Копии переменных

Для начала давайте рассмотрим такой фрагмент кода:

```cpp
#include <iostream>
#include <string>

int main() {
    std::string s1 = "Elementary, my dear Watson!";
    std::string s2 = s1;

    s1.clear();  // s2 никак не изменится

    std::cout << s1 << "\n";  // пустая строка
    std::cout << s2 << "\n";  // Elementary, my dear Watson!
}
```

Важно понимать, что здесь `s2` будет совершенно новой строкой, которая проинициализирована значением `s1`, но более никак с `s1` не связана. Это отличает C++ от некоторых других языков программирования — например, языка Python. В них после аналогичного присваивания строка осталась бы той же самой.

Создание новой строки `s2` требует ресурсов: нужно выделить новый блок памяти и скопировать туда старую строку.

Ссылки

Впрочем, в C++ есть возможность обращаться к уже существующему в памяти объекту под другим именем. Рассмотрим это на примере целых чисел:

```cpp
#include <iostream>

int main() {
    int x = 42;
    int& ref = x;  // ссылка на x

    ++x;
    std::cout << ref << "\n";  // 43
}
```

Здесь `ref` — псевдоним для `x`. Это не самостоятельная переменная, а просто ссылка на объект, уже живущий в памяти. Формально типом `ref` является `int&` — ссылка на `int`.

Аналогично для строк:

```cpp
#include <iostream>
#include <string>

int main() {
    std::string s1 = "Elementary, my dear Watson!";
    std::string& s2 = s1;  // тут ссылка!

    s1.clear();

    std::cout << s2.size() << "\n";  // напечатает 0
}
```

Ссылка должна быть проинициализирована сразу в момент объявления. Например, так написать нельзя:

```
1  int main() {
2      int my_variable = 42;
3      int& ref;  // ошибка!
4      // ...
5      ref = my_variable;
6  }
```

Ссылка привязана к одному и тому же объекту со своего рождения. Переназначить её нельзя:

```
1  int main() {
2      int x = 42, y = 13;
3      int& ref = x;  // OK
4      ref = y;  // ссылка останется привязанной к x, значение x поменяется
5  }
```

Ссылки удобны там, где исходное имя слишком громоздко (например, является вложенным полем какой-либо структуры).

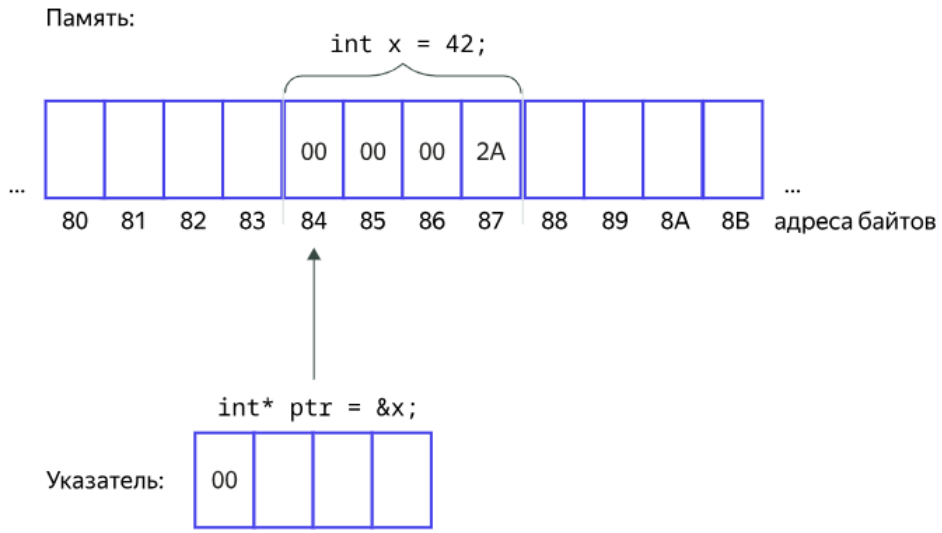
Указатели

Другой (более базовый) способ сослаться на что-то уже существующее в памяти — указатели. Это специальные типы данных, которые могут хранить адрес какой-либо другой переменной в памяти. Здесь мы можем представлять себе память как длинную ленту с пронумерованными ячейками (байтами). Сам адрес переменной можно получить с помощью унарного оператора `&`:

```
1  int main() {
2      int x = 42;
3      int* ptr = &x;  // сохраняем адрес в памяти переменной x в указатель ptr
4
5      ++x;  // увеличим x на единицу
6      std::cout << *ptr << "\n";  // 43
7  }
```

Пример вывода:

```
0x7ffdfee3188c
0x7ffdfee31888
0x7ffdfee31884
```



Формально указатель — это не номер ячейки памяти, а отдельный тип. Но обычно он может быть преобразован к целому числу. Вот такой код напечатает адреса переменных в шестнадцатеричном виде:

```
1  #include <iostream>
2
3  int main() {
4      int x = 1, y = 2, z = 3;
5      std::cout << &x << "\n";
6      std::cout << &y << "\n";
7      std::cout << &z << "\n";
8  }
```

Кроме адреса ячейки памяти переменная-указатель обладает ещё и типом данных, значение которого в этой ячейке лежит. Это позволяет компилятору правильно интерпретировать обращение к памяти по этому адресу. Поэтому мы используем не какой-либо абстрактный тип «указатель», а именно «указатель на `int`».

Оператор разыменования (унарная звёздочка) противоположен оператору взятия адреса (унарному амперсанду). Сравните: `&x` — это адрес `x` в памяти, а `*ptr` — это значение, живущее по адресу, записанному в `ptr`.

Указатели, в отличие от ссылок, можно переназначать. Кроме того, есть выделенное значение никуда не ссылающегося указателя — `nullptr` («нулевой» указатель):

```cpp
#include <iostream>

int main() {
    int x = 42, y = 13;
    int* ptr;  // по умолчанию не инициализируется, тут лежит «случайный» адрес
    ptr = nullptr;  // «нулевой» указатель
    ptr = &x;  // теперь в ptr лежит адрес переменной x
    std::cout << *ptr << "\n";  // 42
    ptr = &y;  // можно поменять адрес, записанный в ptr
    std::cout << *ptr << "\n";  // 13
}
```

Указатель `nullptr` нельзя разыменовывать: это приведёт к неопределённому поведению.

Отдельно рассмотрим указатели на структуру. Для обращения к полям структуры через указатель есть отдельный оператор `->` :

```cpp
#include <iostream>

struct Point {
    double x, y, z;
};

int main() {
    Point p = {3.0, 4.0, 5.0};

    Point* ptr = &p;

    std::cout << (*ptr).x << "\n";  // обращение через * и . требует скобок
    std::cout << ptr->x << "\n";  // то же самое, но чуть короче
}
```

Константность

Константа — это переменная, предназначенная только для чтения. Её значение должно быть зафиксировано в момент присваивания. При этом оно не обязательно должно быть известно в момент компиляции:

```cpp
#include <iostream>

int main() {
    const int c1 = 42;  // эта константа известна в compile time

    int x;
    std::cin >> x;
    const int c2 = 2 * x;  // значение становится известным только в runtime

    c2 = 0;  // ошибка компиляции: константе нельзя присвоить новое значение
}
```

У константного вектора или строки нельзя будет вызвать функции, которые их будут изменять:

```cpp
#include <iostream>
#include <vector>

int main() {
    const std::vector<int> v = {1, 3, 5};
    std::cout << v.size() << "\n";  // OK, напечатает 3
    v.clear();  // ошибка компиляции: константный вектор нельзя изменять
    v[0] = 0;  // тоже ошибка компиляции
}
```

Ссылки и указатели можно комбинировать с константностью:

```
int main() {
    int x = 42;

    int& ref = x;  // обычная ссылка
    const int& cref = x;  // константная ссылка
    ++x;  // OK
    ++ref;  // OK
    ++cref;  // ошибка компиляции: псевдоним cref предназначен только для чтения

    int* ptr = &x;  // обычный указатель
    const int* cptr = &x;  // указатель на константу
    ++*ptr;  // OK
    ++*cptr;  // ошибка компиляции: разыменованный cptr — константа!
}
```

Если исходная переменная уже была константной, то взять обычную ссылку или указатель на неё не получится. Другими словами, константность нельзя просто так отменить, её можно только добавить:

```
int main() {
    const int cx = 42;

    int& ref = cx;  // ошибка компиляции: константность нельзя убрать
    const int& cref = cx;  // OK

    int* ptr = &cx;  // тоже ошибка компиляции
    const int* cptr = &cx;  // OK
}
```

Базовый тип и слово `const` можно менять местами. Так что `const T` и `T const` — это одно и то же. Но следует различать указатель на константу (`const T*`) и константу типа «указатель» (`T* const`):

```
int main() {
    int x = 42;
    const int cx = 13;

    int* ptr = &x;  // обычный указатель
    ptr = &cx;  // ошибка компиляции

    const int* cptr = &x;  // OK: через *cptr нельзя будет изменить x
    cptr = &cx;  // OK

    int* const ptrc = &x;  // OK: *ptrc можно менять, но сам ptrc менять нельзя
    ptrc = nullptr;  // ошибка компиляции

    const int* const cptrc = &x;  // OK, для &cx тоже бы сработало
}
```

Пример в последней строке похож на константную ссылку: указатель `cptrc` не позволяет менять содержимое ячейки `&x` (первый `const`) и в него нельзя записать адрес другой переменной (второй `const`).

Добавление константности

«Висячие» ссылки и указатели

Может так оказаться, что переменная, на адрес которой ссылается указатель, уже вышла из своей области видимости. Похожая ситуация может произойти и со ссылками. В таком случае обращаться к памяти через ссылку или указатель нельзя — это приведёт к неопределённому поведению.

```cpp
#include <iostream>

int main() {
    int* ptr = nullptr;

    {
        int x = 42;
        ptr = &x;
    }

    // обращаться к памяти, в которой жила переменная x, уже нельзя:
    std::cout << *ptr << "\n";  // неопределённое поведение!
}
```

Аналогичная ситуация произойдёт при обращении к уже несуществующему элементу вектора:

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<std::string> words = {"one", "two", "three"};

    std::string& ref = words[0];  // псевдоним для начального элемента вектора

    words.clear();

    // обращаться к ссылке ref уже нельзя!
    std::cout << ref << "\n";  // неопределённое поведение!
}
```

Передача параметров в функцию

Аргументы, которые представляют переменные или константы, могут передаваться в функцию **по значению** (by value) и **по ссылке** (by reference).

Передача аргументов по значению

При передаче аргументов **по значению** функция получает копию значения переменных и констант. Например:

```
1  #include <iostream>
2
3  void square(int);   // прототип функции
4
5  int main()
6  {
7      int n {4};
8      std::cout << "Before square: n = " << n << std::endl;
9      square(n);
10     std::cout << "After square: n = " << n << std::endl;
11 }
12
13 void square(int m)
14 {
15     m = m * m;  // изменяем значение параметра
16     std::cout << "In square: m = " << m << std::endl;
17 }
```

Вывод на экран

```
Before square: n = 4
In square: m = 16
After square: n = 4
```

Функция square принимает число типа `int` и возводит его в квадрат. В функции main перед и после выполнения функции square происходит вывод на консоль значения переменной n, которая передается в square в качестве аргумента.

И при выполнении мы увидим, что изменение параметра m в функции square действуют только в рамках этой функции. Значение переменной n, которое передается в функцию, никак не изменяется:

Передача аргументов по ссылке

При передаче параметров **по ссылке** передается ссылка на объект, через которую мы можем манипулировать самим объектов, а не просто его значением. Так, перепишем предыдущий пример, используя передачу по ссылке:

При передаче параметров **по ссылке** передается ссылка на объект, через которую мы можем манипулировать самим объектов, а не просто его значением. Так, перепишем предыдущий пример, используя передачу по ссылке:

```
#include <iostream>

void square(int&);   // прототип функции

int main()
{
    int n {4};
    std::cout << "Before square: n = " << n << std::endl;
    square(n);
    std::cout << "After square: n = " << n << std::endl;
}
void square(int& m)
{
    m = m * m;  // изменяем значение параметра
    std::cout << "In square: m = " << m << std::endl;
}
```

Вывод на экран

```
Before square: n = 4
In square: m = 16
After square: n = 16
```

Теперь параметр m передается **по ссылке**. Ссылочный параметр связывается непосредственно с объектом, поэтому через ссылку можно менять сам объект. То есть здесь при вызове функции параметр `m` в функции `square` будет представлять тот же объект, что и переменная `n`

Передача по ссылке позволяет возвратить из функции сразу несколько значений. Также передача параметров по ссылке является более эффективной при передаче очень больших объектов. Поскольку в этом случае не происходит копирования значений, а функция использует сам объект, а не его значение.

От передачи аргументов по ссылке следует отличать передачу ссылок в качестве аргументов:

```cpp
#include <iostream>

void square(int);   // прототип функции

int main()
{
    int n = 4;
    int &nRef = n;  // ссылка на переменную n
    std::cout << "Before square: n = " << n << std::endl;
    square(nRef);
    std::cout << "After square: n = " << n << std::endl;
}
void square(int m)
{
    m = m * m;  // изменяем значение параметра
    std::cout << "In square: m = " << m << std::endl;
}
```

Вывод на экран

```
Before square: n = 4
In square: m = 16
After square: n = 4
```

Если функция принимает аргументы по значению, то изменение параметров внутри функции также никак не скажется на внешних объектах, даже если при вызове функции в нее передаются ссылки на объекты.

Указатели в параметрах функции

Параметры функции в C++ могут представлять указатели. Указатели передаются в функцию по значению, то есть функция получает копию указателя. В то же время копия указателя будет в качестве значения иметь тот же адрес, что оригинальный указатель. Поэтому используя в качестве параметров указатели, мы можем получить доступ к значению аргумента и изменить его.

```cpp
#include <iostream>

void increment(int*);

int main()
{
    int n {10};
    increment(&n);
    std::cout << "main function: " << n << std::endl;
}
void increment(int *x)
{
    (*x)++; // получаем значение по адресу в x и увеличиваем его на 1
    std::cout << "increment function: " << *x << std::endl;
}
```

Вывод на экран

```
increment function: 11
main function: 11
```

Для изменения значения параметра применяется операция разыменования с последующим инкрементом: `(*x)++`. Это изменяет значение, которое находится по адресу, хранимому в указателе x.

Поскольку теперь функция в качестве параметра принимает указатель, то при ее вызове необходимо передать адрес переменной: `increment(&n);`.

В итоге изменение параметра x также повлияет на переменную n, потому что оба они хранят адрес на один и тот же участок памяти:

В то же время поскольку аргумент передается в функцию по значению, то есть функция получает копию адреса, то если внутри функции будет изменен адрес указателя, то это не затронет внешний указатель, который передается в качестве аргумента:

```cpp
#include <iostream>

void increment(int*);

int main()
{
    int n {10};
    int *ptr {&n};
    increment(ptr);
    std::cout << "main function: " << *ptr << std::endl;
}
void increment(int *x)
{
    int z {6};
    x = &z;     // переустанавливаем адрес указателя x
    std::cout << "increment function: " << *x << std::endl;
}
```

Вывод на экран

```
increment function: 6
main function: 10
```

В функцию increment передается указатель ptr, который хранит адрес переменной n. При вызове функция increment получает копию этого указателя через параметр x. В функции изменяется адрес указателя x на адрес переменной z. Но это никак не затронет указатель ptr, так как он предствляет другую копию. В итоге поле переустановки адреса указатели x и ptr будут хранить разные адреса.