

Лекция 2. Функции. Многофайловая компоновка

Программирование на C++

Мехмат ЮФУ

Осенний семестр 2026

- 1 Функции: объявление, определение, прототипы.
- 2 Перегрузка функций.
- 3 Аргументы по умолчанию.
- 4 inline-функции и накладные расходы.
- 5 Ссылки и передача параметров.
- 6 Многофайловая компоновка: заголовочные файлы.
- 7 Директивы препроцессора и условная компиляция.
- 8 Стражи включения.
- 9 Этапы сборки и ошибки линковки.
- 10 extern и глобальные переменные.
- 11 Пространства имён.
- 12 Модули C++20 (обзор).

Что такое функция?

Определение

Функция — это именованный блок кода, который выполняет конкретную задачу и может возвращать результат.

- В математике: $f(x) = x^2$ — функция.
- В программировании: функция — это «мини-программа» внутри программы.
- Принимает входные данные (параметры), возвращает результат.

Пример

```
int square(int x) { return x * x; }
```

Зачем нужны функции?

- **Повторное использование** — написали один раз, вызываем много раз.
- **Модульность** — программа разбивается на логические блоки.
- **Упрощение отладки** — каждая функция тестируется отдельно.
- **Абстракция** — скрываем детали реализации.
- **Читаемость** — код становится понятнее.

Аналогия

Функция как рецепт: есть ингредиенты (параметры), есть инструкции (тело), есть результат (возвращаемое значение).

тип имя(параметры) { тело }

Пример

```
int my_max(int a, int b) {  
    return (a > b) ? a : b;  
}
```

- `int` — тип возвращаемого значения.
- `my_max` — имя функции.
- `int a, int b` — параметры.
- `return` — возврат результата.

auto для возврата значения (C++14)

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

auto add(int a, int b) {
    return a + b; // тип выводится как int
}

auto square(double x) {
    return x * x; // тип выводится как double
}

int main() {
    cout << add(5, 3) << endl; // 8
    cout << square(2.5) << endl; // 6.25
    return 0;
}
```

- Тип возвращаемого значения выводится автоматически.
- Удобно для коротких функций с одним return.
- Доступно с C++14.

Объявление и определение функции

Объявление (прототип): сообщает компилятору о существовании функции.

```
double area(double r);    // прототип
int my_max(int a, int b); // прототип
void printMessage();      // прототип
```

Определение: содержит тело функции.

```
double area(double r) {
    return 3.14159 * r * r;
}
```

Важно

Прототип заканчивается точкой с запятой! Определение — нет.

Зачем нужны прототипы?

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

int my_max(int a, int b); // прототип

int main() {
    cout << my_max(5, 3); // OK
    return 0;
}

int my_max(int a, int b) {
    return (a > b) ? a : b;
}
```

- Без прототипа — ошибка компиляции.
- Прототипы обычно помещают в заголовочные файлы.
- Позволяют использовать функции до их определения.

Вопрос: что выведет этот код?

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

int main() {
    cout << my_max(5, 3);
    return 0;
}

int my_max(int a, int b) {
    return (a > b) ? a : b;
}
```

Вопрос: что выведет этот код?

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

int main() {
    cout << my_max(5, 3);
    return 0;
}

int my_max(int a, int b) {
    return (a > b) ? a : b;
}
```

Ответ: Ошибка компиляции (нет прототипа `my_max` до вызова).

Перегрузка функций

- Несколько функций с одним именем, но разными параметрами.
- Компилятор выбирает нужную по типу и количеству аргументов.
- Упрощает интерфейс: одно имя для логически схожих операций.

```
int my_max(int a, int b) { return (a > b) ? a : b; }  
double my_max(double a, double b) { return (a > b) ? a : b; }  
char my_max(char a, char b) { return (a > b) ? a : b; }  
int my_max(int a, int b, int c) { return my_max(my_max(a,b), c); }
```

Правила перегрузки

- Функции должны отличаться:
 - ❶ Количеством параметров.
 - ❷ Типами параметров.
 - ❸ Порядком параметров (если типы разные).
- Тип возвращаемого значения **не** влияет на перегрузку.
- Перегрузка разрешается на этапе компиляции (раннее связывание).

Ошибка

```
int f(int a);  
double f(int a); // ОШИБКА!
```

Вопрос: что выведет этот код?

```
int my_max(int a, int b) { return (a > b) ? a : b; }
double my_max(double a, double b) { return (a > b) ? a : b; }

int main() {
    cout << my_max(5.5, 3.2);
    return 0;
}
```

Вопрос: что выведет этот код?

```
int my_max(int a, int b) { return (a > b) ? a : b; }
double my_max(double a, double b) { return (a > b) ? a : b; }

int main() {
    cout << my_max(5.5, 3.2);
    return 0;
}
```

Ответ: 5.5 (вызовется версия для double).

Аргументы по умолчанию

```
void printMessage(const string& msg = "Hello, World!") {  
    cout << msg << endl;  
}  
  
int main() {  
    printMessage();           // Hello, World!  
    printMessage("Hi!");     // Hi!  
    return 0;  
}
```

- Значения по умолчанию задаются в объявлении (прототипе).
- Должны идти в конце списка параметров.
- Упрощают вызов часто используемых функций.

Ошибка с аргументами по умолчанию

```
void f(int a = 1, int b); // ОШИБКА!  
void f(int a, int b = 2, int c = 3); // OK  
void f(int a = 1, int b = 2, int c = 3); // OK
```

Правило

Аргументы по умолчанию могут быть только в конце списка параметров.

```
inline int square(int x) {  
    return x * x;  
}  
  
int main() {  
    int a = square(5); // вместо вызова - подстановка кода  
    return 0;  
}
```

- Рекомендация компилятору встроить тело функции.
- Уменьшает накладные расходы на вызов.
- Не гарантируется — компилятор решает сам.

inline-переменные (C++17)

Проблема

Глобальные переменные, определённые в заголовочных файлах, приводят к ошибкам линковки (multiple definition).

Решение: inline-переменные

```
// config.h
#ifndef CONFIG_H
#define CONFIG_H

inline int globalCounter = 0;    // OK!
inline const double PI = 3.14159; // OK!

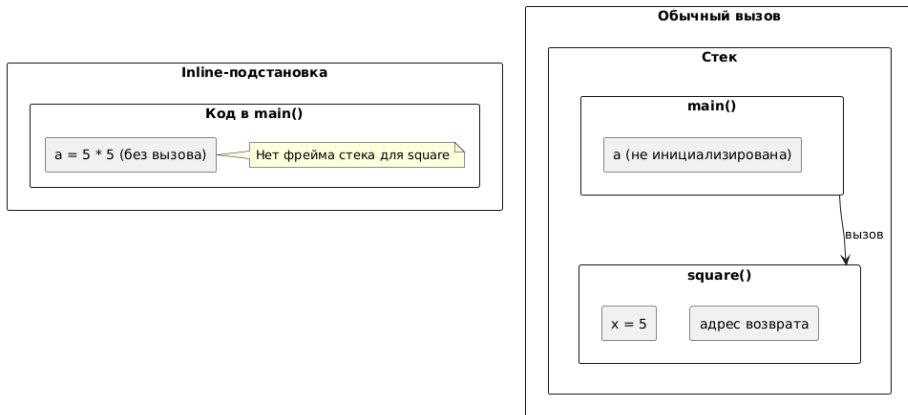
#endif
```

- Позволяет определять глобальные переменные в заголовочных файлах.
- Компилятор гарантирует, что будет только одно определение.
- Аналог inline-функций для переменных.

- При вызове функции:
 - 1 Создаётся запись активации (фрейм стека).
 - 2 Копируются аргументы.
 - 3 Сохраняется адрес возврата.
 - 4 Выделяется память для локальных переменных.
- Для маленьких функций накладные расходы могут превышать полезную работу.
- `inline` помогает этого избежать.

Накладные расходы на вызов

Сравнение: обычный вызов vs inline-подстановка



Когда inline может не сработать

- Рекурсивные функции — компилятор обычно не встраивает.
- Функции с большим телом — компилятор может проигнорировать.
- Виртуальные функции (будет позже в курсе).
- Функции, адрес которых берётся (указатели на функции).

Совет

Используйте inline только для маленьких функций (1–5 строк).

Накладные расходы для обычной функции

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

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

int main() {
    int x = 5, y = 3;
    int s = sum(x, y);
    cout << s;
    return 0;
}
```

Накладные расходы для обычной функции

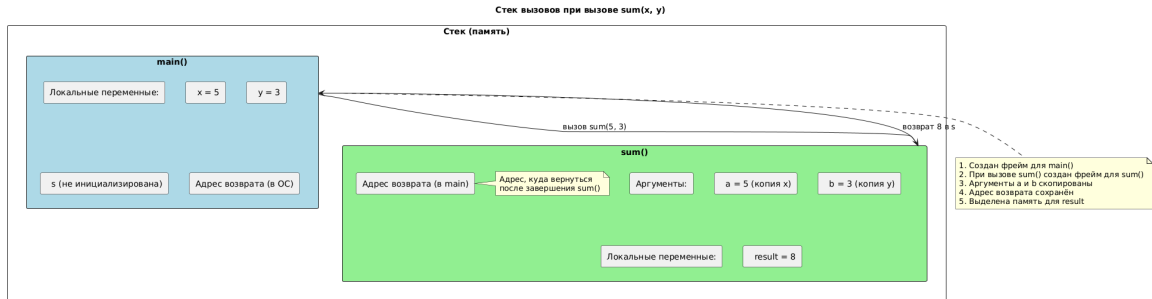


Рис. 1: Стек вызовов для функции `sum()`

Ссылки (references)

```
int a = 5, b = 20;  
int& ref = a; // ref - ссылка на a  
ref = 10; // a стало 10  
cout << a << endl; // 10  
ref = b; // a стало 20  
cout << a << endl; // 20
```

- Ссылка — псевдоним (другое имя) для существующей переменной.
- Должна быть инициализирована при объявлении.
- Не может быть перенаправлена на другой объект.
- Ссылка не занимает памяти (в отличие от указателя).

Передача параметров по ссылке

```
void swap(int& a, int& b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}  
  
int main() {  
    int x = 5, y = 10;  
    swap(x, y);  
    cout << x << " " << y; // 10 5  
    return 0;  
}
```

- Изменения в функции влияют на оригинал.
- Нет копирования — эффективно для больших объектов.

Передача по значению vs по ссылке

```
void byValue(int a) { a = 100; }    // копия
void byReference(int& a) { a = 100; } // оригинал

int main() {
    int x = 5;
    byValue(x);
    cout << x; // 5

    byReference(x);
    cout << x; // 100

    return 0;
}
```

- По значению — безопасно, но дорого для больших объектов.
- По ссылке — эффективно, но можно изменить оригинал.

```
void print(const string& s) {  
    cout << s << endl;  
    // s = "new"; // ОШИБКА!  
}  
  
int main() {  
    string text = "Hello";  
    print(text); // Hello  
    print("World"); // World, литерал тоже можно!  
    return 0;  
}
```

- const защищает объект от изменения.
- Позволяет передавать временные объекты (литералы).
- Предпочтительный способ передачи больших объектов.

Ссылки на структуры

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

void move(Point& p, double dx, double dy) {
    p.x += dx;
    p.y += dy;
}

int main() {
    Point p = {1.0, 2.0};
    move(p, 0.5, 1.5);
    cout << p.x << " " << p.y; // 1.5 3.5
    return 0;
}
```

- Передача структуры по ссылке — эффективно.
- Избегает копирования всей структуры.

Вопрос: что выведет этот код?

```
void f(int& a) { a++; }  
void g(const int& a) { cout << a; }  
  
int main() {  
    int x = 5;  
    f(x);  
    g(x);  
    return 0;  
}
```

Вопрос: что выведет этот код?

```
void f(int& a) { a++; }  
void g(const int& a) { cout << a; }  
  
int main() {  
    int x = 5;  
    f(x);  
    g(x);  
    return 0;  
}
```

Ответ: 6 (f увеличила x, g вывела его).

void-функции и досрочный выход

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

void printPositive(int x) {
    if (x <= 0) return; // досрочный выход
    cout << x << " is positive" << endl;
}

int main() {
    printPositive(5); // 5 is positive
    printPositive(-3); // ничего не выводит
    return 0;
}
```

- return; в void-функции прерывает выполнение.
- Полезно для проверки предусловий.

Зачем нужны несколько файлов?

- **Организация кода** — логическое разделение на модули.
- **Повторное использование** — один модуль можно использовать в разных проектах.
- **Ускорение компиляции** — перекомпилируются только изменённые файлы.
- **Совместная разработка** — разные люди могут работать над разными файлами.
- **Библиотеки** — готовые модули можно подключать без перекомпиляции.

Пример проекта из трёх файлов

math_utils.h:

```
#ifndef MATH_UTILS_H
#define MATH_UTILS_H
int add(int a, int b);
int sub(int a, int b);
#endif
```

math_utils.cpp:

```
#include "math_utils.h"
int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
```

main.cpp:

```
#include <iostream>
#include "math_utils.h"
int main() {
    std::cout << add(5, 3) << std::endl;
    return 0;
}
```

Что должно быть в заголовочном файле?

Можно	Нельзя
Прототипы функций	Определения функций (кроме inline)
Объявления переменных (extern)	Определения переменных
Определения констант	—
inline-функции	—
Шаблоны	—
Директивы препроцессора	—
Определения типов (typedef, using)	—

Файлы реализации (.cpp)

```
// math_utils.cpp
#include "math_utils.h"

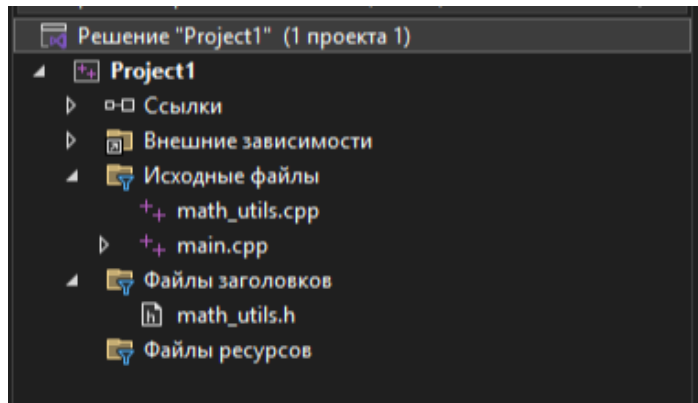
int my_max(int a, int b) {
    return (a > b) ? a : b;
}

double area(double r) {
    return 3.14159 * r * r;
}
```

- Содержат **определения** функций.
- Компилируются отдельно.
- Линковщик собирает всё в один исполняемый файл.

Структура проекта

- `math_utils.h` — объявления (прототипы).
- `math_utils.cpp` — реализация.
- `main.cpp` — точка входа.



Вопрос: что выведет этот код?

```
// math_utils.h
int add(int a, int b);

// math_utils.cpp
int add(int a, int b) { return a + b; }

// main.cpp
#include "math_utils.h"
int main() { cout << add(2, 3); return 0; }
```

Вопрос: что выведет этот код?

```
// math_utils.h
int add(int a, int b);

// math_utils.cpp
int add(int a, int b) { return a + b; }

// main.cpp
#include "math_utils.h"
int main() { cout << add(2, 3); return 0; }
```

Ответ: 5 (всё корректно собрано).

Директивы препроцессора

Директива	Назначение
<code>#include</code>	Вставить содержимое файла
<code>#define</code>	Определить макрос или константу
<code>#undef</code>	Отменить определение макроса
<code>#ifdef</code>	Проверить, определён ли макрос
<code>#ifndef</code>	Проверить, не определён ли макрос
<code>#else</code>	Ветка условной компиляции
<code>#endif</code>	Конец условной компиляции

```
#define DEBUG_MODE

#ifdef DEBUG_MODE
    cout << "Debug: x = " << x << endl;
#else
    cout << x << endl;
#endif
```

- Если DEBUG_MODE определён — компилируется первый блок.
- Если не определён — второй блок.
- Используется для отладки, платформозависимого кода.

Пример с #undef

```
#define PI 3.14
cout << PI; // 3.14
#undef PI
// теперь PI не определён
// cout << PI; // ошибка!
```

- #undef отменяет определение макроса.
- Полезно для переопределения макросов.

Стражи включения (include guards)

Вариант 1 (классический):

```
#ifndef MATH_UTILS_H
#define MATH_UTILS_H
// содержимое заголовка (прототипы)
#endif
```

Вариант 2 (современный):

```
#pragma once
// содержимое заголовка (прототипы)
```

- Защищают от повторного включения одного заголовка.
- `#pragma once` — проще, но не стандартизирован.

Что произойдёт без стражей?

```
// a.h
struct A { int x; };

// b.h
#include "a.h"

// main.cpp
#include "a.h"
#include "b.h" // a.h включён дважды!
```

Что произойдёт без стражей?

```
// a.h
struct A { int x; };

// b.h
#include "a.h"

// main.cpp
#include "a.h"
#include "b.h" // a.h включён дважды!
```

Ошибка

Повторное определение структуры A.

Вопрос: что выведет этот код?

```
// a.h
#ifndef A_H
#define A_H
struct A { int x; };
#endif

// main.cpp
#include "a.h"
#include "a.h"
int main() { return 0; }
```

Вопрос: что выведет этот код?

```
// a.h
#ifndef A_H
#define A_H
struct A { int x; };
#endif

// main.cpp
#include "a.h"
#include "a.h"
int main() { return 0; }
```

Ответ: Компилируется без ошибок (стражи сработали).

Этапы сборки программы

- ❶ **Преппроцессинг:** обработка `#include`, `#define`, удаление комментариев.
- ❷ **Компиляция:** перевод в объектный код (`.obj` или `.o`) — для каждого `.cpp` отдельно.
- ❸ **Линковка:** сборка объектных файлов и библиотек в исполняемый файл (`.exe`).

Важно

При многофайловой сборке линковка становится ключевым этапом — она связывает определения из разных файлов.

Ошибка линковки: повторное определение

file1.cpp:

```
int global = 42;
```

file2.cpp:

```
int global = 100; // ОШИБКА!
```

Ошибка

Линковщик найдёт два определения глобальной переменной `global` и выдаст ошибку.

Ошибка линковки: отсутствие определения

main.cpp:

```
void f(); // объявление  
int main() { f(); return 0; }
```

other.cpp: — нет определения f.

Ошибка

Линковщик не найдёт определение функции f.

Решение: либо определить f, либо убрать вызов.

Демонстрация: undefined reference

math_utils.h:

```
#pragma once  
int add(int a, int b);
```

main.cpp:

```
#include <iostream>  
#include "math_utils.h"  
  
int main() {  
    std::cout << add(2, 3);  
  
    return 0;  
}
```

Ошибка

undefined reference to 'add(int, int)'

Решение: добавить math_utils.cpp или скомпилировать оба файла.

extern — объявление глобальной переменной

global.cpp:

```
int counter = 0; // определение
```

main.cpp:

```
extern int counter; // объявление

int main() {
    counter++;
    return 0;
}
```

- extern сообщает: «переменная определена в другом файле».
- Без extern — ошибка повторного определения.

Вопрос: что выведет этот код?

```
// file1.cpp
int x = 5;

// main.cpp
#include <iostream>
extern int x;

int main() {
    std::cout << x;

    return 0;
}
```

Вопрос: что выведет этот код?

```
// file1.cpp
int x = 5;

// main.cpp
#include <iostream>
extern int x;

int main() {
    std::cout << x;

    return 0;
}
```

Ответ: Напечатает 5 (всё корректно).

Пространства имён (namespace)

Определение

Пространство имён — это декларативная область, в рамках которой определяются различные идентификаторы (имена типов, функций, переменных, и т. д.).

- Позволяют группировать имена (функции, классы, переменные).
- Предотвращают конфликты имён в больших проектах.
- Синтаксис: `namespace MyLib { ... }`
- Доступ: `MyLib::function()`

Пример

Например, все стандартные сущности в `std`.

Вложенные пространства имён

```
#include <iostream>

namespace MyLibrary {
    namespace Math {
        double pi = 3.14159;
    }
    namespace IO {
        void print(const std::string& s) { std::cout << s; }
    }
}

int main() {
    // Использование:
    double p = MyLibrary::Math::pi;
    MyLibrary::IO::print("Hello");

    return 0;
}
```

Анонимные пространства имён

```
namespace {  
    int internalCounter = 0;  
    void helper() { /* ... */ }  
}  
  
int main() {  
    internalCounter++; // OK  
    helper();          // OK  
    // helper() видна только в этом файле  
  
    return 0;  
}
```

- Всё внутри анонимного пространства видно только в текущем файле.
- Замена статических глобальных переменных.

Псевдонимы пространств имён

```
namespace VeryLongNamespaceName {  
    void f() { /* ... */ }  
}  
  
namespace VLN = VeryLongNamespaceName;  
  
int main() {  
    VLN::f(); // Коротко и удобно  
    return 0;  
}
```

- Упрощает работу с длинными именами.
- Удобно для библиотек с глубокой вложенностью.

using-директива

```
#include <iostream>
using namespace std; // все имена из std становятся доступны

int main() {
    cout << "Hello"; // ОК, без std::
    return 0;
}
```

Осторожно

using namespace std; в заголовочных файлах — плохая практика!

Конфликт имён

```
namespace A { int x = 5; }  
namespace B { int x = 10; }  
using namespace A;  
using namespace B;  
int main() {  
    cout << x; // ОШИБКА: неоднозначность!  
    return 0;  
}
```

Решение

Использовать явную квалификацию: A::x или B::x.

Вопрос: что выведет этот код?

```
namespace A { int x = 5; }  
namespace B { int x = 10; }  
int main() {  
    cout << A::x << " " << B::x;  
    return 0;  
}
```

Вопрос: что выведет этот код?

```
namespace A { int x = 5; }  
namespace B { int x = 10; }  
int main() {  
    cout << A::x << " " << B::x;  
    return 0;  
}
```

Ответ: 5 10.

Модули C++20 (краткий обзор)

math.ixx:

```
export module math;  
  
export int add(int a, int b) {  
    return a + b;  
}
```

main.cpp:

```
import math;  
int main() {  
    int x = add(2, 3); // OK  
    return 0;  
}
```

- Альтернатива заголовочным файлам.
- Компилируются один раз, не зависят от порядка включения.
- Экспорт через `export`.

Преимущества модулей перед #include

- **Скорость компиляции:** модуль компилируется один раз.
- **Изоляция:** импорт не транзитивен (не видно внутренностей).
- **Отсутствие макросов:** макросы не влияют на модули.
- **Упорядоченность:** импорт не зависит от порядка.

Важно

Модули — будущее C++. Но пока заголовочные файлы остаются основным способом.

Сравнение модулей и заголовочных файлов

Характеристика	#include	Модули
Скорость компиляции	медленнее (повторная обработка)	быстрее (один раз)
Изоляция	нет (макросы влияют)	да
Порядок включения	важен	не важен
Поддержка компиляторами	все	C++20 (не все)

Типичные ошибки с функциями и компоновкой

- ❶ Забыли прототип функции перед использованием.
- ❷ Перегрузка только по возвращаемому типу (ошибка).
- ❸ Аргументы по умолчанию не в конце списка.
- ❹ Аргументы по умолчанию указаны в заголовочном файле (прототипе) и в `.cpp` (определении).
- ❺ Забыли `#include` нужного заголовочного файла.
- ❻ Повторное определение глобальной переменной.
- ❼ Не добавили все `.cpp` файлы в проект.

- Функции: объявление, определение, перегрузка, аргументы по умолчанию, inline.
- Ссылки: передача по ссылке, const-ссылки.
- Многофайловая компоновка: заголовочные файлы, стражи включения.
- Директивы препроцессора и условная компиляция.
- Этапы сборки и ошибки линковки.
- Пространства имён (вложенные, анонимные, псевдонимы).
- Модули C++20 (краткий обзор).

На следующей лекции: структуры, перечисления, кортежи.

Вопросы?