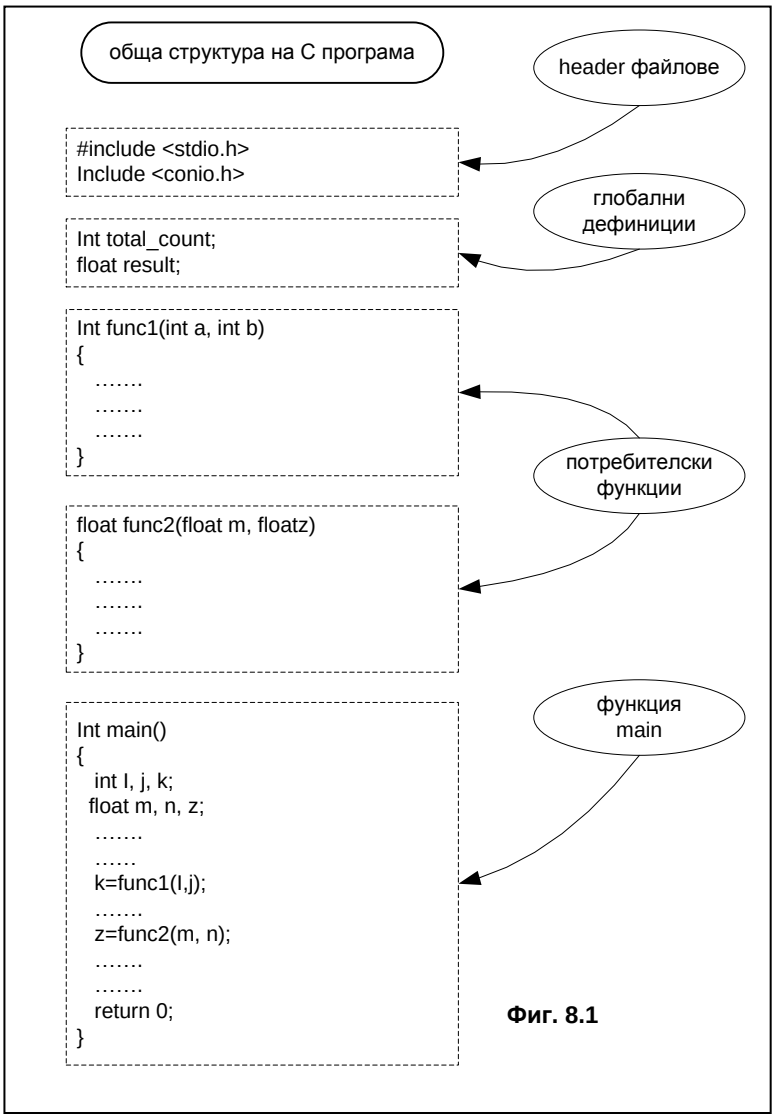
<u>тема 8</u>: Функции. Класове памет.

1. Въведение – понятие за функция

Най-малките единици на които може да се декомпозира една С-програма са функциите. Тя изглежда като линейна структура (фиг.8.1). ***<u>Функцията</u>*** е *именована логически самостоятелна е компонента от потребителката програма, проектирана за решаване на определена задача, която може да обменя данни с други за нея (външни) програмни единици (функции).*

Функциите са два вида – библиотечни и потребителски. Библиотечните функции са структурирани по функционален признак в отделни header файлове (например – вход/изход – **scanf(..)**, **printf(…)** – във файла **stdio.h**). За тяхното използуване в потребителските програми е достатъчно в тях да се добавят препроцесорните директиви за включване на съответните header файлове (фиг.8.1), в които се намират декларациите им. Те са (декларациите) достатъчни за С-компилатора при генериране на изпълнимия код на потребителската програма. Свързващия редактор излича изпълнимия им код от библиотеката с изпълнимите кодове (LIB) при създаване на изпълнимия код на потребителската програма (EXE).



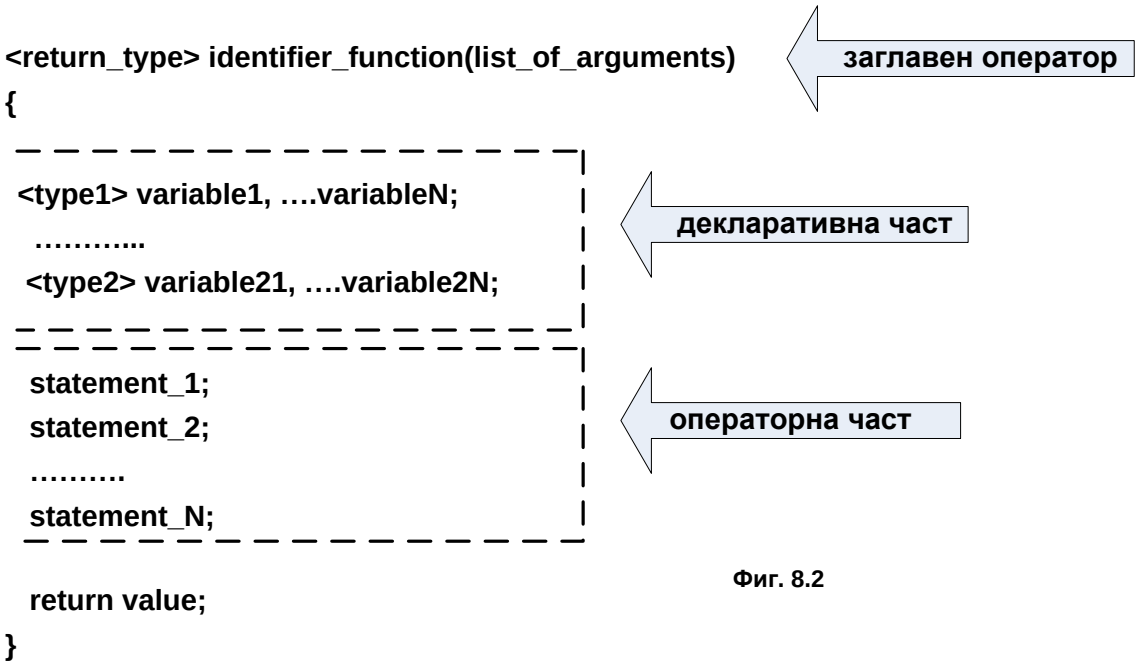
Фиг. 8.1

Използуването на потребителски функции се налага от необходимостта за многократни еднотипни обработки върху различни множества от входни данни. Например – еднотипната обработка на 3 целочислени масива би увеличило изпълнимия код на потребителската програма с описанието на еднакъв в алгоритмично отношение код върху еднотипни данни. При реализирането на функция за обработка на един масив и последващото й използуване (трикратното и извикване във функцията **main** ще намали кода й), чрез изпълнение на един и същи код (кода на функцията).

Анализирайки **source** кода от фиг.8.1 и използувайки абстракцията на т. нар. **черна кутия** функцията е самостоятелна единица преработваща постъпилите входни данни (подадени чрез входните параметри) по вложения в нея алгоритъм до определено множество от изходни данни. Ако се използува аналогията от математическа гледна точка функцията може да се раглежда като автомат работещ по закона**: f(arg1,arg2,…)**, т.е функция зависеща от определен брой аргументи.

~~Понятие за функция е дадено в тема 1. при разглеждане на общата структура на С-програма.~~

Съгласно стандарта на езика С дефиницията на функцията (пълното описание) се осъществява с формата показан на фиг.8.2, състоящ се от заглавен оператор и тяло на функцията (блок - съставен оператор – {….}):



Фиг. 8.2

<u>където:</u>

<return_type>	- тип на връщания резултат от функцията при нейното изпълнение;
identifier_function	- идентификатор (име) на функцията;
<list_of_arguments>	-списък от аргументи(параметри) на функцията:

<u>Забележки:</u>

1. Типът на функцията може да бъде един от предефинираните типове в езика С (**int, float**, …) или потребителски тип дефиниран с оператора **typedef** . Ако потребителската функция не връща резултат то тя се описва с тип **void**, при което в дефиницията й може да се пропусне оператора **return** в нейния край.

2. Типът на функцията е незадължителен елемент в заглавния оператор на функцията, т.е може да се пропусне, но тогава функцията се счита за функция връщаща резултат **int.**

3. Идентификаторът на функцията се подчинява на общите правилата в езика С за запис на идентификатори , т.е. буква, последвана от буква или цифра, без наличие на специални символи.

4. Списъкът от аргументи има вида: **<type> arg1, <type> arg2, ….** Той не е задължителен и зависи от функционалността на проектираната функция – може да е празен или от тип **void.** Той се нарича списък на формалните аргументи (параметри), които се заменят с фактически в точката на извикването на функцията.

5. Декларативната част (ако е необходима) се намира в началото на блока на функцията и не е задължителен.Той е последван от процедурната (операторната) част на функцията, в коятро се описва алгоритъма за съответната обработка. В тази част не се разрешават ново дефиниции или декларации.

6. Идентификаторът на функцията е особен вид указател ” указател към функция ” . Той съдържа адреса на входната точка на изпълнимия код на функцията.

7. При всяко извикване на функцията компилаторът на С генерира код за предаване на управлението на изчислителния процес с възврат в точката на извикване, а след завършването й управлението се предава към следващия изпълним оператор след точката на извикване.

8. *Извикването на функция* е операция с най-висш приоритет.

2. Дефиниция и декларация на функция

Потребителските програми обикновено се състоят от определен брой малки по обем потребителски функции, което от една страна осигурява по-добрата читаемост на програмата, а от друга по-лесната и настройка. При използуване на разделно компилиране (виж. Т.5) потребителските функции се оформят в отделни файлове, което улеснява последващото им използуване в други потребителски проекти.

В езика С при описанието на различните програмни обекти (самостоятелни програмни единици, каквито са например променливите, структурите и функциите) се използуват два начина - **дефиниция** и **декларация,** между които има съществена концепсуална разлика:

- ***Дефиницията*** *е пълното и описание програмен обект, при обработката на който компилаторът на С заделя памет за съхраняване текущите му стойности по време на достъпа му до него*.

- ***Декларация или прототип на функция*** *е формално описание на програмния обект, който компилаторът на С използува при обработката на потребителската програма в етапа на генериране на кода и синтактическия анализ на изходния код –без заделяне на памет*.

Декларацията на функцията, съгласно синтаксиса на езика С се реализира със формата:

<return_type> indentifier_function(list_of_arguments);

т.е. запис на заглавния оператор на функцията, последван от **;**

Декларациите на потребителските функции се записват в началото на блока за декларации съгласно структурата на С-програмата (тема 1). От прототипа на функцията компилаторът на С извлича необходимата информация за синтактичния разбор и генериране на код в точката на извикването й, дори когато все още не е генерирал кода от дефиницията. Правилото в езика С е, че на етапа на компилация , в точката на извикване на потребителска функция, компилаторът трябва да има информация за нея, чрез декларация или дефиниция. При наличие само на декларации етапа на *компилация* завършва успешно – без съобщения за грешка, но на следващия етап – *свързващото редактиране* – свързващия редактор (линкерът) ще даде съобщение за грешка, тъй като не може да намери междинния код на потребителската функция.

3. Взаимодействие между функции

Всъщност С програмата е съвкупност от равно поставени самостоятелни програмни единици, които си взаимодействат по между си, като при това обменят една със друга информация в най-общ смисъл. Извикването на функция е операция с най-висш приоритет от всички операции в езика С. Всяка потребителска функция може да извиква друга функция. Първата се нарича ”*извикваща”*, а в точката на извикване втората се нарича ”*извикана”*. Това взаимодействие може да се разгледа в два аспекта:

8.1. ***предаване на данни*** между извикващата и извикана функции;

8.2. ***предаване на управлението*** между извикващата и извикана функции;

За по-подробното разглеждане на това взаимодействие нека да разгледаме конкретен пример на проста С програма състояща се от три функции – фиг 8.3. Две потребителски функции: (1) - **int area** , (2)- **int perimeter** , като и двете имат да формални параметра страните на правоъгълник, като първата изчислява лицето а втората – периметъра му и главна функция (3) - **main**, която зарежда данни от клавиатурата и извиква последователно двете функции. Примерното изпълнение на програмата **test1** е показано на фиг.8.4

```
#include <stdio.h>
int a, b;
int area(int side1, int side2)
{
  int surface;
  surface=side1*side2;
  return surface;
}

int perimeter(int side1, int side2)
{
  int p;
  p=2*side1+2*side2;
  return p;
}

int main(void)
{
  int i, j, result;
  printf("\nside1="); scanf("%d",&i);        /* 01 */
  printf("side2="); scanf("%d",&j);         /* 02 */
  result=area(i,j);                         /* 03 */
  printf("result area=%d\n",result);        /* 04 */
  result=perimeter(i,j);                    /* 05 */
  printf("result perimeter=%d\n",result);   /* 06 */
  printf("a="); scanf("%d",&a);             /* 07 */
  printf("b="); scanf("%d",&b);             /* 08 */
  printf("area(a,b)=%d\n",area(a,b));       /* 09 */
}
```

фиг.8.3

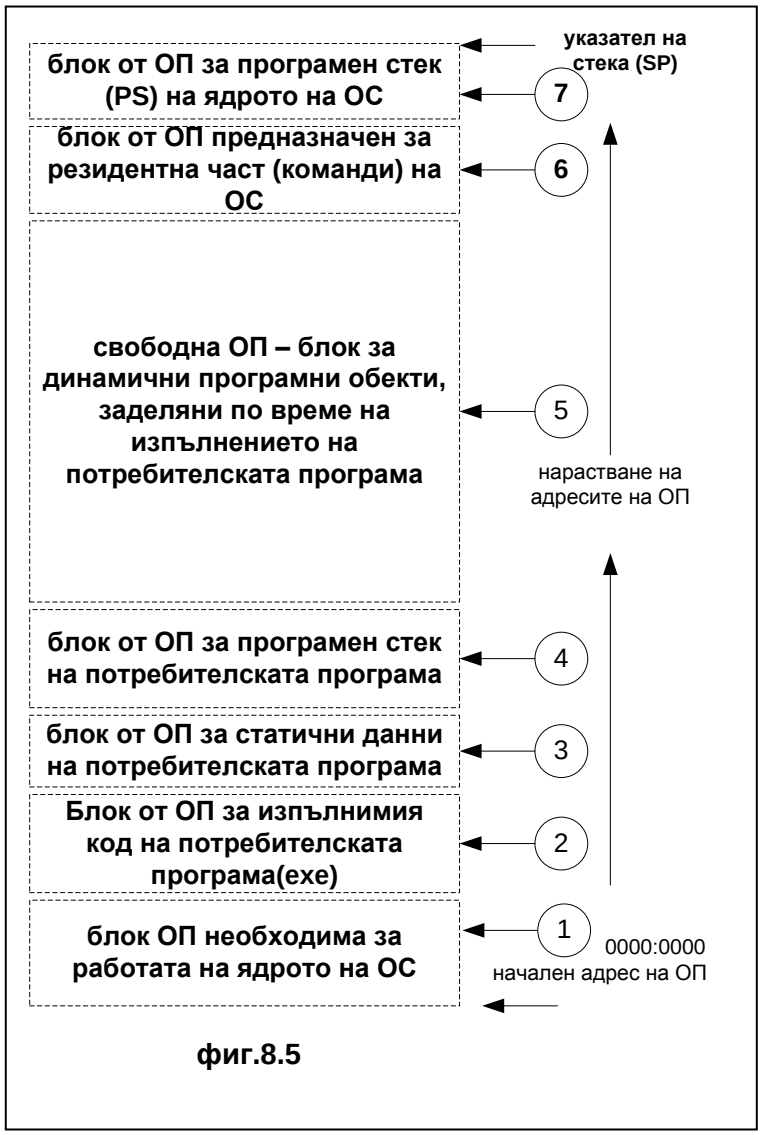
Примерно изпълнение на програмата:

```
side1=1 2
side2=6
result area=72
result perimeter=36
a=3
b=4
area(a,b)=12
```

фиг.8.4

Предаването на управлението в определена точка на дадена функция към друга функция се нарича обръщение към функция или извикване на функция. При стартирането на програмата **test1**, изпълнението й започва от функцията **main** (коментарен ред 01), като след въвеждането на конкретните стойности за променливите **i**, **j** (примерното изпълнение програмата е показано на фиг.8.4) се извиква функцията **area(i,j)**. - коментарен ред – 03. Това предизвиква предаване управлението на изчислителния процес към кода на функцията **area**, като преди това компилаторът е генерирал последователност от машинни инструкции с които съхранява адреса на следващата машинна инструкция, с която започва реализирането на следващия оператор от кода на **main** (**result=…** изпълнението на оператора за присвояване на стойността която се изчислява от **area** . В момента на предаване на управлението към изпълнимия код на функцията може да се счита че се активира самостоятелна програмна единица, работеща по алгоритъма записан със С-операторите в тялото й. По време на изпълнение на функцията трябва да се отбележи, че всички програмни обекти (променливи, константи), дефинирани в тялото на извикващата функция са недостъпни (невидими) за извиканата функция (в конкретния случай това са променливите **i, j, result**). Последният оператор от тялото на **area return p;** завършва изпълнението й, като заедно с това зарежда възвратния адрес за продължение изпълнението на **main** – зареждане на променливата **result** с изчислената стойност. Последва извеждането на резултата на екрана, последващо извикване на функцията **perimeter** и т.н...

За да се разбере в по-голяма дълбочина механизмите за взаимодействие между функциите в една потребителска С-програма е необходимо да се разгледа как се изпълняват изпълнимия й код в средата на операционната система(**ОС**) и как се разпределя наличната оперативна памет (**ОП**).



фиг.8.5

В ядрото на **ОС** съществуват два основни модула, които управляват така да се каже разпределението й – *системна зареждаща програма* и *програма за динамично разпределение на паметта*.

При еднопрограмните **ОС** (каквато е например **MS DOS**) разпределението на наличната **ОП** е показано на фиг.8.5. След началното зареждане на ядрото на **ОС** тя създава два основни блока : блок за временно съхранение на данните необходими за функционирането на ядрото на **ОС** (1) и блока за поддържане на системния стек на **ОС** (7).

Всяка **ОС** притежава някакъв набор от команди (директиви – например за **MS DOS** – **dir, cd** …) и фонова програма, реализираща тяхното изпълнение (за **MS DOS** това е **command.com**). Този дял от **ОП**, наричан резидентна част (фиг.8.5 – 6) на ядрото, която се зарежда при началната инициализация на **ОС**, присъства непрекъснато в **ОП**.

При мултипрограмните **ОС**, като **Windows, Unix, Linux** и др., работещи в режим на времеделене, системната зареждаща програма заделя за всеки активиран процес отделен блок памет (обикновено наричан дял на **ОП**), който с известна условност има структурата показана на фиг.8.5.

Независимо от особеностите на различните ОС на които се изпълнява приложното програмно осигуряване изпълнимите модули се изграждат от съответните свързващи редактори със сходна структура. Стартирането на изпълнимия код на потребителската програма, се изпълнява от *Системната Зареждаща програма* на ОС, което се осъществява на няколко етапа:

- зареждането на изпълнимия код (намиране на подходящ по дължина свободен блок от **ОП**);

- последващата му обработка, налагаща се от факта че съвременните компилатори генерират т.н. *преместваем код* – т.е. след прехвърлянето на изпълнимия код в ОП, всички обръщения в асемблерския код към абсолютни адреси трябва да се коригират спрямо началната точка (адрес) на конкретното зареждане;

- настройка на възвратния адрес след завършване на потребителската програма към ядрото на **ОС** (**command.com**)

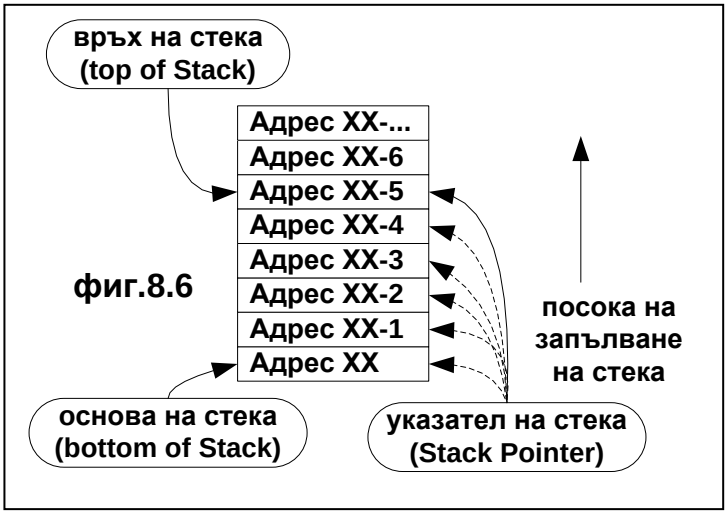
- предаване на управлението на изчислителния процес във входната точка на функцията **main** – първия изпълним машинен код от нея;

Изпълнимият код на потребителската програма се състои от няколко фрагмента;

- ***блок, съдържащ асемблерския код на програмата***, включващ кода на функцията **main** и на всички потребителски и библиотечни функции използувани в нея – блок 2 от фиг.8.5;

- ***блок на статичните*** (глобални***) данни*** – блок 3 от фиг.8.5. Това е област, която компилаторът е заделил за данни (в зависимост от типа им – т.е броя байтове с които се представят), които съществуват и са достъпни по време на изпълнението на потребителската програма (виж т.X от тема 8);

- ***блок на програмния стек*** – блок 4 от фиг.8.5. Стекът *е динамична структура* от еднотипни данни, която компилаторът е заделя от **ОП** за временно съхранение на данни, управлявана по дисциплината **FILO** (First Input, Last Output - първи влязъл, последен излязъл). Организацията на структурата stack е показана на фиг.8.6. Стекът е организиран така че осигурява следните 3 операции:



фиг.8.6

- инициализация на стека **(Init Stack**);

- вмъкване на елемент в стека (**Push Stack**);

- извличане(сваляне) на елемент от стека (**Pull Stack**);

Постъпването на нови елементи за съхранение и извличането на съхранените в него данни се осъществява само през върха на стека, чиято текуща стойност се съхранява в отделен регистър на централния процесор – **stack pointer(SP)**.

Стекът е част от **ОП**, чрез която извикващата и извиканата функция си предават данни и управлението на изчислителния процес. Компилаторът на С, а в последствие и свързващия редактор добавя машинен код към изпълнимия код на потребителската програма, с която регистъра **SP** на процесора се инициализира с началния адрес на областта предназначена за стек, т.е:

stack pointer:= bottom of stack;

Временното съхранение и последващото извличане на информация от стека се осъществява само през неговия текущ връх, които се поддържа като стойност в указателя на стека (особеност е че стекът се инициализира със най-старшия адрес на сегмента от паметта, предназначен за него и при вмъкване на елемент той се намалява като стойност, а при извличане се увеличава - фиг.8.6). При вмъкване и извличане на определен тип данни в стека, т.е при копиране на текущите стойности на фактическите параметри при извикване на функцията, макар и байтово ориентиран, той се коригира със толкова байтове с колкото се представят те в **ОП** (например данни тип **int** – 4 байта). При операцията **push** абсолютната текуща стойност на стека се намалява а при операцията **pull** се увеличава с толкова байта с колкото съответния обект се представя в **ОП**.

- ***блок на свободната памет от*** **ОП** (**heap** - куп) – блок 5 от фиг.8.5. Това е област, която се резервира за динамични програмни обекти (структури, списъци, стекове, декове и др.), чиито размер се определя по време на изпълнението на програмата. Тази област от **ОП** се управлява от модула *за динамично разпределение на паметта* от ядрото на **ОС**);

4. Класове памет и област на видимост

Всеки програмен обект (ПО) - променлива, константа или функция, който се описва в езика С, може формално да се опише с тройката:

по:=(идентификатор,тип,стойност)

където:

- ***идентификатор*** е характеристиката на обекта, която осигурява възможността за достъп и опериране с него със средствата на програмния език от високо ниво(езика С);

- ***тип*** е характеристиката на обекта, която осигурява е свързана с машинното представяне на данните в ОП – пряко е свързана с броя байтове от ОП, в която се съхранява текущата му стойност;

- ***стойност*** е количествената характеристиката на обекта, т.е. неговата текуща стойност;

Компилаторът на С осъществява връзката между имената на обектите и адресите от ОП, в които се съхраняват текущите им стойности. В процеса на компилация той, наред с гореописаната информация за името, типа и евентуалната инициализация на променливите, трябва да има и допълнителна информация, както за местоположението на променливата в изпълнимия код на потребителската програма, така и за начина на достъп до нея. В този смисъл в стандарта на езика С дефинирано понятието **name scope** (област на видимост – обхват или обсег), което се свързва с две допълнителни характеристики на програмните обекти: **storage (*клас памет)*** и **duration** (***период на активност*** - време, през което обекта съществува).

табл. 4.1 – класове памет, дефинирани в езика С

клас памет (дефиниция)	**описание на типа памет**
Auto	автоматичен – локален обект по дефиниция – необходимата памет се заделя в областта на програмния стек
Static	статичен – необходимата памет за обекта се заделя в сегмента за статични данни и обекта съществува през цялото време на изпълнение на програмата
register	регистров – за обекта не се заделя памет от ОП, а той се съхранява в някой от вътрешните регистри на централния процесор
extern	външен – не се заделя памет за обекта

Класът памет осигурява информация за местоположението на обекта (променливата) и пряко е свързан с тяхната "**видимост**", т.е. възможността за достъпа до тях от отделните програмни единици, изграждащи С програмата. **Периодът на активност** осигурява информация за времето, през което в **С** програмата съответната променлива физически съществува или другояче казано докога към идентификатора (името) има прикрепена област от ОП, в която се съхранява текущата им стойност. В стандарта са дефинирани четири типа клас памет (табл. 4.1):

8.3. (1) - *scope block* - област на видимост блок (съставен оператор);

8.4. (2) - *scope function* – област на видимост дефиниция на функция;

8.5. (3) - *scope function prototype* – област прототип на функция;

8.6. (4) - *scope file* – област файл;

Променливите в една С програма могат да се разглеждат като външни и вътрешни по отношение на функциите, от които тя е изградена, т.е могат да се разглеждат като *локални* или *глобални* програмни обекти. Правило в езика е: всички вътрешни променливи дефинирани в областите на видимост *scope block* (1) , *scope function*(2) и *scope function prototype*(3) са от тип **auto**, ако в явен вид

за тях не е определен друг тип клас памет. За тях се заделя памет от областта на програмния стек, т.е те са видими (достъпни) само в блока, в който са дефинирани, като времето на съществуване съвпада с времето на видимост и могат да се считат за ”*локални*” за блока. Такива са променливите **int surface** от функцията **area(...)** и **int p** от **perimeter(...)** – фиг.8.4.

Когато една променлива се дефинира извън даден блок тя се счита за външна за него и обикновено нейната област на видимост е *scope file*(4). Тя се разполага в областта от изпълнимия код на програмата предназначен за съхраняване на статичните данни. В този смисъл тези променливите могат да се считат за ”*глобални*”. Такива са променливите **int a, b;** – фиг.8.3.

Забележки:

1. Автоматичните променливи не се инициализират автоматично в момента на тяхната дефиниция, тъй като те се разполагат в областта на стека и получават случайна стойност.

2. В зависимост от режима на работа на компилатора или по-точно от начина на оптимизация на генерирания от него код, той може да предостави някои от вътрешните регистри на централния процесор (**ЦП**) за променлива от тип **auto.** Това зависи обикновено от броя на активните променливи.

3. Типът **registe**r може да се разглежда като частен случай на тип **auto.** Когато той се употреби в явен вид при дефиницията на променлива, това указва на компилатора да задели някои от вътрешните регистри на **ЦП** за съхранение на променливата. Регистровите променливи могат да бъдат прости типове данни (не могат да бъдат данни от структурен тип – масиви или структури). Когато компилаторът е генерирал код, при който вътрешните регистри на ЦП са заети, въпреки дефиницията за регистрова променлива, той заделя памет от стека. Предимството при използуване на регистрови променливи се състои в това, че се постига по-голямо бързодействие на изчислителния процес, тъй като операциите за достъп до тях е хай-бързо.

4. При променливите от тип **auto** и **registe**r областта на видимост и времето на съществуване – периода на активност се покриват – в областта на блока в който са дефинирани.

5. Обектите от тип **static** се заделят в ОП предназначена за съхраняване на статични данни. Те са достъпни от всички програмни обекти, за които те са външни (например променливите **int a,b;** от фиг.8.4). Техният *период на активност* е времето на изпълнение на програмата, т.е. те съществуват до завършване на програмата, но достъпът до тях е областта на файла в който се дефинират. От примера на фиг.8.4 (променливите **int a,b;**) са достъпни както в блока на функцията а **main()**, така и във блоковете на функциите **area()** и **perimeter()**.

6. Всички дефинирани извън блоковете на функциите в този смисъл могат да се считат като глобални и за тях компилаторът заделя памет в областта за статични данни, т.е дори явно да не са дефинирани от тип **static**. Ако няма явна инициализация на статичните данни в точката на тяхната дефиниция, компилаторът генерира допълнителен код за тяхното нулиране.

7. Променлива от тип **static** може да се дефинира и в блок на функция като локален обект. За нея се заделя памет в областта за статични данни но видимостта и е само в локалната област на функцията, в която е дефинирана, т.е областта на видимост и времето на съществуване в този случай не се покриват – макар че съществува локалната статична променлива не е достъпна за другите програмни обекти (функции), освен в блока в който е дефинирана.

8. Програмните обекти дефинирани като клас памет **extern** са външни за блока в които са описани като такива. Записът от вида: **extern <u>name</u>;** е декларация за описването на програмен обект, който е дефиниран в областта на друг файл (- *scope file*(4)). При обработката му компилаторът не заделя памет за обекта, а го използува като мнемонично име при генериране на кода на програмата – адресът на **extern** променливата се настройва в етапа на свързващото редактиране. Този тип описание се използува преимуществено при реализиране на потребителско програмно осигуряване при така нареченото разделно компилиране – когато изпълнимият код се изгражда от отделни C-source файлове. При необходимост от използуването на глобални променливи, дефинирани е един от файловете, изграждащи потребителската програма, в друг от файловете се използува декларацията **extern.**

5. Обмен на информация между функции

Обменът на данни между функциите в една С-програма може да се осъществи по два начина:

- използуване на общи (външни) променливи

При дефиниране на глобални променливи достъпът до тях в блока на функциите е възможен, тъй като те са видими. Това в известен смисъл ограничава независимостта (мобилността) им и последващото им използуване в други програми или включването им като единици в други проекти, използуващи разделна компилация. Това ограничение се налага от факта че използуването на външни променливи в блока на функция се осъществява, чрез техните конкретни идентификатори и при пренасянето й в друг проект налага задължително дефинирането на външен обект със същото име в новия програмен проект.

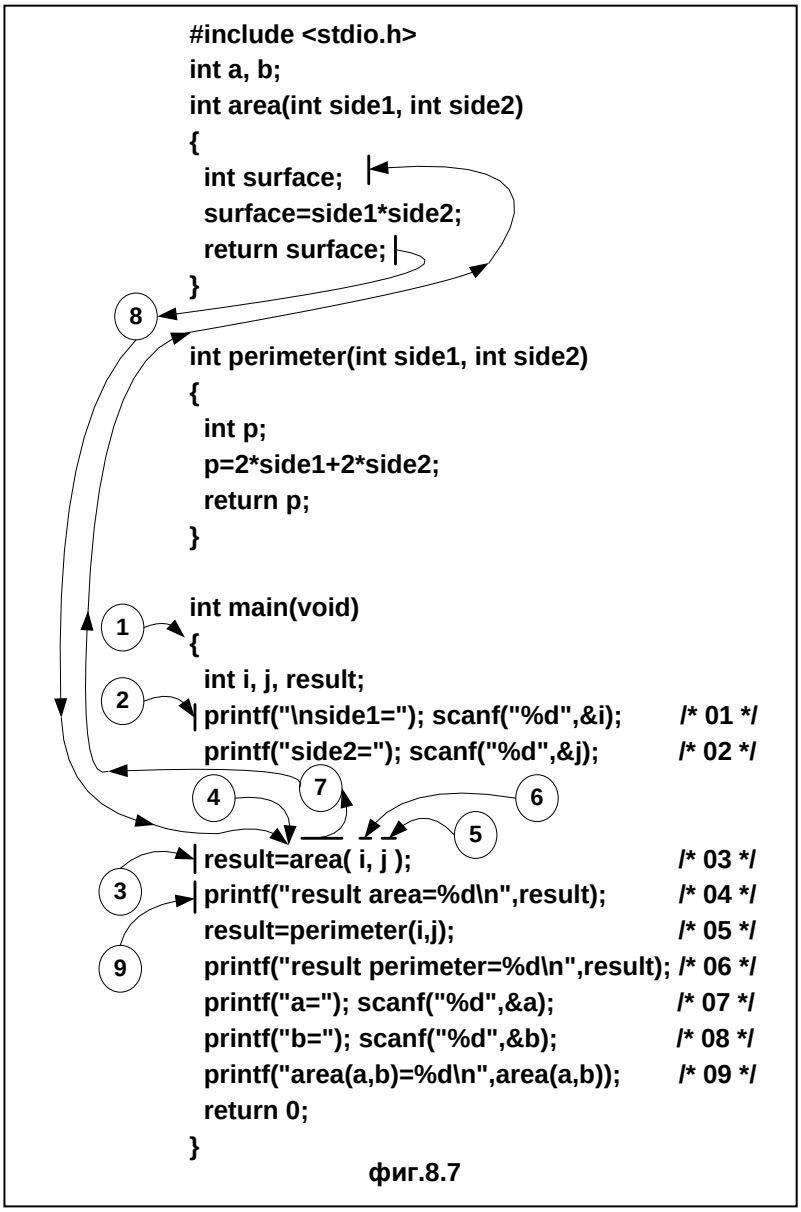
Макар и по-ограничен в използуването си този подход за предаване на данни може да се разглежда като целесъобразен в случаите, при които се реализират по-големи групи от потребителски функции, използуващи едни и същи глобални променливи за предаване на данни – това съкращава броя на формалните им параметри и осигурява пряк достъп до тях. Типичен пример за използуването на този подход е съвкупността от входно/изходните библиотечни функции (**prinf(…)**, **scanf**(…), **fopen(…)**), които зареждат променливата **errno** след всеки обмен, без да си я предават като параметър.

- предаване на данни чрез входни аргументи на извиканата функция

Това е основния подход, който се използува при за предаване на данни между извиканата и извикващата функции. Това предаване на данни, съгласно стандарта на С се **<u>предават по стойност (Call By Value)</u>**. При него формалните параметри, описани в прототипа на функцията получава посредством механизма на извикване **<u>копия на стойностите на фактическите параметри</u>**, които се използуват при изпълнението на извиканата функция. Това предаване (копиране) се осъществява чрез програмния стек (**SP**). По този начин в С се осигурява възможността за връщане на повече от една стойност в извикващата функция – ако формалния аргумент е указател, копирайки го по стойност в **SP** , извиканата функция получава достъп до адреса на фактическия аргумент и би могла да промени неговата стойност. Ако формалният параметър е указател към данна от агрегатен тип (масив или структура), то извиканата функция получава достъп до нея и може да върне съвкупност от данни в зависимост от алгоритъма й за функциониране.

Детайлизацията на механизма за взаимодействие между извикващата и извиканата функция при предаване на данни и използуването на програмния стек в този механизъм е показана на фиг.8.7, използуващ изпълнимия код на примерната програма от т.8.3. и фиг.8.4. В него е показан

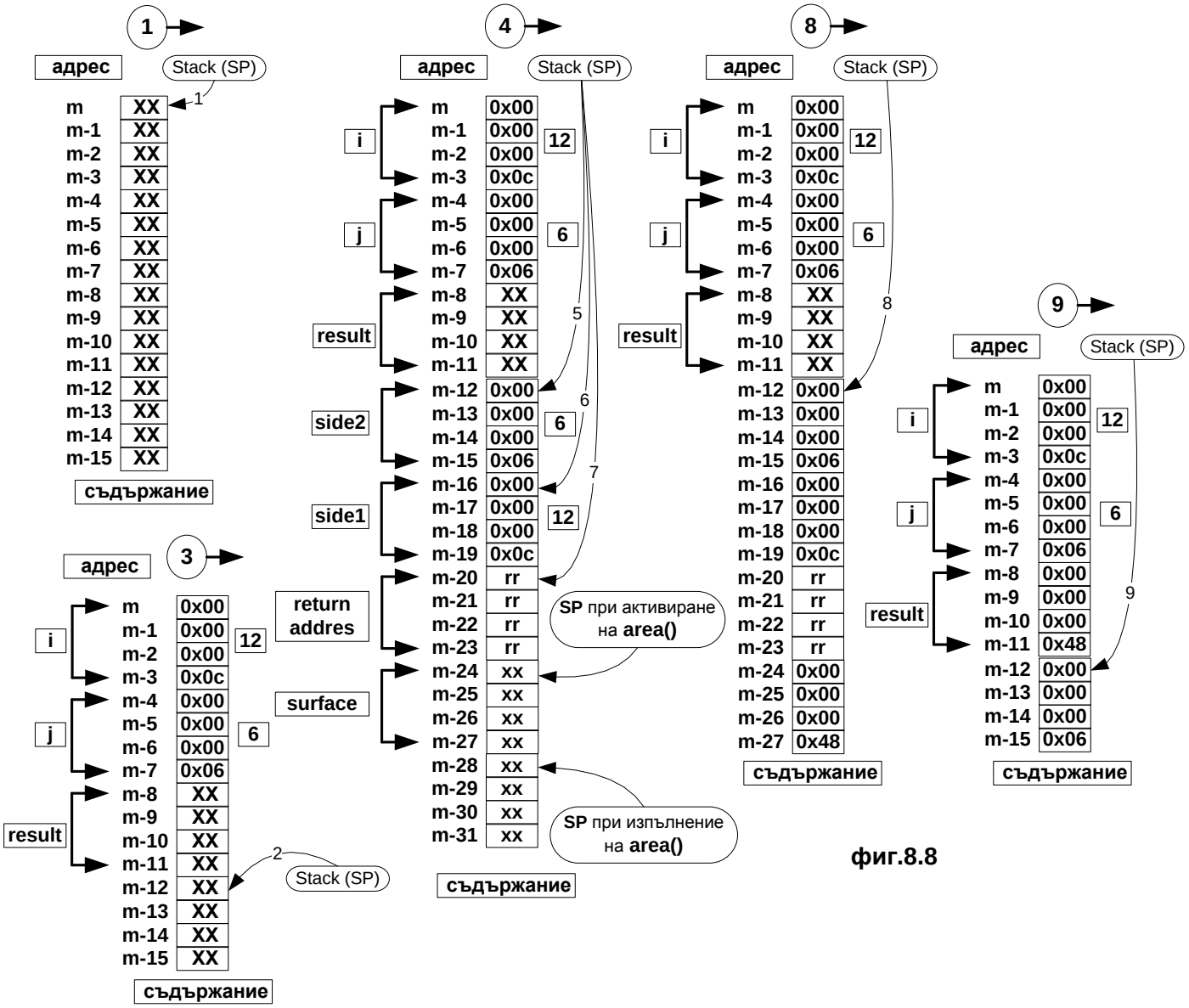
схематично процеса на взаимодействие и текущото състояние на **SP.** Трите функции: **area(…), perimeter(…)** и **main()**, са разположени в сегмента на изпълнимия код на програмата е показан в начално състояние фиг. 8.8-1. Входната точка на изпълнимия код е функцията **main(),** в която **SP** е инициализиран със адреса **m** от ОП. В последващата дефиниционна част компилаторът генерира код, с който се заделя необходимата памет за локалните променливи **i**, **j**, **result**, като ги разполага в областта на стека в реда в които среща техните дефиниции – фиг.8.8 – 2, като те имат случаини стойности (означени с **xx**). Преди изпълнението на първия процедурен ред **SP:=m-12** (коментарен ред /*01*/). Следва извикването на библиотечните функции **printf** и **scanf,** с които се осигурява диалога с потребителя за зареждане на входните променливи **i** и **j** с конкретни стойности от клавиатурата. Изменението на стековия указател при тези извиквания не е показан на фиг.8.8, но той не се различава от по-долу разгледания при извикването на потребителската функция **area()** – т.3 от фиг.8.7. Подробното разглеждане на взаимодействието между извикващата **main()** и извиканата **area()** функция започва с ред **/* 03 */** от кода на С-програмата – т.3 от фиг.8.7, а текущите стойности на променливите са дадени на фиг.8.8-1. Той по същество е оператор за изчисление на израз (на променливата **result** трябва да се присвои върнатия от **area()** резултат. Операцията присвояване (фиг.8.7-4) се извършва съгласно синтаксиса на С от дясно на ляво, което би следвало да означава, че компилаторът трябва да генерира код за извикване на потребителската функция **area().** Ако приемем, че е извършено въвеждането на входните данни от фиг.8.4, т.е. **i=12** и **j=6**, състоянието на стековата памет е показано на фиг. 8.8-2. На нея е показано примерното разпределение на адресното пространство отделено за стека, както следва: за променливата **i –** начален адрес **m**, за променливата **j** съответно **m-4** и **result m-8.** Текущата стойност на стека е **sp:=m-12**. В точка 3 от фиг.8.7 променливите **i**, **j** вече имат конкретни стойности, докато променливата **result** е неопределена (паметта е показана в състояние **xx**). Преди да се извика(активира) функцията **area(),** компилаторът генерира код, с който се подготвя това извикване, състоящо се в следните стъпки (фиг. 8.7 – точки 5,6 ,7):



фиг.8.7

- заделяне на памет за формалните параметри (**side1**, **side2**) от областта на програмния стек, последвано от копиране стойностите на фактическите параметри(**i** и **j**). Това копиране се извършва като параметрите се обработват от дясно на ляво, т.е. първо в стека се заделя памет и се копира параметъра **side2**(адрес **m-12**), а след това за **side1** (адрес **m-16) –** фиг.8.8 – 4,5.

- извикването на функцията **area()** по същество е предаване на управлението към самостоятелен програмен обект, завършващ с оператор **return** (завръщане от ”изпълнение на функция”), т.е. трябва да е възможно възстановяване на изчислителния процес в точката (инструкцията) непосредствено след извикването й. В конкретния случай е изпълнението на операцията за присвояване (**result** = … - фиг.8.8-9). Това се реализира като компилаторът генерира код за съхраняване на възвратния адрес в стека (адрес **m-20**), с които ще продължи изпълнението на функцията **main()**.



фиг.8.8

- на следващата стъпка се активира функцията **area()**, която изчислява лицето на правоъгълника със страни **side1**, **side2** – фиг.8.7-7. В този момент указателят на стека е настроен към адрес **m-24**, а след като се задели памет за локалната променлива **surface** (адрес **m-28).** Това е и актуалния му адрес по време на изпълнението на процедурната част на функцията **area()**. Съществено на тази стъпка е да се отбележи, че променливите **side1**, **side2** и **surface** са от клас *scope function* и по същество са **auto** променливи.
- при достигане на последния оператор **return** на функцията **area()** се осъществява зареждането на програмния брояч на ЦП с адреса за възврат, запомнен в стека, така че изпълнението на следващата ще се извлече от този адрес. Това по същество означава че управлението се връща

отново във функцията **main()** – фиг.8.7-8 в която се изпълнява присвояването на изчислената стойност в променливата **result**.

Гореописаният механизъм за извикване, изпълнение и завръщане от функция се повтаря в **main()** в последващите програмни редове - съответно за извикването на функцията **perimeter (i,j)** – коментарен ред /* 05 */ и **area(a,b)** - /*09 */.

Извикването на функция съгласно стандарта на езика С се подчинява на следните правила:

- преди активирането на дадена функция, *извикващата функция* осъществява зареждането на стойностите на фактическите аргументи в програмния стек в ред от ***дясно наляво***, т.е. най-десния аргумент се вмъква пръв в стека;
- всички стойности се ***копират по стойност*** – с други думи *извиканата функция* получава от *извикващата* **копия** на фактическите аргументи, които са локални и видими само в нейния блок;
- *извикващата функция* зарежда в стека възвратния адрес за възстановяване на изчислителния процес след изпълнението на *извиканата функция*;
- *извикващата функция* коригира върха на стека в точката на възврат;
- с механизма на **return** (възврат от изпълнение на функция) може да се върне само един една стойност в точката на извикване – ако тя е от прост тип (int, float,..) се съхранява в някои от вътрешните регистри на ЦП, а ако е от структурен тип се предава чрез програмния стек;
- възможност за предаване(връщане) на повече от една стойност от извиканата към извикващата функция в езика С се реализира, чрез използуване на *указатели като формални аргументи* в дефиницията на потребителската функция;
- предаването на адреси като аргументи от извикващата функция към извиканата *също става по стойност* (в стека се копират адресите на програмните обекти, които са притежание на извикващата функция) , което по същество дава достъп на извиканата функция до има достъп до програмни обекти, притежание на извикващата функция.
- предаването на адреси като аргументи е единствения начин за достъп до данни от агрегатен тип, каквито са масивите, масиви от структури или масиви от дефинирани от потребителя данни;

Пример за връщане на повече от една стойност е показан на фиг.8.9. В него е дефинирана функцията **float area_square (float *size1, float size2, int per_cent)**, чрез която се пресмята лицето на правоъгълна област като в последствие се правоъгълника се модифицира до квадрат, като преди това лицето се коригира с процента указан в **per_cent**. Преизчислената стойност на страната на квадрата, с коригираното лице се връща чрез формалния параметър ***size1**, а самата функция с механизма на **return** връща коригираното лице на правоъгълната област. Във функция **main** след зареждане на входните за **i,j** и **percernt** се извиква функцията **area_square**.

Преди да се генерира кода за извикването й, компилаторът копира по стойност фактическите параметри в програмния стек в следния ред - първо стойността на променливата **percernt=76** (съгласно примерното изпълнение, показано на фиг.8.10), след това се вмъква стойността на **j**(6) и последен се вмъква адреса на променливата **i**(не нейната конкретна стойност). Накрая в стека се записва възвратния адрес на операцията за присвояване и чак тогава се генерира кода за активиране на функцията **area_square.** В тялото си тя използува индиректно стойността на формалния параметър ***side1**, т.е. чрез указателя тя има достъп до паметта, в която е разположена променливата i на функцията **main()**.

Изразът ***sise1=(float) sqrt(new_serface);** в тялото на функцията осигурява достъпа до променливата и ма функцията **main()**

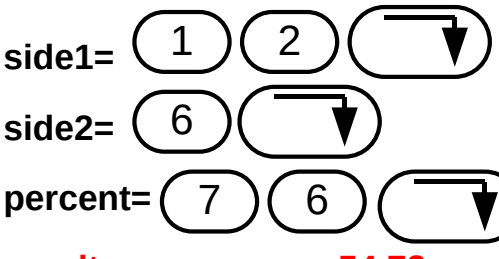
```
#include <stdio.h>
#include <math.h>

float area_square(float *side1, float side2, int per_cent)
{
  float surface, new_serface;
  surface=*side1*side2;
  new_serface=surface*per_cent/100;
  *side1=(float)sqrt(new_serface);
  return new_serface;
}
```

фиг.8.9

```
int main(void)
{
  float i, j, result;
  int percent;
  printf("\nside1=");       scanf("%f",&i);
  printf("side2=");         scanf("%f",&j);
  printf("percent=");       scanf("%d",&percent);
  result=area_square(&i,j,percent);
  printf("result area_square=%.2f -> new square side=%.2f\n",result,i);
  return 0;
}
```

Примерно изпълнение на програмата:



фиг.8.10

result area_square=54.72 -> new square side=7.40

6. Предаване на параметри. Модификатори cdecl и pascal

Като обобщение може да се заключи, че компилаторът на С при обработка на изходния код на потребителската програма, когато срещне операция ”извикване на функция”, генерира две групи машинни инструкции типа в изпълнителния й код:

- зареждане в програмния стек на копия на променливи или константи записани като фактически параметри в списъка с аргументи в точката на извикване, ако той не е празен (**void**). При това

копиране се осъществява и необходимото преобразуване на стойностите съгласно указаните типове на аргументите в прототипа на извикваната функция;

- извикване на функцията чрез машинната инструкция ”извикване на процедура със възврат”.чрез която се предава управлението към кода на извиканата функция, като преди това в стека се записва възвратния адрес за връщане в извикващата функция.

Важна особеност в този процес е това ,че извикващата функция подрежда в стека копираните стойности от дясно на ляво, т.е първи в стека се вмъква (операция **push**) най-десният фактически аргумент , като всеки път указателя на стека се намалява толкова байта, с колкото се представия типа на вмъквания аргумент. Този начин за разполагане на данните които се обменят между извикващата и извиканата функции се нарича “**С-ред предаване на аргументи**”. Явното обявяване на този начин за предаване на данни може да се укаже с модификатора **cdecl**, които се прилага към типа на функцията, като например:

int cdecl func_1(int a, char *t);

Съществува и друга възможност за взаимодействие между извиканата и извикващата функции – т.н. ”**pascal - ред предаване на параметри** “. При него това взаимодействие се осъществява съгласно конвекциите, приети в езика **Pascal,** при които аргументите се предават в обратен ред – от ляво на дясно. Той може да се укаже явно с модификатора **pascal,** указан явно в декларацията на потребителската функция:

int pascal func_1(int a, char *t);

Обикновено компилаторите на С обработват изходните текстове на програмите като по премълчаване считат, че към тях е приложен модификатора **cdecl**. Някои потребителски среди за създаване на С софтуер предоставят системни средства за промяна на режима на обработка, т.е. по премълчаване да се генерира код с приложен модификатор **pascal.**

7. Функции с променлив брой аргументи

В стандарта на езика С е разрешено използуването на и проектирането на функции с променлив брой аргументи. Това позволява допълнителна гъвкавост при реализиране на потребителското програмно осигуряване.

В библиотечните функции използувани от всички компилатори на С има доста примери за използуването на подобен подход при обработката на информация като например – повечето функции за форматиран вход/изход, позволяващи с едно извикване на се обменят данни с един или няколко програмни обекта. Типичен пример са функциите **printf(...)** и **scanf(…)** и много други, които са декларирани в header файла **stdio.h** както следва:

int __ cdecl printf(const char *, ...);

int __cdecl scanf(const char *, ...);

В този случай компилаторът на С при обработката на **source** кода не е в състояние да осъществява синтактически контрол върху съответствието между формалните и фактическите параметри при генериране кода на потребителската програма. Това се осъществява като в явен вид в декларацията и дефиницията на функцията с променлив брой аргументи завършва с последователност от три точки (...). Този запис информира компилатора да игнорира проверките за съответствие между тях. Естествено е, че със тялото на функцията с променлив брой аргументи трябва да е вграден механизъм за определяне броя на аргументите на активираната функция в точката на извикване. От практическа

гледна точка са възможни два варианта за определяне броя на аргументите в тялото на извиканата функция:

- при първият се използува допълнителен аргумент (първия), показващ актуалния брой на аргументите в точката на извикване;

- при втория се използува броя на аргументите се определя от стойността им, в смисъл установяване на типична стойност - например 0 или друга стойност, която не може да се срещне в реалните данни обработвани във извиканата функция;

И в двата случая се използува дефинирания в стандарта на С начин за взаимодействие между функциите посредством програмния стек. За демонстрирането им на фиг.8.11 е показан прост пример състоящ се от две потребителски функции. Първата **suma_dots(),** изчислява сумата на определен брой събираеми, които се зареждат в точката на извикване – като за целта се използува паразитния аргумент **counter** за указване броя на аргументите. В примера тя се извиква като в него (**counter**) първия път се зарежда константата 3, а втория път – 6, последвани от съответния брой фактически аргументи. При генериране на кода за извикване, както беше обяснено в т.5, в стека се зареждат фактическите параметри от дясно наляво, така че в него при активиране на функцията указателя на стека сочи към областта от ОП в която е копирана стойността за аргумента **counter**, т.е. той стои на върха на стека**.** В тялото на функцията локалния параметър **int *ptr** (указател към обект от цял тип) се настройва към адреса на **counter**. Достъпът до останалите аргументи се осъществява чрез този работен указател. Самото изчисляване на сумата се осъществява в цикъла **for**, като в инициализиращата му секция се нулира **suma** и еднократно се

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

int suma_dots(int counter,...)
{
  int suma;
  int *ptr=&counter;
  printf("\nfunction <suma_dots> call with %d arguments",counter);
  for(suma=0, ptr++;counter>0; counter--)
    suma+=*ptr++;
  return suma;
}

int suma_null(int arg,...)
{
  int suma=0,count=0;
  int *ptr=&arg;
  printf("\nfunction <suma_null> ");
  while(*ptr)
  {
    count++;
    suma+=*ptr++;
  }
  printf(" -> count elements=%d",count);
  return suma;
}

int main(void)
{
  int a1=5, a2=13, a3=8, a4=2, a5=-5, a6=2, a7=10;
  printf(" ->result=%d",suma_dots(3,a1,a2,a3));
  printf(" ->result=%d",suma_dots(6,a1,a2,a3,a4,a5,a6,a7));
  printf(" ->result=%d",suma_null(a1,a2,a3,a4,0));
  printf(" ->result=%d\n",suma_null(a1,a2,a3,a4,a5,a6,0));
  return 0;
}
```

Фиг. 8.11

Примерно изпълнение на програмата: **фиг.8.12**

```
function <suma_dots> call with 3 arguments ->result=26
function <suma_dots> call with 6 arguments ->result=25
function <suma_null>  -> count elements=4 ->result=28
function <suma_null>  -> count elements=6 ->result=25
```

увеличава **ptr**, така че да сочи към следващия фактически аргумент(първият аргумент след **counter**), лежащ в стека. При цикличното събиране **ptr** се увеличава с 1, като по този начин се премества към следващия аргумент в стека. Примерното изпълнение на програмата е показано на фиг.8.12.

Втората функция - **suma_null() -** примера на фиг.8.11, демонстрира втория подход за реализиране на функции с променлив брой аргументи, когато характерна стойност на фактическия аргумент спира извличането на аргументи от стековата област – в конкретния случай това е стойност 0. Тя се извиква два пъти – първият път с 4 аргумента, последван от константата 0, а вторият път с 6 аргумента, последван от 0.

За улеснение на проектирането на потребителски функции с променлив брой аргументи, при използуване на втория подход, описан по-горе (функцията **suma_null()**), в пакета с библиотечни функции ,в хедър файловете **stdio.h** и **stdarg.h** са дефиниран типа: **typedef char * va_list**; и макросите:

```
#define va_start(ap,v)  ( ap = (va_list)&v + _INTSIZEOF(v) )
#define va_arg(ap,t)    ( *(t *)((ap += _INTSIZEOF(t)) - _INTSIZEOF(t)) )
#define va_end(ap)      ( ap = (va_list)0 )
```

Чрез тях се улеснява обработката при извличане на аргументите във функциите с променлив брой аргументи. Проектирането на функция **suma_null_va()** при използването **va_list** и **va_arg,** която реализира функционалността на **suma_null** е показано на фиг. 8.13. Тя използува променливата **marker** за пресмятане на сума от списък на аргументи, завършващ със стойност 0. Стопиращият аргумент може да има и друга стойност, стига тя да не се среща в реалните данни, които се обработват – например ако се обработват само положителни числа за край на извличането на аргументи може да се използува стойността -1.

```
#include <stdarg.h>

int suma_null_va(int arg,...)
{
 int suma=0,count=0,val=arg;
 va_list marker;
 printf("\nfunction <suma_null_va> ");
 va_start( marker,arg );/* инициализацияна променливите аргументи */
 while( val != 0 )
 {
   suma += val;
   count++;
   val = va_arg( marker, int); /* извличане на нов аргумент */
 }
 va_end( marker);          /* ресет на променливите аргументи */
 printf(" -> count elements=%d",count);
 return suma;
}
```

фиг.8.13

8. Аргументи на функцията main.

Съгласно синтаксиса на езика С програмата е линейна структура от функции, от които задължително е присъствието на функция **main()**. Тя обикновено се нарича *главна функция*, но тя не е с някакъв по-висш приоритет спрямо останалите библиотечни или потребителски функции. Компилаторът на С генерира междинния (обектния) код на програма в реда, в които среща декларациите и дефинициите на функциите, от които е изградена потребителската програма. В последващата обработка свързващият редактор трябва да изгради изпълнимия код на програмата, като осигури връзката между всички функции, от които тя е изградена. При това за правилното

свързващо редактиране на него му е необходима информация за входната точка (началото), от която трябва да започне изпълнението на програмата. В този смисъл функцията **main()** е задължителна за всяка С програма - това е началото на изпълнимия код, която я определя като ”главна” функция. Това е функцията към която се предава управлението при зареждането на изпълнимия код от зареждащата програма на ОС.

От гледна точка на всяка ОС стартираната потребителска програма е активиран потребителски процес, който в най-общия случай може да бъде извикан (стартиран) от друг процес (наричан родителски), като при това често се налага предаването на съвкупност от параметри за изпълнението на извиканата програма. От друга страна при завършване на изпълнението тя може да върне определена стойност на родителския процес – обикновено това е статуса на завършване, който да бъде обработен от родителския процес, например типа на грешката при обработката на входните данни.

Съгласно езика С, за реализиране на достъпа до външни аргументи на **main(),** се описва като функция от цял тип. Тя се дефинира като функция в два варианта:

```
 int main( int argc, char *argv[ ])
{
  операции;
 return xxx;
}

int main( int argc, char *argv[ ], char *envp[ ] )
{
  операции;
 return xxx;
}
```

<u>където:</u>

int argc - брой на аргументите от командния ред, който е стартирана потребителската програма, включително и името на програмата;

char *argv[] - масив от указатели към **ASCIIZ** низове съдържащ командния ред, с който е стартирана програмата;

char *envp[] - масив от указатели към **ASCIIZ** низове съдържащ глобалните променливи на обкръжението на ОС;

xxx - резултат, които се връща на процеса, активирал потребителската програма;