Покажувачи. Алоцирање на меморија

Секој податок во еден компјутерски систем, без разлика дали се работи за цел број, знак, текстуална низа или сложена структура, си има своја мемориска локација преку која може да биде пристапен. Во C++, преку креирање на променливи од соодветен тип, овозможена е работа со податоци без контрола на локацијата на која тие се сместени: всушност, во сите програми кои што ги напишавме досега, воопшто не се грижевме за локацијата на која се сместуваат податоците.

Во ова предавање ќе зборуваме за покажувачи: специјален тип на податок кој овозможува работа со мемориски локации. Во C++, покажувач се креира така што помеѓу типот на податок до кој се покажува и името на променливата која ќе служи како покажувач се поставува знакот '*' (ѕвезда). Меморискатa локација на една променлива се добива со поставување на знакот '&' пред името на променливата чија адреса сакаме да ја дознаеме. Да разгледаме неколку примери:

Програма 15.1

```
#include <iostream>
using namespace std;
int main()
{
int a = 5; //promenliva 'a' so vrednost 5
int *b; //pokazhuvach kon podatok od tip int
b = &a; //b ja sodrzhi adresata na a ("b pokazhuva kon a")
/*
double *k;
k = &a; //GRESHKA: pogreshen tip na pokazhuvach (double)
*/
cout << a << endl; //pechati '5' (vrednosta na a)
cout << b << endl; //pechati '0x27ff44' (adresata na a)
return 0;
}
```

Најпрвин, важно е да се каже дека, при креирање на покажувачи (int *x), позицијата на која се наоѓа знакот '*' нема никакво влијание на извршувањето на програмата. Имено, знакот '*' може да се наоѓа веднаш по податочниот тип (int* x), помеѓу податочниот тип и името на покажувачот (int * x) или до името на покажувачот (int *x): сите наредби ќе имаат ист ефект.

Доколку ја извршите оваа програма на вашиот компјутер, таа веројатно ќе испечати некоја сосема друга адреса за променливата a (со последниот ред за печатење). Не се грижете: тоа е очекуваното однесување на програмата.

Кога се креираат повеќе покажувачи од еден ист тип, треба да се внимава да се наведе знакот '*' за секој покажувач посебно:

Извадок 15.1

```
int a, b; //a e od tip 'int', b e od tip 'int'
int *a, *b; //a i b se pokazhuvachi kon podatok od tip 'int'
int *a, b; //a e pokazhuvach, b e obichna promenliva od tip 'int'
int a, *b; //a e od tip 'int', b e pokazhuvach
```

Често, кога користиме покажувачи, сакаме да знаеме која е вредноста која се чува на одредена мемориска локација - односно, која е вредноста на променливата до која покажува нашиот покажувач. Во C++, тоа го правиме на начин што пред името на покажувачот го поставуваме знакот '*' (истиот знак што го користевме за негово креирање). Тука е важно да се знае дека со знакот '*' всушност пристапуваме до податок на одредена мемориска локација и, не само што може да ја видиме неговата вредност, туку истата можеме и да ја промениме!

Програма 15.2

```
#include <iostream>
using namespace std;
int main()
{
int a = 5; //promenliva 'a' so vrednost 5
int *p; //pokazhuvach kon podatok od tip int
p = &a; //p ja sodrzhi adresata na a ("p pokazhuva kon a")
cout << a << endl; //pechati '5'
cout << *p << endl; //pechati '5'
*p = 3; //promena na vrednost
cout << p << endl; //pechati '0x27ff44' (adresa)
cout << *p << endl; //pechati '3' (vrednost)
cout << a << endl; //pechati '3'
return 0;
}
```

Разликата помеѓу операторите '&' и '*' може да ја дефинираме со следниве два изрази:

- '&' се чита "адреса на" и ја враќа адресата на одреден податок. На пример, "&x" ја враќа адресата на x.
- '*' се чита "вредноста покажувана од" и ја враќа вредноста која се чува на одредена локација. На пример, "*p" служи за пристап до вредноста покажувана од покажувачот p.

Пред да продолжиме понатаму, важно е да наведеме дека еден покажувач може да се искористи повеќе пати (во една иста програма) и притоа (во различни моменти) да покажува кон различни променливи и мемориски локации. Доколку сакаме експлицитно да изразиме дека одреден покажувач не покажува кон ништо, него може да му доделиме вредност 0 (т.н. NULL покажувач). Стандардот гарантира дека не постои податок во компјутерската меморија со адреса 0.

Да разгледаме една едноставна програма која ги илустрира сите концепти кои ги споменавме досега:

Програма 15.3

```
#include <iostream>
using namespace std;
int main()
{
int a = 5, b = 2;
int *pa = &a, *pb; //inicijalizacija
pb = &b; //pb pokazhuva na b
*pa = 3;
cout << a << " " << *pa << " " << *pb << endl; //pechati '3 3 2'
*pb = -1;
cout << b << " " << *pb << endl; //pechati '-1 -1'
*pa = *pb;
cout << a << " " << b << endl; //pechati '-1 -1'
pa = 0;
double c = 8.0123, d = 0.0000;
double *px;
px = &c;
(*px) += 2.0;
cout << (*px) << endl; //pechati '10.0123'
px = &d;
cout << (*px) << endl; //pechati '0'
return 0;
}
```

Големината на еден покажувач зависи од архитектурата на компјутерскиот систем на кој се извршува програмата и од оперативниот систем: 32-битен компјутерски систем ќе користи 32-битни мемориски адреси (и, соодветно, покажувачите ќе зафаќаат 4 бајти податочен простор), додека 64-битен компјутерски систем ќе користи 64-битни мемориски адреси (и, соодветно, покажувачите ќе зафаќаат 8 бајти податочен простор). Ова на никој начин не влијае на однесувањето на покажувачите: тие и понатаму можат да се искористат за пристап до податокот на кој покажуваат - без разлика на неговата големина.

Програма 15.4

```
#include <iostream>
using namespace std;
struct typeX
{
int t1, t2, t3, t4;
double t5, t6, t7, t8;
};
int main()
{
int a = 5, b = 2;
double c = 3.8;
char d = 'Z';
```

```
typeX e;
int *pa = &a, *pb = &b;
double *pc = &c;
char *pd = &d;
typeX *pe = &e;
cout << sizeof(pa) << endl; //pechati '4'
cout << sizeof(pb) << endl; //pechati '4'
cout << sizeof(pc) << endl; //pechati '4'
cout << sizeof(pd) << endl; //pechati '4'
cout << sizeof(pe) << endl; //pechati '4'
return 0;
}
```

Низи и покажувачи

Во C++, низите се многу тесно поврзани со покажувачите. Всушност, кога креираме низа со одредена големина, името на променливата која ја користиме за пристап до елементите на таа низа не претставува ништо друго освен покажувач до првиот елемент на низата.

Програма 15.5

```
#include <iostream>
using namespace std;
int main()
{
int array[] = {1, 2, 3, 4, 5};
cout << *array << endl; //pechati '1'
*array = 0; //sega, array[] = {0, 2, 3, 4, 5}
int *parr;
parr = array;
cout << *parr << endl; //pechati '0'
return 0;
}
```

Во програмата дадена погоре, откога parr ќе покажува на истата локација како и array, нема разлика помеѓу можностите кои ги нуди array и оние кои ги нуди parr - и двата покажувачи може да се искористат за пристап до елементите на низата. Единствената разлика е што parr, во понатамошниот тек на програмата, може да се промени и да покажува на нешто друго (на некоја друга локација), додека array е т.н. константен покажувач и не може да се менува – тој секогаш ќе покажува на низата со која што е креиран.

Во C++, овозможена е употреба на т.н. аритметика со покажувачи. Имено, секој покажувач во C++ е дефиниран со соодветен тип на податок кон кој покажува, и истиот може да се промени на начин што во иднина ќе покажува на некој претходен или следен податок во меморијата. Ова се прави со едноставно користење на операторите + и -.

На пример, доколку array покажува на првиот елемент од одредена низа, тогаш array+1 покажува на вториот елемент од низата, array+2 покажува на третиот елемент од низата, итн.

Следнава програма ќе испечати неколку елементи од низата array:

Програма 15.6

```
#include <iostream>
using namespace std;
int main()
{
int array[] = {1, 2, 3, 4, 5, 6, 7};
int *parr = array;
//kje ispechati '2 3 4 5 6 '
for (int x=0; x<5; x++)
{
parr = parr+1; //parr kje pokazhuva na sledniot element
cout << (*parr) << " ";
}
parr--; //parr kje pokazhuva na prethodniot element
cout << (*parr) << endl; //kje ispechati '5'
parr = parr - 2;
cout << (*parr) << endl; //kje ispechati '3'
return 0;
}
```

Забележете дека, во меморијата, елементите од одредена низа се наоѓаат на однапред познато растојание - т.н. големина на податочниот тип. На пример, во програмата дадена претходно, низата array содржи елементи од тип int кои, соодветно, имаат големина од по 4 бајти. Доколку користевме некој друг податочен тип, елементите би се наоѓале на некое друго растојание. Задача на самиот компајлер е да ја зголеми (или намали) вредноста на покажувачот за точно онаа вредност која соодветствува на големината на податочниот тип.

На пример, доколку првиот елемент од низата (parr) има мемориска локација 100000, вториот ќе има мемориска локација 100004 (растојание од 4 бајти), третиот ќе има 100008, итн. Поради тоа, parr+1 треба да врати 100004 (еден елемент по parr, односно 4 бајти по parr), parr+2 треба да врати 100008, parr+3 треба да врати 100012, итн. Доколку низата содржеше елементи од тип double (со големина од 8 бајти) и првиот елемент имаше мемориска локација 100000, тогаш parr+1 треба да врати 100008, parr+2 треба да врати 100016, итн.

Интересно е да се каже дека изразот *(array+5) има комплетно исто значење како и изразот array[5]. Всушност, "array[5]" е само еден поубав начин на запишување на операцијата *(array+5). Во позадина, компајлерот секако врши операции со покажувачи.

Фактот што променливата преку која пристапуваме до елементите на одредена низа е покажувач има значаен ефект на користењето на низите како аргументи на една функција: имено, кога испраќаме низа како аргумент на функција, системот не врши копирање на елементите, туку, на функцијата и се испраќа покажувач до првиот елемент на низата. Поради тоа, сите промени на елементите кои ќе се извршат во функцијата ќе бидат видливи и по нејзиното завршување:

Програма 15.7

```
#include <iostream>
using namespace std;
void func(int array[])
{
array[0] = 10;
}
int main()
{
int array[] = {1, 2, 3, 4, 5, 6, 7};
func(array);
cout << array[0] << endl; //pechati '10'
return 0;
}
```

Структури и покажувачи

Покажувачите често се користат во комбинација со структури на податоци. На пример, еден од начините на кои може да се предаде структура на податоци како аргумент на одредена функција е преку испраќање на покажувач до самата структура.

Притоа, важно е да се забележи дека се можни два начини на пристап до членовите на структурата:

- со користење на операторот '.' - (*nekojPokazuvacDoStruktura).imeNaElement
- со користење на операторот '->' - nekojPokazuvacDoStruktura->imeNaElement

Притоа, операторот '->' е специјален оператор кој има значење единствено кај покажувачите до структури на податоци.

Програма 15.8

```
#include <iostream>
using namespace std;
struct newType
{
```

```
int x, y;
};
int func(newType *p)
{
//val = (*p).x + (*p).y
int val = p->x + p->y;
return val;
}
int main()
{
newType p;
p.x = 3;
p.y = 7;
//argument na funkcijata func e pokazhuvach, pa
//potrebno e da ja ispratime adresata na (p)
int res = func(&p);
cout << res << endl; //pechati '10'
return 0;
}
```

Алокација на меморија

Досега разгледавме неколку програми кои користеа низи за чување на повеќе податоци од еден ист тип. Но, едно нешто што беше константно во сите тие програми беше фактот што големината на низата ја определувавме однапред - како параметар при нејзиното креирање (int array[10]). Овој параметар, барем според C++ стандардот, мора да биде константен цел број.

Но, што доколку потребната големина не може да се специфицира пред самото извршување на програмата? На пример, што доколку сакаме да креираме низа од N елементи од тип студент, а бројот на студенти може да се открие единствено за време на извршување на програмата - како параметар кој го специфицира корисникот?

Решението на овој проблем е т.н. динамичко алоцирање на меморија. Во C++, тоа се прави со користење на операторите new (доколку сакаме да алоцираме простор за само еден елемент) или new[N] (доколку сакаме да алоцираме простор за N елементи). Резултатот од овие операции е покажувач до блокот меморија кој бил резервиран.

Многу е важно, по користењето на резервираната меморија (откако истата повеќе не е потребна), да се направи нејзино бришење. На тој начин, оперативниот систем потоа може да ја додели таа меморија на процесите кои навистина имаат потреба од неа. Во C++, ова се прави со користење на операторите delete (за еден елемент) и delete[] (за повеќе елементи).

Да разгледаме една програма која користи динамичко алоцирање на меморија:

Програма 15.9

```
#include <iostream>
using namespace std;
int main()
{
int *parr;
parr = new int;
*parr = 2;
cout << *parr << endl; //pechati '2'
delete parr; //MNOGU VAZHNO!!!
int N = 100;
parr = new int[N]; //niza od 100 elementi
parr[5] = 3;
cout << parr[5] << endl; //pechati '3'
delete [] parr; //MNOGU VAZHNO!!!
return 0;
}
```

Една од најчестите грешки при користењето на динамички алоцирана меморија е бришење на низа од N елементи со помош на операторот delete array - наместо delete[] array. Внимавајте: доколку ја искористите погрешната верзија на операторот за бришење, можно е да изгубите дел од податоците со кои работи вашата програма. Слично, доколку воопшто не ги ослободувате алоцираните ресурси, можно е, поради немање на слободна меморија, да дојде до назадување на перформансите на вашата програма - или на целиот компјутерски систем.

Поради фактот што секој компјутерски систем има ограничено количество на меморија, понекогаш е можно да не успее динамичкото алоцирање на меморија (new int[N]). Притоа, доколку системот не успее да резервира доволно количество меморија, нашата програма ќе прекине со извршување и ќе јави грешка (runtime error).

Доколку сакаме самите да се справуваме со евентуалниот недостаток на меморија, тогаш треба да го искористиме специјалниот параметар (nothrow) - дефиниран во датотеката <new>, и самите да провериме дали алокацијата на меморија завршила успешно - доколку се случил проблем, покажувачот ќе има вредност 0 (NULL):

Програма 15.10

```
#include <iostream>
#include <new>
using namespace std;
int main()
{
int N = 200000000;
int *parr = new (nothrow) int[N]; //~750MB memorija
if (!parr) //parr == 0
{
cout << "Greshka: Nema dovolno memorija!" << endl;
return 0;
}
parr[N-1] = 3;
cout << parr[N-1] << endl;
```

```
delete [] parr;
return 0;
}
```