

# Указатель

C++ унаследовал от C возможность работы на низком уровне

Пусть мы имеем ячейку памяти `int i = 5`

Объявление

```
int *p;
```

вводит указатель, то есть переменную, которая может хранить адрес ячейки памяти с любым `int`, например, `i`

```
int i = 5;
```

```
int *p;
```

```
p = &i // В указателе p хранится адрес ячейки памяти i
```



Можно использовать **нулевой указатель**, чтобы показать, что переменная-указатель пока не хранит никакого адреса

Для этого есть несколько способов

```
p = NULL;    // Так часто делали в С. Макрос NULL определён в <stdlib>.  
p = 0;       // Так советует Страуструп для С++98.  
p = nullptr; // С++11
```

# Объявление vs разыменование

Если \* после типа переменной, то это объявление указателя

```
int *p;
```

Если \* пишется перед именем переменной, то эта переменная — указатель,

а \* — операция разыменования

```
*p = 6 // Операция разыменования
```



# Указатели и ссылки

//Указатели

```
int *p = &i;
```

```
*p = 6;
```

// Ссылки

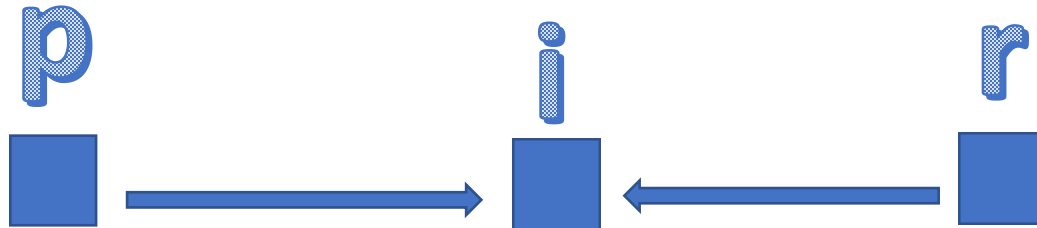
```
int &r = i; // i и r — одна ячейка памяти
```

```
r = 6;
```

Если & пишется после названия типа, то это ссылка.

В противном случае, если он пишется перед именем переменной, то это адрес этой переменной.

Ссылку можно трактовать, как указатель, который постоянно находится в разыменованном состоянии.



# Передача параметров в функции

## По ссылке:

```
void q(int &r) {  
    r++;  
}
```

```
int i = 5;  
q(i);    // i == 6
```

## По указателю:

```
void q(int *p) {  
    (*p)++; // Скобочки  
            важны!!!  
}
```

```
int i = 5;  
q(&i);    // i == 6
```

Производительность в обоих случаях одинаковая.

# Указатель void\*

`void *p; // указатель на область памяти`

`int i = 5;  
p = &i;`

`double d = 3.14;  
p = &d;`  
т.е. p может хранить адрес любого объекта

```
void *p = &i
```

```
// Ошибка компиляции: нельзя разыменовывать p  
*p = 6
```

```
// Явное приведение к типу int* в стиле C  
*(int*)p = 6;
```

```
// Использование представляет опасность, если i не является int
```

```
// Стил С++ более явно заявляет об этой опасности:
```

```
*static_cast<int*>(p) = 6;    // но работает точно так же, как и выше
```

```
double d = 3.14;
```

```
p = &d;
```

```
*static_cast<double*>(p) = 2.8;
```

# Приведение типов

Невозможно выполнить:

```
int *pa;  
double *pb;  
pa = pb;  
pb = pa;
```

Но можно с помощью явного приведения типов:

в стиле C или

в стиле C++, но не `static_cast`, а `reinterpret_cast`

```
pa = reinterpret_cast< int * > (pb);
```



# Указатели на структуры

```
struct Person {  
    string name;  
    int age;  
};
```

```
Person p {"Иванов", 19};
```

```
Person *pp = &p;
```

```
(*pp).age = 20;
```

```
pp -> age = 20; // Операция доступа к полю в памяти
```

# Указатели и константность

```
int i = 5;  
int *p = &i;
```

```
const int *cp = &i;    // Указатель на константу  
cout << *cp;  
(*cp)++;              // Ошибка компиляции
```

Это используется при передаче аргументов в функции.

Объявление `const int *p` заведомо не позволяет написать функцию, которая изменяет переменную, переданную через указатель.

```
void q(const int *p) {  
    (*p)++;    // Ошибка компиляции  
    cout << *p;  
}
```

# Константные указатели

```
int i = 5;  
int j = 7;  
const int n = 10;           // Обычная константа  
int* const pc = &i;         // Константный указатель  
pc = &j;                     // Ошибка компиляции  
(*pc)++;                    // А здесь ошибки не будет
```

Другой пример. Нельзя обычному указателю присваивать адрес константы:

```
const int n = 10;  
int* pn = &n;               // Ошибка компиляции
```

```
const int *pn = &n;  
*pn = 11;                   // Ошибка компиляции  
cout << *pn;
```

Однако возможно заставить компилятор снять константность:

```
*const_cast<int*>(pn) = 11;
```

# Константные указатели на константу

```
const int* const pn = &n;
```

# Константный указатель и указатель на константу

```
int a=100;    //два обычных объекта типа int  
int b=222;
```

```
int *const P2=&a; //Константный указатель  
*P2=987;         //Менять значение разрешено  
//P2=&b;         //Но изменять адрес не разрешается
```

```
const int *P1=&a; //Указатель на константу  
//*P1=110;       //Менять значение нельзя  
P1=&b;           //Но менять адрес разрешено
```

```
const int *const P3=&a; //Константный указатель на константу  
//*P3=155;           //Изменять нельзя ни значение  
//P3=&b;             //Ни адрес к которому такой указатель привязан
```

# Указатели и массивы

В C++ указатели и массивы тесно связаны

```
int a[10];
```

```
int* p = &a[0]; // адрес первого элемента  
*p = 5;
```

**a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]**

5

\*p

p=a



# Операции при работе с указателями

Выполним следующую операцию r++;

Теперь указатель r указывает на следующий (второй) элемент массива.

**a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]**

5

\*p

Операция r++ увеличивает адрес в зависимости от типа указателя т.е.  
r++ равносильно тому, что r передвинется на sizeof(int) байт



# Операции при работе с указателями

```
r += 1;    // Переход на следующий элемент массива  
r += n;    // Увеличение на n элементов массива  
r+1        // Адрес следующего элемента  
r1 = r+n;  // Записать в `r1` адрес n-го элемента  
r1 - r = n; // Количество элементов между указателями
```

А вот складывать указатели нельзя!!!



# Связь массивов и указателей

$*(p+0)$  и  $a[0]$  - являются ссылками на первый элемент массива

$*(p+1)$  и  $a[1]$  - являются ссылками на второй элемент массива

$*(p+2)$  и  $a[2]$  - являются ссылками на третий элемент массива

$a[0]$   $a[1]$   $a[2]$   $a[3]$   $a[4]$   $a[5]$   $a[6]$   $a[7]$   $a[8]$   $a[9]$

5

$*(p+0)$

$a[0]$   $a[1]$   $a[2]$   $a[3]$   $a[4]$   $a[5]$   $a[6]$   $a[7]$   $a[8]$   $a[9]$

5

$*(p+1)$

# Связь массивов и указателей

Имя массива **a** может быть неявно преобразовано к указателю на свой первый элемент.

Т.е на самом деле, **a** — это указатель на первый элемент массива.  
Значит, мы можем написать проще:

```
int* p = a;
```

**a** является константным указателем на свой первый элемент, т. е он как бы описан таким образом

```
int* const a;
```

Отсюда становится понятно, почему **нельзя** писать **a = a1**.

Т.к. имя массива константный указатель, то нельзя присваивать один массив другому.

# Связь массивов и указателей

Вообще говоря, более строгая связь массивов и указателей выглядит следующим образом:

**$a[n] == *(a+n)$**

Отсюда следует вывод (крамольная истина):

В языке C/C++ массивов нет - есть только указатели!!!

**Следствие 1.** Понятно, почему нет контроля выхода за границы массива.

**Следствие 2.** Понятно, почему массивы индексируются с нуля. Это самое эффективное по этой формуле.

**Следствие 3.**  $a[n] == *(a+n) == *(n+a) == n[a]$

```
int a[]={1,2,3,4,5};  
int n = 2;  
cout<<a[n]<<' '<< *(a+n)<<endl;  
cout<< *(n+a)<<' '<< n[a]<<endl;
```

# Идиома \*p++

Идиома - устойчивое выражение, которое воспринимается как единое целое.

Теперь допустим, нам необходимо сделать следующее:

```
int a[10];  
int* p = a;  
*p = 3;  
p++;
```

А что, если записать \*p++=3 ?

Как это можно воспринимать? Как \*(p++) или как \*(p)++ или как (\*p)++ ?

В C/C++ унарные операции ассоциируются справа налево, поэтому в данном случае ++ относится к указателю,

```
// *p++ ~ *(p++) верно!!!
```

```
// *p++ ~ (*p)++ неверно!!!
```

## Пример 1. Заполнить массив а нулями

```
int a[10];  
//int* p = a; // Однако, это можно перенести в раздел инициализации  
for(;;)
```

```
for(int* p = a; p != a+10; *p++ = 0);
```

```
for(int* p = a; p != a+10; p++)  
    *p = 0;
```

```
for(int* p = a; p != a+10; ++p)  
    *p = 0;
```

## Пример 2

Даны 2 массива `int a[10], b[10]`.

Необходимо заполнить 3-й массив `c[10]` суммой элементов массивов `a[10]` и `b[10]`.

```
int *pa = a, *pb = b, *pc = c;
```

```
//for(; pa != a + 10;)
```

```
while(pa != a + 10)
```

```
    *pc++ = *pa++ + *pb++;
```

# Передача массива в функцию с помощью указателя

```
void InitZero(int* a, int n) {  
    for(int* p = a; p != a+n;)   
        *p++ = 0;  
}
```

```
// int* a ~ int a[] ?
```



# Как читать сложные объявления

# Функция и указатель на функцию

```
void (*funcPtr)();  
void *funcPtr();
```

# Правило прочтения объявлений

*Используйте для прочтения сложных объявлений прием «движения изнутри наружу чередуя направо и налево».*

# Перенасыщенные скобками объявления

- `int (*pf)(); // *pf; указатель на функцию, возвращающую int`
- `char ** argv;`
- `int (* daytab)[13];`
- `void *comp();`
- `void (*comp)();`

```
char (*(x[3]))[5];
```

```
char (*(pArrPChar ())[5]);
```

```
void (*(fp1)(int))[10];
```

fp1 – указатель на функцию, которая получает аргумент типа int и возвращает указатель на массив из 10 указателей типа void;

```
float (*(fp2)(int,int,float))(int);
```

fp2 – указатель на функцию, которая получает три аргумента (int, int и float) и возвращает указатель на функцию, получающую аргумент int и возвращающую float;

```
int (*(fp3())[10]);
```

fp3 – функция, которая возвращает указатель на массив из 10 указателей на функции, возвращающие значения типа int

в конструкции typedef

```
typedef double (*( *(*fp4)())[10])();
```

```
fp4 a,b,c;
```

```
typedef double (*f) (int a, int b);  
f x, y;
```

# Ссылки на константы

```
int i = 5;  
int& ci = i;
```

```
const int& cci = i; // Здесь все будет нормально
```

```
const int n = 10;  
int& cn = n; // Такое компилятор запретит  
int& cn = const_cast<int&>(n);
```

```
const int& ccn = n;
```