за компилатора?**

   Типът на една променлива носи следната информация:

   - Видът на данните, които тази променлива може да съхранява.
   - Видовете операции, които могат да се прилагат върху тази променлива.

4. **Къде се дефинират променливите?**

   Променливите могат да се дефинират на две места:

   - Извън всички функции.
   - В началото на функция или блок с код преди операторите за изпълнение.

5. **Кои са базовите типове данни в езика C?**

   C89 дефинира следните базови типове данни: char, int, float, double, void.

6. **Кои са модификаторите на базовите типове и как се прилагат към тях?**

   C89 дефинира следните модификатори: unsigned, signed, short и long. Те могат да се прилагат към базовите типове както е показано по-долу.

unsigned char unsigned int
signed char signed int
long int unsigned long int signed long int long double
short int unsigned short int signed short int

7. **Опишете правилата за създаване на име на променлива (от какви символи може да се състои и т.н.)**

Името на една променлива може да се състои от следните символи:

- Малки и големи букви от латинската азбука (a÷z и A÷Z).
- Арабски цифри (0÷9).
- Подчертаващо тире (_).

Името на променливата НЕ може да започва с цифра.

8. Ако имате променлива с име symbol, покажете как ще изглежда дефиницията от тип char.

char symbol;

9. Как се присвоява стойност на една променлива?

Присвояването на стойност на променлива става с помощта на операцията присвояване '=', т.е. .

име-променлива = стойност;

10. Опишете областта на видимост и времето на живот на локалните променливи (auto и register). Каква е стойността на локална променлива след дефинирането при липса на явен инициализатор?

Областта на видимост на локална променлива е тялото на блока, в който е дефинирана. Времето на живот на локалните променливи съвпада с областта им на видимост. При липса на явен инициализатор, локалните променливи съдържат произволна стойност.

11. Опишете областта на видимост и времето на живот на локалните статични променливи. Каква е стойността на локална статична променлива след дефинирането при липса на явен инициализатор?

Областта на видимост на локална статична променлива е тялото на блока, в който е дефинирана. Времето на живот е докато трае изпълнението на програмата. При липса на явен инициализатор, локалните статични променливи се инициализират с нула.

12. Опишете областта на видимост и времето на живот на глобалните статични променливи. Каква е стойността на глобалната статична променлива след дефинирането при липса на явен инициализатор?

Областта на видимост на глобална статична променлива е файлът, в който е дефинирана. Времето на живот е докато трае изпълнението на

програмата. При липса на явен инициализатор, глобалните статични променливи се инициализират с нула.

13. Опишете областта на видимост и времето на живот на глобалните външни променливи. Каква е стойността на глобалната външна променлива след дефинирането при липса на явен инициализатор?

Областта на видимост на глобална външна променлива е цялата програма (всички файлове, от които е изградена програмата). Времето на живот е докато трае изпълнението на програмата. При липса на явен инициализатор, глобалните външни променливи се инициализират с нула.

14. За какво служи спецификатора typedef?

Спецификаторът typedef служи за създаване на синоним (друго име) на вече съществуващ тип.

6 Константи

1. Какво е константа ?

Константата е фиксирана стойност, която не може да се изменя в програмата.

2. Какви видове константи има?

В зависимост от типа, константите могат да се разделят на:

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

3. Ако тип int е 16 бита, а long е 32 бита, какъв тип ще назначи компилатора на следните целочислени константи?

a) 10 - int
b) 10u - unsigned int
c) 10L - long
d) 10ul - unsigned long
e) 50000 - long

f) **0xC350 (десетично 50000) – unsigned int**

4. **Какъв е разделителният символ в константите с плаваща запетая?**

   точка

5. **Какъв тип ще назначи компилатора на следните константи с плаваща запетая?**

   a) **10.2 - double**
   b) **10.2f - float**
   c) **10.2L - long double**

6. **Какво е низова константа?**

   Низова константа се нарича всяка последователност от символи, затворена в двойни кавички: "ANSI C. Beginner's course."

7. **Как се обозначава край на низ?**

   Всеки низ завършва с нулев символ '\0'.

7 Извеждане на данни на екран

1. **Напишете обобщената форма на извикване на функцията printf().**

   printf(форматиращ-низ, списък-променливи);

2. **Кои са форматиращите последователности за отпечатване на char, int, float и double с помощта на printf()?**

   %c, %d, %f, %f

3. **Изведете на екрана съобщението "ANSI C. Beginner's course!" с puts().**

   puts("ANSI C. Beginner's course!");

8 Четене на данни от клавиатурата

1. **Напишете обобщената форма на извикване на функцията scanf().**

   scanf(форматиращ-низ, списък-променливи);

2. **Кои са форматиращите последователности за четене на данни тип char, int, float и double с помощта на scanf()?**

   %c, %d, %f, %lf

3. **Какъв е синтаксиса на списък-променливи на функцията scanf().**

   &име-променлива1, &име-променлива2, ... , &име-променливаN

9 Операции в езика C

1. **Избройте категориите операции в езика C.**

   Аритметични операции
   Операции за сравнение
   Логически операции
   Битови операции
   Операции за присвояване
   Условна операция
   Операция sizeof
   Операция за последователно изчисляване на изрази
   Други

2. **Какъв ще бъде резултатът от делението на две цели числа?**

   Цяло число.

3. **Какъв ще е резултатът от делението 13 / 2. Напишете примерна програма, която изобразява резултата на екрана.**

   В контекста на езика C 13 / 2 = 6

```c
#include <stdio.h>

int main(void)
{
   printf("13/2 = %d\n", 13/2);

   return 0;
}
```

4. **Операторът % може да се използва с цели и дробни числа. Вярно или невярно?**

   Невярно. Може да се използва само с цели числа.

5. **Какъв ще е резултатът от делението по модул 121 % 21. Напишете примерна програма, която изобразява резултата на екрана.**

   121 % 21 = 16

```c
#include <stdio.h>

int main(void)
{
   printf("121%%21 = %d\n", 121%21);

   return 0;
}
```

6. **Какъв е резултатът от логическите операции?**

   0 или 1.

7. **Каква ще бъде стойността на y в следната програма?**

```c
#include <stdio.h>

int main(void)
{
    int x,y;

    x = 10;
    y = 0;
    (x < 0) && (y = --x + 1);
    printf("y = %d\n", y);

    return 0;
}
```

   y = 0;

8. **Ако имате променлива x тип int със стойност 0x2355, напишете код, който установява в 1 бит 7 и нулира бит 0, без да променяте останалите битове.**

```c
#include <stdio.h>

int main(void)
{
    int x = 0x2355;

    x |= 0x0080; /* Установяване на бит 7 в 1 */
    x &= 0xFFFE; /* Нулиране на бит 0         */
    printf("x = %#x\n", x);
    return 0;
}
```

9. **Каква ще бъде стойността на x в следната програма?**

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

```c
int main(void)
{
    int x;

    x = (3 != 0) ? 1 : 0;
    printf("x = %d\n", x);

    return 0;
}
```

x = 1;

10. Ако тип int е 4 байта, определете стойностите на променливите x и y в следния пример:

```c
int y, x = 1;
y = sizeof(++x);
```

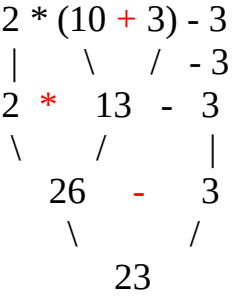
y = 4 и x = 1

11. Обяснете за какво се използва приоритета и асоциативността на операциите.

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.

12. Какъв е редът на изчисление на операциите в следния израз?

2 * (10 + 3) - 3



10 Изрази и преобразувания на типове

1. Какво е израз в езика C ?

Израз е комбинация от операнди и операции. Операндите могат да бъдат променливи, константи, функции връщащи резултат и т.н. В най-простия случай израз може да бъде само променлива или само константа.

2. Какво представлява целочисленото повишаване и кога се прилага ?

Всички променливи от тип char (signed или unsigned), short (signed или unsigned), enum-тип и битово поле се преобразуват в тип int (при необходимост и в тип unsigned int) преди изчисление на израза, в който участват. Следващата таблица описва кога операндите се преобразуват в int и кога 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]	●	

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 имплементацията.

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.

---

3. Опишете правилата за преобразуване на операнди от различни типове в аритметичен израз.

№	ако единият операнд е	а другият е	компилаторът преобразува операндите в
1	long double	double, float, unsigned long, signed long, unsigned int, signed int	long double
2	double	float unsigned long signed long unsigned int	double

		signed int	
3	float	unsigned long signed long unsigned int signed int	float
4	unsigned long	signed long unsigned int signed int	unsigned long
5	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]
6	signed long	signed int	signed long
7	unsigned int	signed int	unsigned int
8	signed int	char (signed или unsigned) short (signed или unsigned) битово поле enum	signed int или unsigned int съгласно Табл.42 и правило 7
9	char (signed или unsigned) short (signed или unsigned) битово поле enum	char (signed или unsigned) short (signed или unsigned) битово поле enum	signed int или unsigned int съгласно Табл.42 и правило 7

Забележка[1] : Ако тип long и int са с еднакъв размер (32 бита), signed long не може да представи всички стойности на unsigned int.

---

4. Ако променливата d е тип double, а i е тип int, какъв ще е типът на резултата от изчислението на израза d + i?

double

5. Каква ще е стойността на променливата res в следната програма?

```
#include <stdio.h>

int main(void)
{
    unsigned char op1, op2, res;

    op1 = 200;
    op2 = 100;
    res = op1 + op2;
    printf ("res = %d\n", res);
```

```
    return 0;
}
```

res = 44

6. Опишете правилата за преобразуване при присвояване.

Ако стойността на десния операнд се побира в диапазона от стойности на левия операнд, то стойността на десния операнд се копира в левия без промяна.

Ако стойността на левия операнд е от знаков тип, а стойността на десния операнд е от знаков или беззнаков тип и излиза извън диапазона от стойности на левия операнд, то резултатът зависи от имплеметацията. Тази ситуация е известна като целочислено препълване.

Ако стойността на левия операнд е от беззнаков тип, а стойността на десния операнд е от беззнаков тип и излиза извън диапазона от стойности на левия операнд, то излишните старши байтове на десния операнд се изхвърлят, а младшите се копират в левия операнд.

Ако стойността на левия операнд е от беззнаков тип, а стойността на десния операнд е от знаков тип и излиза извън диапазона от стойности на левия операнд, то са възможни два сценария:

1. Ако стойността на десния операнд е неотрицателна, от нея се изважда число с 1 по-голямо от максималното число, което може да се представи от левия операнд, толкова пъти, докато получената стойност попадне в обхвата му от стойности.

2. Ако стойността на десния операнд е отрицателна, към нея се добавя число с 1 по-голямо от максималното число, което може да се представи от левия операнд, толкова пъти, докато получената стойност попадне в обхвата му от стойности.

Ако левият операнд е цяло число, а десният е число с плаваща запетая, то дробната част на десния операнд се изхвърля. Ако цялата част на десния операнд излиза извън диапазона от стойности на левия операнд, резултатът е недефиниран. Това е вид препълване.

7. Каква ще е стойността на променливата res в следната програма?

```
#include <stdio.h>5

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

```
    res = 10.345;
    printf ("res = %d\n", res);

    return 0;
}
```

res = 10

8. **Какъв е резултатът от следните явни преобразувания?**

```
(unsigned char)0xaa55 -> 0x55
(int)20.34 -> 20
```

11 Управляващи оператори

1. **Как работи операторът за цикъл while?**

   Ако стойността на **управляващ-израз** е различна от нула, програмата преминава към изпълнение на операторите в тялото на while. След това отново се проверява стойността на **управляващ-израз**. Ако стойността е различна от нула, операторите в тялото на while се изпълняват отново. Процесът се повтаря, докато стойността на **управляващ-израз** е различна от нула. Ако стойността на **управляващ-израз** е нула, операторът while се прескача и програмата преминава към изпълнение на първия оператор след while.

2. **Как работи операторът за цикъл do-while?**

   При достигане до оператора do-while, програмата първо изпълнява операторите в тялото на цикъла, след което се изчислява стойността на **управляващ-израз.** Ако стойността на **управляващ-израз** е различна от нула, програмата отново преминава към изпълнение на операторите в тялото на do-while. Процесът се повтаря, докато стойността на **управляващ-израз** е различна от нула. Ако стойността на **управляващ-израз** е нула, операторът do-while се прескача и програмата преминава към изпълнение на първия оператор след do-while.

3. **Как работи операторът за цикъл for?**

   При достигане до оператора for, програмата първо изчислява **израз1**, след което преминава към изчисляване на **управляващ-израз**. Ако стойността на **управляващ-израз** е различна от нула, програмата преминава към изпълнение на операторите в тялото for. След като изпълни операторите, програмата изчислява стойността на **израз3**, след което отново изчислява **управляващ-израз**. Ако стойността е различна от нула, операторите в тялото на for се изпълняват отново. Процесът се

повтаря, докато стойността на **управляващ-израз** е различна от нула. Ако стойността на **управляващ-израз** е нула, операторът for се прескача и програмата преминава към изпълнение на първия оператор след for.

4. **Коригирайте следната програма така, че да отпечата само числата от 0 до 5, без да променяте управляващия израз на цикъла.**

```
#include <stdio.h>

int main(void)
{
   int x;

   for (x = 0; x < 100 ; x++)
   {
         printf("x = %d\n", x);
   }

   return 0;
}
```

```
#include <stdio.h>

int main(void)
{
   int x;

   for (x = 0; x < 100 ; x++)
   {
         if ( x <= 5)
         {
               printf("x = %d\n", x);
         }
         else
         {
               break;
         }
   }

   return 0;
}
```

12 Масиви

1. **Покажете как се дефинира едномерен масив.**

   тип име-масив[размер];

2. **Дефинирайте масив с име digits, който има 10 елемента от тип int.**

   int digits[10];

3. **Номерата (индексите) на елементите на масива започват от 1. Вярно или невярно?**

   Невярно. Първият елемент на масив има индекс 0.

4. **Как се достигат елементите на масив?**

   име-масив[индекс]

5. **Ако е дадена дефиницията int a[5] = {1};, какви ще бъдат стойностите на елементите a[0], ... , a[4]?**

   a[0] = 1, a[1] = 0, a[2] = 0, a[3] = 0, a[4] = 0

6. **Какво е двумерен масив ?**

   Двумерният масив е едномерен масив, чиито елементи също са едномерни масиви.

7. **Създайте двумерен символен масив и го инициализирайте със имената на месеците. Отпечатайте масива на екрана.**

```c
#include <stdio.h>

#define MONTHS     12
#define LENGTH     10

int main(void)
{
   char months[MONTHS][LENGTH] =
   {
          "January", "February", "March", "April",
          "May", "June", "July", "August",
          "September", "October",   "November", "December"
   };
   int index;

   /* Отпечатване на всеки низ от таблицата */
   for(index = 0 ; index < MONTHS ; index++)
   {
          printf("%s\n", months[index]);
   }

   return 0;
}
```

13 Указатели

1. **Покажете как се дефинира указател.**

   тип * име-указател;

2. **Дефинирайте указател pd, който сочи към променливи от тип double.**

   double * pd;

3. **Към обекти от кои типове може да сочи един указател?**

   - всеки един от базовите типове;
   - друг указател;
   - структурен тип;
   - union-тип;
   - enum-тип.

4. **Кой е операторът за извличане на адрес на променлива и как се използва?**

   &име-променлива

5. **Как се осъществява косвен достъп до променлива посредством указател?**

   *име-указател

6. **Напишете програма, която дефинира променлива d от тип double и указател pd към променливи от тип double. Задайте косвено стойност на d и отпечатайте стойността на екрана директно.**

```c
#include <stdio.h>

int main(void)
{
   double d, *pd = &d;

   *pd = 2.3;
   printf("d = %f\n", d);

   return 0;
}
```

7. **Кои операции могат да се прилагат към указателите?**

   - събиране и изваждане на указател с цяло число;
   - изваждане между указатели;

- сравнение на указатели;
- сравнение на указател и цяло число;
- присвояване на 0 на указател.

8. **Дефинирайте указател към функция, която приема един аргумент тип int и не връща стойност.**

   void (*ptr_to_f) (int i);

9. **Покажете как може да опростите горната дефиниция с помощта на typedef.**

   typedef void (*t_ptr_to_f) (int i);
   t_ptr_to_f ptr_to_f;

10. **Обяснете връзката между указатели и масиви.**

    Името на масив е указател към първия елемент на масива. Компилаторът не заделя памет за името на масива, а го асоциира с адреса на първия елемент.

14 Структури

1. **Обяснете какво представлява структурата.**

   Структурата е комплексен тип данни, който се дефинира от потребителя. Тя комбинира различни типове данни в един общ тип.

2. **Покажете как се дефинира структура.**

```
struct  име-структура
{
   тип    име-променлива1;
   тип    име-променлива2;
   . . .
   тип    име-променливаN;
};
```

3. **Как се достъпват членовете на структура?**

   име-структурна-променлива **.** име-променлива
   име-указател-към-структура -> име-променлива

4. **Какъв тип могат да бъдат членовете на една структура?**

   - променливи от базови типове;
   - масиви;
   - указатели;
   - променливи от други структури;
   - променливи от union-тип;
   - битови полета;
   - променливи от enum-тип.

5. **Декларирайте структура с име mystr, която има членове d от тип double и i от тип int. Дефинирайте структурна-променлива с име s. Задайте стойности на членовете на структурата и ги отпечатайте на екрана.**

```
#include <stdio.h>

struct mystr
{
   double d;
   int    i;
};

int main(void)
{
   struct mystr s;

   s.d = 2.3;
   s.i = 10;
   printf("s.d = %f\n", s.d);
   printf("s.i = %d\n", s.i);

   return 0;
}
```

6. **Как може да опростите декларацията на структурата от т.5 с помощта на typedef?**

```
#include <stdio.h>

typedef struct mystr
{
   double d;
   int    i;
}t_mystr; /* типът t_mystr е еквивалентен на типа struct mystr */

int main(void)
{
```

```
    t_mystr s;

    s.d = 2.3;
    s.i = 10;
    printf("s.d = %f\n", s.d);
    printf("s.i = %d\n", s.i);

    return 0;
}
```

7. **Може ли една структура да съдържа член от себе си?**

   Членовете на една структура не могат да бъдат от същия тип като структурата, но могат да бъдат указатели към обекти от този тип.

15 Обединения

1. **Обяснете какво представлява обединението.**

   Обединението е потребителски тип данни, в който всички членове използват една и съща памет.

2. **Покажете как се дефинира обединение.**

```
union  име-обединение
{
    тип име-променлива1;
    тип име-променлива2;
    . . .
    тип име-променливаN;
};
```

3. **Какъв тип могат да бъдат членовете на едно обединение?**

   - променливи от базови типове;
   - масиви;
   - указатели;
   - структурни-променливи;
   - променливи от друго обединение;
   - битови полета;
   - променливи от enum-тип.

4. **Как се достъпват членовете на обединение?**

   име-union-променлива **.** име-променлива
   име-указател-към-обединение -> име-променлива

5. **Декларирайте обединение с име myunion, което има членове d от тип double и i от тип int. Дефинирайте union-променлива с име u. Задайте стойности на членовете на обединението и ги отпечатайте на екрана.**

```c
#include <stdio.h>

union myunion
{
   double d;
   int    i;
};

int main(void)
{
   union myunion u;

   u.d = 2.3;
   printf("u.d = %f\n", u.d);
   u.i = 10;
   printf("u.i = %d\n", u.i);

   return 0;
}
```

6. **Как може да опростите декларацията на обединението от т.5 с помощта на typedef?**

```c
#include <stdio.h>

typedef union myunion
{
   double d;
   int    i;
}t_myunion; /* Типът t_myunion е еквивалентен на типа union myunion */

int main(void)
{
   t_myunion u;

   u.d = 2.3;
   printf("u.d = %f\n", u.d);
   u.i = 10;
   printf("u.i = %d\n", u.i);

   return 0;
}
```

16 Битови полета

1. **Покажете как се дефинира битово поле.**

   тип име-битово-поле: широчина;

2. **Какво представлява битово поле без име и широчина различна от 0?**

   Неименуваните битови полета с широчина различна от 0 играят ролята на уплътняващи битове и не могат да се достъпват от кода.

3. **Какво представлява битово поле без име и широчина равна на 0?**

   Неименувано 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 началото на следващия сегмент.

4. **Какви ограничения налага стандартът C на битовите полета?**

   - Битово поле не може да съществува самостоятелно извън тялото на структура или обединение.
   - Не може да се извлича адреса на битово поле.
   - Не може да се дефинира указател към битово поле.
   - Не може да се дефинира масив от битови полета.

17 Изброявания

1. **Обяснете какво представлява изброяването.**

   Изброяването е средство, чрез което могат да се създават именувани целочислени константи.

2. **Покажете как се декларира изброяване.**

```
enum име-изброяване
{
   име-константа1,
   име-константа2,
   ...
   име-константаN
```

};

3. **Ако е дадено изброяването enum rgb {RED, GREEN, BLUE};, какви стойности компилаторът ще присвои на enum-константите RED, GREEN и BLUE ?**

RED == 0
GREEN == 1
BLUE == 2

18 Функции

1. **Как се дефинира функция?**

```
тип-резултат  име-функция(списък-параметри)
{
/**************************************
* Между тези две фигурни скоби        *
* се разполага кодът, който се изпълнява *
* от функцията                        *
**************************************/
}
```

2. **Кои са съставните компоненти на една функция.**

   - Заглавие на функция (тип-резултат, име-функция, списък-параметри).
   - Тяло на функция.

3. **За какво служи прототипът?**

   Прототипът дава информация на компилатора за типа на резултата и броя и типа на параметрите на функцията. Тази информация позволява на компилатора да сравнява типа и броя на аргументите с тези на параметрите. Ако типът на някой аргумент се различава от този на параметъра, компилаторът ще го преобразува. Ако преобразуването е невъзможно, компилаторът ще генерира грешка.

4. **Напишете функция, която не връща стойност, но има един параметър от тип double. Нека функцията отпечатва стойността на параметъра на екрана.**

```
#include <stdio.h>

void print_value(double d)
```

```
{
   printf("%lf", d);
}
```

5. **Какви са механизмите на подаване на аргументи на функция?**

   - по стойност (чрез копие);
   - по адрес.

6. **Какъв тип може да бъде тип-резултат на функцията?**

   - базов тип;
   - структурен-тип;
   - union-тип;
   - enum-тип;
   - указателен тип.

7. **Какъв тип могат да бъдат параметрите на функцията?**

   Могат да бъдат всеки един от типовете данни в C (базов тип, структурен-тип, union-тип, enum-тип, указателен тип, масиви).

19 Предпроцесор

1. **Каква е ролята на предпроцесора ?**

   Предпроцесорът анализира кода преди неговата компилация. Предпроцесорът не прави синтактичен анализ на кода, а най-общо казано преобразува програмния текст в текст готов за компилация. За целта той използва предпроцесорни команди, наречени директиви.

2. **Какво прави конструкцията #include <file.h>?**

   Добавя съдържанието на хедър-файла file.h.

3. **Покажете как се дефинира макрос.**

   #define име-макрос заменящ-текст

   или

#define име-макрос(списък-параметри) заменящ-текст

4. **Опишете как работят директивите #ifdef-#else-#endif.**
   Предпроцесорът проверява дали **име-макрос** е дефиниран с директивата #define. Ако **име-макрос** е дефинирано с директивата #define, програмният текст, заключен между #ifdef и #else, се подава на компилатора, а програмният текст, заключен между #else и #endif, не се компилира. Ако **име-макрос** не е дефинирано с директивата #define или е бил отдефиниран с #undef, програмният текст, заключен между #ifdef и #else, не се компилира, а програмният текст, заключен между #else и #endif, се подава на компилатора.

5. **Опишете как работят директивите #ifndef-#else-#endif.**

   Предпроцесорът проверява дали **име-макрос** е дефинирано с директивата #define. Ако **име-макрос** не е дефинирано с директивата #define или е било отдефинирано с #undef, програмният текст, заключен между #ifdef и #else, се подава на компилатора, а програмният текст, заключен между #else и #endif, не се компилира. Ако **име-макрос** е дефинирано с директивата #define, програмният текст, заключен между #ifdef и #else, не се компилира, а програмният текст, заключен между #else и #endif, се подава на компилатора.

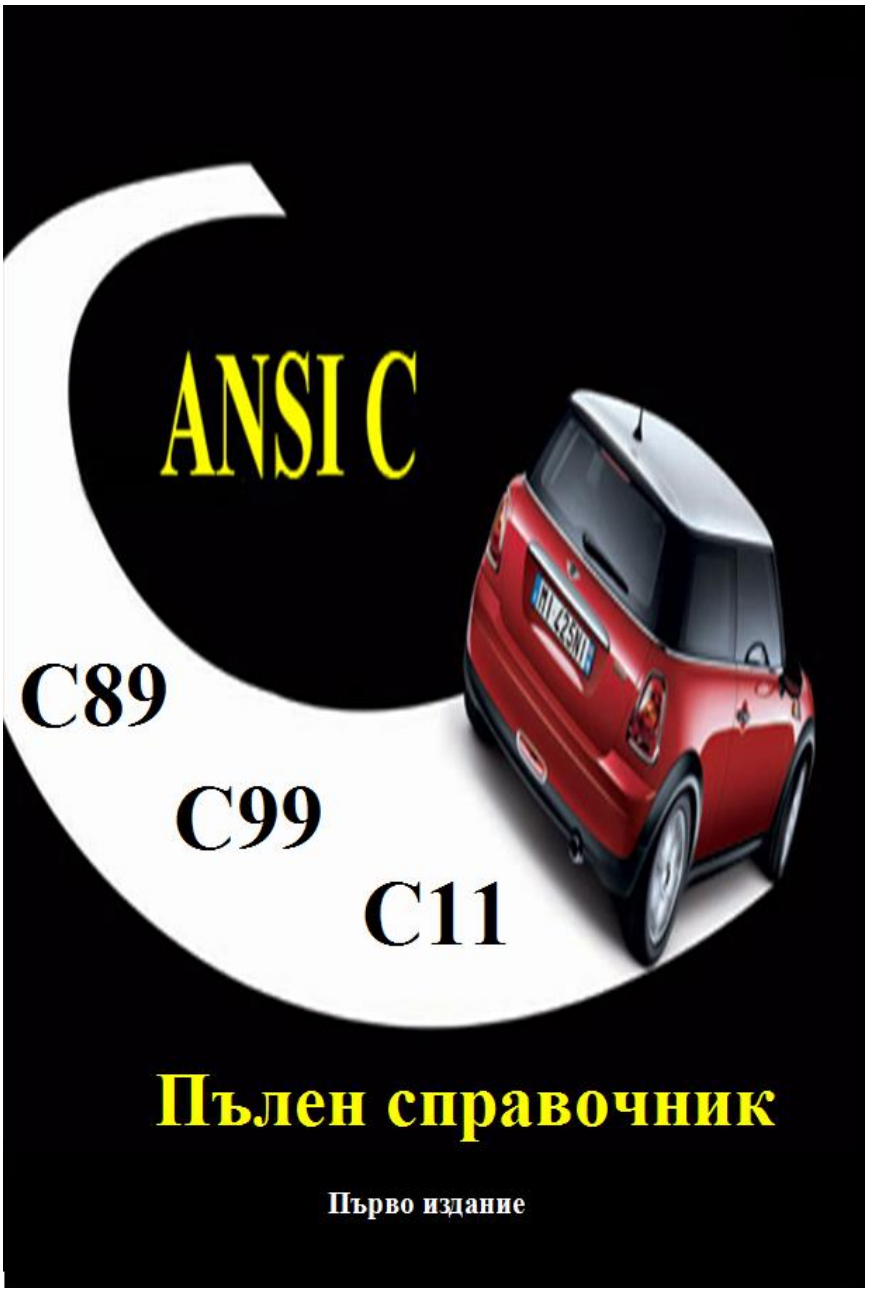
6. **Опишете как работят директивите #if-#else-#endif.**

   Предпроцесорът изчислява **управляващ-израз** на директивата #if. Ако резултатът е ненулева стойност, програмният текст, заключен между #if и #esle, се подава на компилатора, а програмният текст, заключен между #else и #endif, не се компилира. Ако резултатът е нула, програмният текст, заключен между #if и #else, не се компилира, а програмният текст, заключен между #else и #endif, се подава на компилатора.

7. **Опишете как работи предпроцесорният оператор defined.**

   Ако **име-макрос** е дефинирано с #define, резултатът от оператора defined е 1, а ако не е дефинирано или е бил отдефиниран с #undef, резултатът е 0.

Очаквайте следващата книга от поредицата.



Тази книга описва C в детайли и е предназначена за хора, които са запознати с основите на езика, но искат да задълбочат своите познания. Книгата е подготвена така, че може да се използва и като справочник от опитни C-програмисти.

C стандарти

Езикът C е стандартизиран първо през 1989 г. от ANSI (American National Standards Institute) комитета. През 1990 г. този стандарт е приет и от международната организация ISO (International Standards Organization). Затова той се нарича още ANSI/ISO стандарт на езика C. Тези два стандарта са еднакви и често се наричат C89 или C90 и се означават съответно като **ANSI X3.159-1989** и **ISO/IEC 9899:1990**. 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 C95)

ISO/IEC 9899:1990/Cor 2:1996

През 1999 г. е приет нов стандарт **ISO/IEC 9899:1999**, който заменя предходния с всички корекции и добавки. Този стандарт е известен още като C99. 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 2011 г. е приет нов стандарт **ISO/IEC 9899:2011**, формално наричан C11.

Стандартът C95 (ISO/IEC 9899:1990/Amd 1:1995) е основа на езика C++, който пък е основа на двата най-популярни езика за обектно-ориентирано програмиране: C# и Java.