

Лекция 8

Указатели. Ссылки

Указатели	1
Получение адреса переменной	1
Определение переменной-указателя на некоторый объект	2
Инициализация переменной-указателя	3
Операции над указателями	5
Примеры работы с указателями	5
доступ по указателю	5
присваивание и приведение типа	5
особенности операции явного приведения типов	7
сложение с константой и вычитание константы	8
операции инкремента и декремента	9
сравнение	9
инициализация указателя с помощью функции malloc	10
Виды указателей	10
указатель на константу	10
указатель-константа	11
указатель-константа на константу	11
указатель на указатель	12
Ссылки	12

Указатели

Получение адреса переменной

Операция **&** позволяет получить адрес переменной:

//вывод на экран адресов переменных

```
int main()
{
    int var1 = 11;
    int var2 = 22;
    int var3 = 33;
    cout << &var1 << endl    //вывод адресов
        << &var2 << endl
        << &var3 << endl;
    _getch();
    return 0;
}
```

Обратите внимание на особенности получения адреса константы:

```
#define PI 3.141592
int main()
{
    //cout << &PI << endl; //ошибка !!!, адрес константы получить нельзя
    _getch();
    return 0;
}
```

```
int main()
{ const int n=5;
  cout << &n << endl;    //ОК !!! выводит адрес типизированной константы n
  _getch();
  return 0;
}
```

В C++, помимо доступа к переменной по ее имени, существует возможность использования механизма указателей. Различают указатели-переменные и указатели-константы.

Указатель-переменная (указатель) – это переменная, предназначенная для хранения адреса. Указатель занимает 4 байта памяти. Значением переменной-указателя является адрес данного, адрес участка памяти, выделенного для объекта конкретного типа. Указатель не является самостоятельным типом, он всегда связан с каким-либо другим конкретным типом. В определении указателя всегда присутствует обозначение соответствующего ему типа.

Для доступа к объекту через указатели в C++ определена операция ***** (операция разыменования, т.е. обращения к содержимому участка памяти, адрес которого хранится в указателе). Чтобы операция разыменования выполнялась правильно, **указатель должен иметь некоторое значение. Получить его он может в результате инициализации или присваивания.**

Указатели делятся на две категории: указатели на объекты и указатели на функции. Рассмотрим сначала указатели объектов (на объекты).

Определение переменной-указателя на некоторый объект

Определение в простейшем случае имеет вид:

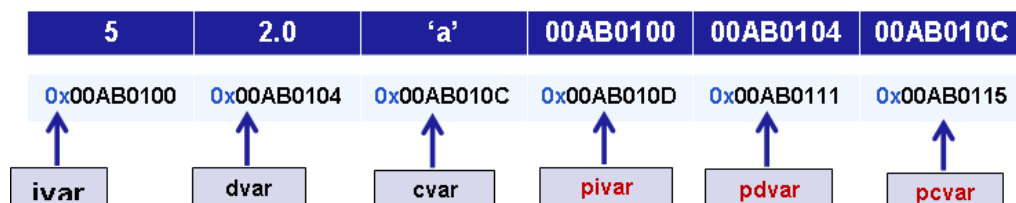
*тип *имя_указателя;*

```
int ivar=5;           // определение и инициализация переменной типа int
double dvar=2.0;     // определение и инициализация переменной типа double
char cvar='a';       // определение и инициализация переменной типа char
int *pivar = &ivar;  // определение и инициализация указателя на объект типа int
int *pivar1, *pivar2, *pivar3; // определение трех указателей на объекты типа int
                                // определение и инициализация указателя на объект типа double
double *pdvar=&dvar;
char *pcvar = &cvar; // определение и инициализация указателя на объект типа char
```

В совокупности **имя типа и символ *** перед именем воспринимаются как обозначение особого типа данных **«указатель на объект данного типа»**

Стиль определения указателей:

- каждый указатель определяется на отдельной строке; символ '*' пишется сразу (без промежуточных пробелов) за названием типа объектов, которые определяемый указатель может адресовать: `int* p;`
- если в строке определяется более одного объекта, то символ '*' необходимо записывать непосредственно перед определяемым указателем: `int *p.`



Указатель типа void * является особым типом указателя. Ключевое слово void говорит об отсутствии данных о размере объекта в памяти. Указателю на void можно присвоить значение указателя любого типа, а также сравнивать его с любыми указателями. Но компилятору для корректной интерпретации ссылки на память через указатель нужна информация о числе байтов, участвующих в операции. Поэтому во всех случаях использования указателя, описанного как void*, необходимо выполнить операцию явного приведения типа указателя, т.е. преобразовать указатель к конкретному типу явным образом.

Инициализация переменной-указателя

Присвоение значения указателю – это возможность избежать ошибки !!!!!
Существует несколько способов присвоить значение переменной-указателю:

- **инициализация указателя адресом переменной, определенной ранее**

```
int a=5;
int *iptr = &a;    //определение указателя и его инициализация адресом переменной a
int *iptr1(&a);    // определение указателя и другая форма его инициализации
```

- **инициализация указателя значением другого указателя, ранее проинициализированного**

```
int *kptr = iptr;    //определение указателя и его инициализация значением указателя iptr
```

- **определение указателя вне любой функции с предписанием static**
начальное его значение – нулевой адрес памяти:

```
static int *kptr;

• инициализация указателя явным адресом памяти с последующим приведением типа
int main ()
{ int a, b[5]={1,2,3,4,5};
  int *pi = reinterpret_cast <int*> (0x0012ff44);
                                     //адрес массива(известен, например, при отладке)

  for (int i=0; i<5; i++)
      {cout << *(pi+i) << endl;          //1 2 3 4 5
       _getch();
      }
  return 0;
}
```

Заметим, что операция **reinterpret_cast** применяется для преобразования не связанных между собой типов (указателей в целые или наоборот, указателей в указатели и пр.) и позволяет программисту **обозначать опасные преобразования**, результат которых остается на его совести.

- **инициализация указателя именем массива или присвоение ему имени функции, которые трактуются как адреса-константы**

```
int b[10];           //массив
int *ptr = b;        //инициализация указателя адресом массива

void f (int a) {/*тело функции*/}    //определение функции f, имя функции в C++
                                     // - это указатель-константа на функцию, равный
                                     //адресу первой машинной команды функции

void (*pf)(int)=f;    //pf - указатель на функцию с аргументом типа int
                     // и не возвращающую значения
                     //присваивание указателю pf адреса функции f
```

- **инициализация указателя нулевым значением**

```
int *pi = NULL;           //pi - указатель инициализируется константой, определенной как
                          //нулевой (пустой) указатель
int *pi = 0;              // нулевой указатель, еще одна форма инициализации
```

Нулевой указатель не обязательно связан с участком памяти, имеющим нулевой адрес или хранящим нулевое значение. Он просто отличен от указателя на любой объект. Попытка разыменования нулевого указателя бесперспективна.

- **инициализация указателя адресом участка динамической памяти выделяемого с помощью операции new:**

```
int *nptr = new int;      //определение статической переменной-указателя nptr и
                          //инициализация ее адресом памяти, выделенной для динамической переменной типа int;
delete nptr;              // уничтожение динамической переменной;
int *kptr = new int(10);  //определение статической переменной kptr, инициализация
                          // ее адресом памяти, выделенной для динамической переменной типа int;
                          //и инициализация этой переменной значением 10;
delete kptr;              // уничтожение динамической переменной;

int *mptr= new int[10];   //выделение памяти под динамический массив из 10 элементов типа
                          // int, запись адреса этой памяти в статическую переменную mptr
delete [ ] mptr;          // освобождение памяти, занимаемой массивом;
```

или с помощью функции malloc:

```
int *iptr; //определение статической переменной-указателя
iptr = reinterpret_cast <int*> ( malloc( sizeof(int) ) ); //выделение sizeof(int) байт памяти,
//и приведение типа значения, возвращаемого функцией malloc ( void*, указатель на
// объект типа void), к типу int* (указатель на объект типа int)
*iptr = 421; //присваивание значения динамической переменной
free (iptr); // уничтожение динамической переменной;
```



Операции над указателями

Для указателей-переменных разрешены операции:

- доступа по указателю (разыменования или косвенного обращения к объекту (*)),
- присваивания,
- приведение типов,
- сложение с константой,
- инкремента или декремента,
- вычитание двух указателей,
- сравнение указателей (одного типа),
- получения адреса (&).

Примеры работы с указателями

доступ по указателю

//пример 1

```
int main()
{
    int ivar1, ivar2;           //определение переменных целого типа
    int* iptr;                 //определение указателя на объект типа int
    iptr = &ivar1;             //инициализация указателя на int адресом переменной var1
    *iptr = 37;                //присваивание переменной var1=37
    ivar2 = *iptr;             //присваивание переменной var2 значения переменной var1
    cout << ivar2 << endl;     //37
    _getch();
    return 0;
}
```

//пример 2

```
int main()
{
    int ivar1 = 11;
    int ivar2 = 22;
    cout << &ivar1 << endl ;   //вывод на экран значения адреса переменной ivar1
    cout << &ivar2 << endl ;   //вывод на экран значения адреса переменной ivar2
    int *iptr = &ivar1;        //инициализация указателя на int iptr адресом ivar1
    cout << iptr << endl;       //вывод содержимого iptr (значения адреса)
    cout << *iptr << endl;      //вывод разыменованного содержимого iptr
                                //(значения переменной ivar1=11)
    iptr = &ivar2;              //указателю iptr присваивается адрес ivar2
    cout << iptr << endl;       //вывод нового содержимого iptr (значения адреса)
    cout << *iptr << endl;      //вывод разыменованного содержимого iptr
                                //(значения переменной ivar2=22)
    _getch();
    return 0;
}
```

присваивание и приведение типа

//1

```
int main()
{
    int intvar;
    float flovar;
    int* ptring;                //определение указателя на int
    float* ptrflo;              //определение указателя на float
    void* ptrvoid;              //определение указателя на void

    ptring = &intvar;           //ok, int* to int*
```

```

//ptrint = &flovar;           //error, float* to int*
//ptrflo = &intvar;           //error, int* to float*
    ptrflo = &flovar;         //ok, float* to float*

    ptrvoid = &intvar;         //ok, int* to void*
    ptrvoid = &flovar;         //ok, float* to void*
    _getch();
    return 0;
}
//2
int main()
{ int intvar=5; float flovar=5.0f; double dvar=5.0;
  int* ptrint =NULL;          //определение и инициализация указателя на объект типа int
  float* ptrflo =NULL;        // определение и инициализация указателя на объект типа float
  double* ptrd =NULL;         // определение и инициализация указателя на объект типа double
  void* ptrvoid =NULL;        // определение и инициализация указателя на void

                                //Операция присваивания выполнена верно:
  ptrint = &intvar;
                                //присваивание указателю на объект типа int адреса объекта типа int
  ptrflo = &flovar;
                                //присваивание указателю на объект типа float адреса объекта типа float
  ptrd = &dvar;
                                // присваивание указателю на объект типа double адреса объекта типа double
  ptrvoid = &intvar;
                                //присваивание указателю на void адреса объекта типа int
  ptrvoid = &flovar;
                                // присваивание указателю на void адреса объекта типа float
  ptrvoid = &dvar;
                                // присваивание указателю на void адреса объекта типа double

                                //Операция присваивания выполнена неверно:
  //ptrflo = &intvar;          //ошибка !!!
                                //присваивание указателю на объект типа float адреса переменной типа int
  //ptrint = &flovar;          //ошибка !!!
                                //присваивание указателю на объект типа int адреса переменной типа float
  //ptrint = &dvar;            //ошибка !!!
                                //присваивание указателю на объект типа int адреса переменной типа double

  //Разрешено неявное (умалчиваемое) преобразование любого неконстантного и
  //не имеющего модификатора volatile1 указателя к указателю типа void*
                                //Неявное приведение типа выполняется верно:
  ptrvoid = ptrint;
                                // присваивание указателю на void указателя на объект типа int
  ptrvoid = ptrflo;
                                // присваивание указателю на void указателя на объект типа float
  ptrvoid = ptrd;
                                // присваивание указателю на void указателя на объект типа double

                                //Неявное приведение типа выполняется неверно:
  //ptrint = ptrvoid;          //ошибка !!!
                                // указателю на объект типа int присваивается указатель на void
  //ptrflo = ptrvoid;          //ошибка !!!

```

¹ Квалификатор *volatile* указывает транслятору, что в процессе выполнения программы значение объекта может изменяться между явными обращениями к нему (может повлиять какое-то внешнее событие) и что он при оптимизации не должен делать никаких предположений о содержимом переменной:

```

volatile int i;
...
i=i;
while (i);

```

– транслятор будет проверять значение *i* перед каждой итерацией.

```

// указателю на объект типа float присваивается указатель на void
//ptrd = ptrvoid; //ошибка !!!
// указателю на объект типа double присваивается указатель на void

//Явное приведение типа выполняется верно:
ptrint = reinterpret_cast <int*> (ptrvoid);
// указатель на void приводится к указателю на объект типа int
ptrflo = reinterpret_cast <float*> (ptrvoid);
// указатель на void приводится к указателю на объект типа float
ptrd = reinterpret_cast <double*> (ptrvoid);
// указатель на void приводится к указателю на объект типа double

ptrvoid = reinterpret_cast <void*> (ptrint);
// указатель на объект типа int приводится к указателю на void
ptrvoid = reinterpret_cast <void*> (ptrflo);
// указатель на объект типа float приводится к указателю на void
ptrvoid = reinterpret_cast <void*> (ptrd);
// указатель на объект типа double приводится к указателю на void

ptrint = reinterpret_cast <int*> (&dvar);
// адрес переменной типа double приводится к указателю на объект типа int
ptrflo = reinterpret_cast <float*> (&intvar);
// адрес переменной типа int приводится к указателю на объект типа float
ptrint = reinterpret_cast <int*> (&floatvar);
// адрес переменной типа float приводится к указателю на объект типа int
_getch();
return 0;
}

```

особенности операции явного приведения типов

Как мы уже знаем, операция **static_cast** используется для преобразования родственных типов. Операция **reinterpret_cast** обеспечивает преобразование независимых типов, поэтому компилятор не может проверить допустимость заданного преобразования, и вся ответственность ложится на программиста

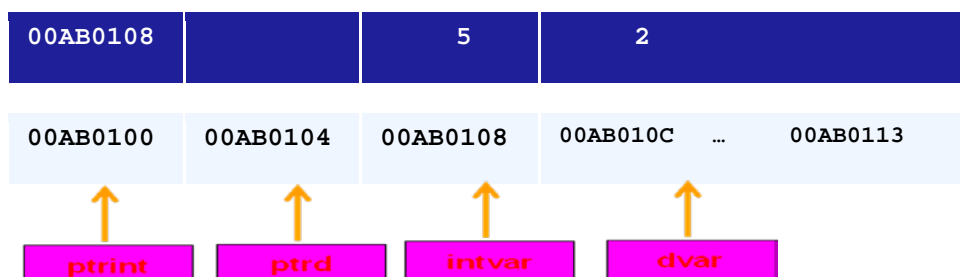
Используйте reinterpret_cast с пониманием последствий!!!

Проанализируйте приведенный пример и его схематическое изображение и определите ошибочные действия программы:

```

int main()
{ int intvar=5;
  double dvar=2.0;
  int *ptrint = &intvar; double *ptrd=NULL;
  ptrd = reinterpret_cast <double*> (&intvar);
  //явное приведение адреса переменной типа int к указателю на double
  _getch();
  return 0;
}

```




//3.определить результат:

```
int main()
{
    unsigned long block = 0xffeeddccL;    // переменная block занимает 4 байта
    void *ptr = &block;                  //указатель на переменную block
    unsigned char ch;
    unsigned short two_bytes;
    unsigned long four_bytes;            //адрес переменной есть адрес ее младшего байта
    ch = *( reinterpret_cast <unsigned char*>(ptr));
        //приведение указателя ptr к типу «указатель на unsigned char» и взятие значения
    printf ("%x", ch);                    // cc (1 байт)

    two_bytes = *( reinterpret_cast <unsigned short *>(ptr));
        //приведение указателя к типу «указатель на unsigned short»
        // и взятие значения (2 байта)
    printf ("%x", two_bytes);              // ddcc

    four_bytes = *( reinterpret_cast <unsigned long *>(ptr));
        //приведение указателя к типу «указатель на unsigned long» и
        // взятие значения (4 байта)
    printf ("%x", four_bytes);              // ffeeddcc

    _getch();
    return 0;
}
```



The diagram illustrates a memory layout. A horizontal rectangle is divided into four equal segments, each containing two lowercase letters: 'ff', 'ee', 'dd', and 'cc' from left to right. To the right of this rectangle is a vertical line. An arrow points from this vertical line to the left edge of the first segment ('ff'). Above the arrow, the word 'адрес' (address) is written.

СЛОЖЕНИЕ С КОНСТАНТОЙ И ВЫЧИТАНИЕ КОНСТАНТЫ

type * ptr1, *ptr2;
ptr1 + N → ptr1 + N * sizeof (type);
ptr1 - N → ptr1 - N * sizeof (type);
ptr2 - ptr1 = (адрес2 - адрес1) / sizeof (type);

char *cptr;

если адрес равен 2000, то **cptr++** будет равно 2001;

int *iptr;

если адрес равен 2000, то **iptr++** будет равно 2004;

long *lptr;

если адрес равен 2000, то **lptr++** будет равно 2004.

$ptr + N \rightarrow ptr + N * \text{sizeof} (type);$

$ptr - N \rightarrow ptr - N * \text{sizeof} (type);$

Такие правила арифметических операций с указателями вытекают из того, что указатель в C++ неявно рассматривается как указатель на начало массива однотипных элементов. Продвижение указателя вперед или назад совпадает с увеличением или уменьшением индекса элемента.

операции инкремента и декремента

!!! Помним об особенностях префиксных операций инкремента и декремента: они lvalue, поэтому выражения вида ++p =...; являются верными. В то же время, выражения вида p++ =...; где p – указатель, дадут ошибку компиляции, т.к. постфиксные операции инкремента и декремента не есть lvalue.

```
int main ()
{int i=5, j=6, *pi =&i, *pj =&j;
*pi=*pj;
cout << pi << endl;      // 0012FF60
cout << pj << endl;      // 0012FF54
++pi=&j;
cout << pi << endl;      // 0012FF54
cout << pj << endl;      // 0012FF54
_getch();
return 0;
}

int main ()
{int i=5, j=6, *pi =&i, *pj =&j;
*pi=*pj;
cout << pi << endl;
cout << pj << endl;
//pi++=&j;                error C2106: '=' : left operand must be l-value
cout << pi << endl;
cout << pj << endl;
_getch();
return 0;
}
```

Кроме того, если name есть имя некоторого объекта, то недопустимы записи:

++&name; //ошибка, требуется l-выражение

--&name++; // ошибка, требуется l-выражение

Адрес участка памяти есть константа (т.е. не есть леводопустимое выражение) и его нельзя изменять.

Благодаря приоритетам и ассоциативности операторов ++, -- и *, можно в одном выражении совместить доступ к элементу и смещение указателя:

*p++=8; // 8 засылается в текущий элемент, затем указатель сдвигается на следующий элемент

*p--=8; // 8 засылается в текущий элемент, затем указатель сдвигается на предыдущий элемент

*++p=8; // сначала указатель сдвигается на следующий элемент, затем в этот элемент засылается 8.

*--p=8; // сначала указатель сдвигается на предыдущий элемент, затем в этот элемент засылается 8.

А вот как меняем не указатель, а значение переменной, на которую он указывает:

```
(*p)++;
(*p)--;
--*p;
++*p;
```

сравнение

```
int main ()
{int i=5, j=6, *pi =&i, *pj =&j;
*pi=*pj;                                // i=j;

if (pi==pj) pi=0;                       //сравниваем значения указателей
else if (*pi==*pj) pj=pi;               //они не равны, сравниваем разыменованные значения
                                         // (они равны), тогда
                                         // присваиваем указателю pj значение pi
```

```

if (pi==pj) *pj=4;           //раз значения указателей равны, j присваиваем значение 4
cout << *pi<< endl;         // i=4, т.к. значения указателей равны
_getch();
return 0;
}

int main ()
{int i=66, j=65, *pi =&i, *pj =&j;      //65 - ASCII-код символа A
  char c='a', *pc =&c;
  pi=pj;                               //ок!!! указателю pi присваивается адрес переменной j
//pj=pc;                               // ошибка! указателю pj нельзя присвоить адрес переменной c
//pj=*pi;                              // ошибка! указателю присваивается значение переменной
//*pi=NULL;                            //ошибка! значению присваивается нулевой адрес
*pc=*pi;                               //ок!!! значению c присваивается значение i
// if (pc !=NULL) *pc=NULL;            //ошибка! разыменовывается нулевой указатель
if (pj !=NULL) *pj=*pi;                //ок!!!
if (pj !=NULL) {cin >> *pi; *pj=*pi;}   //ок!!! вводится значение i
if (pj == pi) cout << pj <<endl;        //ок!!!
//if (pj != pc) cin >> *pc;            // ошибка! содержимое указателя pj сравнивается с
// содержимым указателя pc
cout << *pc<< endl;                    //ок!!! A
_getch();
return 0;
}

```

инициализация указателя с помощью функции malloc

```

#include <cstdlib>
int main () {
int *xiptr, *wiptr, yi, zi;           //память под указатель выделяется на этапе компиляции
xiptr=(int *) malloc (sizeof(int));    //память для инициализации указателя
//выделяется в процессе выполнения программы
//еще лучше, вдруг в куче нет места:
// if ( (xiptr=( reinterpret_cast <int *>( malloc (sizeof (int)) ) ) !=0){....}
*xiptr=16;
yi=-15;
wiptr=&yi;
printf ("\nsizeof(xiptr) = %d", sizeof(xiptr));
printf ("xiptr = %p", xiptr);
printf ("*xiptr = %d", *xiptr);
printf ("\n&yi = %p", &yi);
printf ("\n&zi = %p", &zi);
printf ("\n*wiptr = %d\n", *wiptr);
_getch();
return 0;
}

```

Результат:

```

sizeof(xiptr) = 4   xiptr = 004301F0   *xiptr = 16
&yi = 0012FF74
&zi = 0012FF70
*wiptr = -15

```

Виды указателей

указатель на константу

```

int var1=5, *pvar1=&var1;
int var2=6;

```

```

const int cvar1 = 1; // cvar1 – целая константа
                        // pvar1, pvar11 – указатели на константу
const int *pvar1 = &cvar1; // const запрещает изменение значения *pvar1
const int *pvar11 = &var2; // const запрещает изменение значения *pvar11
cout << *pvar1 << " " << *pvar1 << " " << *pvar11 << endl; // 5 1 6

// *pvar11=*pvar1; // l-value specifies const object
// *pvar1=*pvar11; // l-value specifies const object

pvar11 = pvar1; // OK!!! изменение самого указателя разрешено
pvar1 = pvar1; // OK!!! изменение самого указателя разрешено

cout << *pvar1 << " " << *pvar1 << " " << *pvar11 << endl; // 5 5 1

```

указатель-константа

```

int var1=5, *pvar1=&var1;
int var2=6;
const int cvar1 = 1; // cvar1 – целая константа
const int *pvar1 = &cvar1; // pvar1 – указатель на константу
                        // cp1 – константный указатель
int *const cp1 = &var1; // const запрещает изменение значения указателя
cout << *cp1 << endl; // 5
// cp1 = pvar1; // l-value specifies const object
*cp1 = var2; // OK!!! изменение содержимого по указателю разрешено
cout << *cp1 << endl; // 6
*cp1 = *pvar1; // OK!!! изменение содержимого по указателю разрешено
cout << *cp1 << endl; // 1

// int *const cp2 = &cvar1; // cannot convert from 'const int *' to 'int *const '
// cp1 = pvar1; // cannot convert from 'const int *' to 'int *const '
// cp1 = reinterpret_cast<int *const>(pvar1);
// 'reinterpret_cast': cannot convert from 'const int *' to 'int *const '

```

не спасает положение

указатель-константа на константу

```

int var1=5, *pvar1=&var1;
int var2=6;
const int cvar1 = 1; // cvar1 – целая константа
                        // cp1, cp2 – константные указатели на константу
                        // модификатор const запрещает изменение значения
const int *const cp1 = &var1; // и указателя, и содержимого *cp1, *cp2
const int *const cp2 = &cvar1;
cout << *cp1 << endl; // 5 1

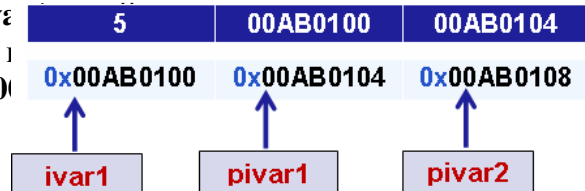
```

```
// cpc1 = pvar1; // l-value specifies const object
// *cpc1 = 8; // l-value specifies const object
```

указатель на указатель

Сам указатель также имеет адрес:

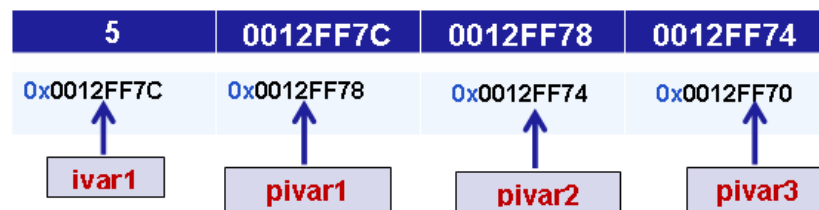
```
int ivar1=5;
int *pivar1 = &ivar1, **pivar2=&pivar1;
cout << *pivar1<< endl; // 5, 1
cout << *pivar2 <<endl; //00AB0100
cout << **pivar2 <<endl; //5
```



указатель на указатель на указатель на ...

```
int ivar1=5;
int *pivar1 = &ivar1;
int **pivar2 = &pivar1;
int ***pivar3 = &pivar2;
cout << ivar1 << " " << &ivar1 << endl; //5 0x0012FF7C
cout << *pivar1 << " " << &pivar1 << endl; //5 0x0012FF78
cout << **pivar2 << " " << &pivar2 << endl; //5 0x0012FF74
cout << ***pivar3 << " " << &pivar3 << endl; //5 0x0012FF70
```

```
cout << pivar1 << " " << *pivar1 << endl; // 0x0012FF7C 5
cout << pivar2 << " " << *pivar2 << " " << **pivar2 << endl;
// 0x0012FF78 0x0012FF7C 5
cout << pivar3 << " " << *pivar3 << " " << **pivar3 << " " << ***pivar3 << endl;
//0x0012FF74 0x0012FF78 0x0012FF7C 5
```



Ссылки

Ссылка, в отличие от указателя, не занимает дополнительного пространства в памяти и является просто другим именем (псевдонимом) для переменной. Ссылка представляет собой синоним имени переменной, указанной при ее инициализации, т.е. объявленной ранее. Имя ссылки может использоваться вместо имени переменной. Ссылку можно рассматривать как указатель, который всегда разыменовывается. В отличие от указателя ссылка не может быть изменена, чтобы представлять другую переменную. Но на данную переменную может быть объявлено несколько ссылок.

Формат объявления ссылки:

Базовый_тип &Имя_Ссылки = Имя_Переменной;

Например:

```
int kol;
```

```
int &pal = kol; //ссылка pal – альтернативное имя для ko
```

Инициализация ссылки:

```
const char& CR = '\n'; // CR – ссылка на симв.константу инициализируется симв.константой !!!
```

```
int i = 0;
```

```
int &rint = i; // ссылка rint синоним переменной i
```

```
int *pint = &i; // указатель pint
```

```
// int &jj = 1; ошибка!!!
```

```
const int j(17);
```

```
// int& jjj = j; ошибка!!!
```

Ссылка на переменную не может инициализироваться константой:

компилятор выдаст ошибку, так как предполагает, что последует изменение величин, на которые ссылаются jj и jjj, и на всякий случай устраняет искушение. Иначе говоря, константность – свойство переменной, а не данных, поэтому **неконстантная переменная не может сослаться на константную величину.**

Ссылку, можно использовать вместо имени исходной переменной:

```
int main () {  
int i = 0;  
int &rint = i; // ссылка rint синоним переменной i  
int *pint = &i; // указатель pint проинициализирован адресом i  
rint +=10; //ссылка не требует разыменования  
*pint +=10; // для указателя требуется разыменование  
int &rr = i; //вторая ссылка – rr – синоним переменной i  
int *p = &rr; //еще один указатель на i, но через ссылку!!!  
_getch();  
return 0;  
}
```

Правила:

- Ссылка явно инициализируется при ее описании, кроме случаев, когда она является параметром функции, описана как extern или ссылается на поле данных класса.
- Ссылке после инициализации не может быть присвоена другая переменная, но на данную переменную может быть объявлено несколько ссылок.
- Тип ссылки должен совпадать с типом величины, на которую она ссылается.
- **Не разрешается определять указатели на ссылки, создавать массивы ссылок и ссылки на ссылки.**
- **Указатель можно инициализировать адресом ссылки.**
- Ссылки применяются чаще всего в качестве параметров функций и типов значений, возвращаемых функциями.