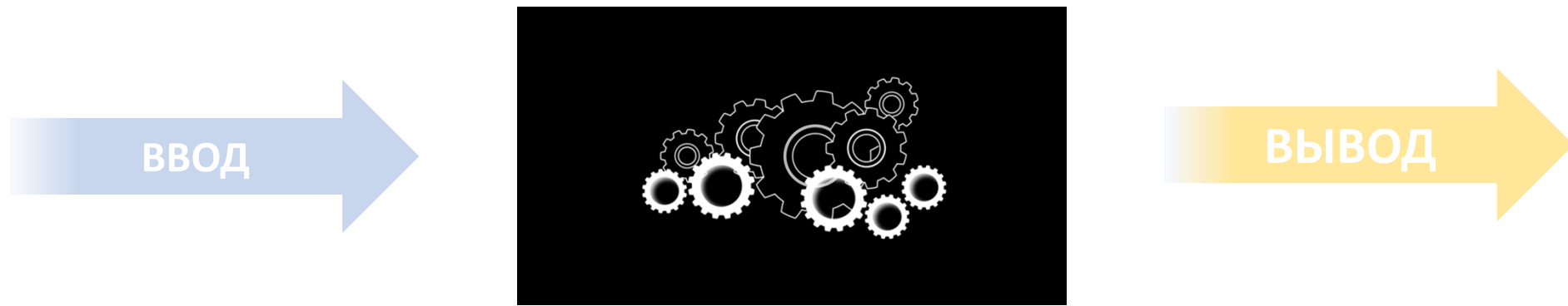


Алгоритмизация и программирование

Лекция 3 (C++)

Модель программы



* МОНОЛИТ С ВВОДОМ И ВЫВОДОМ ТОЛЬКО В КОНСОЛЬ

Простой код

```
#include <iostream>
```

```
int main() {  
    int a, b;  
    std::cin >> a >> b;  
    int c = a + b;  
    std::cout << c << std::endl;  
}
```

Код побольше

```
#include <iostream>
#include <vector>
#include <string>

int main() {
    std::cout << "Введите числа (введите 'stop' для завершения ввода):"<< std::endl;

    std::vector<int> numbers;
    while (true){
        std::string input;
        std::cout << "> ";
        std::cin >> input;

        if (input == "stop") break;

        auto num = std::stoi(input);
        numbers.push_back(num);
    }

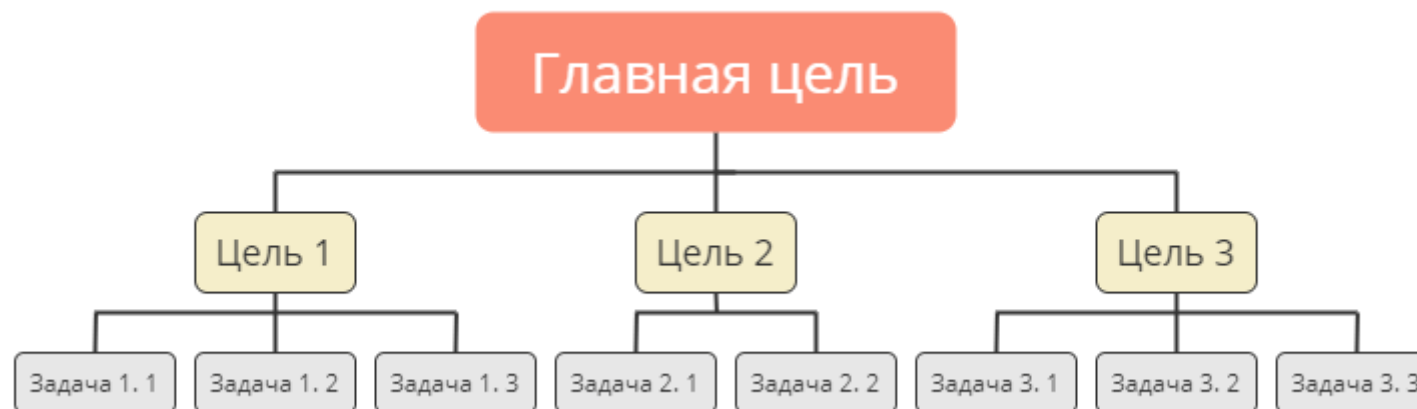
    int total = 0;
    for (const auto& num : numbers) {
        total += num;
    }

    double average = total / numbers.size();
    std::cout << "Среднее значение: " << average << std::endl;
}
```

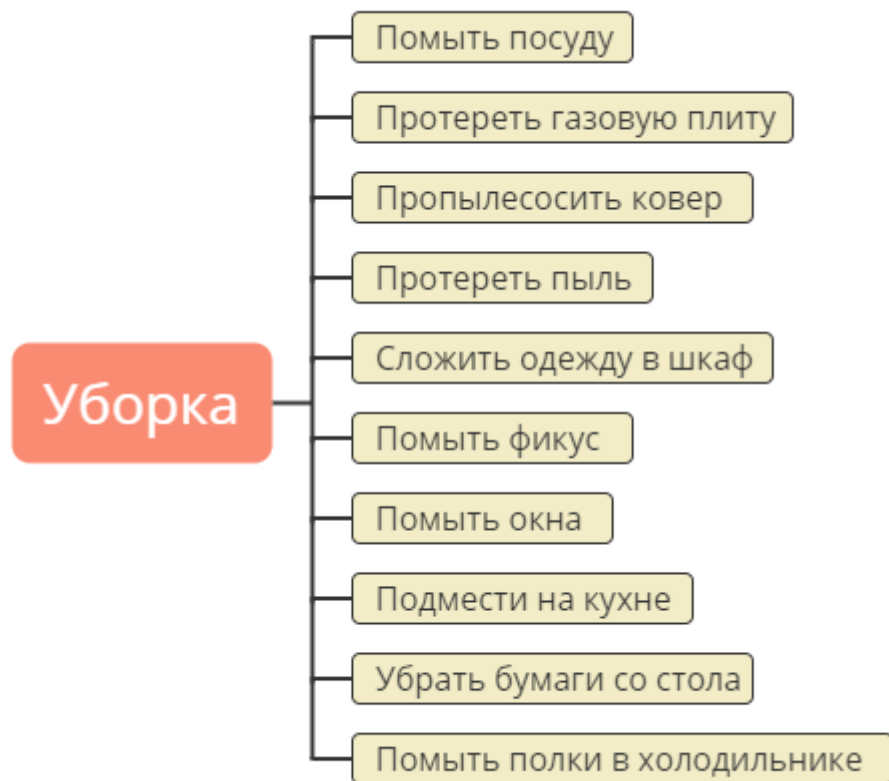
Снимаем ограничение на
возможность писать код только в
функции `main`

Декомпозиция

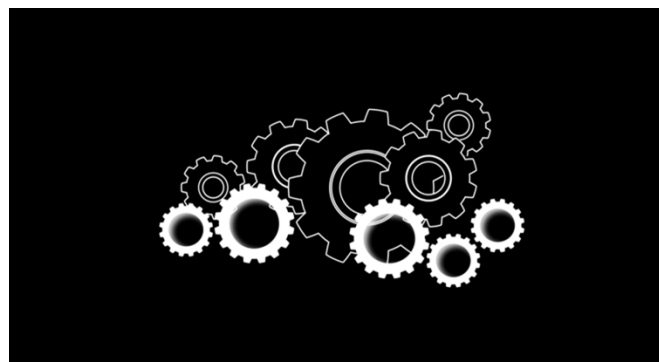
Декомпозицией называют разделение задачи на отдельные небольшие шаги — подзадачи



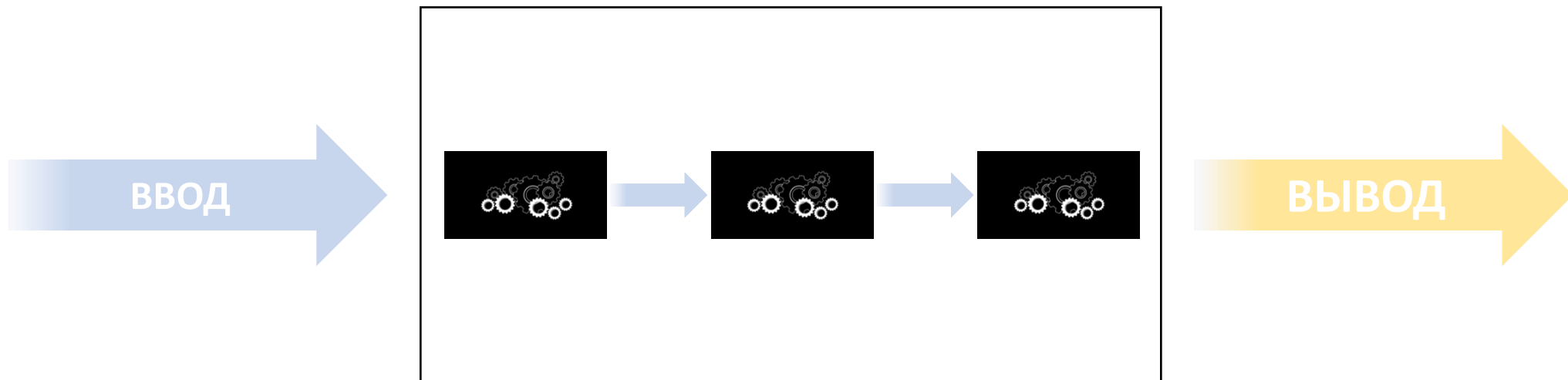
Декомпозиция



Подпрограмма



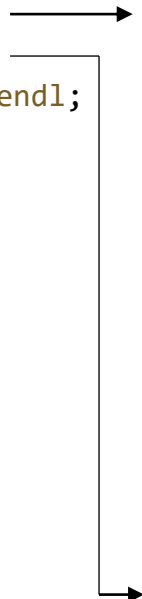
Программа



Код побольше + декомпозиция

```
#include <iostream>
#include <vector>
#include <string>

int main() {
    std::vector<int> numbers = readNumbersFromTerminal();
    double average = calculateAverage(numbers);
    std::cout << "Среднее значение: " << average << std::endl;
}
```



```
std::vector<int> readNumbersFromTerminal(){
    std::cout << "Введите числа (введите 'stop' для завершения ввода):"
               << std::endl;

    std::vector<int> numbers;
    while (true){
        std::string input;
        std::cout << "> ";
        std::cin >> input;

        if (input == "stop") break;

        auto num = std::stoi(input);
        numbers.push_back(num);
    }

    return numbers;
}
```

```
double calculateAverage(const std::vector<int>& numbers){
    int total = 0;
    for (const auto& num : numbers) {
        total += num;
    }

    return total / numbers.size();
}
```

Виды подпрограмм

Функция — это подпрограмма которая выполняет указанную последовательность действий и возвращает результат. Вызов функции может использоваться в других выражениях или в качестве правой части присваивания.

Процедура — это подпрограмма которая выполняет указанную последовательность действий и НЕ возвращает результат. Вызов процедуры нельзя использовать в выражениях или в качестве правой части присваивания.

Плюсы разделения на подпрограммы

- Уменьшение сложности:
 - Структура основной программы стала проще для понимания;
 - Каждая отдельная подпрограмма тоже обладает небольшой сложностью;
 - Разработка происходит небольшими, законченными этапами;
 - Проще покрыть текстами.
- Повторное использование;
- Совместная работа над решением

Магическое число семь плюс-минус два («кошелёк Миллера») — закономерность, обнаруженная американским учёным-психологом Джорджем Миллером, согласно которой кратковременная человеческая память, как правило, не может запомнить и повторить более 7 ± 2 элементов.

Сложный код

```
#include <iostream>

int main() {
    // Сложный код
    std::cout << "Многа букв ...";
    std::cout << std::endl;

    // Простой код
    std::cout << 2 + 2;
}
```

Используем подпрограммы

Нужно выполнить 2 действия:

- Объявить и определить функцию. В С++ процедуры не выделяют в отдельный тип подпрограмм, а тоже называют функциями. Чтобы в С++ создать функцию соответствующую определению процедуры, необходимо указать тип возвращаемого значения `void` (ничто);
- Вызвать функцию.

Простой код

```
#include <iostream>

// Объявляем и определяем функцию
void print_words(){
    std::cout << "Многа букв ...";
    std::cout << std::endl;
}

int main() {
    print_words(); // Вызываем функцию
    std::cout << 2 + 2;
}
```

Объявление (англ. declaration)

Объявление создаёт идентификатор и связывает его **с типом** некоторой программной сущности (переменной, функцией, типом, и т.д.).

Объявление используется, чтобы уведомить компилятор о существовании программной сущности **без её создания**, т.к. по не некоторым причинам она создаётся в другом месте.

Такие объявления обычно предшествуют определениям, поэтому их называют forward declarations.

Объявление (англ. declaration)

1. Объявление переменной типа int из другого файла :

```
extern int age;
```

2. Объявление функции:

```
int add(int a, int b);
```

3. Объявление класса:

```
class MyClass;
```

4. Объявление шаблона функции:

```
template <typename T> void print(T value);
```

5. Объявление перечисления:

```
enum Color;
```

6. Объявление пространства имён:

```
namespace MyNamespace;
```

7. Объявление шаблона класса:

```
template <typename T> class MyTemplateClass;
```

Определение (англ. definition)

Определение создаёт идентификатор, программную сущность и устанавливает связь между ними.

Во время определения **создаётся** переменная, функция, класс, и т.д., а также описывается их свойства и поведение.

Определение включает в себя объявление и создание сущности.

В сленге часто используют слово "объявление" даже в тех случаях когда код - это "определение".

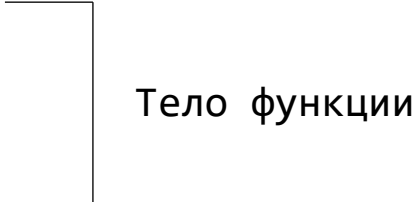
```
string personName; // идентификатор personName – переменная
class myClass{      // идентификатор myClass - тип
};
int main(){         // идентификатор main – функция
}
```

Определение функции

Прежде чем функцией можно будет пользоваться её нужно определить в коде. Функция не может быть определена внутри другой функции, только в глобально.

Общая структура определения:

```
тип_результата идентификатор_функции (список_параметров){  
    выполняемые_операторы  
}
```



Тело функции

```
int min(int a, int b){  
    if (a < b) return a;  
    return b;  
}
```

Прототип функции

Прототип функции – это часть определения с точкой с запятой вместо тела функции. Прототип – это объявление функции без её определения. Используется, чтобы добавить идентификатор в область видимости. Компилятор использует прототип, чтобы проверить правильность использования идентификатора в коде и сгенерировать правильный код, который в дальнейшем будет использовать линкер.

```
int min(int a, int b){  
    if (a < b) return a;  
    return b;  
}
```

```
int min(int a, int b);    // Прототип  
int min(int one, int two); // Прототип  
int min(int, int);        // Прототип
```

Прототип функции

Прототип функции может быть указан любое количество раз в коде. Прототип можно указывать не только на глобальном уровне, но и внутри функций, пространств имён, блоков и т.д.

```
#include <iostream>

int min(int a, int b); // Используем прототип, чтобы компилятор знал о функции min

int main() {
    std::cout << min(1, 2);
}

int min(int a, int b){
    if (a < b) return a;
    return b;
}
```

Сигнатура функции

Часть объявления функции которая используется компилятором для того, чтобы однозначно отличить одну функцию от другой. Функции с одинаковой сигнатурой с точки зрения компилятора не различимы и считаются переопределением.

Обычно в сигнатуру входит: название функции и типы параметров, но в зависимости от языка понятие сигнатуры может меняться.

```
int min(int a, int b){  
    if (a < b) return a;  
    return b;  
}
```

```
min(int, int)           // Тип возвращаемого значения не входит в сигнатуру
```

Тип функции

Функции в C++ имеют определённый тип, так же как и переменные. Тип функции можно получить из прототипа функции если отбросить все идентификаторы указанные в нём:

```
int min(int a, int b)    // заголовок функции
int(int, int)           // тип функции
```

Создать переменную или параметр функционального типа нельзя, но можно создать переменную или параметр типа указатель на функцию.

```
int min(int a, int b){
    if (a < b) return a;
    return b;
}
```

```
int (*other_min)(int, int) = min;
other_min(5, 10)           // Результат: 5
```

```
using two_int_to_int = int (*)(int, int);
two_int_to_int other_min = min;
```

https://youtu.be/PEIMSz3Er_8?si=AIPTD5d8ZlJgBq0Y

Параметры функции (формальные параметры)

Через параметры функция получает входные данные. С точки зрения функции, параметры это её локальные переменные. Параметры указываются в скобках после имени функции. Для каждого параметра указывается тип и имя (как для обычной переменной).

Параметры разделяются запятыми. У каждого параметра должен быть указан тип индивидуально.

```
double add(int a, double b);
```

```
int min(int a, int b);
```


Аргументы функции (фактически параметры)

Аргументы функции – это конкретные значения или переменные, переданные функции к точке вызова.

Аргументы передаются в функцию только по позиции (именованной передачи нет), т.е. первый аргумент будет советовать первому параметру функции, второй второму и т.д.

```
add(1, 2); // Вызов
```

```
min(a, b); // Вызов
```

Произвольное число аргументов

В C++ произвольное число аргументов используется не часто, т.к. по историческим причинам эта возможно была довольно не безопасна и разработчики привыкли обходится без неё.

В последних стандартах языка ситуация стала лучше, но всё равно использовать механизм произвольного числа аргументов по прежнему не очень удобно.

Произвольное число аргументов(до C++11)

```
#include <iostream>
#include <string>
#include <cstdarg> // Для va_list, va_start, va_end

void simple_printf(const char fmt[], ...)
{
    va_list args;
    va_start(args, fmt);

    for (int i=0; fmt[i] != '\0'; i++){
        if (fmt[i] == 'd') {
            int i = va_arg(args, int);
            std::cout << i << '\n';
        } else if (fmt[i] == 's') {
            char * s = va_arg(args, char*);
            std::cout << s << '\n';
        }
    }

    va_end(args);
}

int main()
{
    std::string str1( "Hello" ), str2( "world" );

    simple_printf("dddd", 10, 20, 30, 40 );
    simple_printf("ss", str1.c_str(), str2.c_str() );
}
```

Произвольное число аргументов (initializer_list)

```
#include <iostream>
#include <string>
#include <initializer_list>

template <class T>
void func( std::initializer_list<T> list ){
    for( auto elem : list )    {
        std::cout << elem << std::endl ;
    }
}

int main(){
    std::string str1( "Hello" ), str2( "world" );

    func( {10, 20, 30, 40} ) ;
    func( {str1, str2} ) ;
}
```

Произвольное число аргументов (variadic templates)

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

template <typename T>
void func(T t) {
    std::cout << t << std::endl ;
}

template<typename T, typename... Args>
void func(T t, Args... args) { // recursive variadic function
    std::cout << t << std::endl;

    func(args...) ;
}

int main() {
    std::string str1( "Hello" ), str2( "world" );

    func(1, 2.5, 'a', str1);
    func(10, 20, 30, 40);
    func(str1, str2);
}
```

Произвольное число аргументов (C++17)

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

template<typename... Args>
void print(Args... args){
    (std::cout << ... << args ) << "\n";
}

int main(){
    std::string str1( "Hello" ), str2( "world" );

    print(1, " ", 2.5, " ", 'a', " ", str1);
    print(10, " ", 20, " ", 30, " ", 40);
    print(str1, " ", str2);
}
```

Возвращаемые значения

В C++ функции могут возвращать результат или не возвращать ничего. Если функция не возвращает ничего, то в качестве типа указывается `void`:

```
void doSomething(int a, int b);
```

Если функция возвращает что-либо одно, то указывается тип возвращаемого значения:

```
int sum(int a, int b);
```

Функция не может возвращать несколько значений одновременно. Чтобы это сделать нужно упаковать возвращаемые значения в кортеж или структуру и вернуть их.

Возвращаемые значения

В C++ есть альтернативный синтаксис записи возвращаемого значения.

Объявление (declaration):

C++11

C++14

```
int sum(int a, int b);
```

```
auto sum(int a, int b) -> int;
```

```
auto sum(int a, int b);
```

Определение (definition):

```
int sum(int a, int b)
{
    int result = a + b;
    return result;
}
```

```
auto sum(int a, int b) -> int
{
    int result = a + b;
    return result;
}
```

```
auto sum(int a, int b)
{
    int result = a + b;
    return result;
}
```


return

```
void doSomething(int a, int b);
```

В теле функции можно, но не обязательно, использовать оператор return без указания значения. Используется для досрочного выхода из функции.

```
int sum(int a, int b);
```

В теле функции обязательно использовать оператор return со значением совпадающим с типом возвращаемого значения указанного в заголовке или преобразуемым в него. Для функции `main` сделано исключение, т.к. если не указать return функция просто сама вернёт 0.

Оператор return, в теле функции, можно использовать произвольное количество раз.

Вызов функции

Вызов функции возможен только в теле другой функции. Для этого нужно указать имя функции и в круглых скобках список аргументов, в том же количестве, в том же порядке и тех же типов, что и параметры в сигнатуре.

В качестве аргументов можно использовать как переменные, так и литералы (значения):

```
int sum(int a, int b);
```

```
int one = 1, two = 2, three;  
sum(one, two)      // Вызов  
three = sum(1, 2)  // Вызов
```

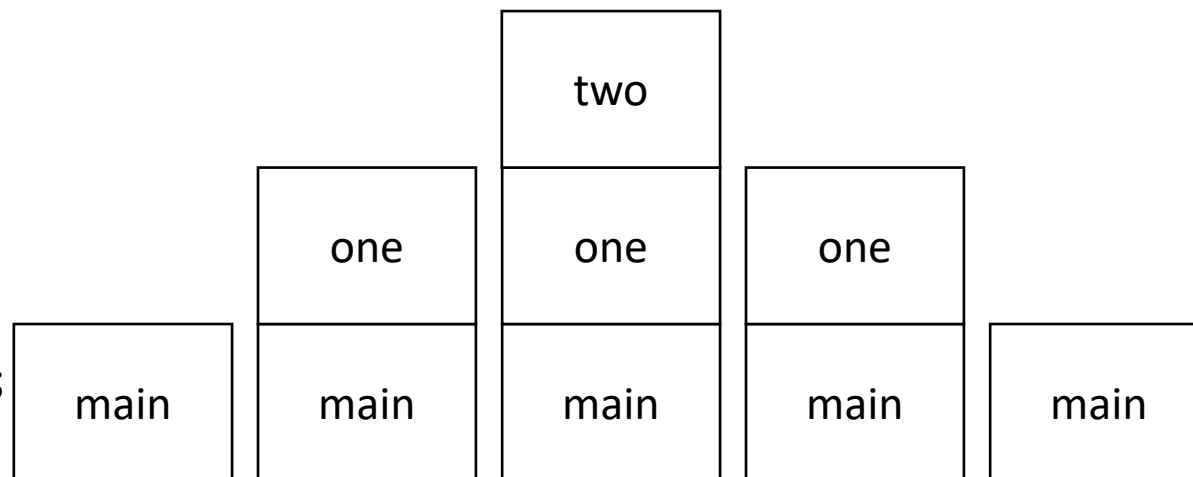
Стек вызовов

При вызове функции `one` исполнение функции `main` приостанавливается и ожидает завершения функции `one`. Функция `one` вызывая функцию `two` тоже приостанавливается и ожидает завершения функции `two`. Т.е. функции как бы складываются в стопку (стек) и в этой стопке исполняется только та функция которая лежит на вершине.

```
void two(){
    std::cout << "two" << std::endl;
}

void one(){
    std::cout << "one" << std::endl;
    two();
}

int main(){
    std::cout << "main" << std::endl;
    one();
}
```



Рекурсия

В качестве вызываемой функции можно указать имя текущей функции. Таким образом, текущая функция станет на паузу, и начнёт исполняться новая "копия" функции (на самом деле копируются только параметры и локальные переменные из-за чего может быть переполнен стек).

```
// Классическая рекурсия - факториал
int factorial(int n) {
    if (n <= 1) return 1;           // базовый случай
    return n * factorial(n - 1);    // рекурсивный случай
}
```

Хвостовая рекурсия (Tail Recursion)

Хвостовая рекурсия – частный случай рекурсии, при котором любой рекурсивный вызов является последней операцией перед возвратом из функции.

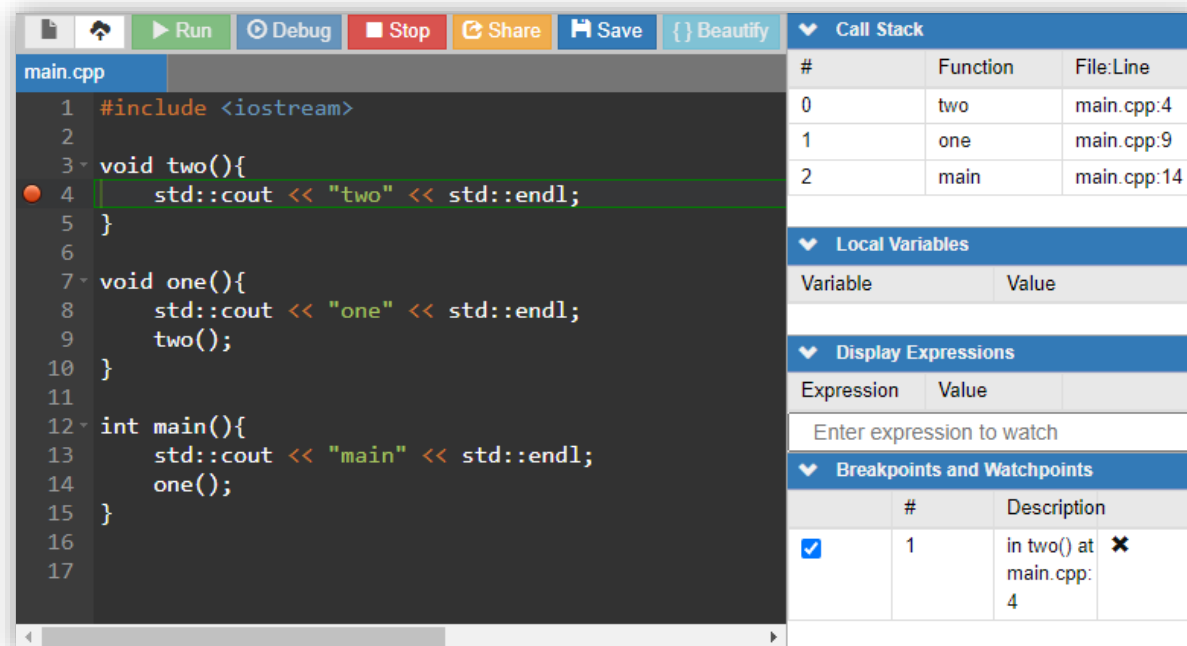
Такой вид рекурсии оптимизируется компилятором C++. Для него не выделяются новые кадры на стеке, а переиспользуются один.

```
// Хвостовая рекурсия для факториала
int factorial_tail(int n, int accumulator = 1) {
    if (n <= 1) return accumulator;
    return factorial_tail(n - 1, n * accumulator);
}
```

Стек-трейс (stack trace)

Стек-трейс (stack trace) – это информация о последовательности вызовов функций и методов. Стек-трейс полезен при поиске ошибок в коде. В настоящий момент стек-трейс без танцев с бубном получить не очень просто (в последующих стандартах обещают добавить).

Самым простой способ получить стек-трейс – использовать breakpoint и инструменты отладки. Большинство сред разработки позволяют это сделать максимально просто.



Вызов функции

В C++ функцию можно вызвать:

- Немедленно. Вызываем функцию обычным образом. При этом поток исполнения «перепрыгивает» из вызывающей функции в вызываемую и после её завершения возвращается обратно.

```
sum(1, 2);
```

- Отложенный вызов. В C++ нет встроенной возможности вызвать функцию отложенным образом. Вместо этого можно воспользоваться идиомой RAII, т.к. рантайм C++ гарантирует вызов деструктора объекта перед выходом из области видимости. Поэтому можно завернуть вызов функции в обёртку:

```
#include <functional>    // для std::function

struct Defer{
    std::function<void()> f;

    Defer(std::function<void()> f): f(f){
    }

    ~Defer(){
        f();    // Вызываем функцию в деструкторе
    }
};
```

Вызов функции

В параллельном потоке. В C++ нет встроенной возможности вызвать функцию в параллельном потоке. Вместо этого можно воспользоваться средствами стандартной библиотеки, например `std::thread`, `std::async` или корутинами (начиная с C++20)

Вызов функции

- Немедленный вызов:

```
void other(){  
    std::cout << "other" << std::endl;  
}  
  
int main(){  
    std::cout << "main start" << std::endl;  
    other();  
    std::cout << "main stop" << std::endl;  
}
```

```
main start  
other  
main stop
```

- Отложенный вызов:

```
void other(){  
    std::cout << "other" << std::endl;  
}  
  
int main(){  
    std::cout << "main start" << std::endl;  
    Defer defer( [](){ other(); } );  
    std::cout << "main stop" << std::endl;  
}
```

```
main start  
main stop  
other
```

Вызов функции

- В параллельном потоке:

```
#include <iostream>
#include <thread>

void other(){
    std::cout << "Concurrent execution in C++ " << std::endl;
}

int main(){
    std::cout << "main start" << std::endl;
    std::thread t1(other);
    std::cout << "main stop" << std::endl;
    t1.join();    // Ждём завершения потока
}
```

```
main start
main stop
Concurrent execution in C++
```

- асинхронно:

```
#include <iostream>
#include <future>

void other(){
    std::cout << "Concurrent execution in C++ " << std::endl;
}

int main(){
    std::cout << "main start" << std::endl;
    auto res = std::async(std::launch::async, other);
    std::cout << "main stop" << std::endl;
    res.get();    // Ждём завершения потока
}
```

```
main start
main stop
Concurrent execution in C++
```

Передача данных в функцию

Передача данных **по значению**. Создаёт локальную копию передаваемых данных.

```
void swap(int a, int b){  
    int t = a;  
    a = b;  
    b = t;  
}
```

Передача данных **по ссылке**. Создаёт дополнительное имя для переменной переданной в качестве аргумента.

```
void swap(int& a, int& b){  
    int t = a;  
    a = b;  
    b = t;  
}
```

Передача данных **по указателю**. Создаёт копию, но не данных, а адреса по которому они находятся.

```
void swap(int* a, int* b){  
    int t = *a;  
    *a = *b;  
    *b = *t;  
}
```

Передача данных в функцию

В C++ данные в функцию можно передать:

- по значению. В этом случае функция получает **копии** передаваемых **данных**. Следовательно вызываемая функция не имеет доступа к данным из вызывающей функции.
Стоит отметить, что происходит не глубокое копирование, т.е. если передать структуру, которая хранит указатель или ссылку на данные, то сама структура будет скопирована, а данные на которые она указывает нет.
- по указателю. В этом случае функция получает **копии адресов** передаваемых данных. Зная адрес размещения оригинальных данных вызываемая функция может прочитать и изменить данные вызывающей функции.
Чтобы передать данные по указателю в качестве параметра функции используются указатели, а в качестве аргумента – адреса данных

```
void swap(int a, int b){  
    int t = a;  
    a = b;  
    b = t;  
}  
  
int main(){  
    int first = 10, second = 20;  
    std::cout << first << " " << second << std::endl;  
    swap(first, second);  
    std::cout << first << " " << second << std::endl;  
}
```

10 20
10 20

```
void swap(int* a, int* b){  
    int t = *a;  
    *a = *b;  
    *b = t;  
}  
  
int main(){  
    int first = 10, second = 20;  
    std::cout << first << " " << second << std::endl;  
    swap(&first, &second);  
    std::cout << first << " " << second << std::endl;  
}
```

10 20
20 10

Передача данных в функцию

В C++ данные в функцию можно передать:

- по ссылке. Ссылки на исходные данные. Следовательно вызываемая функция имеет доступа к данным из вызывающей функции и может их изменять. Чтобы передать данные по ссылке к типу параметра добавляется амперсанд, весь остальной код остаётся таким же как и при передаче по значению.

```
void swap(int& a, int& b){  
    int t = a;  
    a = b;  
    b = t;  
}  
  
int main(){  
    int first = 10, second = 20;  
    std::cout << first << " " << second << std::endl;  
    swap(first, second);  
    std::cout << first << " " << second << std::endl;  
}
```

```
10 20  
20 10
```

Передача данных в функцию - const

Квалификатор **const** запрещает изменять параметры.

```
void swap(const int a, const int b){  
    int t = a;  
    a = b;  
    b = t;  
}
```

Переменные в памяти

Состояние переменных в памяти сразу после вызова функции `swap`

```
void swap(int* a, int* b){
    int t = *a;
    *a = *b;
    *b = t;
}

int main(){
    int first = 10, second = 20;
    std::cout << first << " " << second << std::endl;
    swap(&first, &second);
    std::cout << first << " " << second << std::endl;
}
```

Имя	Тип	Значение	Адрес
...			
b	int*	0xc000074018	0xc00007e028
a	int*	0xc000074010	0xc00007e020
...			
t	int	мусор	0xc000074038
...			
second	int	20	0xc000074018
first	int	10	0xc000074010
...			

Указатели

Переменная-указатель предназначена для хранения адреса в памяти. Чтобы создать переменную-указатель нужно после типа данных поставить символ звёздочка:

```
тип_данных* pointer;
```

```
int* pointer1;           // Адрес int-a
```

```
double* pointer2;        // Адрес double-a
```

```
std::vector<int>* pointer3; // Адрес вектора из 3х int-ов
```

Переменная-указатель, как и любая другая, по умолчанию имеет мусорное значение, т.е. какой-то случайный адрес. Существует специальное значение `nullptr` которое говорит о том, что этот указатель не хранит адрес (никуда не указывает). Из языка C доступны ещё 2 способа "занулить" указатель, значение `NULL` и просто 0;

```
int* pointer1 = nullptr; // Нулевой указатель
```


Указатели – получение адреса

Чтобы сохранить в переменную адрес его сначала нужно получить. Для этого используется оператор взятия адреса – амперсанд (&). Его нужно написать перед именем объекта адрес которого нам нужен (например переменной) или получить как результат работы функции.

```
int value = 10;  
int* pointer = nullptr;  
pointer = &value;           // Получаем адрес переменной value
```

Положить в переменную что-то кроме адреса нельзя

```
int* pointer2 = value;      // Ошибка! pointer2 ждёт адрес, а не значение
```

Не у всех программных объектов есть адреса

```
int* pointer3 = &5;         // Ошибка! У 5 нет адреса в памяти
```

Совместимы только адреса объектов одного типа

```
double valuef = 10.0;  
int* pointer4 = &valuef;    // Ошибка! Не тот тип адреса
```

Указатели – доступ к значению через адрес

Зная адрес можно получить доступ к данным расположенным по этому адресу. Через адрес можно как читать данные, так и изменять их. Для получения доступа используется оператор разыменования (звёздочка) который нужно написать перед **переменной-указателем**:

```
int value = 10;
int* pointer = &value;    // Получаем адрес переменной value
std::cout << *pointer;    // Читаем данные из value через её адрес

*pointer = 20;             // Меняем данные в value через её адрес
std::cout << value;        // Проверяем
```

Не имеет значение каким образом получен адрес, если к нему применить оператор разыменования, то мы получим то, на что указывает адрес. Если к результату применить оператор взятия адреса, то мы снова получим адрес и т.д.

```
std::cout << *&*value;    // Можно так. Это тоже самое, что и просто value
```

Здесь операторы применяются справа налево (<--) начиная от имени переменной.

Ссылки

Переменная-ссылка с точки зрения языка является **указателем**, к которому автоматически применяется оператор разыменования. Поэтому поменять адрес записанный в переменную-ссылку не получится, т.к. идентификатор будет автоматически разыменован и даст доступ к значению по хранимому адресу. Поэтому переменную-ссылку нужно инициализировать при создании.

С точки зрения программиста можно воспринимать переменную-ссылку как второе имя для переменной. Чтобы создать переменную-ссылку нужно после типа данных поставить символ амперсанд (&):

```
тип_данных& pointer = адресуемый_объект;
```

```
int& ref1;           // Так нельзя
```

```
int value = 10;
int& ref = value;    // Создаём ссылку на value
std::cout << ref;    // Читаем данные из value через ссылку
```

```
ref = 20;            // Меняем данные в value через ссылку
std::cout << value;  // Проверяем
```

Ссылки — инициализация

Под "капотом" ссылки используют адреса для доступа к памяти, поэтому ссылку можно взять только на объект у которого есть адрес.

```
int value = 10;  
int& ref = value;      // У переменной value есть адрес
```

Нельзя взять ссылку на ссылку, здесь мы получим адрес переменной value

```
int& ref2 = ref;       // ref2 – ещё одна ссылка на value
```

Не у всех программных объектов есть адреса

```
int& ref3 = 5;         // Ошибка! У 5 нет адреса в памяти
```

Брать ссылки можно только на объекты одного типа

```
double valuef = 10.0;  
int& ref4 = valuef;    // Ошибка! Не тот тип объекта
```

```
std::vector<int> values = {1, 2, 3};  
int& ref5 = values[0]; // Не безопасно, но можно  
std::cout << ref5;
```

Возврат результата

Возврат данных **по значению**. Создаёт копию возвращаемых данных и отдаёт наружу.

```
int sum(int a, int b){  
    int result = a + b;  
    return result;  
}
```

Возврат данных **по ссылке**. Даёт доступ вызывающему коду к локальной переменной функции.

```
int& sum(int a, int b){  
    int result = a + b;  
    return result;  
}
```

Возврат данных **по указателю**. Передаёт наружу информацию об адресе, по которому лежат данные.

```
int* sum(int a, int b){  
    int result = a + b;  
    return &result;  
}
```

Возврат результата

В C++ данные из функции можно возвращать:

- по значению. В этом случае вызывающая функция получит **копию данных** которые были в вызываемой функции. Существуют оптимизации RVO и NRVO, позволяющие пропустить операцию копирования и создать объект сразу в точке возврата.
Для объектов существует оптимизация copy elision, позволяющая не вызывать конструктор копирования в ряде случаев.
- по указателю. В этом случае вызывающая функция получает **копии адресов** объектов созданных или переданных в вызываемую функцию. В C++ возврат результата по указателю НЕ безопасен в случае возврата адреса локального объекта вызывающая функция получит непригодный для использования адрес, т.к. при завершении функция удалит все свои локальные переменные.
Можно возвращать адреса объектов созданных в куче или за пределами функции.

```
std::vector<int> createBord(int count) {  
    std::vector<int> bord;  
    if (count <= 0) bord = std::vector<int>();  
    else bord = std::vector<int>(count, 0);  
    return bord;  
}  
  
int main(){  
    std::vector<int> bord = createBord(5);  
}
```

```
std::vector<int>* createBord(int count) {  
    std::vector<int> bord;  
    if (count <= 0) bord = std::vector<int>();  
    else bord = std::vector<int>(count, 0);  
    return &bord;  
}  
  
int main(){  
    std::vector<int>* bord = createBord(5);  
}
```

Возврат результата

В C++ данные из функции можно возвращать:

- по ссылке. В этом случае вызывающая функция получает **ссылку** на возвращаемый объект. В C++ возврат результата по ссылке НЕ безопасен в случае возврата ссылки на локальный объект вызывающая функция получит непригодный для использования объект, т.к. при завершении функция удалит все свои локальные переменные.
Можно возвращать ссылку на объекты созданные в куче или за пределами функции.

```
std::vector<int>& createBord(int count) {  
    std::vector<int> bord;  
    if (count <= 0) bord = std::vector<int>();  
    else bord = std::vector<int>(count, 0);  
    return bord;  
}  
  
int main(){  
    std::vector<int> bord = createBord(5);  
}
```

Возврат нескольких значений

В C++ нет встроенного способа вернуть из функции несколько значений сразу. Чтобы это сделать нужно упаковать возвращаемые значения в кортеж или структуру и вернуть их:

- упаковка в структуру.
- упаковка в кортеж.

```
struct Human{
    int age;
    std::string name;
    double salary;
};

Human getHuman(){
    return {20, "John", 100000};
}

int main(){
    Human res = getHuman();
    std::cout << res.age << res.name << res.salary;
}
```

```
struct Human{
    int age;
    std::string name;
    double salary;
};

Human getHuman(){
    return {20, "John", 100000};
}

int main(){
    // C++17 structured binding:
    auto [age, name, salary] = getHuman();
    std::cout << age << name << salary;
}
```


Возврат нескольких значений - кортеж

В качестве результата функции указываем кортеж и перечисляем в угловых скобках нужные типы.

В вызывающей функции получаем кортеж или сразу распаковываем по переменным

```
// Как вернуть кортеж
std::tuple<int, int> foo_tuple()
{
    return {1, -1}; // Error until N4387
    return std::tuple<int, int>{1, -1}; // Always works
    return std::make_tuple(1, -1);      // Always works
}
```

```
#include <tuple>

std::tuple<int, std::string, double> getHuman(){
    return {20, "John", 100000};
}

int main(){
    auto res = getHuman();
    std::cout << std::get<0>(res)
               << std::get<1>(res)
               << std::get<2>(res);
}
```

```
#include <tuple>

std::tuple<int, std::string, double> getHuman(){
    return {20, "John", 100000};
}

int main(){
    int age;
    std::string name;
    double salary;

    std::tie(age, name, salary) = getHuman();
    std::cout << age << name << salary;
}
```

```
#include <tuple>

std::tuple<int, std::string, double> getHuman(){
    return {20, "John", 100000};
}

int main(){
    // C++17 structured binding:
    auto [age, name, salary] = getHuman();
    std::cout << age << name << salary;
}
```

Функция main

Параметры функции main

В стандарте определены следующие прототипы для функции main:

- `int main();`
- `int main(int argc);`
- `int main(int argc, char* argv[]);`

Параметр `argc` позволяет узнать сколько аргументов было передано программе. Параметр `argv` позволяет получить аргументы переданные программе в виде массива строк за завершающим нулевым символом (C-строки).

```
#include <iostream>
```

```
int main(int argc, char* argv[]){  
    for (int i=0; i<argc; i++){  
        std::cout << argv[i] << std::endl;  
    }  
}
```

Аргументы функции main

После компиляции программы мы получаем исполняемый файл. Если его запустить в терминале и через пробел указать значения, то эти значения будут переданы в программу как её аргументы.

```
prog.exe hello world 42
```

В результате программа получит массив строк из 4-х значений:

```
prog.exe hello world 42
```

По соглашению, первым передаётся имя программы (точнее команда которой мы её запустили. Здесь только имя, но может быть ещё и путь), а затем оставшиеся аргументы разделяя их по пробелам. Если нужно передать аргумент содержащий пробел, то необходимо заключить его в кавычки:

```
prog.exe "hello world" 42
```

Значение возвращаемое функцией main

Значение возвращаемое функцией main интерпретируется операционной системой и может быть использовано в консольных скриптах.

По стандарту нулевое значение возвращённое функцией main интерпретируется как успешный выход без ошибок, а любое не нулевое как признак возникновения ошибки в процессе выполнения программы.

Функция main должна возвращать значение в пределах от 0 до 255 (другие значения обрежутся до байта), но для лучше совместимости стоит ограничиться 127. Коды ошибок каждый разработчик определяет сам и указывает в документации. Стандартизированы только 0 EXIT_SUCCESS и 1 EXIT_FAILURE

В терминале можно проверить код завершения последней запущенной программы:

Linux/Mac

`$?`

Windows (cmd)

`%ERRORLEVEL%`

Windows (PowerShell)

`$LASTEXITCODE`

Параметры со значениями по
умолчанию

Значения по умолчанию

Параметрам можно указать значения по умолчанию. В лучшем случае, если аргумент для параметра не будет передан, он получит это значение.

```
int min(int a, int b=0){  
    if (a < b) return a;  
    return b;  
}
```

```
min(1, 2); // 1  
min(1);    // 0
```

Значения по умолчанию должны устанавливаться справа на лево. Не допускается "дырки" при указании значений по умолчанию

```
int min(int a=0, int b, int c=0); // ошибка
```

Значения по умолчанию

Значения по умолчанию создаются каждый раз при вызове функции (отличие от Python)

```
int sum(std::vector<int> array = std::vector<int>{1,2,3}){  
    int acc = 0;  
    for (int v: array) acc += v;  
  
    return acc;  
}
```

```
std::cout << sum({3,3,3});  
std::cout << sum();
```


Область видимости (scope)

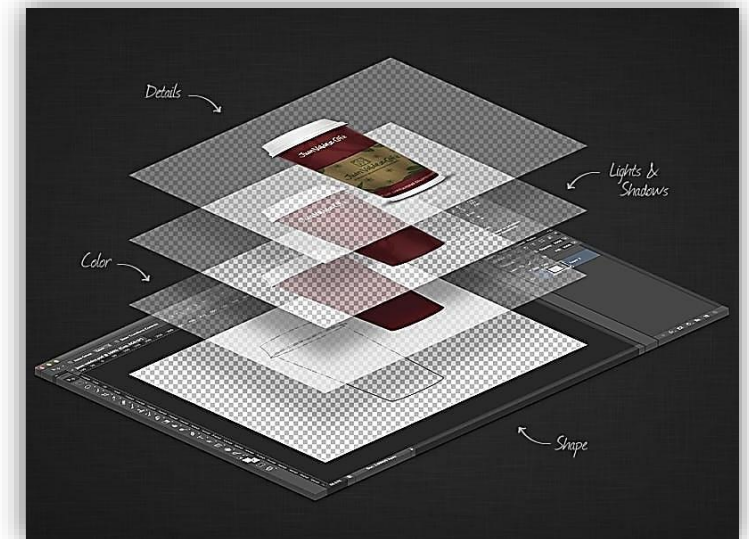
Область видимости (scope)

Область видимости — **часть программы** (кода), в пределах которой идентификатор, объявленный как имя некоторой программной сущности (обычно — переменной, типа данных или функции), остаётся связанным с этой сущностью, то есть позволяет посредством себя обратиться к ней.

Говорят, что идентификатор объекта «виден» в определённом месте программы, если в данном месте по нему можно обратиться к данному объекту. За пределами области видимости тот же самый идентификатор может быть связан с другой переменной или функцией, либо быть свободным (не связанным ни с какой из них). Область видимости может, но не обязана совпадать с областью существования объекта, с которым связано имя.

В пределах одной области видимости идентификатор может быть связан только с одной программной сущностью. Добавление идентификатора в область видимости происходит через объявление/определение. Таким образом, в одной области видимости может быть только одно определение.

Области видимости входят друг в друга и **составляют иерархию**, от локальной области видимости, ограниченную функцией (или даже её частью), до глобальной, идентификаторы которой доступны во всей программе. Также в зависимости от правил конкретного языка программирования области видимости могут быть реализованы двумя способами: лексически (статически) или динамически



Shadowing

Говорят, что идентификатор скрывает "затеняет" другой идентификатор, если он переопределяет его в более конкретной области видимости, т.е. ближе по иерархии к точке, в которой указано обращение к идентификатору.

Т.к. в процессе связывания выбирается идентификатор из ближайшей области видимости, одноимённая переменная становится, как бы, на время невидимой (скрытой в тени).

C++

```
int main(){  
    int x = 0;  
    for (int x=0; x<3; x++)  
        std::cout << x << std::endl;  
    std::cout << x << std::endl;  
}
```

0
1
2
0

Python

```
x = 0  
  
def outer():  
    x = 1  
    print("outer:", x)  
  
outer()  
print("global:", x)
```

1
0

Связывание идентификатора (binding)

Связывание идентификатора — **процесс** определения программного объекта, доступ к которому даёт идентификатор в конкретном месте программы и в конкретный момент её выполнения.

- **лексическое (статическое) связывание** (англ. *lexical (static) binding*), **лексическая область видимости**, или **лексический контекст** (англ. *lexical scope*): поиск объявлений подходящих для связывания происходит только с учётом размещения идентификатора в коде.
- **динамическое связывание** (англ. *dynamic binding*), или **динамическая область видимости**, или **динамический контекст** (англ. *dynamic scope*): поиск объявлений подходящих для связывания происходит с учётом истории выполнения кода.

C++

```
void print_x(){  
    std::cout << x;  
}  
  
int x = 10;  
int main(){  
    print_x();  
}
```

Ошибка
компиляции

Python

```
def print_x():  
    print(x)  
  
x = 10  
def main():  
    print_x()  
  
main()
```

10

Начало области видимости

В зависимости от языка программирования точка **начала** действия объявления внутри области видимости может определяться по разному:

- Начало действия от точки объявления и ниже по коду. В этом случае в процессе связывания просматриваются объявления текущей области видимости от точки обращения к идентификатору и выше по коду. Если объявлений не обнаружено, то поиск продолжается в следующей по иерархии области видимости.
- Идентификатор действует сразу во всей области видимости независимо от того, в каком её месте находится объявление. В этом случае в процессе связывания просматриваются вся текущей области видимости и если объявление найдено, то происходит связывание, а если не найдено, то поиск продолжается в следующей по иерархии области видимости

C++

```
auto scope = "global\n";  
void f() {  
    std::cout << scope;  
    auto scope = "local\n";  
    std::cout << scope;  
}
```

global
local

JavaScript

```
var scope = "global";  
function f() {  
    alert(scope);  
    var scope = "local";  
    alert(scope);  
}
```

undefined
local

Блочная область видимости

Ещё один нюанс в семантике лексической области видимости — наличие или отсутствие так называемой «блочной видимости», то есть возможности объявить локальную переменную не только внутри функции, процедуры или модуля, но и внутри отдельного блока команд

- В C++ блок операторов образует локальную область видимости, и объявляемая внутри цикла переменная `x` — это новая переменная, областью видимости которой является только тело цикла.
- В JavaScript нет блочной области видимости (в версиях, предшествующих ES6), а повторное объявление локальной переменной работает просто как обычное присваивание.

C++

```
void f() {  
    int x = 3;  
    std::cout << x << std::endl;  
    for (int i=10; i<30; i += 10){  
        int x = i;  
        std::cout << x << std::endl;  
    }  
    std::cout << x << std::endl;  
}
```

3
10
20
3

JavaScript

```
function f() {  
    var x = 3;  
    alert(x);  
    for (var i = 10; i < 30; i += 10) {  
        var x = i;  
        alert(x);  
    }  
    alert(x);  
}
```

3
10
20
20

Блоки в C++

The file block. Каждый файл содержит файловый блок, содержащий весь исходный код в этом файле. Файловый блок создаёт глобальную область видимости.

В глобальная область видимости может быть выделен отдельный фрагмент – namespace.

The function block. Каждая функция создаёт собственный блок.

Struct, Class and Enum block. Структура, класс и перечисление создаёт отдельный блок.

The control structure block. Считается, что каждый оператор "if", "for", "while" и "switch" находится в своем собственном неявном блоке. Блок начинается перед ключевым словом и заканчивается после последней закрывающей фигурной скобки.

Clause block. Каждое предложение "case" в операторе "switch" действует как неявный блок.

Блоки могут быть вложенными и создают отдельные области видимости.

Области видимости в C++

- Идентификаторы, обозначающего константу, тип, переменную или функцию (но не метод), объявленную на верхнем уровне (вне какой-либо функции) видны от точки объявления и до конца файлового блока.
- Областью видимости идентификатора параметра функции начинается от точки объявления и далее до конца тела функции.
- Область видимости идентификатора константы или переменной начинается в конце объявления и действует до конца блока.
- Область видимости идентификатора типа начинается с идентификатора в объявлении типа и действует до конца блока.

Области видимости в C++ (метки)

Метки используются в операторах "goto".

В отличие от других идентификаторов, метки не ограничены областью блока и не конфликтуют с идентификаторами, которые не являются метками. Область видимости метки - это тело функции, в которой она объявлена исключая тело любой вложенной функции.

Запрещено "перепрыгивать" с помощью метки через объявления **новых** переменных. Т.е. перепрыгивать назад можно.

```
int main(){  
    int x = 0;  
    x:  
        std::cout << x++ << std::endl;  
        if (x > 10) return 0;  
        goto x;  
}
```

0
1
2
3
4
5
6
7
8
9
10

Коллега-программист благодарит вас за прекрасно написанные код:



Анонимные функции

Анонимные функции

Анонимные функции - это функции, которым не назначен идентификатор. Они отличаются от обычных функций также тем, что они могут определяться внутри других функций и также могут иметь доступ к контексту выполнения.

```
int main() {  
    auto f = [](int x, int y)->int{ return x + y; };  
    std::cout << f(3, 4) << std::endl;    // 7  
    std::cout << f(6, 7) << std::endl;    // 13  
}
```

Синтаксис лямбда-функций

C++11	<div>[CAPTURES] {BODY}</div> <div>[CAPTURES] -> RETURN-TYPE {BODY}</div>					
C++23	<div>[CAPTURES] SPECIFIERS {BODY}</div> <div>[CAPTURES] SPECIFIERS -> RETURN-TYPE {BODY}</div>					
C++11	<div>[CAPTURES] (PARAMETERS) {BODY}</div> <div>[CAPTURES] (PARAMETERS) -> RETURN-TYPE {BODY}</div> <div>[CAPTURES] (PARAMETERS) SPECIFIERS {BODY}</div> <div>[CAPTURES] (PARAMETERS) SPECIFIERS -> RETURN-TYPE {BODY}</div>					
C++20	<div>[CAPTURES] <TYPE-PARAMETERS> (PARAMETERS) {BODY}</div> <div>[CAPTURES] <TYPE-PARAMETERS> (PARAMETERS) -> RETURN-TYPE {BODY}</div> <div>[CAPTURES] <TYPE-PARAMETERS> (PARAMETERS) SPECIFIERS {BODY}</div> <div>[CAPTURES] <TYPE-PARAMETERS> (PARAMETERS) SPECIFIERS -> RETURN-TYPE {BODY}</div>					
SPECIFIERS	C++11			C++17	C++20	
	mutable	noexcept	[[attribute]]	constexpr	constexpr	requires...

Синтаксис лямбда-функций

```
auto f1 = [] { return 47; };          auto x = f1();          // ⇔ int x = 47; C++11
auto f2 = [] (int x, int y) { return 0.5*(x+y); }; auto y = f2(2,3); // ⇔ double y = 2.5;
auto f3 = [] (int x) -> float { return x*x; }; auto z = f3(2); // ⇔ float z = 4.0f;
```

Capturing of variables from the surrounding scope

by reference

```
int x = 1;
auto f = [&] (int y) { x += 2; return x*y; };
cout << f(2); // 6
cout << x; // 3
```

by value

```
int x = 1;
auto f = [=] (int y) { return x*y; };
cout << f(2); // 2
cout << x; // 1
```

[=]	capture all by value
[&]	capture all by reference
[=, &x]	x by reference, all others by value
[&, x]	x by value, all others by reference
[x, &y]	only x by value and y by reference
[&x, y]	only x by reference and y by value
[&x, y, &z]	only x by ref, y by value and z by ref

init captures define lambda-local variables

C++14

```
auto f = [x=2] (int y) { return x*y; };
std::vector<char> v (1000, 'a');
auto g = [w=std::move(v)] () { /* use w */ };
```

Immediately Invoked Function Expressions (IIFE)

```
int z = [] (int x) { return x*x; } (2); // create lambda and call it ⇔ int z = 4;
```

Mutable Lambdas

```
int y = 1;
auto f = [=] () {
    y += 2;
    return y;
};
```

COMPILER ERROR:
local variable
'y' is const!

```
int y = 1;
auto f = [=] () mutable {
    y += 2;
    return y;
};
cout << f(); // 3
cout << f(); // 5
cout << y; // 1
```

```
auto f = [i=0] () mutable {
    ++i; return i;
};
cout << f(); // 1
cout << f(); // 2
```

Generic Lambdas

C++14

```
auto print = [] (auto const& x) { std::cout << x; };
print(5); ✓
print(std::string{"it works!"}); ✓
```

```
[] (auto value, auto const& cref, auto& ref) { ... }
[] (auto... args) { return g(args...); }
```

Perfect Forwarding preserving constness, l/r-valueness

```
[] (auto&& x) { g( std::forward<decltype(x)>(x) ); }
[] (auto&&... args) {
    g( std::forward<decltype(args)>(args)... ); }
```

Constrained auto Parameters

C++20

```
#include <concepts>
[] (std::copyable auto x) { ... }
```

Explicit Template Parameters

C++20

```
[] <typename T> (T x, T y) { ... }
```

Анонимная функция как аргумент функции

Очень удобно использовать анонимные функции в качестве аргументов других функций. Лямбда-функцию без захвата можно передать через указатель:

```
#include <iostream>

using func = int (*)(int, int);

void action(int n1, int n2, func operation){
    auto result = operation(n1, n2);
    std::cout << result << std::endl;
}

int main() {
    action(10, 25, [](int x, int y){ return x + y; });    // 35
    action(5, 6, [](int x, int y){ return x * y; });     // 30
}
```

Анонимная функция как аргумент функции

Лямбда-функцию с захватом нельзя передать через указатель, но можно, через обёртку `std::function`

```
#include <iostream>
#include <functional>

void action(int n, std::function<int(int)> operation){
    auto result = operation(n);
    std::cout << result << std::endl;
}

int main() {
    int y = 10;
    action(10, [y](int x){ return x + y; });    // 20
    action(5, [y](int x){ return x + y; });    // 15
}
```

Анонимная функция как результат функции

Анонимная функция может быть результатом другой функции:

```
#include <iostream>
#include <functional>

std::function<int(int, int)> selectFn(int n){
    if (n==1) {
        return [](int x, int y){ return x + y;};
    }else if (n==2){
        return [](int x, int y){ return x - y;};
    }else{
        return [](int x, int y){ return x * y;};
    }
}

int main() {
    auto f = selectFn(1);
    std::cout << f(2, 3) << std::endl; // 5
    std::cout << f(4, 5) << std::endl; // 9
    std::cout << f(7, 6) << std::endl; // 13
}
```


Доступ к окружению (замыкания)

Преимуществом анонимных функций является то, что они имеют доступ к окружению, в котором они определяются.

Замыкание (англ. closure) в программировании — функция первого класса, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции в окружающем коде и не являющиеся её параметрами. Говоря другим языком, замыкание — функция, которая ссылается на свободные переменные в своей области видимости.

Замыкание — это особый вид функции. Она определена в теле другой функции и создаётся каждый раз во время её выполнения. Синтаксически это выглядит как функция, находящаяся целиком в теле другой функции. При этом вложенная внутренняя функция содержит ссылки на локальные переменные внешней функции. Каждый раз при выполнении внешней функции происходит создание нового экземпляра внутренней функции, с новыми ссылками на переменные внешней функции.

Нелокальные переменные — с точки зрения функции это переменные которые не являются её локальными переменными, параметрами, переменными возврата, глобальными переменными.

C++

```
std::function<int(int)> add(int x) {  
    return [x](int y){  
        auto z = x + y;  
        std::cout << x << "+" << y << "=" << z << std::endl;  
        return z;  
    };  
}  
  
int main(){  
    auto add_3 = add(3);  
    add_3(6);  
}
```

add_3(6)
вернёт 9 и
выведет в
терминал
3+6=9

JavaScript

```
const add = function(x) {  
    return function(y) {  
        const z = x + y;  
        console.log(x + '+' + y + '=' + z);  
        return z;  
    };  
};  
  
const add_3 = add(3);  
  
add_3(6);
```

add_3(6)
вернёт 9 и
выведет в
терминал
3+6=9

Доступ к окружению (замыкания)

В C++ замыкания работают не совсем ожидаемым образом, если смотреть на них с точки зрения других языков.

Захватить переменную из окружающего контекста можно двумя способами:

- по значению. В этом случае значения захваченных переменных будут **копироваться в объект лямбда-функции** и использоваться внутри нее. После выхода из родительской функции захваченное значение продолжает "жить" внутри объекта лямбда-функции.
- по ссылке. Это означает, что лямбда-функция будет использовать оригинальные переменные, а не их копии. После выхода из родительской функции всё её переменные будут уничтожены, а лямбда продолжит ссылаться на том место в памяти где была переменная, т.е. получится "висячая" ссылка связанная со свободной областью памяти. Это может привести к потенциальным ошибкам.

Способы захватить контекст лямбда-функцией:

[=]	capture all by value
[&]	capture all by reference
[=, &x]	x by reference, all others by value
[&, x]	x by value, all others by reference
[x, &y]	only x by value and y by reference
[&x, y]	only x by reference and y by value
[&x, y, &z]	only x by ref, y by value and z by ref

Классификация функций

Функцию называют функцией **первого класса**, если она является объектом первого класса.

Объект называют «объектом первого класса», если он:

- может быть сохранен в переменной или структурах данных;
- может быть передан в функцию как аргумент;
- может быть возвращен из функции как результат;
- может быть создан во время выполнения программы;
- не зависит от именования.

Функция высшего порядка — в программировании функция, принимающая в качестве аргументов другие функции или возвращающая другую функцию в качестве результата. Основная идея состоит в том, что функции имеют тот же статус, что и другие объекты данных.

Чистая функция — это функция, которая:

- является детерминированной. Функция является детерминированной, если для одного и того же набора входных значений она возвращает одинаковый результат. Т.е. функция не зависит не от чего, кроме своих параметров и не имеет внутреннего состояния.
- не обладает побочными эффектами. В императивных языках некоторые функции в процессе выполнения своих вычислений могут модифицировать значения глобальных переменных, осуществлять операции ввода-вывода, реагировать на исключительные ситуации, вызывая их обработчики. Такие функции называются **функциями с побочными эффектами**. Другим видом побочных эффектов является модификация переданных в функцию параметров (переменных), когда в процессе вычисления выходного значения функции изменяется и значение входного параметра.