



РАЗРАБОТКА НА C++

Урок 2.
Область видимости. Передача параметров в
функцию



О ЧЕМ ПОГОВОРИМ СЕГОДНЯ



- Что такое область видимости и время жизни переменной
 - Как классифицируются переменные по области видимости
 - Подробнее рассмотрим передачу параметров в функцию
- 
- 



Время жизни и область видимости

Все переменные имеют определенное **время жизни (lifetime)** и **область видимости (scope)**.

Время жизни начинается с момента определения переменной и длится до ее уничтожения.

Область видимости представляет часть программы, в пределах которой можно использовать объект.

В зависимости от области видимости переменные могут быть **глобальными** и **локальными**.

Глобальные переменные

Это переменные определенные **вне блоков кода**. Новички часто **злоупотребляют** такими переменными, что приводит к ошибкам в больших проектах.

Хорошим тоном является создание **const** переменных в глобальной видимости (например число π). Мы знаем что это число никогда не изменится и часто его используем в расчетах.

```
#include <iostream>
using namespace std;

int n = 1; // глобальная
const double PI = 3.14; // глобальная

int main(){
    cout << n << endl;
    cout << PI << endl;
}
```

Любимые ошибки :)

```
#include <iostream>
using namespace std;
int n = 1; // глобальная

int main() {
    //тут много кода
    int n = 3; //забыли (не заметили) что такое имя уже есть
    //тут ещё много кода
    cout << n << endl; // 3 или 1 ???
    //Правильный ответ: 3, а глобальную теперь можно увидеть только через ::
    // cout << ::n << endl; //НО!! лучше просто следить за именами
}
```

Локальные переменные

Это переменные внутри блоков кода. Локальные переменные бывают **автоматическими (обычными)** или **статическими**.

```
int main()
{
    int n = 1; // локальная переменная
    cout<< n <<endl;
}
```

Автоматические локальные переменные

На протяжении всего первого модуля мы использовали именно такие переменные (в функции `main`, в цикле `for` и так далее)

Автоматические переменные так называются потому, что они **существуют только от момента объявления и до следующей закрывающей фигурной скобки**

Специального ключевого слова не требуется!

```
#include <iostream>
using namespace std;

int main(){
    if (true) {
        int n = 1; // момент объявления
    } // следующая закрывающая скобка!

    cout << n << endl; // больше n не существует
}
```

идентификатор "n" не определен

[Поиск в Интернете](#)

Ключевое слово static

```
#include <iostream>
using namespace std;

void exampleFunction() {
    static int staticVar = 0; // Статическая локальная переменная
    // Увеличиваем значение переменной при каждом вызове
    staticVar++;
    cout << "Static Local Variable: " << staticVar << endl;
}

int main() {
    exampleFunction(); // Выведет "Static Local Variable: 1"
    exampleFunction(); // Выведет "Static Local Variable: 2"
    cout << staticVar << endl; // Ошибка! Компилятор не видит
    return 0;
}
```

Статические (static) переменные доступны только в том месте кода где были объявлены (как и автоматические)

Но при этом **они НЕ уничтожаются после выхода за пределы фигурных скобок, но как и автоматические становятся недоступными**



Передача параметров


Обмен информацией между вызываемой и вызывающей функциями осуществляется с помощью **механизма передачи параметров**

! ВАЖНО

Передача параметров в функцию может осуществляться **по значению и по адресу**.

При передаче данных **по значению** функция работает с копиями параметров, и **доступа к исходным значениям переменной, у нее нет**

При передаче **по адресу** в функцию **передается не переменная**, а ее адрес, и, следовательно, функция имеет доступ к ячейкам памяти, в которых хранятся данные, благодаря чему **можно менять значение исходной переменной**.



Передача по значению

Создается копия оригинальной переменной.

Т.е. в функции Add и main у нас **две разные переменные n**:

- при изменении одной, вторая остается прежней
- они хранятся по разным адресам в памяти
- переменная в функции Add будет уничтожена после закрывающей фигурной скобки



отличие тут (см. следующий слайд)

```
void Add(int n){  
    n++;  
    cout << n << endl;  
}
```


```
int main(){  
    int n = 0;  
    cout << n << endl; // 0  
    Add(n);             // 1  
    cout << n << endl; // 0  
}
```

Передача по адресу

Благодаря оператору "&" перед именем переменной мы передаем *не значение, а адрес* ячейки в памяти, где хранится та самая переменная `int n = 0`; из функции `main`

Т.е. в функции `Add` и `main` у нас **одна и та же переменная `n`**:

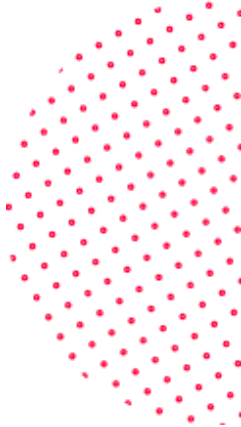
- при изменение одной, вторая тоже изменится
- один и тот же адрес памяти
- переменная в функции `Add` не будет уничтожена



отличие тут

```
void Add(int& n){  
    n++;  
    cout << n << endl;  
}
```

```
int main(){  
    int n = 0;  
    cout << n << endl; // 0  
    Add(n);             // 1  
    cout << n << endl; // 1  
}
```



Обратите внима

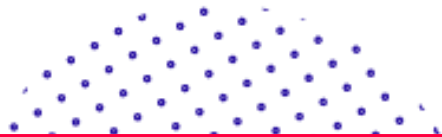
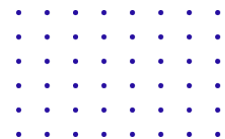
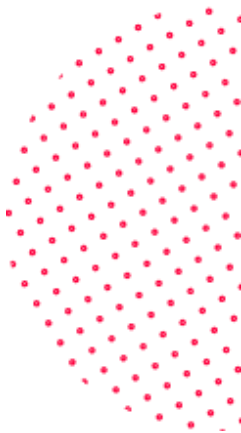


Почему это важно?

У вас есть тетрадь с домашней работой на сегодня. Учитель говорит вам сдать тетради на проверку. Вы сдадите тетрадь, в которой вы сделали сегодняшнюю домашнюю работу? или перепишите в точно такую же тетрадь новую домашку и все свои предыдущие домашки?

Если передадим объект или переменную (`int a`) в функцию, то создадим его точную копию. Это достаточно дорогая операция в плане производительности при работе с большими данными (например 3D модели, тексты книг, массивы на 10000+ элементов и т.п.)

Если используем ссылки (`int& a`) или указатели (`int* a`), то передаем ссылку на этот самый объект



Константы в аргументах

Когда мы передаем аргументы по ссылке мы можем изменить те данные, которые нельзя было менять. Для того, чтобы это не допустить, используются константы.

Это особенно важно когда над проектом работает несколько человек!

```
int func(const int &a){  
    a++; //ошибка! пытаемся изменить константу  
    return a;  
}
```

Что такое указатель?

Давайте рассмотрим на примере: В памяти компьютера все данные хранятся в своих проиндексированных ячейках.

```
int x = 10;
```

объект типа `int` занимает 4 байта

<u>0x60FE98</u>	0x60FE99	0x60FE9A	0x60FE9B
1 байт	1 байт	1 байт	1 байт
10			
переменная x			

Примерно так выглядит ячейка в памяти, которую занимает переменная после создания.

Каждый байт имеет индекс в 16-ричной системе счисления.

В современных компьютерах таких ячеек очень много, компьютер каждую секунду времени обращается к данным по этим индексам

Что такое указатель?

И указатель, и ссылка ссылаются на самый первый байт, а информацию о том, сколько байт занимает переменная внутри они получают когда мы пишем тип данных для них .

Например `int*` `pa` знает что хранит в себе адрес типа `int`, который как правило занимает 4 байта (зависит от архитектуры)

```
/*  
a - переменная которая хранит число 5  
pa - указатель на область в памяти где хранится переменная a  
&a - ссылка на область в памяти где хранится переменная a  
*/  
int a = 5;  
int* pa = &a;
```





Что такое указатель?

В языке C++ идет очень тонкая работа с памятью, если программист не будет использовать указатели и ссылки, то он рискует лишиться производительности и многих удобств. Например:

- **Выделение памяти под динамические массивы** - мы можем создавать массив любого размера, даже если изначально не знаем какой размер будет. *Пользователь вводит размер массива или размер зависит от вычислений в программе.*
- **Аргументы в функциях** - если используем ссылки и указатели, то передаем ссылку на этот самый объект.
- **Связь между элементами.** В C++ структуры где связь между объектами хранят сами объекты, например: stack, queue, list. (о таких структурах будем говорить в этом модуле)



Ну и зачем нам указатели?

В первую очередь, указатели - это оптимизация, так как C++ разработка предполагает скорость и рациональное использование памяти, например:

- В играх с большим количеством объектов на экране, таких как множество врагов или частицы, можно выделить память только для активных объектов, в то время как неактивные объекты могут быть удалены и их память может быть возвращена в пул свободной памяти.
- Когда игровой персонаж входит в новую область все игровые текстуры загружаются в память, а когда он покидает область - они выгружаются. Это помогает экономить память и ускорять процесс загрузки игры.

Без работы с памятью программа будет занимать практически всю свободную оперативную память для хранения объектов

Майнкрафт :)

Чанки обычно загружаются в память **только тогда, когда они необходимы** для отображения. Это управление памятью «до тех пор, пока оно не понадобится» обычно используется в играх с процедурно-генерируемым ландшафтом, чтобы компьютерам игроков не приходилось одновременно отслеживать и обновлять сотни растений и мобов.

Интересный факт: А вы замечали, что поставив переплавляться в печке стак руды, идете вы очень далеко от печки, к примеру копать в шахте, спустя 2 часа возвращаетесь, в надежде, что руда уже полностью переплавилась, а на самом-то деле переплавилось всего-то пару слитков.

Бывало? Это тоже объясняется очень просто. Когда вы ставите печку и отходите очень далеко, то чанки, можно сказать “пропадают”, но не пропадает то, что было на этих чанках, и печка из-за этого просто останавливается.

