Лекция 1. Вводная

Илья Макаров

CS центр
7 сентября 2021
Санкт-Петербург

Некоторые вводные

Ожидается, что люди, пришедшие на курс:

- Имеют опыт программирования на любом языке программирования.
- Имеют базовые представления об ООП: классы, объекты, икапсуляция, наследование и полиморфизм.
- Знакомы с базовыми алгоритмами и структурами данных: поиск, сортировка, линейный массив, хеш-таблица, …

Какие результаты ожидаем

- Сможем самостоятельно решать задачи с использованием C++.
- Научимся самостоятельно находить ответы на свои вопросы.
- Будем писать эффективный, простой и быстрый код, применяя естественные для C++ подходы.

Где искать ответы на вопросы

Опишем примерный алгоритм поиска ответы на вопросы:

- Гуглим проблему и идем на `https://stackoverflow.com`.
- Читаем человекочитаемую документацию на `https://en.cppreference.com/w/`.
- Смотрим видеозаписи курсов на `https://stepik.org/course/7/` и `https://stepik.org/course/3206/`.
- Читаем актуальную версию черновика стандарта `https://github.com/cplusplus/draft`.

Особенности C

- **Эффективность.**
  Язык C позволяет писать программы, которые напрямую работают с железом.
- **Стандартизированность.**
  Спецификация языка C является международным стандартом.
- **Относительная простота.**
  Стандарт языка C занимает 230 страниц
  (против 700+ для Java и 1300+ для C++).

Совместимость C и C++

- Один из принципов разработки стандарта C++ — это сохранение совместимости с C.
- Синтаксис C++ унаследован от языка C.
- C++ не является в строгом смысле надмножеством C.
- Можно писать программы на C так, чтобы они успешно компилировались на C++.
- C и C++ сильно отличаются как по сложности, так и по принятым архитектурным решениям, которые используются в обоих языках.

Стандартизация C++

- Лишь в 1998 году был ратифицирован международный стандарт языка C++: ISO/IEC 14882:1998 “Standard for the C++ Programming Language”.
- В 2003 году был опубликован стандарт языка ISO/IEC 14882:2003, где были исправлены выявленные ошибки и недочёты предыдущей версии стандарта.
- С 2005 года началась работа над новой версией стандарта, которая получила кодовое название C++0x. В конце концов в 2011 году стандарт был принят и получил название C++11 ISO/IEC 14882:2011.
- В 2014 году вышел C++14: ISO/IEC 14882:2014.
- В 2017 году вышел C++17: ISO/IEC 14882:2017.
- В 2020 году вышел C++20: ISO/IEC 14882:2020.
- В данный момент готовится к публикации C++23.

Характеристики языка C++

Характеристики C++:

- сложный,
- мультипарадигмальный,
- эффективный,
- низкоуровневый,
- компилируемый,
- статически типизированный.

Сложность

- Описание стандарта занимает более 1300 страниц текста.
- Нет никакой возможности рассказать “весь C++” в рамках одного, пусть даже очень большого курса.
- В C++ программисту позволено очень многое, и это влечёт за собой большую ответственность.
- На плечи программиста ложится много дополнительной работы:
  - проверка корректности данных,
  - управление памятью,
  - обработка низкоуровневых ошибок,
  - …

Мультипарадигмальный

На C++ можно писать программы в рамках нескольких парадигм программирования:

- **процедурное программирование**
  (код “в стиле C”),
- **объектно-ориентированное программирование**
  (классы, наследование, виртуальные функции, …).
- **обобщённое программирование**
  (шаблоны функций и классов),
- **функциональное программирование**
  (функторы, безымянные функции, замыкания),
- **генеративное программирование**
  (метапрограммирование на шаблонах).

Эффективный

Одна из фундаментальных идей языков C и C++ — *отсутствие неявных накладных расходов*, которые присутствуют в других более высокоуровневых языках программирования.

- Программист сам выбирает уровень абстракции, на котором писать каждую отдельную часть программы.
- Можно реализовывать критические по производительности участки программы максимально эффективно.
- Эффективность делает C++ основным языком для разработки высоконагруженных приложений, приложений с компьютерной графикой и …

Низкоуровневый

Язык C++, как и C, позволяет работать напрямую с ресурсами компьютера.

- Позволяет писать низкоуровневые системные приложения (например, драйверы операционной системы).
- Неаккуратное обращение с системными ресурсами может привести к падению программы.

В C++ отсутствует автоматическое управление памятью.

- Позволяет программисту получить полный контроль над программой.
- Необходимость заботиться об освобождении памяти.

Компилируемый

C++ является компилируемым языком программирования.

Для того, чтобы запустить программу на C++, её нужно сначала *скомпилировать*.

Компиляция — преобразование текста программы на языке программирования в машинный код.

- Нет накладных расходов при исполнении программы.
- При компиляции можно отловить некоторые ошибки.
- Требуется компилировать для каждой платформы отдельно.

Статическая типизация

C++ является статически типизированным языком.

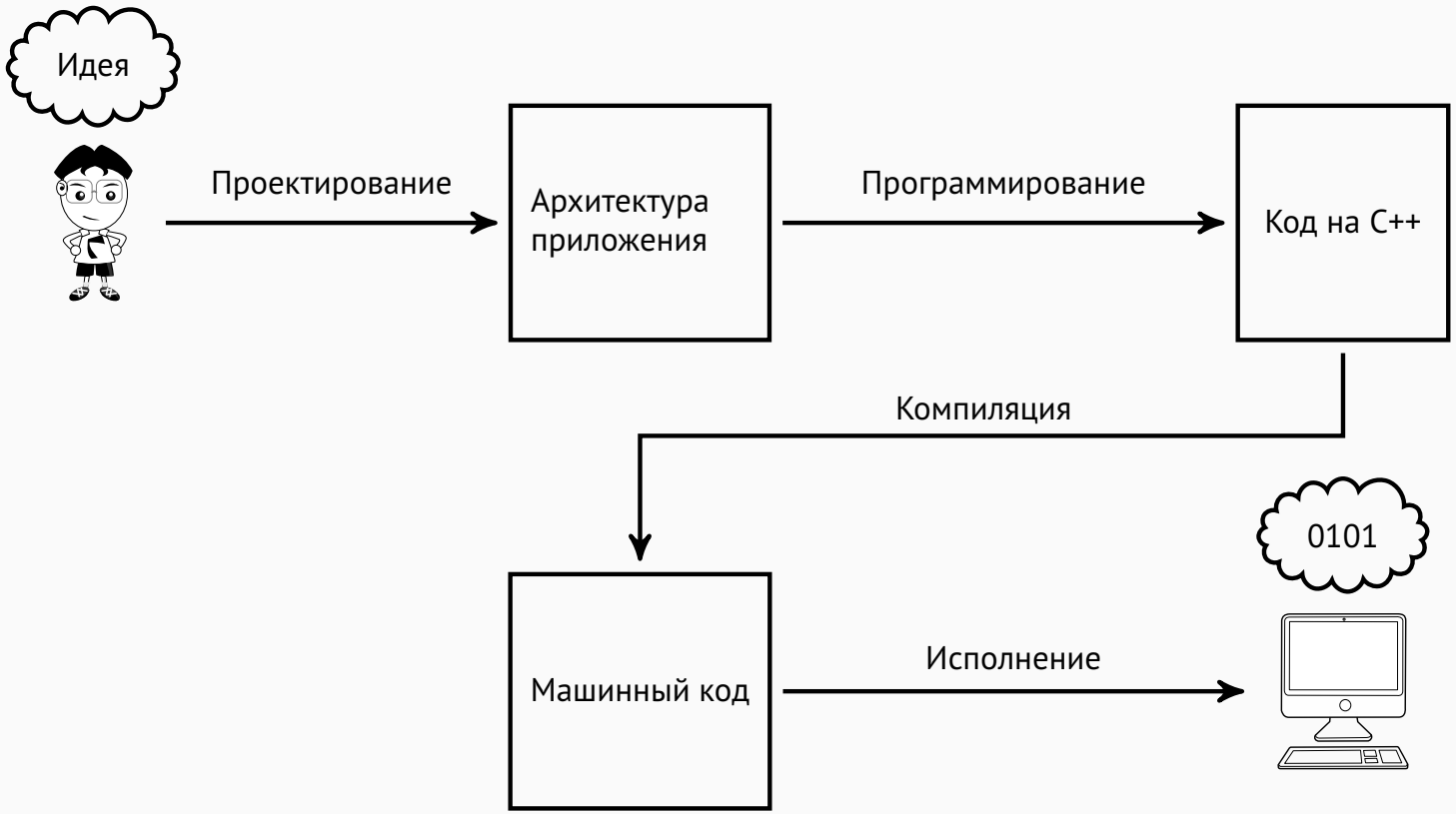
1. Каждая сущность в программе (переменная, функция и пр.) имеет свой тип,
2. и этот тип определяется на момент компиляции.

Это нужно для того, чтобы

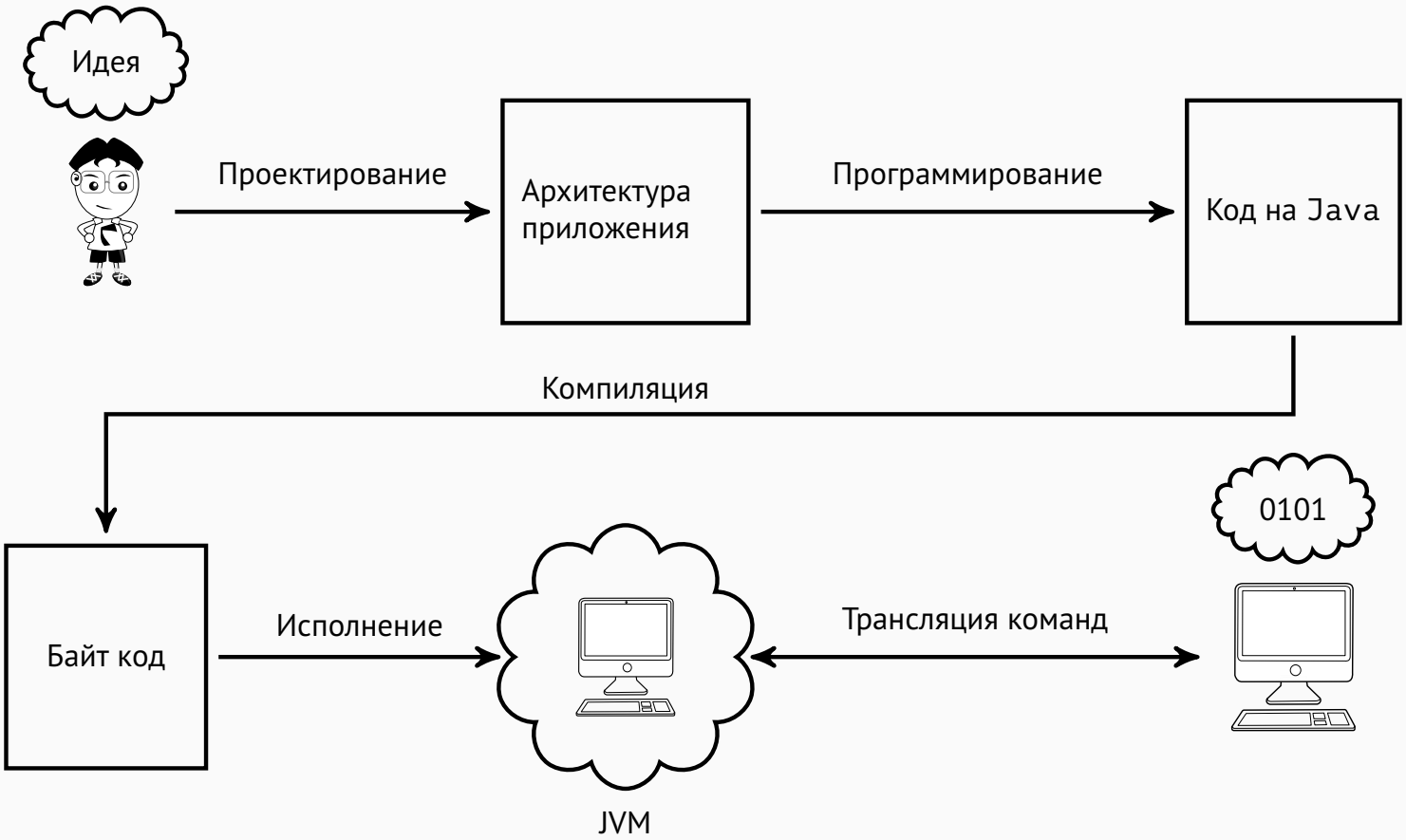
1. вычислить размер памяти, который будет занимать каждая переменная в программе,
2. определить, какая функция будет вызываться в каждом конкретном месте.

Всё это определяется на момент компиляции и “зашивается” в скомпилированную программу.
В машинном коде никаких типов уже нет — там идёт работа с последовательностями байт.

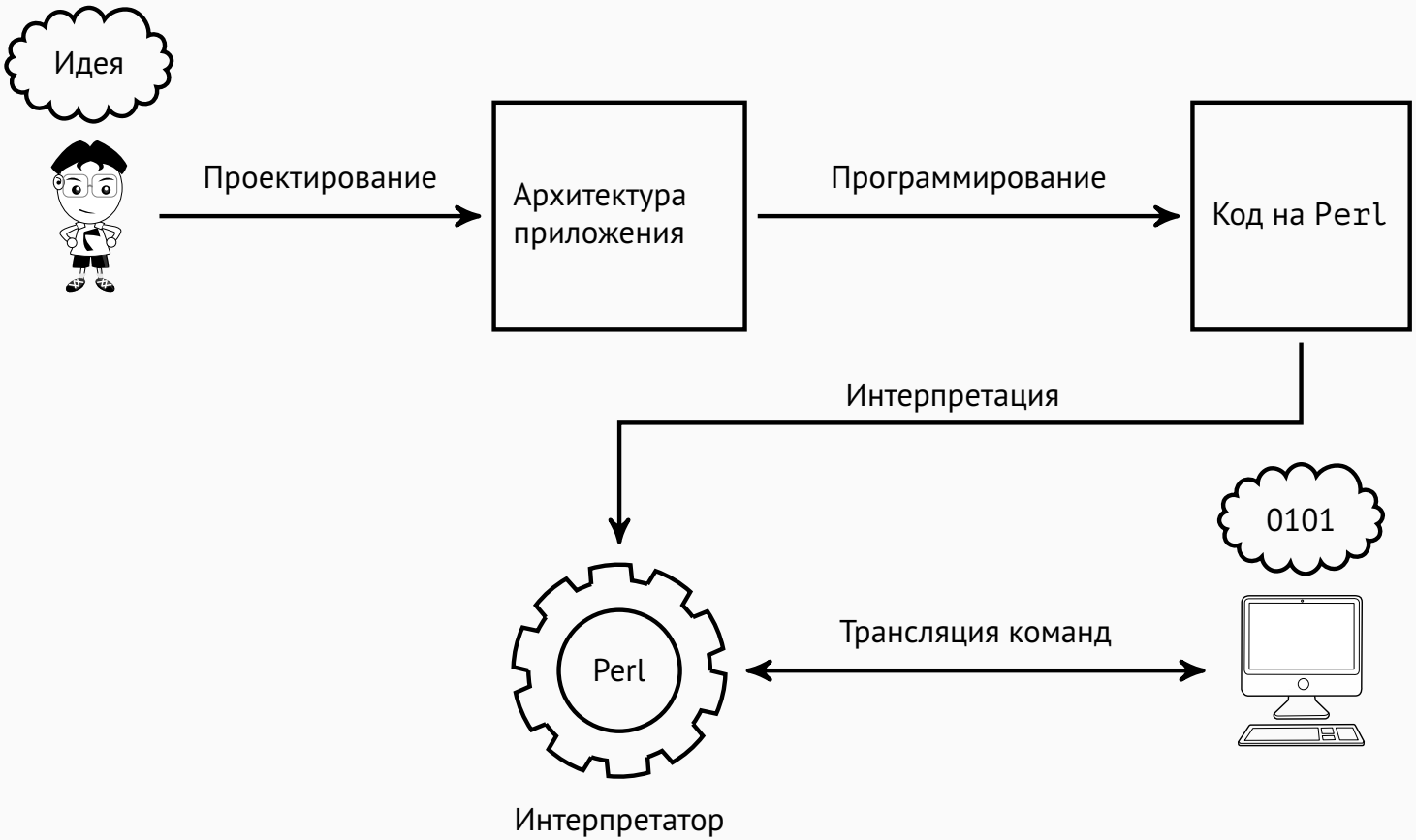
Что такое компиляция?



Что такое компиляция?



Что такое интерпретация?



Плюсы и минусы компилируемости в машинный код

Плюсы:

- эффективность: программа компилируется и оптимизируется для конкретного процессора,
- нет необходимости устанавливать сторонние приложения (такие как интерпретатор или виртуальная машина).

Минусы:

- нужно компилировать для каждой платформы,
- сложность внесения изменения в программу — нужно перекомпилировать заново.

Важно: компиляция — преобразование одностороннее, нельзя восстановить исходный код.

Разбиение программы

Зачем разбивать программу на файлы?

- С небольшими файлами удобнее работать.
- Разбиение на файлы структурирует код.
- Позволяет нескольким программистам разрабатывать приложение одновременно.
- Ускоряет компиляцию программы.
- Ускорение повторной компиляции при небольших изменениях в отдельных частях программы.

Файлы с кодом на C++ бывают двух типов:

1. файлы с исходным кодом (расширение `.cpp`, иногда `.C`),
2. заголовочные файлы (расширение `.hpp` или `.h`).

Простейшая программа на C++

- Файл `foo.hpp`:

```
#pragma once // страж включения

// объявление (declaration) функции foo
void foo();
```

- Файл `foo.cpp`:

```
#include "foo.hpp"
#include "bar.hpp"

// определение (definition) функции foo
void foo()
{
    bar(); // вызываем функцию bar
}
```

Простейшая программа на C++

- Файл `bar.hpp`:

```
#pragma once // страж включения

// объявление (declaration) функции bar
void bar();
```

- Файл `bar.cpp`:

```
#include "bar.hpp"

// определение (definition) функции bar
void bar()
{
    // полезный код
}
```

Простейшая программа на C++

- *Точка входа* — функция, вызываемая при запуске программы. По умолчанию — это функция `main`:

```
#include "foo.hpp"

int main()
{
    foo();
}
```

или

```
#include "foo.hpp"

int main(int argc, char ** argv)
{
    foo();
}
```

Замечания

- Не забывайте разбивать программу на части.
- Не забывайте про стражи включения.
- Отличайте определения (.cpp) от объявлений (.hpp).

Компилиция этап №1: препроцессор

- Язык препроцессора – это специальный язык программирования, встроенный в C++.
- Препроцессор работает с кодом на C++ как с текстом.
- Команды языка препроцессор называют директивами, все директивы начинаются со знака `#`.
- Директива `#include` позволяет подключать заголовочные файлы к файлам кода.
  1. `#include <foo.h>` – библиотечный заголовочный файл,
  2. `#include "bar.h"` – локальный заголовочный файл.
- Препроцессор заменяет директиву `#include "bar.h"` на содержимое файла `bar.h`.

Компилиция этап 2: компиляция

- На вход компилятору поступает код на C++ после обработки препроцессором.
- Каждый файл с кодом компилируется отдельно и независимо от других файлов с кодом.
- Компилируется только файлы с кодом (т.е. `*.cpp`).
- Заголовочные файлы сами по себе ни во что не компилируются, только в составе файлов с кодом.
- На выходе компилятора из каждого файла с кодом получается “объектный файл” — бинарный файл со скомпилированным кодом (с расширением `.o` или `.obj`).

Компилиция этап 3: линковка (компоновка)

- На этом этапе все объектные файлы объединяются в один исполняемый (или библиотечный) файл.
- При этом происходит подстановка адресов функций в места их вызова.

```
void foo()
{
    bar();
}
```

- По каждому объектному файлу строится таблица всех функций, которые в нём определены.

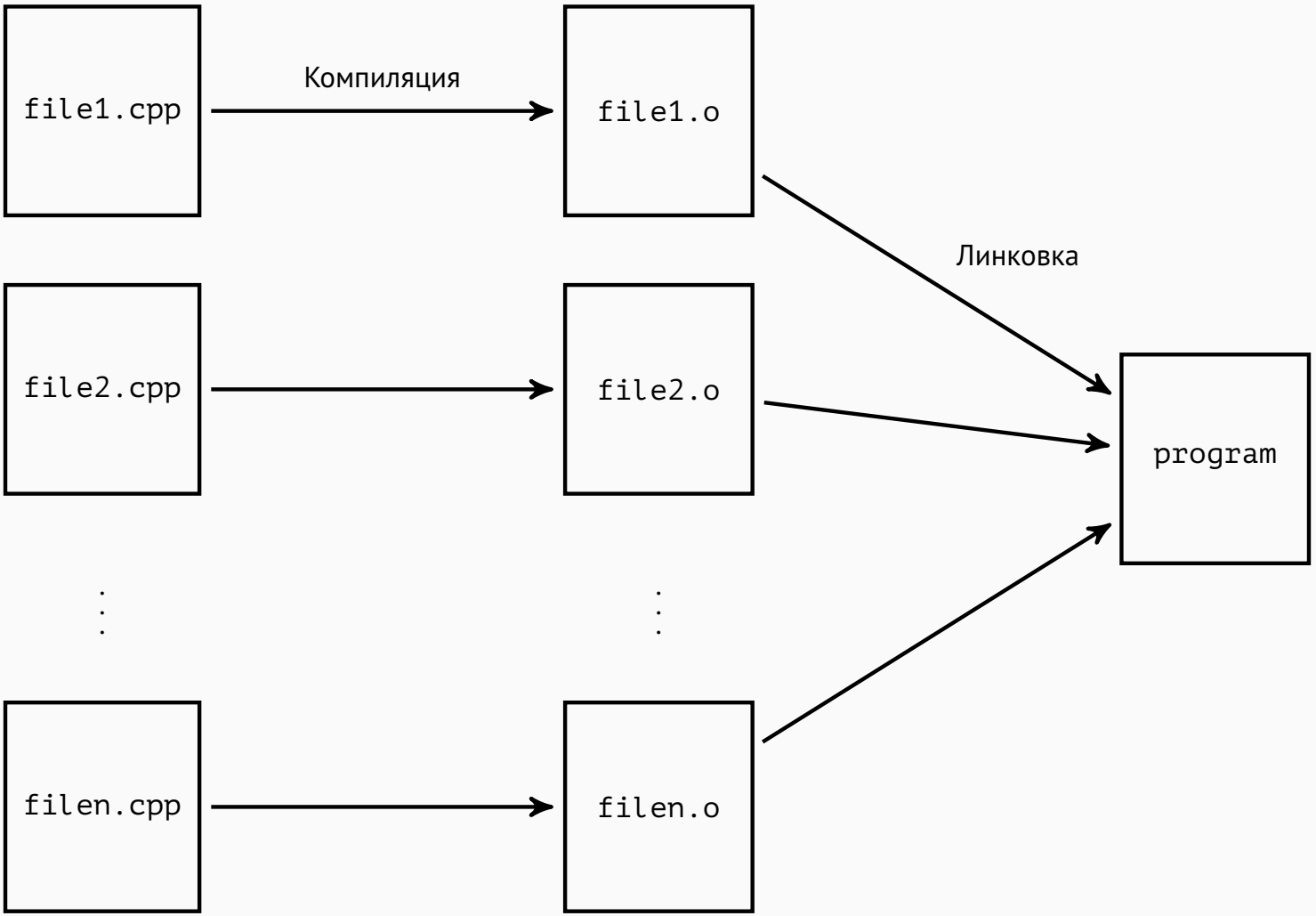
Компилиция этап 3: линковка (компоновка)

- На этапе компоновки важно, что каждая функция имеет уникальное имя.
- В C++ может быть две функции с одним именем, но разными параметрами.
- Имена функций искажаются (mangle) таким образом, что в их имени кодируются их параметры.
  Например, компилятор GCC превратит имя функции `foo`

```
void foo(int, double) {}
```

  в `_Z3fooid`.
- Аналогично функциям в линковке нуждаются глобальные переменные.

Общая схема компиляции



Простые типы данных C++

- Целочисленные:
  1. `char` (символьный тип данных)
  2. `short int`
  3. `int`
  4. `long int`

  Могут быть беззнаковыми (`unsigned`).
  - $-2^{n-1} \dots (2^{n-1} - 1)$ (n — число бит)
  - $0 \dots (2^n - 1)$ для `unsigned`
- Числа с плавающей точкой:
  1. `float`, 4 байта, 7 значащих цифр.
  2. `double`, 8 байт, 15 значащих цифр.
- Логический тип данных `bool`.
- Пустой тип `void`.

Замечания касательно простых типов

- Размеры многих типов ограничены только снизу так, например, тип int может быть размера 4 и 8 байт в зависимости от целевой платформы.
- Существуют алиасы, гарантирующие размер типа, они расположены в заголовочном файле `<cstdint>`:
  1. `std::int32_t` — знаковый, размера 32 бита (4 байта),
  2. `std::uint32_t` — беззнаковый, размера 32 бита (4 байта),
  3. `std::int16_t` — знаковый, размера 16 бит (2 байта),
  4. `std::int8_t` — знаковый, размера 8 бит (1 байта),
  5. `std::int64_t` — знаковый 64 бита (8 байт),
  6. и т.д.

Замечания касательно простых типов

- Для получения размера конкретного типа можно использовать оператор `sizeof`.
- `sizeof(std::int32_t) == 4`.
- Для получения максимального, минимального значения, можно использовать шаблон из заголовочного файла `<limits>`.
- Например:
  1. `std::numeric_limits<std::uint32_t>::max()` $= 2^{32} - 1$
  2. `std::numeric_limits<std::int32_t>::max()` $= 2^{31} - 1$

Литералы

- Целочисленные:
  1. `'a'` — код буквы 'a', тип `char`,
  2. `42` — все целые числа по умолчанию типа `int`,
  3. `1234567890L` — суффикс '`L`' соответствует типу `long`,
  4. `1703U` — суффикс '`U`' соответствует типу `unsigned int`,
  5. `2128506UL` — соответствует типу `unsigned long`.
- Числа с плавающей точкой:
  1. `3.14` — все числа с точкой по умолчанию типа `double`,
  2. `2.71F` — суффикс '`F`' соответствует типу `float`,
  3. `3.0E8` — соответствует $3.0 \cdot 10^8$.
- `true` и `false` — значения типа `bool`.
- Строки задаются в двойных кавычках: `"Text string"`.

Переменные

- При определении переменной указывается её тип. При определении можно сразу задать начальное значение (инициализация).

```
int      i = 10;
short    j = 20;
bool     b = false;

unsigned long l = 123123;

double x = 13.5, y = 3.1415;
float z;
```

- Нужно всегда инициализировать переменные.
- Нельзя определить переменную пустого типа `void`.

Операции

- Оператор присваивания: `=`.
- Арифметические:
  1. бинарные: `+ - * / %`,
  2. унарные: `++ --`.
- Логические:
  1. бинарные: `&& ||`,
  2. унарные: `!`.
- Сравнения: `== != > < >= <= <=>`.
- C-style приведение типов: `(type)`.
- Сокращённые версии бинарных операторов: `+= -= *= /= %=`.

Операции

```
int i = 10; // initialize i with integer value
i = (20 * 3) % 7; // integer calculations

int k = i++; // post increment
int l = --i; // pre decrement

bool b = !(k == l); // logical expression
// another logical expression
b = (a == 0) || (1 / a < 1);
auto p = k <=> l; // ordereing since C++20

double d = 3.1415; // initialize d with integer value
float f = (int)d; // cast d to float
// floating point calculations
d *= i + k; // equal to d = d * (i + k)
```

Инструкции

- Выполнение состоит из последовательности *инструкций*.
- Инструкции выполняются одна за другой.
- Порядок вычислений внутри инструкций не определён.

```
int i = 10;
i = (i += 5) + (i * 4); // unspecified behavior
```

- Блоки имеют вложенную область видимости:

```
int k = 10;
{
    int k = 5 * i;  // не видна за пределами блока
    i = (k += 5) + 5;
}
k = k + 1;
```

Замечания касательно инструкций

- В C++ существует несколько "особых состояний"программы, которые описываются в стандарте.
- Undefined behaviour — поведение программы не определено (может произойти все что угодно). Например, переполнение знакового целого числа.

```
int i = std::numeric_limits<int>::max() + 1; // UB
```

Замечания касательно инструкций

- Unspecified behaviour — в определённых маргинальных ситуациях программа может выдавать результат, зависящий от реализации компилятора.

```
void foo(int a, int b, int c) {
    std::cout << a << b << c; // prints a, b ,c
}

int i = 0;
foo(++i, i++, i); // unspecified
```

Условные операторы

- Оператор `if`:

```
int d = b * b - 4 * a * c;
if (d > 0) {
    roots = 2;
} else if (d == 0) {
    roots = 1;
} else {
    roots = 0;
}
```

- Тернарный условный оператор:

```
int roots = 0;
if (d >= 0)
    roots = (d > 0 ) ? 2 : 1;
```

Циклы

- Цикл `while`:

```
int squares = 0;
int k = 0;
while (k < 10) {
    squares += k * k;
    k = k + 1;
}
```

- Цикл `for`:

```
for (int k = 0; k < 10; k = k + 1) {
    squares += k * k;
}
```

- Для выхода из цикла используется оператор `break`.

Функции

- В сигнатуре функции указывается тип возвращаемого значений и типы параметров.
- Ключевое слово `return` возвращает значение.

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

- Переменные, определённые внутри функций, — *локальные*.
- Функция может возвращать `void`.
- Параметры передаются по значению (копируются).

```
void strange(double x, double y) {
    x = y;
}
```

Макросы

- Макросами в C++ называют инструкции препроцессора.
- Препроцессор C++ является самостоятельным языком, работающим с произвольными строками.
- Макросы можно использовать для определения функций:

```
int max1(int x, int y) {
    return x > y ? x : y;
}

#define max2(x, y)   x > y ? x : y

a = b + max2(c, d); // b + c > d ? c : d;
```

- Препроцессор “не знает” про синтаксис C++.

Макросы

- Параметры макросов нужно оборачивать в скобки:

```
#define max3(x, y)   ((x) > (y) ? (x) : (y))
```

- Это не избавляет от всех проблем:

```
int a = 1;
int b = 1;
int c = max3(++a, b);
// c = ((++a) > (b) ? (++a) : (b))
```

- Определять константы через макросы — плохая идея.
- Определять функции через макросы — плохая идея.
- Макросы можно использовать для условной компиляции:

```
#ifdef DEBUG
    // дополнительные проверки
#endif
```

Ввод-вывод

- Для консоли будем использовать библиотеку <iostream>.

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

- Ввод:

```
int a = 0;
int b = 0;
cin >> a >> b;
```

- Вывод:

```
cout << "a + b = " << (a + b) << endl;
```

- Аналогичным образов осуществляется ввод-вывод из файла <fstream>

Простая программа

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

int main ()
{
    int a = 0;
    int b = 0;

    cout << "Enter a and b: ";
    cin >> a >> b;

    cout << "a + b = " << (a + b) << endl;

    return 0;
}
```

Контракты при разработке программ

- `assert(<expression>)` или `static_assert(<expression>, <error-text>)` позволяют выполнять проверки во время исполенения и компиляции соотвественно.
- Такие проверки удобны во время разработки больших программных продуктов и рефакторинга.
- Важно отметить, что код для `assert(<expression>)` по умолчанию исполняется только в debug сборке.

Архитектура фон Неймана

Современных компьютеры построены по принципам архитектуры фон Неймана:

1. **Принцип однородности памяти.**
   Команды и данные хранятся в одной и той же памяти и внешне в памяти неразличимы.
2. **Принцип адресности.**
   Память состоит из пронумерованных ячеек.
3. **Принцип программного управления.**
   Все вычисления представляются в виде последовательности команд.
4. **Принцип двоичного кодирования.**
   Вся информация (данные и команды) кодируются двоичными числами.

Сегментация памяти

- Оперативная память, используемая в программе на C++, разделена на области двух типов:
  1. сегменты данных,
  2. сегменты кода (текстовые сегменты).
- В сегментах кода содержится код программы.
- В сегментах данных располагаются данные программы (значения переменных, массивы и пр.).
- При запуске программы выделяются два сегмента данных:
  1. сегмент глобальных данных,
  2. стек.
- В процессе работы программы могут выделяться и освобождаться дополнительные сегменты памяти.
- Обращения к адресу вне выделенных сегментов — ошибка времени выполнения (access violation, segmentation fault).

Как выполняется программа?

- Каждой функции в скомпилированном коде соответствует отдельная секция.
- Адрес начала такой секции — это адрес функции.
- Телу функции соответствует последовательность команд процессора.
- Работа с данными происходит на уровне байт, информация о типах отсутствует.
- В процессе выполнения адрес следующей инструкции хранится в специальном регистре процессора IP (Instruction Pointer).
- Команды выполняются последовательно, пока не встретится специальная команда (например, условный переход или вызов функции), которая изменит IP.

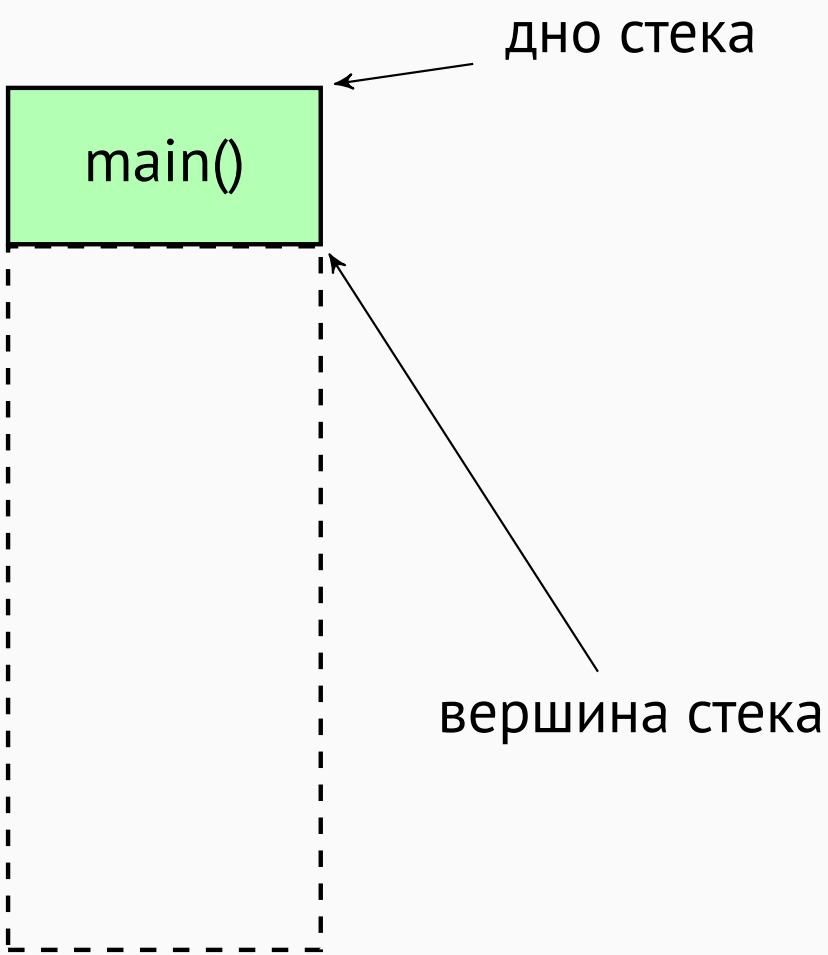
Ещё раз о линковке

- На этапе компиляции объектных файлов в места вызова функций подставляются имена функций.
- На этапе линковки в места вызова вместо имён функций подставляются их адреса.
- Ошибки линковки:
  1. `undefined reference`
     Функция имеет объявление, но не имеет тела.
  2. `multiple definition`
     Функция имеет два или более определений.
- Наиболее распространённый способ получить `multiple definition` — определить функцию в заголовочном файле, который включён в несколько `.cpp` файлов.

Стек вызовов

- Стек вызовов — это сегмент данных, используемый для хранения локальных переменных и временных значений.
- Не стоит путать стек с одноимённой структурой данных, у стека в C++ можно обратиться к произвольной ячейке.
- Стек выделяется при запуске программы.
- Стек обычно небольшой по размеру (4Мб).
- Функции хранят свои локальные переменные на стеке.
- При выходе из функции соответствующая область стека объявляется свободной.
- Промежуточные значения, возникающие при вычислении сложных выражений, также хранятся на стеке.

Устройство стека



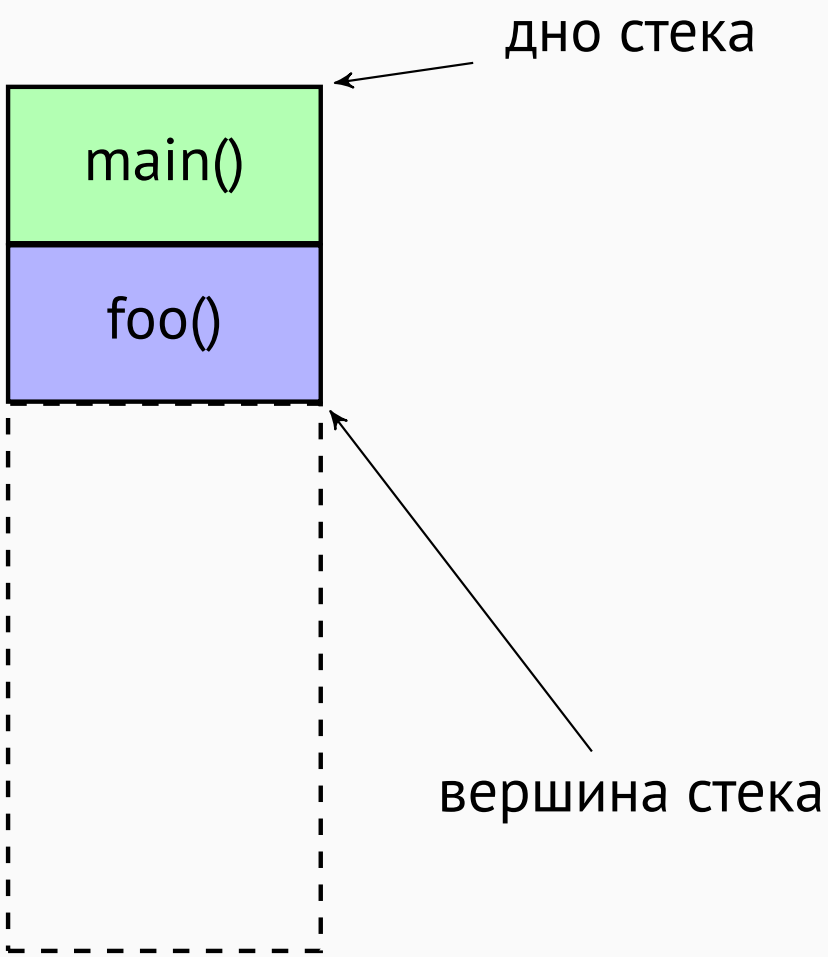

```
void bar( ) {
    int c;
}

void foo( ) {
    int b = 3;
    bar();
}

int main( ) {
    int a = 3;
    foo();
    bar();

    return 0;
}
```

Устройство стека



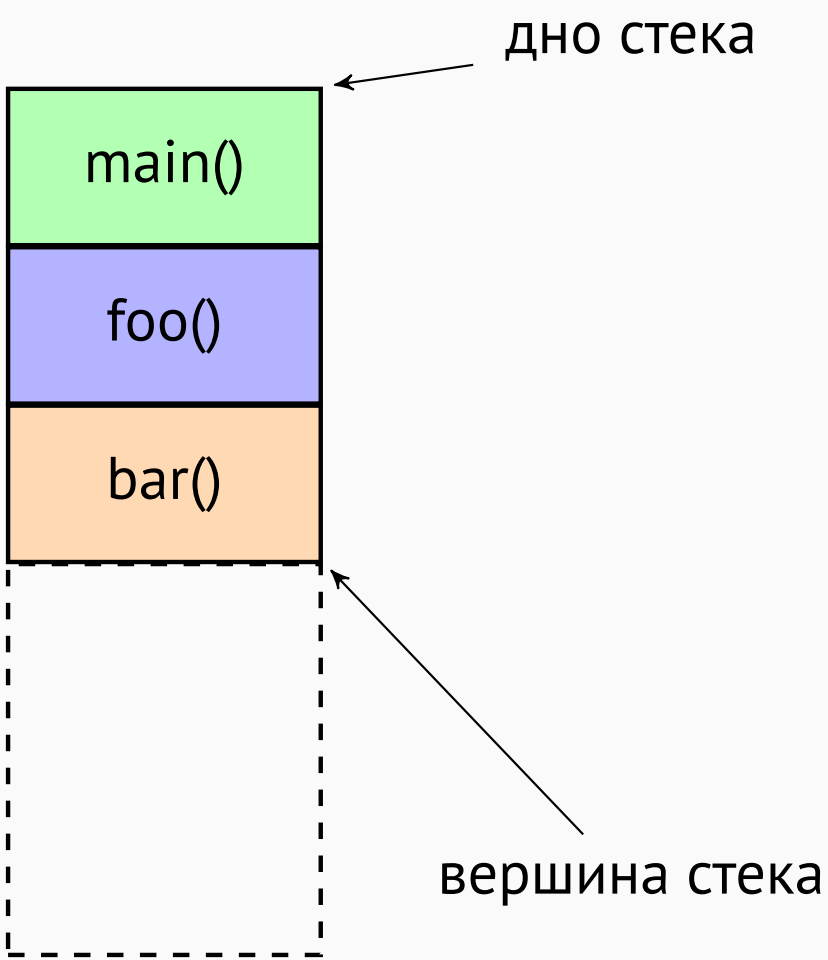

```
void bar( ) {
    int c;
}

void foo( ) {
    int b = 3;
    bar();
}

int main( ) {
    int a = 3;
    foo();
    bar();

    return 0;
}
```

Устройство стека



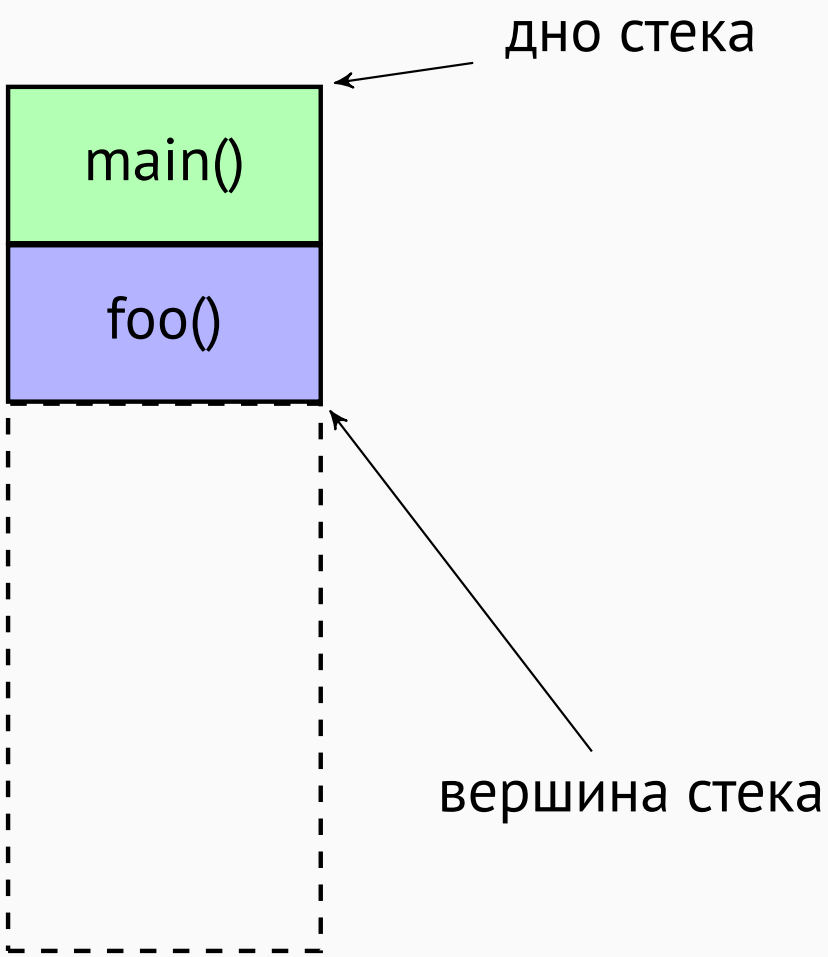

```
void bar( ) {
    int c;
}

void foo( ) {
    int b = 3;
    bar();
}

int main( ) {
    int a = 3;
    foo();
    bar();

    return 0;
}
```

Устройство стека



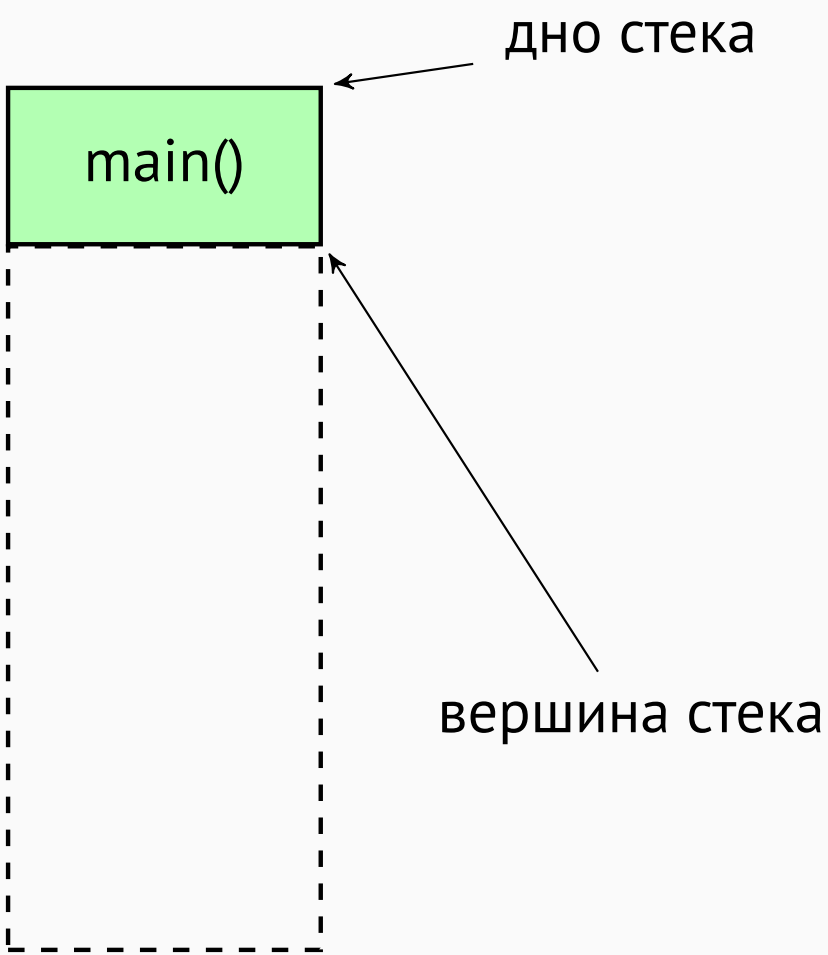

```
void bar( ) {
    int c;
}

void foo( ) {
    int b = 3;
    bar();
}

int main( ) {
    int a = 3;
    foo();
    bar();

    return 0;
}
```

Устройство стека



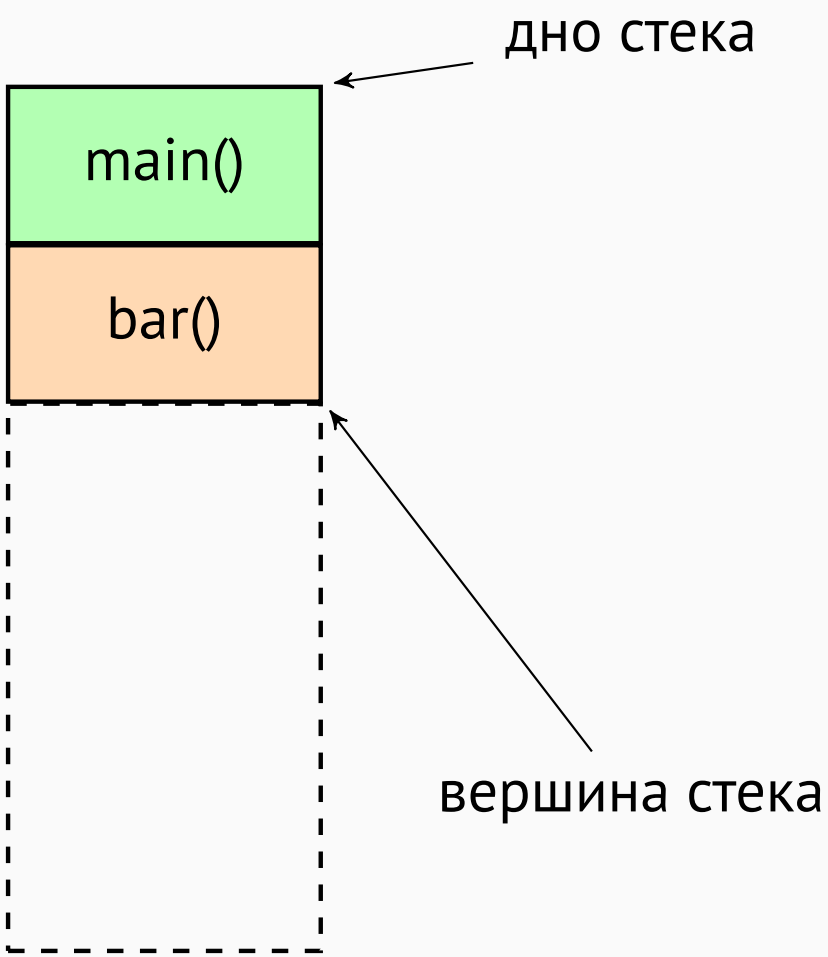

```
void bar( ) {
    int c;
}

void foo( ) {
    int b = 3;
    bar();
}

int main( ) {
    int a = 3;
    foo();
    bar();

    return 0;
}
```

Устройство стека



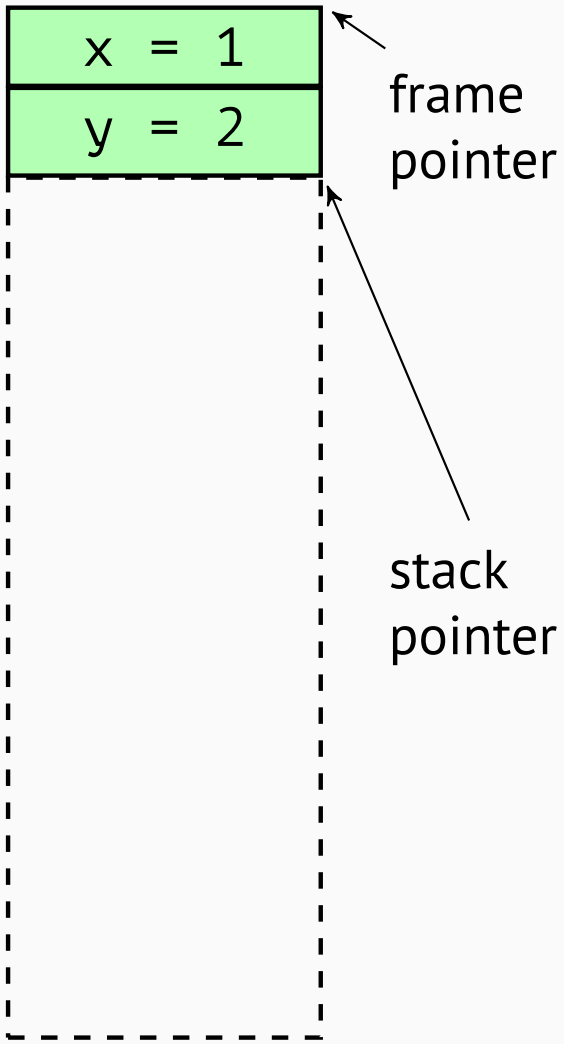

```
void bar( ) {
    int c;
}

void foo( ) {
    int b = 3;
    bar();
}

int main( ) {
    int a = 3;
    foo();
    bar();

    return 0;
}
```

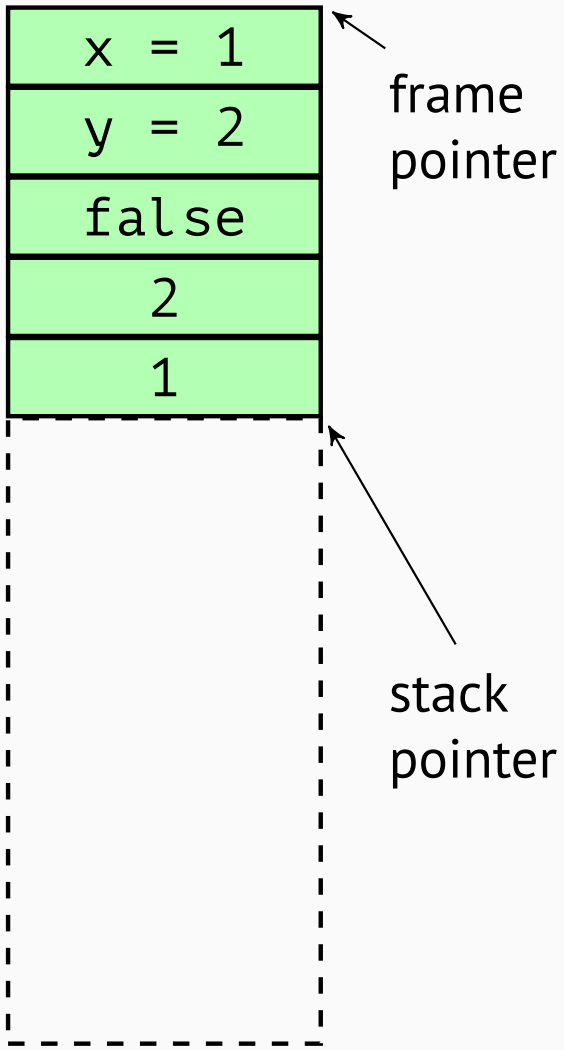
Вызов функции




```
int foo(int a, int b, bool c)
{
    double d = a * b * 2.71;
    int    h = c ? d : d / 2;
    return h;
}

int main( )
{
    int x = 1;
    int y = 2;
    x = foo (x, y, false);
    cout << x;
    return 0;
}
```

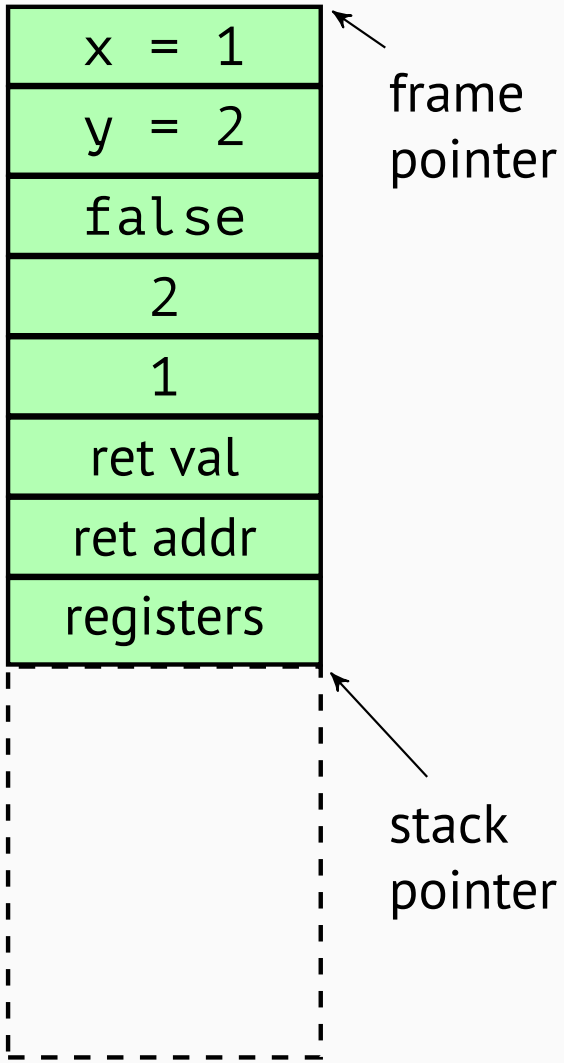
Вызов функции




```
int foo(int a, int b, bool c)
{
    double d = a * b * 2.71;
    int    h = c ? d : d / 2;
    return h;
}

int main( )
{
    int x = 1;
    int y = 2;
    x = foo (x, y, false);
    cout << x;
    return 0;
}
```

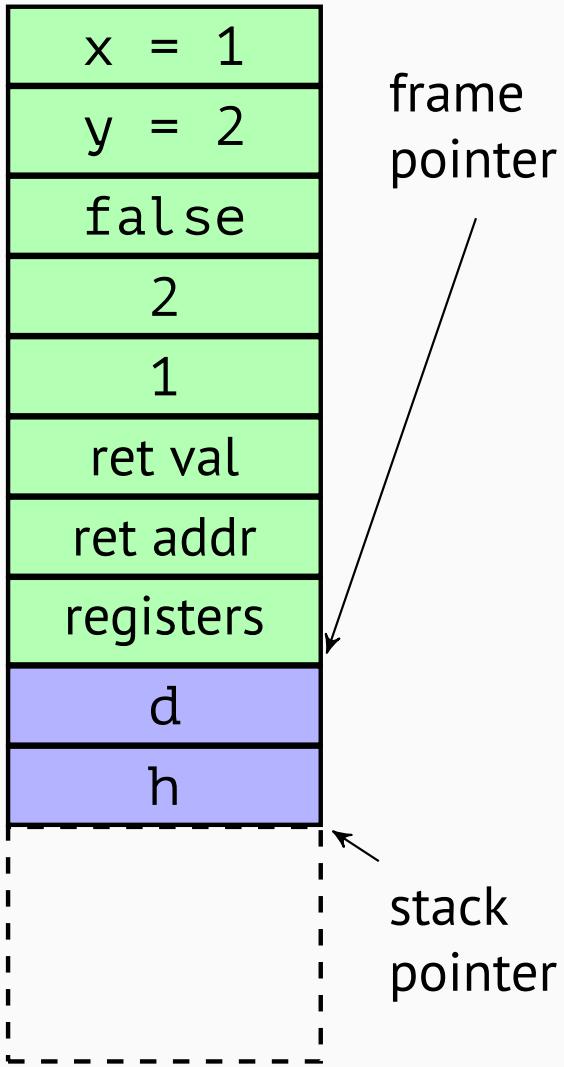
Вызов функции




```
int foo(int a, int b, bool c)
{
    double d = a * b * 2.71;
    int    h = c ? d : d / 2;
    return h;
}

int main( )
{
    int x = 1;
    int y = 2;
    x = foo (x, y, false);
    cout << x;
    return 0;
}
```

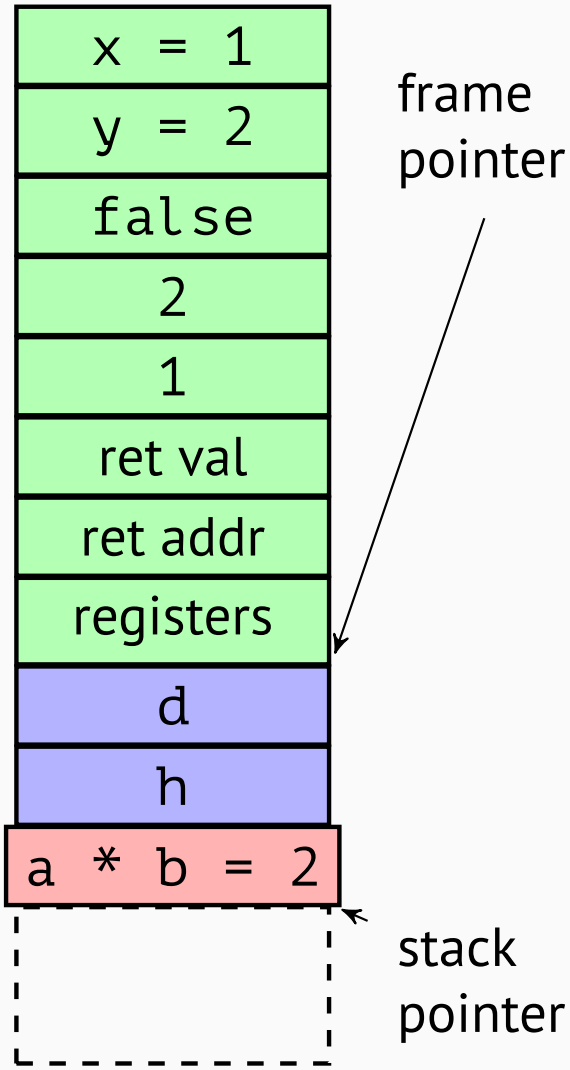
Вызов функции




```
int foo(int a, int b, bool c)
{
    double d = a * b * 2.71;
    int    h = c ? d : d / 2;
    return h;
}

int main( )
{
    int x = 1;
    int y = 2;
    x = foo (x, y, false);
    cout << x;
    return 0;
}
```

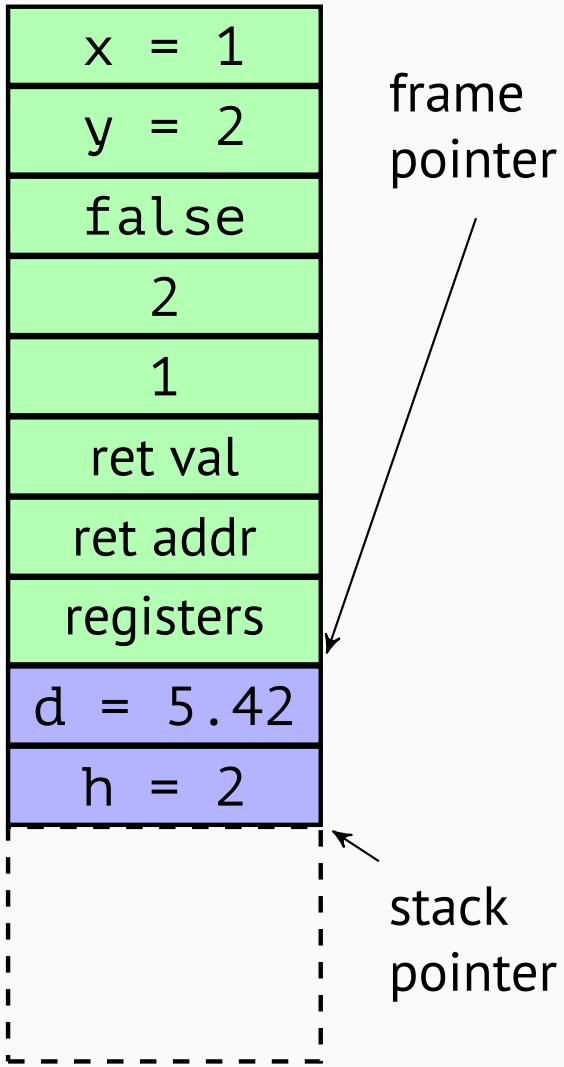
Вызов функции




```
int foo(int a, int b, bool c)
{
    double d = a * b * 2.71;
    int    h = c ? d : d / 2;
    return h;
}

int main( )
{
    int x = 1;
    int y = 2;
    x = foo (x, y, false);
    cout << x;
    return 0;
}
```

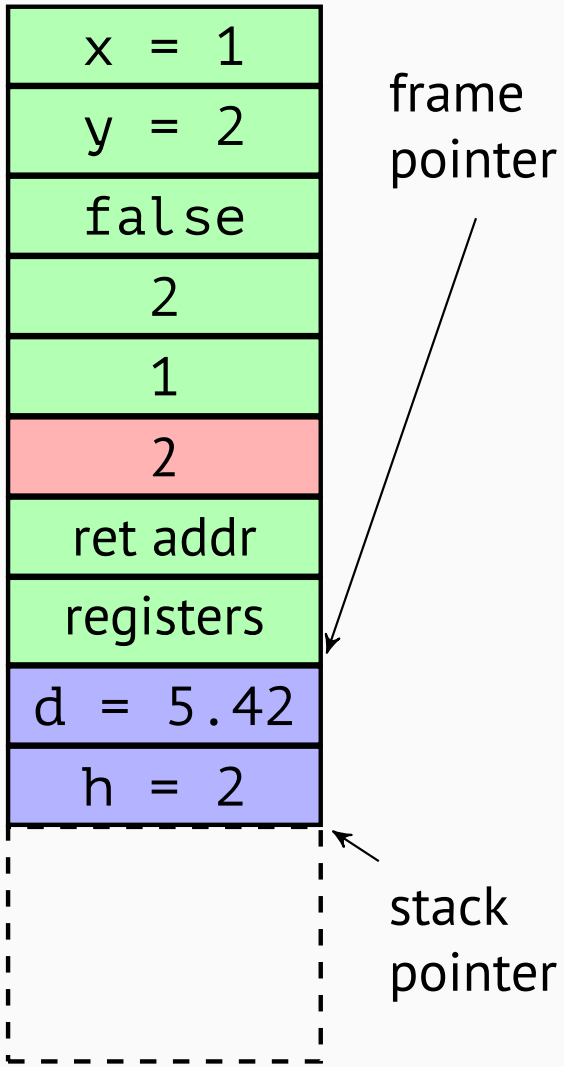
Вызов функции




```
int foo(int a, int b, bool c)
{
    double d = a * b * 2.71;
    int    h = c ? d : d / 2;
    return h;
}

int main( )
{
    int x = 1;
    int y = 2;
    x = foo (x, y, false);
    cout << x;
    return 0;
}
```

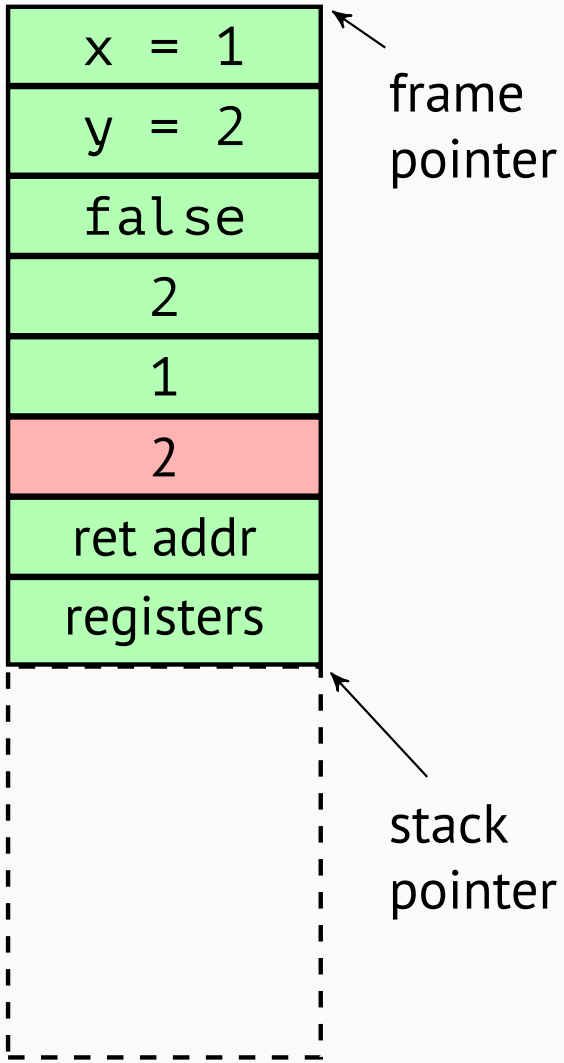
Вызов функции




```
int foo(int a, int b, bool c)
{
    double d = a * b * 2.71;
    int    h = c ? d : d / 2;
    return h;
}

int main( )
{
    int x = 1;
    int y = 2;
    x = foo (x, y, false);
    cout << x;
    return 0;
}
```

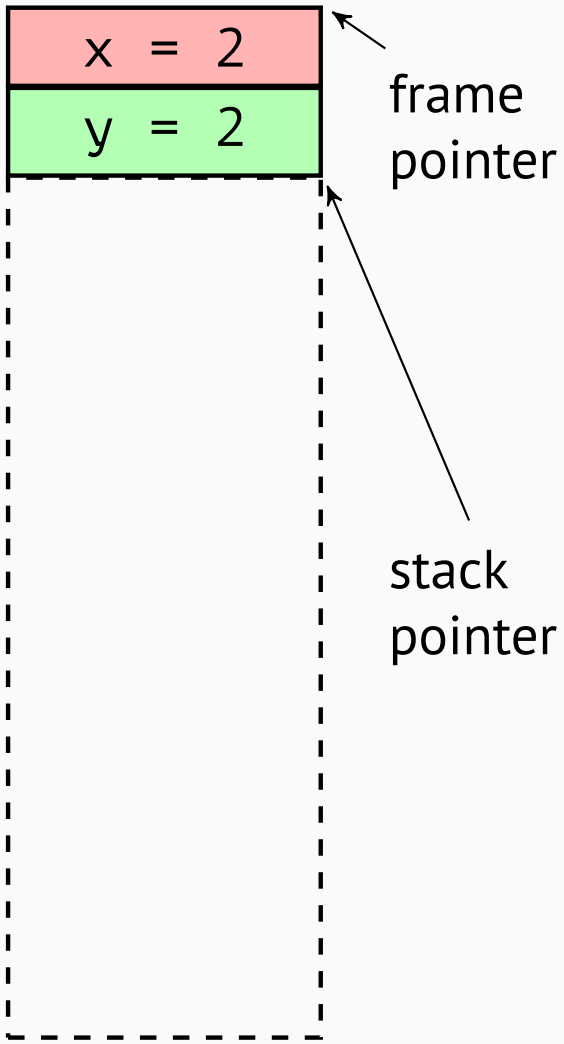
Вызов функции




```
int foo(int a, int b, bool c)
{
    double d = a * b * 2.71;
    int    h = c ? d : d / 2;
    return h;
}

int main( )
{
    int x = 1;
    int y = 2;
    x = foo (x, y, false);
    cout << x;
    return 0;
}
```

Вызов функции




```
int foo(int a, int b, bool c)
{
    double d = a * b * 2.71;
    int    h = c ? d : d / 2;
    return h;
}

int main( )
{
    int x = 1;
    int y = 2;
    x = foo (x, y, false);
    cout << x;
    return 0;
}
```

Вызов функции

- При вызове функции на стек складываются:
  1. аргументы функции,
  2. адрес возврата,
  3. значение frame pointer и регистров процессора.
- Кроме этого на стеке резервируется место под возвращаемое значение.
- Параметры передаются в обратном порядке, что позволяет реализовать функции с переменным числом аргументов.
- Адресация локальных переменных функции и аргументов функции происходит относительно frame pointer.
- Конкретный процесс вызова зависит от используемых соглашений (cdecl, stdcall, fastcall, thiscall).