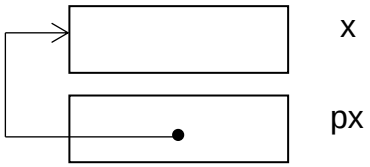
Тема 9
Указатели. Адресна аритметика

1. Указатели

Всеки елемент от програма (константа, променлива) се съхранява в определени клетки от паметта, които се намират на на определено място в адресното пространство на паметта т.е. всяка данна се съхранява *на определен адрес от паметта*. Съответно, достъп до данната може да се осъществи като се използва нейният адрес. Като стойност, даден адрес може да се съхранява на друго място в паметта. Променлива, чиято стойност е адрес от паметта се нарича ***указател***. Стойността на указателя посочва (указва) местоположението на променливата. Така указателят косвено служи за достъп до променлива, за разлика от познатия, директен достъп, чрез нейното име. Като обобщение на казаното, ***указателят е променлива, чиято стойност е адрес на променлива*** (фиг. 10.1).

Използването на указатели е начин за писане на ефективни и компактни програми. От друга страна, ако не бъдат правилно използвани, указателите могат да доведат до разрушаване на данните.



Фиг. 10.1. Указателят px съдържа адреса на променлива x

Както всяка променлива, и указателят трябва да се дефинира. Общият вид на дефиницията на променлива-указател е:

тип *имеУказател [=стойност];

където:

тип определя типа на съдържанието на указателя т.е. това е типа на променливата, чийто адрес ще сочи указателя.

имеУказател е идентификатора на променливата-указател.

Символ * пред името на променливата определя, че променливата е указател. Обикновено, имената на променливите указатели започват с **p**, което се определя от английския термин **pointer** (указател).

Примери за дефиниране на указатели са:

```
int *px;          // px е указател към променлива от тип int
float *p1;        // p1 е указател към променлива от тип float
double *p2;       // p2 е указател към променлива от тип double
```

Както всяка друга променлива, така и указателят може да бъде инициализиран с дадена начална стойност. Тази стойност трябва да бъде адрес на променлива, както е в посочения пример:

```
int x;
int *px=&x;  // указателят px сочи адреса на променливата x
```

Ако указателят не е инициализиран, неговата стойност е ***NULL***. Тази стойност може да бъде присвоена на указател от всеки тип.

2. Операции с указатели. Адресна аритметика.

За работа с указатели са предвидени два оператора, които се наричат адресни:

- **оператор *&*** определя адреса на операнда;
- **оператор *** определя стойността от посочения адрес.

Форматът на оператор **&** е следния:

&променлива

Освен променлива, операндът може да е и елемент на масив. Като резултат операцията връща адреса на променливата.

Форматът на оператор ***** е следния:

****указател***

Резултат от действието на оператора е стойността на променливата, чийто адрес съдържа указателя.

На променливата-указател може да се присвои стойност на друг указател. Например:

```
int x, *pa;
int *px=&x;
…
pa=px;       // pa съдържа адреса на променливата x
```

Указателите, като операнди, могат да участват и в следните аритметични оператори и логически отношения:

+ събиране

- изваждане

++ инкрементиране

-- декрементиране

= присвояване

== равно

> по-голямо

>= по-голямо или равно

< по-малко

<= по-малко или равно

!= различно

Върху указателите могат да се прилагат по-малко на брой аритметични операции, отколкото върху обикновените променливи. Освен това, изпълнението на аритметичните оператори върху указатели е свързано с някои особености, поради което тяхното изпълнение е известно още като ***адресна аритметика***.

Особеност на адресната аритметика е, че при изпълнението на операторите за *събиране, изваждане, инкрементиране и декрементиране* автоматично се извършва ***мащабиране***, което отчита типа на обектите, към които сочат указателите.

Пример:

```
int x;
int *px=&x;
…
px++;
```

Указателят **px** вече сочи към следващ обект и това е следващата променлива от тип ***int***, чийто адрес е след адреса на **x**. Това означава, че като стойност, адресът **px** е увеличен не с един, а два (или четири) байта.

Общото правило, което отличава *адресната аритметика* от останалите операции е следното: *Ако **p** е указател от тип **type**, то **p+k** се изчислява чрез израза: **p+k*sizeof(type)**.* По този начин, новата стойност на указателя вече сочи следващ елемент от същия тип.

Адресната аритметика се прилага най-вече при работа с масиви, тъй като те представляват поредица от еднотипни елементи.

3. Указатели и масиви

В C/C++ съществува пряка връзка между указатели и масиви - ***имената на масивите се явяват указатели***, които съдържат адреса на първия елемент от масива т.е. на елемента с индекс 0. По този начин, достъпът до елемент на масив може да се осъществи и чрез указател.

Например, нека са дефинирани масив от 5 целочислени елемента и указател, който в последствие да получи адреса на масива:

```
int array[5];
int *p;
int x,y,i;
...
// променливите x, y осъществяват достъп до един и същ елемент
x=array[0];
y=*array;
```

```
...
// променливите x, y осъществяват достъп до елемент с индекс i
x=array[i];
y=*(array+i);
...
p=array+i;   // също, достъп до елемент с индекс i
y=*p
```

Основната разлика между името на масива и променливата-указател е, че *името на масива се явява константа* и стойността му не може да бъде променена. Тази особеност се илюстрира чрез следните примери:

```
p=array;            // изразите са допустими
p++;


array=p;            // изразите са недопустими
array++;
array=&x;
```

Достъпът до елементите на масив може да се осъществи както чрез индекс, така и чрез указател. Достъпът чрез указател е много по-бърз, отколкото този чрез индекс и поради тази причина често е предпочитан в програмите на C/C++.

Пример за достъп до елементи на масив чрез указател е програмния код, който намира сумата на елементите на масив в задачата за намиране на средноаритметична стойност от елементи на едномерен масив:

```
#include <stdio.h>
main()
{
      double mB[10];
      int i;
      double suma, average;
      for(i=0;i<10;i++)
      {
            printf("Item[%d]=",i);
            scanf("%lf",&mB[i]);
      }
      suma = 0;
      double *p;
      for(p=mB; p<(mB+10); p++)
            suma += *p; // достъп до елемент на масив чрез указател
      average = suma/10;
      printf("\nArerage = %lf",average);

      return 0;
}
```

Тема 10
Програмни модули в език С. Функции

1. Видове програмни модули

Съвременните програмни системи се разработват въз основа на ***модулно програмиране***. Принципът ***модулност на програмирането*** означава, че дадена глобална задача се разделя на множество подзадачи. Всяка задача се реализира с *отделен, самостоятелен* ***програмен модул***.

Модулното програмиране има следните предимства:

- програмите са по-ясни, по-лесни за тестване, настройка и модификация.
- отделните програмни модули могат да бъдат създадени от различни програмисти, което съществено намалява времето за разработка на една цялостна програмна система;
- веднъж създадени, програмните модули могат да бъдат извиквани и изпълнени многократно. По този начин се избягва писане на едни и същи фрагменти програмен код.

В програмните езици, за видовете програмни модули се използва следната класификация:

- ***главна програма***;
- ***подпрограма*** (процедура, функция).

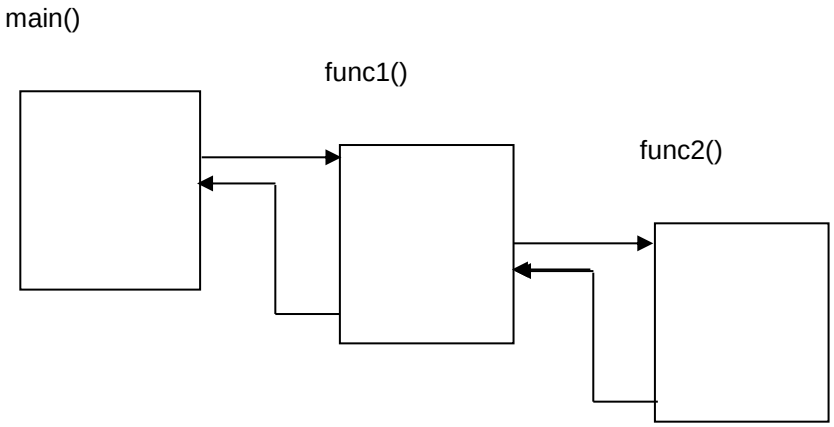
Главна програма е програмен модул, на който операционната система предава управлението при стартиране на изпълнимия код. При изпълнението си, тази програма може да предаде управлението на други програмни модули, наречени ***подпрограми***. След завършване, главната програма връща управлението на операционната система. ***Подпрограма*** е програмен модул, който получава управлението т. е. извиква се от друг програмен модул. След приключване, подпрограмата връща управлението в програмния модул, от където е била извикана.

За разлика от други програмни езици, където подпрограмите се разделят на процедури и функции, в С съществува само един тип програмни модули - ***функции***. ***Функция*** *е самостоятелен фрагмент от програма, съдържащ описания на променливи и набор оператори на езика*. Фрагментът е затворен във скоби {…} и в крайна сметка, се изпълнява като един обобщен оператор.

Обикновено, всяка програма на С се състои от една или повече функции. Задължително трябва да съществува една главна функция. В повечето случаи името на главната функция е ***main().*** Функция ***main()*** изпълнява роля на *главна програма*. Това е функцията, към която първоначално се предава управлението от операционната система. След

приключване на изпълнението, функция ***main()*** връща управлението на операционната система.

Механизмът на извикване на функциите е представен на фигура 11.1. В случая, функция ***main()*** извиква ***func1()***, която от своя страна прави обръщение към ***func2().*** Функциите връщат управлението там, от където са извикани.



Фиг.11.1. Механизъм на извикване на функциите и връщане на управлението

Като самостоятелен програмен модул, функцията може да бъде извиквана многократното, като се стартира с различни входни данни (променливи). Входните променливи се наричат още ***параметри*** на функцията. След завършването си функцията връща ***резултат***. Като програмен модул, функцията връща един резултат, чрез името си. ***Параметрите*** и ***резултатът*** са връзката на функцията с останалите програмни модули.

Възможно е, в частен случай, функция да не изисква входни данни и да не връща резултат. Например: функция, която изчиства екрана; функция за инициализиране на графична среда и др. Възможно е, също, функция да изисква входни данни, но да не връща резултат. Например, функция за изчертаване на окръжност, като входни данни може да изисква координати на центъра **x,y** и радиуса на окръжността **r**. Така, при многократно извикване, функцията може да изчертава различни по големина окръжности, разположени на различно място по екрана. След завършване, не е необходимо функцията да върне резултат.

2. Дефиниране на функции. Оператор *return*

2. 1. Дефиниране на функции

Дефиниция на функция представлява цялостното описание на функцията. Описанието на функцията се състои от две части:

- **заглавна част (*прототип*)**, която описва типа, името на функцията и нейните параметри;

- **тяло,** което съдържа декларации на локални променливи и оператори, конструкции, описващи действията който функцията изпълнява.

Общият вид на дефиницията на функция е следният:

```
                                        //прототип на функция
тип  ИмеФункция (списък формални параметри)
{                                       // тяло на функция
   // описание на локални променливи
  // оператори;
}
```

Тип на функцията е типът на стойността, която функцията ще върне като резултат. По подразбиране функциите са от тип ***int*** т.е. ако не бъде посочен, счита се, че функцията е от тип ***int***. Чрез името си, функцията връща един резултат. Име на функцията е идентификатор за нейното еднозначно определяне.

Формални параметри са списък входни променливи, чрез който се осъществява връзката между функцията и останалите функции. Те имат описателно значение. Списък формални параметри се представя в следния вид:

(тип1 променлива1, тип2 променлива2, ...)

Тялото на функцията включва множество декларации на локални променливи и оператори, реализиращи нейната задача.

Пример за дефиниция на функция е следния:

```
int suma(int x, int y)
{ int z;
   z = x+y;
   return z;
}
```

Функцията ***suma*** пресмята сумата на две числа. Числата се определят от двата входни параметъра x, y, които са променливи от тип ***int***. Функцията връща резултат от тип ***int***. Тъй като функцията е от тип, който е по подразбиране (***int***), възможно е да се пропусне типа пред името на функцията. В случая, също е валидно описанието:

```
suma(int x, int y)
{ int z;
   z = x+y;
   return z;
}
```

По време на изпълнението на функцията, резултатът се съхранява в локалната променлива ***z***. Оператор ***return*** връща стойността на функцията.

2.2. Опрератор *return*

В тялото на всяка функция е необходимо да съществува поне един оператор ***return***. Синтаксисът на оператора е:

return израз;

Оператор ***return*** служи да върне управлението на извикващата функция. Изразът след оператора трябва да е от същия тип какъвто е типът на функцията. Той се явява върнатия резултат.

Ако функцията не връща резултат, в оператора липсва параметър:

return;

Функции в C/C++, който не връщат резултат са от тип ***void.***

2.3. Функции от тип *void*

В C/C++ е предвидено да могат да бъдат дефинирани функции, които не връщат стойност. Тази възможност е предвидена, тъй като има редица операции и действия, които не са свързани с изчисления. Например, входно-изходни операции; задаване на режими на периферни устройства и др. За да се укаже, че дадена функция не връща стойност, тя се дефинира от тип ***void*.** Пример за такава функция е *SaveFile(),* която записва данни във файл:

```
void SaveFile( unsigned *, unsigned *)
{
      …
 return;
}
```

3. Извикване на функция. Предаване на параметри

Извикването на функция става чрез името ѝ. Например**:**

```
y1=suma (x1,y2);
```

В скобите след името на функцията се задават ***фактическите параметри*.** Общия вид на извикване на функция се задава като:

име (списък фактически параметри);

Подобно на *формалните*, *фактическите* параметри са разделени със запетаи.

Фактическите (**действителни)** параметри са *константи*, *променливи* или *изрази*, които при извикване на функцията предават своите стойности на формалните параметри, за да се изпълни функцията. Този процес се нарича ***свързване на формалните с фактическите параметри***.

Фактическите и формалните параметри трябва да си съответстват по брой и по тип. В противен случай, компилаторът ще изведе съобщение за грешка.

Веднъж дефинирана, дадена функция може да бъде извикана многократно, с различни стойности на фактическите параметри, например:

```
m= 3+suma (1,x1);
```

Фактически параметри при извикване на функцията са константата 1 и стойността на променливата x1.

4. Декларации на функции

В случай, че се наложи функцията да бъде извикана преди нейната дефиниция, необходимо е, тя да бъде ***декларирана***. Така компилаторът „знае", че съществува функция с посоченото име, има съответния тип и изисква определените параметри.

Декларацията на функцията е нейният ***прототип*** *(заглавната част),* последван от символ ***;***

Пример:

```
int suma (int x, int y);
```

При декларацията на функция, задължително се задават типовете на формалните параметри, докато имената им могат да се пропуснат.

Например декларацията на функцията ***suma()*** може да се извърши и по следният начин:

```
int suma (int, int);
```

При декларирането на функция, е достатъчно да се посочи броя на параметрите и техния тип. Имената на параметрите не са необходими по време на декларацията на функцията, тъй като компилаторът използва декларацията за проверка на типовете, а не за съответствие на имената. Ако функцията е от типа по подразбиране ***int***, той също може да бъде пропуснат в декларацията й.