# Указатели

```
int b;
```

# Оператор адреса &

Позволяет узнать, какой адрес памяти присвоен определённой переменной. Всё довольно просто:

- int a = 7;
- std::cout << a << '\n'; // выводим значение переменной a
- std::cout << &a << '\n'; // выводим адрес памяти переменной a

Результат:

7

0046FCF0

# Оператор разыменования (*)

- позволяет получить значение по указанному адресу:

- int a = 7;
- std::cout << a << '\n'; // выводим значение переменной a
- std::cout << &a << '\n'; // выводим адрес переменной a
- std::cout << *&a << '\n'; /// выводим значение ячейки памяти переменной a

7

0046FCF0

7

# Указатели

Переменная, значением которой является адрес (ячейка) памяти. Указатели объявляются точно так же, как и обычные переменные, только со звёздочкой между типом данных и идентификатором:

int *iPtr; // указатель на значение типа int

double *dPtr; // указатель на значение типа double

int* iPtr3; // корректный синтаксис (допустимый, но нежелателен)

int * iPtr4; // корректный синтаксис (не делайте так)

int *iPtr5, *iPtr6; // объявляем два указателя для переменных типа int

# Присваивание значений указателю

Поскольку указатели содержат только адреса, то при присваивании указателю значения — это значение должно быть адресом. Для получения адреса переменной используется оператор адреса:

- int value = 5;
- int *ptr = &value; // инициализируем ptr адресом значения переменной

```
#include <iostream>

int main()
{
    int value = 5;
    int *ptr = &value; // инициализируем ptr адресом значения переменной

    std::cout << &value << '\n'; // выводим адрес значения переменной value
    std::cout << ptr << '\n'; // выводим адрес, который хранит ptr

    return 0;
}
```

Результат:

```
003AFCD4
003AFCD4
```

# Тип указателя должен соответствовать типу переменной, на которую он указывает

```
int iValue = 7;

double dValue = 9.0;

int *iPtr = &iValue; // ок

double *dPtr = &dValue; // ок

iPtr = &dValue; // неправильно: указатель типа int не может указывать на адрес переменной типа double

dPtr = &iValue; // неправильно: указатель типа double не может указывать на адрес переменной типа int
```

Следующее не является допустимым:

```
int *ptr = 7;
```

C++ также не позволит вам напрямую присваивать адреса памяти указателю:

- double *dPtr = 0x0012FF7C; // не ок: рассматривается как присваивание целочисленного литерала

# Разыменование указателей

Разыменованный указатель — это содержимое ячейки памяти, на которую он указывает

```
int value = 5;
std::cout << &value << std::endl; // выводим адрес value
std::cout << value << std::endl; // выводим содержимое value

int *ptr = &value; // ptr указывает на value
std::cout << ptr << std::endl; // выводим адрес, который хранится в ptr, т.е. &value
std::cout << *ptr << std::endl; // разыменовываем ptr (получаем значение на которое указывает ptr)
```

Результат:

```
0034FD90
5
0034FD90
5
```

# Указатели и массивы

int array[4] = { 5, 8, 6, 4 }; //фиксированный массив

Для компилятора array является переменной типа int[4]. Переменная array содержит адрес первого элемента массива, как если бы это был указатель!

```
int array[4] = { 5, 8, 6, 4 };

    // Выводим значение массива (переменной array)
    std::cout << "The array has address: " << array << '\n';

    // Выводим адрес элемента массива
    std::cout << "Element 0 has address: " << &array[0] << '\n';
```

# Адресная арифметика

Если ptr указывает на целое число, то ptr + 1 является адресом следующего целочисленного значения в памяти после ptr. ptr - 1 — это адрес предыдущего целочисленного значения (перед ptr).

При вычислении результата выражения адресной арифметики (или ещё «арифметики с указателями») компилятор всегда умножает целочисленный операнд на размер объекта, на который указывает указатель. Например:

```
int value = 8;
    int *ptr = &value;

    std::cout << ptr << '\n';
    std::cout << ptr+1 << '\n';
    std::cout << ptr+2 << '\n';
    std::cout << ptr+3 << '\n';
```

Результат:

002CF9A4

002CF9A8

002CF9AC

002CF9B0

# Индексация массивов

- int array [5] = { 7, 8, 2, 4, 5 };
- std::cout << &array[1] << '\n'; *// выведется адрес памяти элемента под номером 1*
- std::cout << array+1 << '\n'; *// выведется адрес памяти указателя на массив + 1*
- std::cout << array[1] << '\n'; *// выведется 8*
- std::cout << *(array+1) << '\n'; *// выведется 8 (обратите внимание на скобки, они здесь обязательны)*

array[n] — это то же самое, что и *(array + n), где n является целочисленным значением.

# Память

- Статическое

  Память выделяется один раз, при запуске программы, и сохраняется на протяжении работы всей программы

- Автоматическое

  Память выделяется при входе в блок, в котором находятся эти переменные, и удаляется при выходе из него.

- Динамическое

# Динамическое выделение памяти

Это способ запроса памяти из операционной системы запущенными программами по надобности. Эта память не выделяется из ограниченной памяти стека программы, а из гораздо большего хранилища, управляемого операционной системой — кучи.

new int; // динамически выделяем целочисленную переменную и сразу же отбрасываем результат (так как нигде его не сохраняем)

int *ptr = new int; // динамически выделяем целочисленную переменную и присваиваем её адрес ptr, чтобы потом иметь доступ к ней

Затем мы можем разыменовать указатель для получения значения:

*ptr = 8; // присваиваем значение 8 только что выделенной

Когда уже всё, что нужно было, выполнено с динамически выделенной переменной — нужно явно указать C++ освободить эту память. Для переменных это выполняется с помощью оператора delete:

// Предположим, что ptr ранее уже был выделен с помощью оператора new

- delete ptr; // возвращаем память, на которую указывал ptr, обратно в операционную систему
- ptr = 0; // делаем ptr нулевым указателем // Предположим, что ptr ранее уже был выделен с помощью оператора new

Без удаления произойдет утечка памяти!

- int *value = new (std::nothrow) int; // запрос на выделение динамической памяти для целочисленного значения
- if (!value) // обрабатываем случай, когда new возвращает null (т.е. память не выделяется)
- {
- // Обработка этого случая
- std::cout << "Could not allocate memory";
- }

# Динамические массивы

- std::cout << "Enter a positive integer: ";
- int length;
- std::cin >> length;

- int *array = new int[length]; // используем оператор new[] для выделения массива.
- std::cout << "I just allocated an array of integers of length " << length << '\n';
- delete[] array; // используем оператор delete[] для освобождения выделенной для массива памяти
- array = 0;

# Инициализация

- int fixedArray[5] = { 9, 7, 5, 3, 1 }; // инициализируем фиксированный массив
- int *array = new int[5] { 9, 7, 5, 3, 1 }; // инициализируем динамический массив

# Практика