4.6.2. Указатели

Понятие указателя

Указатели - специальные объекты в программах C++, предназначенные для хранения адресов памяти.

Когда компилятор обрабатывает оператор определения переменной, например int i = 10; то в памяти выделяется участок в соответствии с типом переменной (int=>4 байта) и записывает в этот участок указанное значение. Все обращение к этой переменной компилятор заменит адресом области памяти, в которой хранится эта переменная.

Программист может определить свои переменные для хранения адресов областей памяти. Такие переменные называются указателями. Указатель не является самостоятельным типом, он всегда связан с каким-либо другим типом.

Указатели делятся на две категории:

1) указатели на объекты,

2) указатели на функции.

Указатели на объекты сохраняют адрес области памяти, содержащий данные определенного типа.

При объявлении указателя на объект данных перед его именем ставится символ *, являющийся признаком косвенной адресации, а перед символом * задаётся тип косвенно адресуемого объекта:

type *pointer;

Здесь через ***pointer*** обозначено имя переменной – указателя, которое подчиняется тем же правилам именования, что и для обычных переменных.

Тип может быть любым, кроме ссылки.

При объявлении переменной – указателя на функцию имя указателя вместе с предшествующим символом * заключается в круглые скобки, вне которых задаётся профиль косвенно адресуемых функций:

type (*pointer)(список аргументов с их типами);

Под профилем функции здесь и далее понимается тип возвращаемого функцией значения ***type***, а также список и типы каждого аргумента функции.

Замечание. ***С точки зрения синтаксиса языка безразлично, где будет находиться пробел – справа или слева от символа *. Его вообще может не быть.***

<u>Пример 47</u>

```
char *p;                 // указатель на символ
int *q;                  // указатель на целое
int *ip = &ia[2];        // указателю на целое присвоен адрес третьего
                         // элемента массива ia
int **t;                 // указатель на указатель на целую переменную
char *pa[7];             // массив из 7 указателей на символы
double (*fp)(double);    // указатель на функцию, аргументом
                         // которой является число типа  double, и
                         // которая возвращает результат типа double
```

Перед использованием в программе указатель должен содержать адрес объекта требуемого типа. Перед присваиванием указателю адрес переменной, элемента массива, функции или другого объекта может быть получен с помощью оператора взятия адреса **&**. Например:

p = &v[3]; //указатель содержит адрес 4-го элемента массива v
q = &inch; //указатель содержит адрес переменной inch
fp = &sqrt; //указатель содержит адрес функции sqrt()

Для инициализации указателя существуют следующие способы (рис. 20):
Присваивание адреса существующего объекта:

1) с помощью операции получения адреса
int a=5;
int *p=&a; или int p(&a);
2) с помощью проинициализированного указателя
int *r=p;

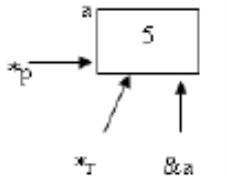

Рис. 20. Инициализация указателя

3) адрес присваивается в явном виде
char *cp=(char*)0x B800 0000;

где *0x B800 0000* – шестнадцатеричная константа, (*char**) – операция приведения типа.

4) присваивание пустого значения:
int *N=NULL;
int *R=0;

ОПЕРАЦИИ С УКАЗАТЕЛЯМИ

С указателями можно выполнять следующие операции:
1) разыменование (*);
2) присваивание;
3) арифметические операции (сложение с константой, вычитание, инкремент ++, декремент --);
4) сравнение;
5) приведение типов.

Операция разыменования предназначена для получения значения переменной или константы, адрес которой хранится в указателе. Если указатель указывает на переменную, то это значение можно изменять, используя операцию разименования.

Пример 50
int a; //переменная типа int
int *pa=new int; //указатель и выделение памяти под динамическую переменную

pa=10; //присвоили значение динамической переменной, на которую указывает указатель

a=*pa; //присвоили значение переменной a

Присваивать значение указателям-константам запрещено.

Операция приведения типов.

На одну и ту же область памяти могут ссылаться указатели разного типа. Если применить к ним операцию разыменования, то получатся разные результаты.

Пример 51

```
int a=123;
int *pi=&a;
char *pc=(char*)&a;
float *pf=(float*)&a;
printf("\n%x\t%i",pi,*pi);
printf("\n%x\t%c",pc,*pc);
printf("\n%x\t%f",pf,*pf);
```

При выполнении программы, представленной в примере 51, получатся следующие результаты:

66fd9c 123
66fd9c {
66fd9c 0.000000

Т.е. адрес у трех указателей один и тот же, но при разыменовании получаются разные значения в зависимости от типа указателя.

В примере при инициализации указателя была использована операция приведения типов. При использовании в выражении указателей разных типов явное преобразование требуется для всех типов, кроме void*. Указатель может неявно преобразовываться в значение типа bool, при этом ненулевой указатель преобразуется в true, а нулевой в false.

Арифметические операции применимы только к указателям одного типа.

1. Инкремент увеличивает значение указателя на величину ***sizeof(mun)***.

Пример 52

```
char *pc;
int *pi;
float *pf;
...
pc++;   //значение увеличится на 1
pi++;   //значение увеличится на 4
pf++;   //значение увеличится на 4
```

2. Декремент уменьшает значение указателя на величину ***sizeof (mun).***

3. Разность двух указателей – это разность их значений, деленная на размер типа в байтах.

4. Суммирование двух указателей не допускается. Можно суммировать указатель и константу.

Пример 54

Если ***p = &v[3];*** //указатель *p* содержит адрес 4-го элемента массива v
то ***p = p + 2;*** //указатель *p* содержит адрес 6-го элемента массива v

При записи выражений с указателями требуется обращать внимание на приоритеты операций.

Указатели и константы

При работе с указателями участвуют два объекта: сам указатель и указываемый им объект. Если в объявлении указателя в качестве префикса есть спецификатор ***const***, то константой объявляется сам объект, но не указатель на него.

Например:

```
char *p = "asdf";            // обычный указатель
const char *pc ="ghjk" ;     // указатель на константу
pc[3] = 'a';                 // ошибка, pc указывает на константу
pc = p;                      // нормально
```

Чтобы объявить как константу сам указатель, а не указываемый объект, нужно использовать оператор объявления *const, а не просто const. Кроме того, оператор *const нужно ставить после типа объекта.

Например:

```
char *p = "asdf";            // обычный указатель
char *const cp ="ghjk" ;     // указатель - константа
cp[3] = 'a';                 // нормально
cp = p;                      // ошибка, указатель cp - константа
```

Чтобы сделать константами и указатель, и объект, надо оба объявить как const, например:

```
char *p = "asdf";                    // обычный указатель
const char *const cpc = "asdf";      // указатель-константа и объект -константа
cpc[3] = 'a';                        // ошибка
cpc = p;                             // ошибка
```

Объект может быть объявлен константой при обращении к нему через один указатель и в то же время быть переменным, если обращаться к нему через другой указатель. Особенно удобно это использовать для аргументов функций.

Объявив для функции аргумент – указатель как const, функция не сможет изменить объект, адресуемый указателем. к примеру:

```
char* strcpy(char*p, const char*q);//функция не может изменять *q
```

Указателю на константу можно присвоить адрес переменной, т. к. это не принесет вреда. Однако адрес константы нельзя присвоить указателю без спецификатора const, иначе можно изменить значение объекта. Например:

```
int a = 1;
const int c = 2;
const int *p1 = &c;    // нормально
const int *p2 = &a;    // нормально
int *p3 = &c;          // ошибка, так как возможна следующая инструкция,
*p3 = 7;               // меняющая значение c
```

ДИНАМИЧЕСКИЕ ПЕРЕМЕННЫЕ

Все переменные, объявленные в программе, размещаются в одной непрерывной области памяти, называемой сегментом данных (64 Кб). Такие переменные не изменяют свой размер в ходе выполнения программы и называются статическими. Размера сегмента данных может быть недостаточно для размещения больших массивов информации. Выходом из сложившейся ситуации является использование динамической памяти.

Динамическая память - это память, выделяемая программе для ее работы за вычетом сегмента данных, стека, в котором размещаются локальные переменные подпрограмм и собственно тела программы.

Для работы с динамической памятью используют указатели, с помощью которых осуществляется доступ к участкам динамической памяти, которые называются динамическими переменными. Динамические переменные создаются с помощью особых функций и операций. Они существуют либо до конца работы программ, либо до тех пор пока не будут уничтожены с помощью специальных функций или операций.

Для создания динамических переменных используют операцию ***new***, определенную в С++:

указатель = new имя_типа[*инициализатор*];

где ***инициализатор*** – выражение в круглых скобках.

Операция **new** позволяет выделить и сделать доступным участок динамической памяти, который соответствует заданному типу данных.

Если задан инициализатор, то в этот участок будет занесено значение, указанное в инициализаторе:

int *x=new int(5);

Для удаления динамических переменных используется операция *delete*, определенная в С++:

delete указатель;

где **указатель** содержит адрес участка памяти, ранее выделенный с помощью операции *new*:

delete x;

В языке С определены библиотечные функции для работы с динамической памятью, они находятся в библиотеке ***<stdlib.h>:***

1) ***void*malloc (unsigned s)*** – возвращает указатель на начало области динамической памяти длиной s байт, при неудачном завершении возвращает *NULL;*

2) ***void*calloc (unsigned n, unsigned m)*** – возвращает указатель на начало области динамической памяти для размещения *n* элементов длиной *m* байт каждый, при неудачном завершении возвращает *NULL*;

3) ***void*realloc (void *p, unsigned s)*** – изменяет размер блока ранее выделенной динамической памяти до размера *s* байт, *p* – адрес начала изменяемого блока, при неудачном завершении возвращает *NULL*;

4) ***void *free (void *p)*** – освобождает ранее выделенный участок динамической памяти, *p* – адрес начала участка.

Пример 49

int *u=(int*)malloc(sizeof(int)); //в функцию передается количество требуемой памяти в байтах, т.к. функция возвращает значение типа *void**, то его необходимо преобразовать к типу указателя *(int*).*

free(u); //освобождение выделенной памяти

4.6.3. Ссылки

Понятие ссылки

Ссылка – это синоним имени объекта, указанного при инициализации ссылки. Формат объявления ссылки

тип &имя =имя_объекта;

Пример 55.

```
int  x;                    // определение переменной
int &sx=x;                 // определение ссылки на переменную x
const char &CR='\n';       //определение ссылки на константу
```

Правила работы со ссылками

1. Переменная ссылка должна явно инициализироваться при ее описании, если она не является параметром функции, не описана как *extern* или не ссылается на поле класса.
2. После инициализации ссылке не может быть присвоено другое значение.
3. Не существует указателей на ссылки, массивов ссылок и ссылок на ссылки.
4. Операция над ссылкой приводит к изменению величины, на которую она ссылается.

Ссылка не занимает дополнительного пространства в памяти, она является просто другим именем объекта.

Пример 56

```
void main()
{
  int I=123;
  int &si=I;
  cout<<"\ni="<<I<<" si="<<si;
  I=456;
  cout<<"\ni="<<I<<" si="<<si;
  I=0;
  cout<<"\ni="<<I<<" si="<<si;
}
```

Результат работы программы:

I=123 si=123
I=456 si=456
I=0 si=0

Ссылки применяются, прежде всего, для параметров функций и для типов возвращаемых функциями значений.

4.6.4. Указатели и массивы

Имя массива можно использовать в качестве указателя на его первый элемент. Например:

```
int v[] = {1,2,3,4};
int* p1 = v;              //указатель на 1-й элемент
```

int* p2 = &v[0]; //указатель на 1-й элемент
int* p3 = &v[4]; //указатель на элемент, следующий за последним.

Доступ к элементам массива может осуществляться при помощи указателя на массив и индекса либо при помощи указателя на элемент массива.

Пример с использованием индекса:

```
void f1(char v[])
{
for (int i = 0; v[i]!=0; i++) cout<<v[i];
}
```

Это эквивалентно использованию указателя:

```
void f2(char v[])
{
for (char* p = v; *p!=0; p++) cout<<*p;
}
```

Префиксный оператор * означает разыменование, поэтому *p есть символ, на который указывает p. Оператор ++ увеличивает значение указателя на размер элемента массива, т.е. обеспечивает переход к следующему элементу.

Результат применения операторов +, -, ++ или – к указателю увеличивает или уменьшает значение указателя на количество элементов, а не на количество байтов.

Т.к. имя_массива – это указатель константа, а индекс определяет смещение от начала массива, то используя указатели, обращение по индексу можно записать следующим образом:

***(имя_массива+индекс).**

Пример 58

```
for (int i=0; i<n; i++)          //печать массива
cout<<*(a+i)<< " ";              /* к имени массива (адресу начала массива) добавляется константа i  и полученное значение разыменовывается */
```

Так как имя массива является константным указателем, то его невозможно изменить, следовательно, запись ***(a++)*** будет ошибочной.

Указатели можно использовать и при определении массивов:

```
int  a[100]={1,2,3,4,5,6,7,8,9,10};
int * na=a;                      //поставили указатель на уже определенный массив
int *b=new int[100];             //выделили в динамической памяти место под массив из 100 элементов.
```

Многомерный массив - это массив, элементами которого служат массивы.

Например, массив с описанием *int a[4][5]* – это массив из 4 указателей типа ***int****, которые содержат адреса одномерных массивов из 5 целых элементов (рис. 21).

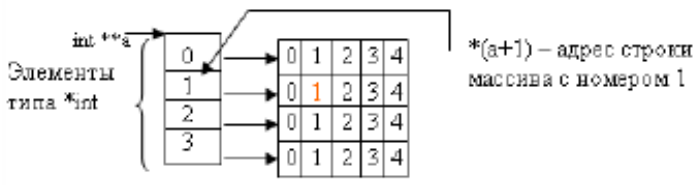

Рис. 21. Доступ к элементам многомерных массивов

Доступ к элементам многомерных массивов возможен и с помощью индексированных переменных и с помощью указателей:

a[1][1] – доступ с помощью индексированных переменных,

****(*(a+1)+1)*** – доступ к этому же элементу с помощью указателей (рис. 21).

4.6.5. Динамические массивы

Операция *new* при использовании с массивами имеет следующий формат:

new тип_массива

Такая операция выделяет для размещения массива участок динамической памяти соответствующего размера, но не позволяет форматировать элементы массива. Операция new возвращает указатель, значением которого является адрес первого элемента массива. При выделении динамической памяти размеры массива должны быть полностью определены.

Пример 60. Выделение динамической памяти:

1. ***int *a=new int[100];*** /*выделение динамической памяти размером ***100*sizeof(int)*** байтов*/

double *b=new double[12]; /* выделение динамической памяти размером 12*sizeof(double) байтов */

2. ***long(*la)[4];*** /*указатель на массив из 4 элементов типа long*/

la=new[2][4]; /*выделение динамической памяти размером 2*4*sizeof(long) байтов*/

3. ***int**matr=(int**)new int[5][12];*** /*еще один способ выделения памяти под двумерный массив*/

4. ***int **matr;***

matr=new int*[4]; /*выделяем память под массив указателей ***int**** */

for(int I=0;I<4;I++) matr[I]=new int[6]; /*выделяем память под строки массива*/

Указатель на динамический массив затем используется при освобождении памяти с помощью операции ***delete***.

Пример 61. Освобождение динамической памяти.

delete[] a //освобождает память, выделенную под массив, если a адресует его начало

delete[]b;

delete[] la;

for(I=0;I<4;I++) delete[] matr[I]; //удаляем строки

delete [] matr; //удаляем массив указателей.

Пример 62. Удалить из матрицы строку с номером K.

```
#include <iostream>
#include <stdlib.h>
#include <conio.h>
using namespace std;
void main()
{
int n,m;          //размерность матрицы
int i,j;
cout<<"\nEnter n";
cin>>n;           //ввод количества строк
```

```
cout<<"\nEnter m";
cin>>m;          // ввод количества столбцов
//выделение памяти
int **matr=new int* [n];        /* массив указателей на строки*/
for(i=0; i<n; i++)
matr[i]=new int [m];            /*память  под  элементы матрицы*/
//заполнение матрицы
for(i=0; i<n; i++)
for(j=0; j<m; j++)
matr[i][j]=rand()%10;  //заполнение матрицы
//печать сформированной матрицы
for(i=0; i<n; i++)
{
for(j=0; j<m; j++)
cout<<matr[i][j]<<" ";
cout<<"\n";
}
//удаление строки с номером к
int k;
cout<<"\nEnter k";
cin>>k;
int**temp=new   int*[n-1];     /*формирование   новой матрицы*/
for(i=0; i<n-1; i++)
temp[i]=new int[m];
//заполнение новой матрицы
int t;
for(i=0, t=0; i<n; i++)
if(i!=k)
{
for(j=0; j<m; j++)
temp[t][j]=matr[i][j];
t++;
}
//удаление старой матрицы
for(i=0; i<n; i++)
delete matr[i];            //удаляем строки
delete[]matr;              //удаляем массив указателей
n--;
//печать новой матрицы
for(i=0; i<n; i++)
{
for(j=0; j<m; j++)
cout<<temp[i][j]<<" ";
cout<<"\n";
}
getch();
}
```

```
#include <iostream>
#include <stdlib.h>
#include <conio.h>
using namespace std;
void main()
{
int n,m;            //размерность матрицы
```

```
int i,j;
cout<<"\nEnter n";
cin>>n;              //ввод количества строк
cout<<"\nEnter m";
cin>>m;              // ввод количества столбцов
//выделение памяти
int **matr=new int* [n];   /* массив указателей на строки*/
for(i=0; i<n; i++)
matr[i]=new int [m];       /*память  под  элементы матрицы*/
//заполнение матрицы
for(i=0; i<n; i++)
for(j=0; j<m; j++)
*(*(matr+i)+j)=rand()%10;  //заполнение матрицы
//печать сформированной матрицы
for(i=0; i<n; i++)
{
for(j=0; j<m; j++)
cout<<*(*(matr+i)+j)<<" ";
cout<<"\n";
}
//удаление строки с номером к
int k;
cout<<"\nEnter k";
cin>>k;
int**temp=new   int*[n-1]; /*формирование   новой матрицы*/
for(i=0; i<n-1; i++)
temp[i]=new int[m];
//заполнение новой матрицы
int t;
for(i=0, t=0; i<n; i++)
if(i!=k)
{
for(j=0; j<m; j++)
*(*(temp+t)+j)=*(*(matr+i)+j);
t++;
}
//удаление старой матрицы
for(i=0; i<n; i++)
delete matr[i];            //удаляем строки
delete[]matr;              //удаляем массив указателей
n--;
//печать новой матрицы
for(i=0; i<n; i++)
{
for(j=0; j<m; j++)
cout<<*(*(temp+i)+j)<<" ";
cout<<"\n";
}
getch();
}
#include "stdafx.h"
#include <iostream>
#include <stdlib.h>
#include <conio.h>
using namespace std;
void main()
{
int n,m;            //размерность матрицы
int i,j;
cout<<"\nEnter n";
cin>>n;              //ввод количества строк
cout<<"\nEnter m";
cin>>m;              // ввод количества столбцов
//выделение памяти
int **matr=new int* [n];   /* массив указателей на строки*/
```

```
for(i=0; i<n; i++)
matr[i]=new int [m];        /*память  под  элементы матрицы*/
//заполнение матрицы
for(i=0; i<n; i++)
for(j=0; j<m; j++)
*(*(matr+i)+j)=rand()%10;  //заполнение матрицы
//печать сформированной матрицы
for(i=0; i<n; i++)
{
for(j=0; j<m; j++)
cout<<*(*(matr+i)+j)<<" ";
cout<<"\n";
}
//сортировка матрицы
for(j=0; j<m; j++)
{for (int i=n-1; i > 0; --i)
       {      int*p = &(*(*(matr)+j)); cout<<*p<<" \n";
int*h = &(*(*(matr+1)+j));
              for (int k=0; k < i; ++k, p++,h++)
                     if (*p < *h)
                     {
                            int tmp = *p;
                            *p = *h;
                            *h = tmp;
                     }

       }
}
//печать сформированной матрицы
cout<<"\nMATRIX\n";
for(i=0; i<n; i++)
{
for(j=0; j<m; j++)
cout<<*(*(matr+i)+j)<<" ";
cout<<"\n";
}
cout<<"\nMATRIX\n";
for(i=0; i<n; i++)
{int*p = &(*(*(matr+i)+0));
for(j=0; j<m; j++,p++)
cout<<*p<<" ";
cout<<"\n";
}


getch();
}
```

Указатель на void

Тип void синтаксически эквивалентен базовым типам, но использовать его можно только косвенно, так как объектов типа void не существует. С помощью типа void задаются указатели на объекты произвольного типа, указывается отсутствие аргументов в функции или объявляются функции, не возвращающие значения.

Например:

void* pv; // указатель на объект произвольного типа
int g(void); // функция без аргументов, равносильно ***int g()***;
void f(); // ***f*** не возвращает значения

Указатель ***void**** может указывать на объект любого типа*, если этот объект объявлен без спецификаторов* ***const*** *или* ***volatile*****.**

Указатель на объект любого типа можно присваивать переменной типа void*, один void* можно присвоить другому void*, указатели void* можно сравнивать на равенство и неравенство. Перед использованием указателя void* нужно явно преобразовать в указатель в необходимый тип.

Поскольку объектов типа void не существует, указатели void* нельзя разименовать.

Тип void* приписывается аргументам функций, которые не должны знать истинного типа этих аргументов. Объекты типа void* могут возвращать функции. Для использования таких объектов необходимо выполнить операцию явного преобразования типа. Такие функции обычно находятся на самых низких уровнях системы, управляющих аппаратными ресурсами.

Например:

```
void* malloc(unsigned size); // библиотечная функция –
                             // аналог оператора new в языке C
void free(void*);            // библиотечная функция –
                             // аналог оператора delete в языке C
void f()                     // распределение памяти в стиле языка C
{
    int* pi = (int*)malloc(10*sizeof(int));
    char* pc = (char*)malloc(10);
    //...
    free(pi);
    free(pc);
}
```

В программе синтаксис: ***(тип) выражение*** - используется для задания операции преобразования выражения к другому типу. Следовательно, перед присваиванием ***pi*** тип ***void****, возвращаемый в первом вызове ***malloc()***, преобразуется к типу ***int***.

Принципиальное отличие указателя от ссылки состоит в том, что указатель может иметь нулевое значение, то есть не указывать ни на какой объект. В то время как ссылка (при корректном использовании) всегда будет ссылаться на настоящий объект.

Хорошим стилем является инициализация указателя, не указывающая ни на какой объект нулевым значением. Даже если это значение не планируется использовать реально – то есть указатель планируется установить на необходимый объект, а затем использовать. В случае, если программистом будет допущена ошибка и указатель будет использован до того, как в него будет помещен адрес реального объекта, значительно проще будет найти ошибку при обращении по нулевому адресу, чем по случайному адресу в памяти.

В общем случае лучше избегать использования указателей. При работе с массивами лучше использовать индексы.

Указатели следует использовать только в строго определенных ситуациях:

- В коде нижнего уровня, работающего непосредственно с оперативной памятью или внешними устройствами.
- Возврат из функций значений нескольких объектов или возврат больших объектов в качестве результатов функций.
- При работе с контейнерами полиморфных объектов, которые будут обсуждаться позже.