Лабораторна тема – функции

(подробно темата е развита тук)

1. ***Въведение – понятие за функция.***

2. ***Дефиниция и декларация на функция.***

3. ***Взаимодействие между функции.***

   предаване на данни между извикващата и извикана функции;

   предаване на управлението между извикващата и извикана функции;

Пример

```
#include <stdio.h>
int a, b;
int area(int side1, int side2)      /* (1) */
{
  int surface;
  surface=side1*side2;
  return surface;
}

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

int main(void)      /* (3) */
{
  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=12
side2=6
result area=72
result perimeter=36
a=3
b=4
area(a,b)=12
```

фиг.8.4

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

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 ѝ. 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 ѝ, 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...

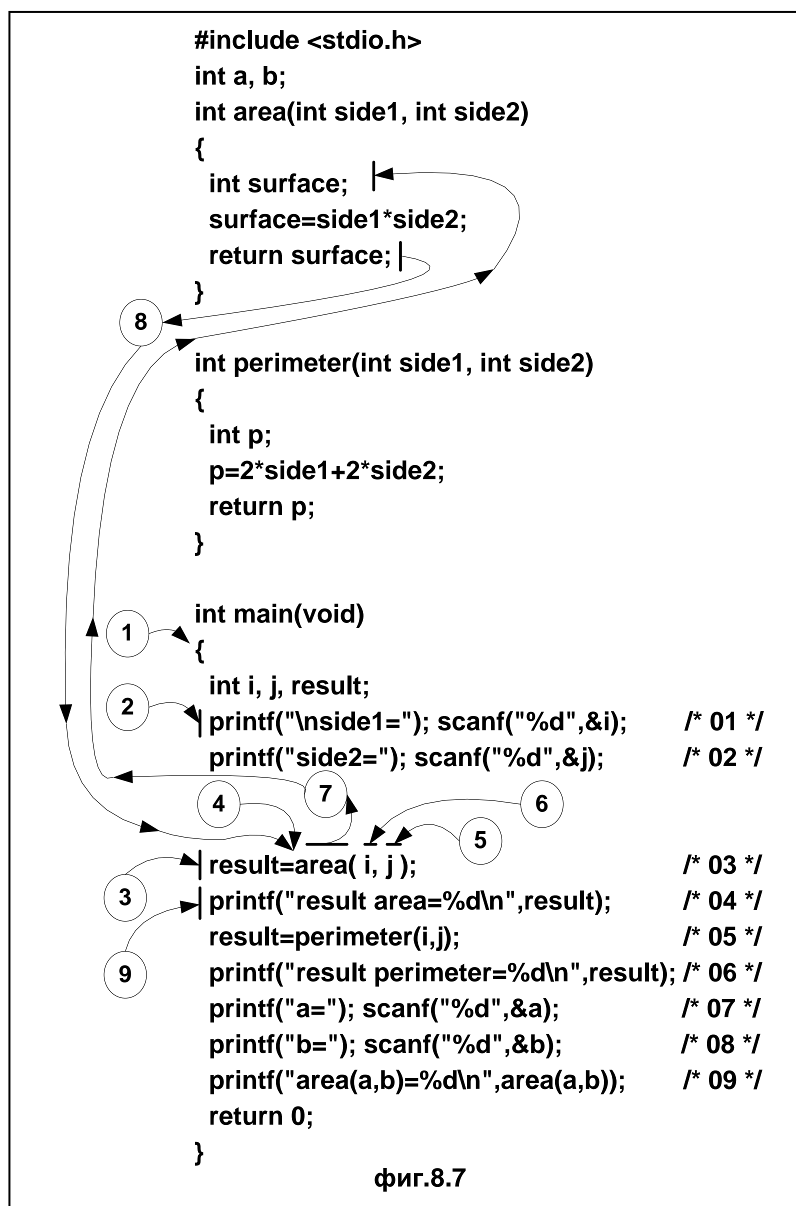
Обяснение в по-голяма дълбочина механизмите за взаимодействие между функциите в една потребителска С-програма може да се намери тук.

4. Класове памет и област на видимост .

5. Обмен на информация между функции.

- използване на общи (външни) променливи
- предаване на данни чрез входни аргументи на извиканата функция

Детайлизацията на механизма за взаимодействие между извикващата и извиканата функция при предаване на данни и използването на програмния стек в този механизъм е показана на фиг.8.7, използващ 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.



фиг.8.7

По-подробно разглеждане на взаимодействието може да се намери тук

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 използва 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;
}
```

фиг.8.10

6. Предаване на параметри Модификатори cdecl и 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 използването 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 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.

```
#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
```

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. Стопиращращият 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.

9. Задачи за самостоятелна работа

1.Въведете и тествайте примера от фиг.8.4, като използувате режима за **Debug** на **IDE** за стъпково изпълнение и проследете моментните стойности на формалните и фактически параметри.

2.Реализирайте функция за пресмятане на средноаритметичната стойност на целочислени едномерни масиви, притежаващи различен брой елементи.

3. Въведете и тествайте примерите от фиг.8.7 и 8.9. Реализирайте нова функция **float circle()**, която да мащабира правоъгълна област до кръг с лице променено с указан процент – входните данни са страните на правоъгълника и процента на корекция, а върнатия резултат е радиуса на кръга.

4.Въведете и тествайте примера от фиг.8.11. Преработете функцията **suma_dots() ,** като реализирате функционалност за натрупване на сума от променлив брой **double** събираеми.

5. Към проекта от фиг.8.11 добавете и тествайте функция double **average(…)**, която изчислява средноаритметичната стойност от променлив брой целочислени аргументи, ползувайки типа **va_list** и **va_arg**.