Программирование

Лекция 3. Указатели.
Использование указателей.
Динамическая память.

Указатели

- Указатель – это переменная, хранящая адрес некоторой ячейки памяти.
- Указатели являются типизированными:

```
int i = 3; // переменная типа int
int * p = 0; // указатель на переменную типа int
```

- Нулевому указателю (которому присвоено значение 0) не соответствует никакая ячейка памяти.
- Существует 2 оператора при работе с указателями:

1) Оператор взятия адреса переменной &
2) Оператор разыменования *.

- p = &i; // указатель p указывает на переменную i (в данном случае в указатель p записывается адрес переменной i)
- *p = 10; // изменяется ячейка по адресу p, т.е. i

Передача параметров по указателю

- Использование указателей позволяет реализовывать функции, которые меняют свои аргументы.
- Допустим, мы хотим написать функцию, которая будет менять значения переменных местами.

```
int main () {
    int k = 10, m = 20;
    swap (k, m);
    cout << k << " " << m << endl; // 10 20
    return 0;
}
void swap (int a, int b) { // функция работает с локальными
копиями                    // переменных
    int t = a;
    a = b;
    b = t;
}
```

Значения k и m не поменялись местами!

Передача параметров по указателю

- Для того, чтобы это исправить, будем передавать не значения типа int, а указатели на эти значения.

```
int main () {
    int k = 10, m = 20;
    swap (&k, &m); // передаем адреса
    cout << k << " " << m << endl; // 20 10
    return 0;
}
void swap (int  * a, int  * b) { // функция работает с
                //адресами переменных
    int t = *a;
    *a = *b;
    *b = t;
}
```

Меняются значения, на которые указывают аргументы функции

Еще раз о массивах

- Массивы – это набор однотипных элементов, расположенных в памяти друг за другом, доступ к которым осуществляется по индексу.

```
// массив 1 2 3 4 5 0 0 0 0 0
int m[10] = {1, 2, 3, 4, 5}; // инициализация массива
```

- Индексация массива начинается с 0, последний элемент массива имеет индекс (n-1)

Массивы часто используются в циклах

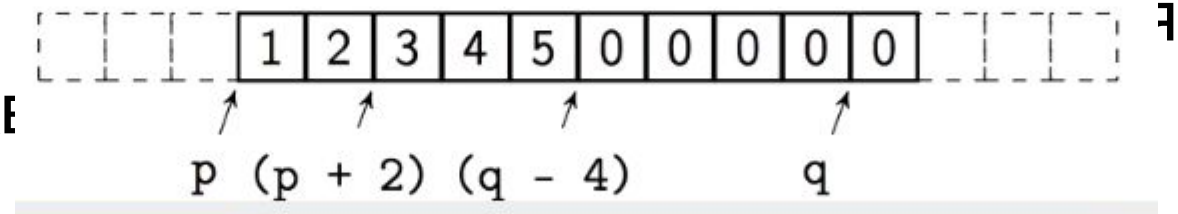
```
for (int i = 0; i < 10; i++)
    cout << m[i] << " ";
```

Связь массивов и указателей

- Массивы тесно связаны с указателями.
- Указатели позволяют передвигаться по массивам.
- Для этого используется арифметика указателей:

```
int m[10] = {1, 2, 3, 4, 5};
int * p = &m[0]; // адрес начала массива
int * q = &m[9]; // адрес последнего элемента массива
```

- (p+k) – сдвиг на k ячеек типа int вправо
- (p-k) – сдвиг на k ячеек типа int влево
- (q-p) – ми
- p[k] эк



Примеры

- Заполнение массива при помощи указателя:

```
int m[10] = {}; // изначально заполнен нулями
//              &m[0]          &m[9]
for (int * p = m ; p <= m + 9; ++p )
    *p = (p - m) + 1;
// Массив заполнен числами от 1 до 10
```

- Передача массива в функцию:

указатель на начало массива

```
int max_element (int * m, int size) {
    int max = *m;
    for (int i = 1; i < size; ++i)
        if (m[i] > max)
            max = m[i];
    return max;
}
```

Работаем с m, как будто это массив. Сначала max – это первый элемент массива

Чтение строк

- Для начала вам нужно подключить библиотеку string. «String» – это строка как последовательность символов, а «line» – последовательность символов, оканчивающаяся переводом строки.
- Решим такую задачу: пользователь вводит свое имя, а программа здоровается с ним.

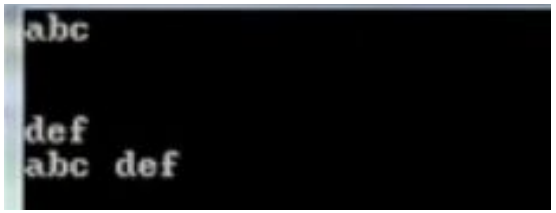
```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string s;
    cin >> s;
    cout << "Hello, " + s;
    return 0;
}
```

Сложение строк

Чтение строк

- При использовании cin чтение будет происходить по словам. Например, если нам понадобится считать два слова, это можно сделать, считав с помощью cin две переменные типа string. Слова могут быть разделены любым количеством пробелов, табуляций и переводом строк, но в переменных окажутся только непробельные символы.

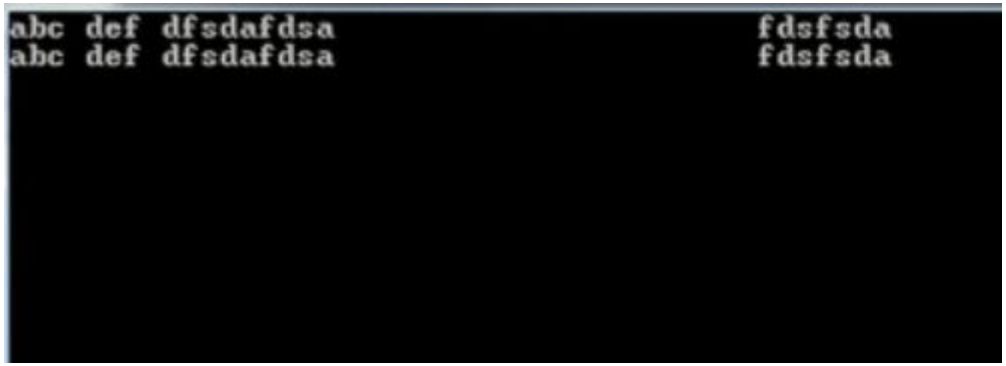
```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string s1, s2;
    cin >> s1 >> s2;
    cout << s1 << " " << s2;
    return 0;
}
```



Чтение строк

- Часто возникает необходимость считать строку (в понимании line) целиком, а не пословно. Для этого есть специальная функция getline(cin, s). Первый параметр в этой функции указывает на поток ввода (cin), а второй – на строку, в которую нужно считывать.

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string s;
    getline(cin, s);
    cout << s;
    return 0;
}
```



Коды символов

- В компьютере всё, в том числе и строки, хранится в виде чисел (строка — набор чисел, которыми кодируются символы). Для хранения одного символа используется тип char (от слова character, символ).
- Можно обращаться к отдельным символам строки, написав после её имени в квадратных скобках номер символа. Нумерация символов в строке начинается с нуля, так же как и в векторах. Узнать длину строки можно с помощью метода size.

```
string s;
char c;
cin >> s;
c = s[0];
cout << c;
```

Когда мы выводим переменную типа char, то выводится символ. Хотя на самом деле char – числовая переменная и обозначает номер символа в кодовой таблице. Вывод кода символа выглядит так:

```
cout << (int) c;
```

Выделение цифр числа

- Задача: получим из html-кода страницы информацию о курсах акций, чтобы заработать на их колебаниях кучу денег. Первым делом нужно выделить из строки только цифры. Мы будем считать, что в строке есть только одно целое число и его и нужно получить. Для решения этой задачи мы будем проходить по всей строке и, если символ – цифра, будем её печатать.

```
string s;
getline(cin, s);
for (auto c : s) {  // тип char
    if (c >= '0' && c <= '9') {
        cout << c;
    }
}
```

В этой программе мы проходим по всем символам строки (так же, по всем элементам вектора). Узнать код конкретного символа – для этого нужно записать этот символ в одинарных кавычках. Если код очередного символа лежит в пределах от 0 до 9, то этот символ – цифра.

Поиск подстроки в строке

- Пусть в скачанном нами файле содержится много строк, но нам интересна только та, где есть название компании, акциями которой мы хотим торговать. Например, это «Рога и копыта» с кодом на бирже rkpt. Дальше наша задача усложняется: среди N строк нужно найти ту, которая содержит подстроку rkpt (то есть где-то внутри строки встречается эта последовательность символов) и вывести число, записанное в этой строке.

```
int n;  // кол-во строк
cin >> n;
string s;
getline(cin, s);
for (int i = 0; i < n; i++) {
    getline(cin, s);
    if (s.find("rkpt") != -1) {
        for (auto c : s) {
            if (c >= '0' && c <= '9') {
                cout << c;
            }
        }
    }
}
```

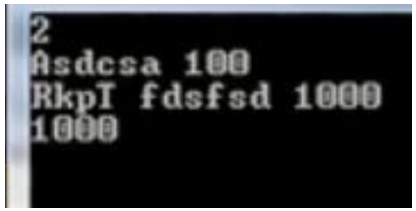
Метод find работает следующим образом: если подстрока нашлась, то она возвращает число, равное номеру символа, с которого началось первое вхождение подстроки в строку. А если подстроки не нашлось, то этот метод возвращает -1.

Обратите внимание на getline перед циклом. Он необходим, потому что после считывания числа в этой строке остается еще и символ перевода строки. Так что когда мы сделаем первый getline, то он считает пустую строку (ведь до перевода строки ничего не осталось).

Изменение регистра символа

```
int n;  // кол-во строк
cin >> n;
string s;
getline(cin, s);

for (int i = 0; i < n; i++) {
   getline(cin, s);
   string s2 = “”;
   for (auto c : s) {  // изменение регистра
      if (c >= ‘a’ && c <= ‘z’) {
        int al_num = c – ‘a’; // номер буквы в алфавите
        s2 += ‘A’ + al_num;
      }
      else
        s2 += c;

}
   if (s2.find(“RKPT") != -1) { // поиск подходящей строки
      for (auto c : s2) {  // s2 – теперь заглавные буквы
         if (c >= '0' && c <= '9') {
            cout << c;
         }
      }
   }
}
```

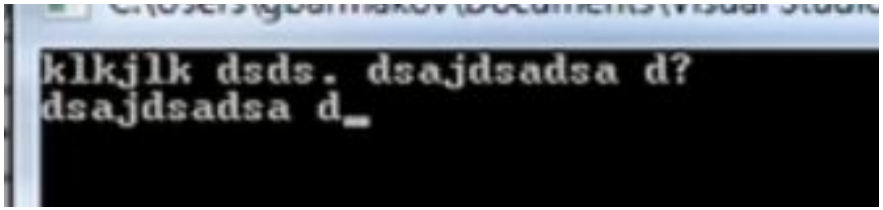
Допустим, название компании может быть написано как большими, так и маленькими буквами или даже вперемешку. А значит нам нужно научиться определять регистр.



Задача

Необходимо вывести символы между первым и вторым знаком препинания.

```cpp
#include <iostream>
#include <string>
using namespace std;
int main() {
    string s;
    getline(cin, s);
    int first = s.find_first_of("?!.");
    int second = s.find_first_of("?!.", first + 1);
    cout << s.substr(first + 2, second - first - 2);
    return 0;
}
```



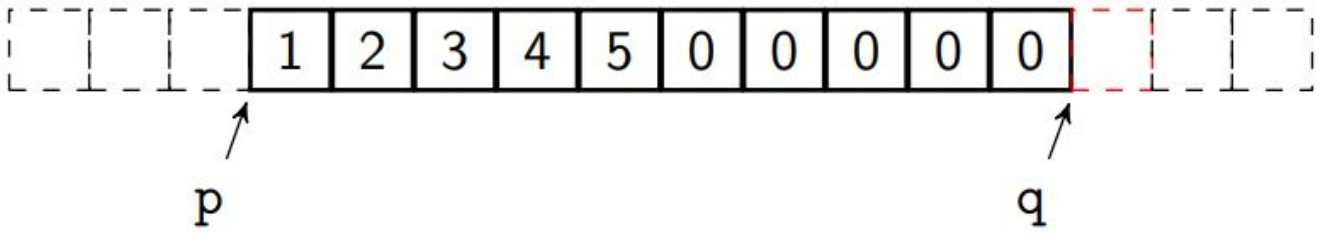
Два способа передачи массива

Функция для поиска элемента в массиве:

```
bool contains(int * m, int size, int value) {
    for (int i = 0; i != size; ++i)
        if (m[i] == value)
            return true;
    return false;
}
bool contains(int * p, int * q, int value) {
    for (; p != q; ++p)
        if (*p == value)
            return true;
    return false;
}
```

*p - указатель на начало массива

Второй способ быстрее!
Так как m[i] это *(m+i)



Возврат указателя из функции

Функция для поиска максимума в массиве:

```
int max_element (int * p, int * q) {
    int max = *p;    // первый элемент
    for (; p != q; ++p)    массива
        if (*p > max)
            max = *p;

    return max;    // возвращаем само значение макс.
}                  элемента
```

```
    int m[10] = {...};
    int max = max_element(m, m + 10);
    cout << "Maximum = " << max << endl;
```

начало массива

Возврат указателя из функции

Но можно вернуть также и место в массиве, на котором находится макс. элемент. То есть указатель на макс. элемент. Эту информацию в дальнейшем можно использовать, например, чтобы переставить макс. элемент в начало.

возвращается указатель!

```
int * max_element (int * p, int * q) {
    int * pmax = p;
    for (; p != q; ++p)
        if (*p > *pmax)
            pmax = p; // меняем указатель на макс. элемент
    return pmax;  // получаем адрес макс. элемента и его значение
}
```

```
    int m[10] = {...};
    int * pmax = max_element(m, m + 10);
    cout << "Maximum = " << *pmax << endl;
```

значение макс. элемента

Возврат значения через указатель

Если функции передали пустой массив, то функция будет сигнализировать об этом.

```
bool max_element (int * p, int * q, int * res) {
    if (p == q)
        return false;
    *res = *p;
    for (; p != q; ++p)
        if (*p > *res)
            *res = *p;
    return true;
}
```

новый параметр – указатель на результат

```
    int m[10] = {...};
    int max = 0;
    if (max_element(m, m + 10, &max))
        cout << "Maximum = " << max << endl;
```

Недостатки указателей

- Использование указателей синтаксически загрязняет код и усложняет его понимание. (Приходится использовать операторы * и &.)
- Указатели могут быть неинициализированными (некорректный код).
- Указатель может быть нулевым (корректный код), а значит указатель нужно проверять на равенство нулю.
- Арифметика указателей может сделать из корректного указателя некорректный (легко промахнуться).

оператор разыменования и взятия адреса

указатели на указатели

если объявлен указатель, но не проинициализирован, то там хранится какой-то адрес

например, выйти за границы массива

обращение к неинициализированному указателю – ошибка.
обращение к нулевому указателю – это ошибка.

Ссылки

Для того, чтобы исправить некоторые недостатки указателей, в C++ введены ссылки.

Ссылки являются "красивой обёрткой" над указателями:

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

int main() {
    int k = 10, m = 20;
    swap (k, m);
    cout << k << ' ' << m << endl; // 20 10
    return 0;
}
```

это ссылки

На самом деле в функции используются не локальные переменные, а ссылки на эти переменные

Внутри ссылок «зашиты» указатели, но синтаксически код выглядит чище, не нужно использовать оператор *.

Различия ссылок и указателей

- Ссылка не может быть неинициализированной.

```
int * p; // OK
int & l; // ошибка
```

для ссылок всегда нужно указывать инициализирующее значение

- У ссылки нет нулевого значения.

```
int * p = 0; // OK
int & l = 0; // ошибка
```

не нужна проверка на 0

- Ссылку нельзя переинициализировать.

```
int a = 10, b = 20;
int * p = &a; // p указывает на a
p = &b;       // p указывает на b
int & l = a;  // l ссылается на a
l = b;        // a присваивается значение b
```

Различия ссылок и указателей

- Нельзя получить адрес ссылки или ссылку на ссылку.

```
int  a = 10;
int  * p = &a;   // p указывает на a
int  ** pp = &p;// pp указывает на переменную p
int  & l = a;    // l ссылается на a
int  * pl = &l; // pl указывает на переменную a
int  && ll = l; // ошибка
```

- Нельзя создавать массивы ссылок.

```
int  * mp[10] = {};   // массив указателей на int
int  & ml[10] = {};   // ошибка
```

- Для ссылок нет арифметики.

Ссылки представляют идею «синонимов»

lvalue и rvalue

- Выражения в C++ можно разделить на два типа:
  1. **lvalue** — выражения, значения которых являются *ссылкой* на переменную/элемент массива, а значит могут быть указаны слева от оператора =.
  2. **rvalue** — выражения, значения которых являются временными и не соответствуют никакой переменной/элементу массива.
- Указатели и ссылки могут указывать только на lvalue.

```
int a = 10, b = 20;
int m[10] = {1,2,3,4,5,5,4,3,2,1};
int & l1 = a;              // OK
int & l2 = a + b;          // ошибка
int & l3 = *(m + a / 2);   // OK
int & l4 = *(m + a / 2) + 1;   // ошибка
int & l5 = (a + b > 10) ? a : b;   // OK
```

= m[a/2]=5

тернарный оператор

Время жизни переменной

Может так случиться, что указатель или ссылка в программе указывает на переменную, которая уже не существует

Следует следить за временем жизни переменных.

```
int * foo() {
    int a = 10;
    return &a;
}

int & bar() {
    int b = 20;
    return b;
}
```

foo возвращает указатель на переменную. Но a – это локальная переменная. При выходе из функции переменная a перестанет существовать.

bar возвращает ссылку на локальную переменную.

```
    int * p = foo();
    int & l = bar();
```

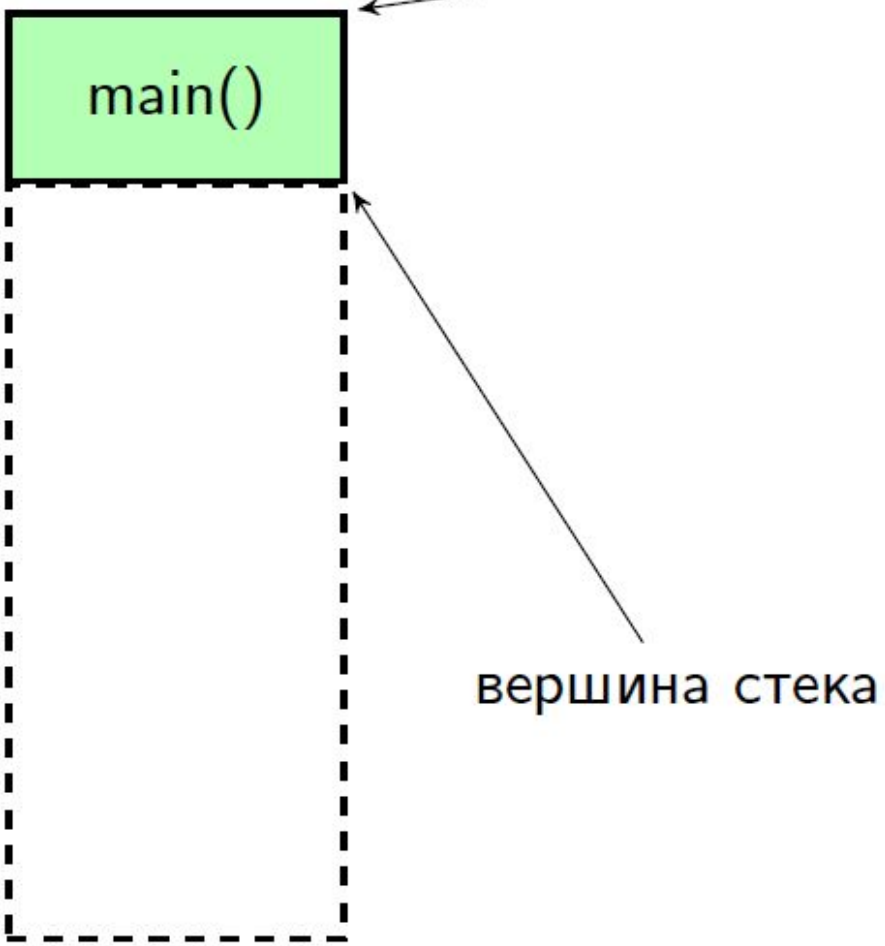
p указывает на переменную, которая не существует

дальнейшее обращение к ней будет некорректно

Стек вызовов

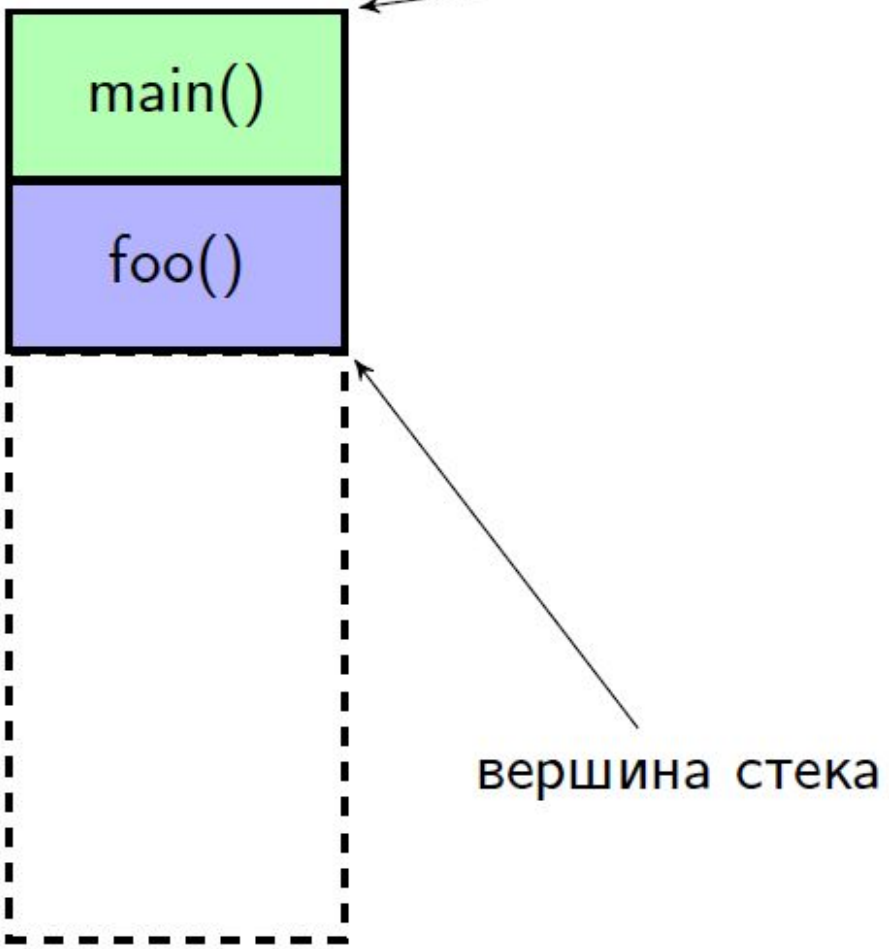
- Стек вызовов — это сегмент данных, используемый для хранения локальных переменных и временных значений.
- Не стоит путать стек с одноимённой структурой данных, у стека в C++ можно обратиться к произвольной ячейке.
- Стек выделяется при запуске программы.
- Стек обычно небольшой по размеру (4Мб).
- 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



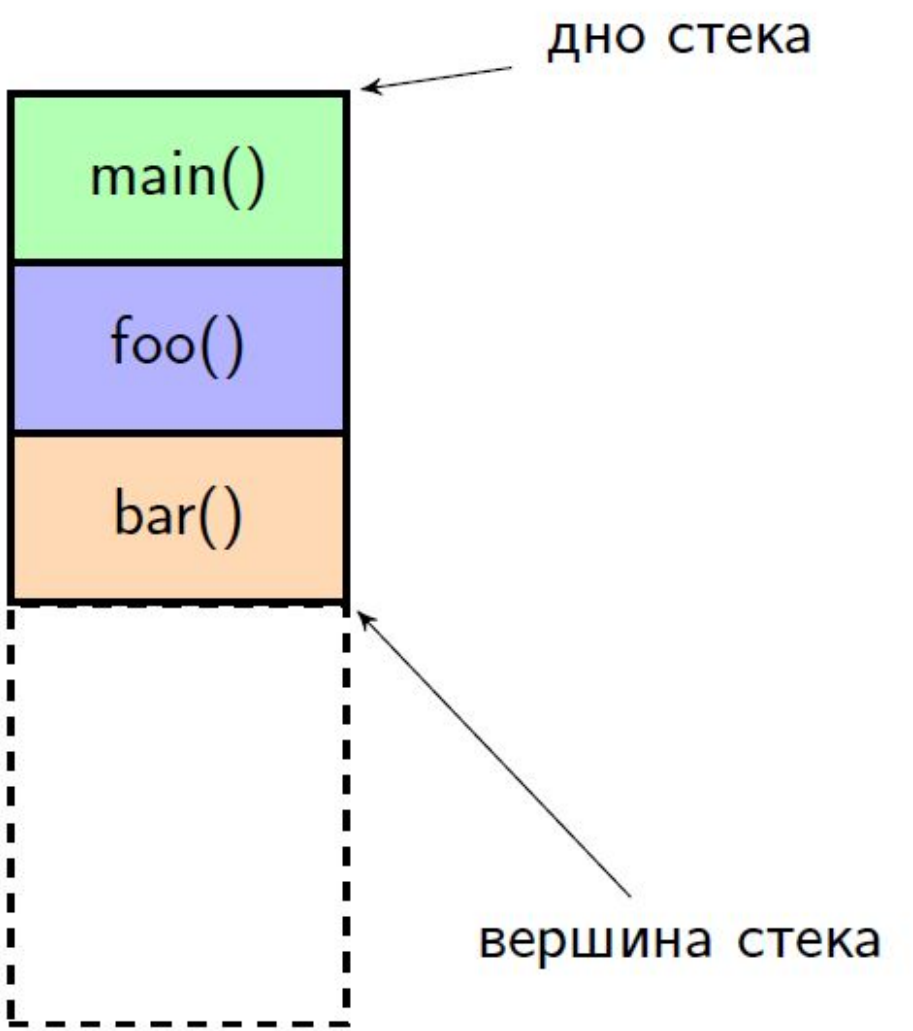

```
void bar( ) {
    int c;
}

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

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

    return 0;
}
```

Устройство стека



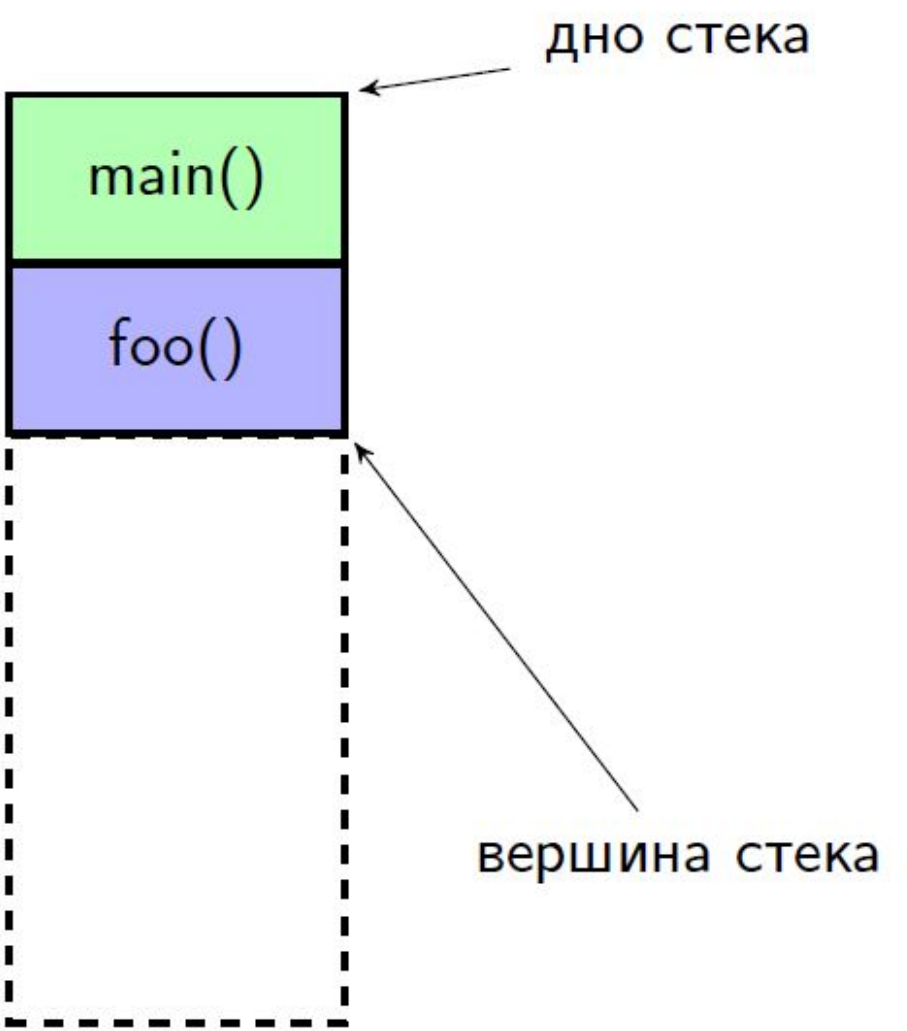

```
void bar( ) {
    int c;
}

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

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

    return 0;
}
```

Устройство стека



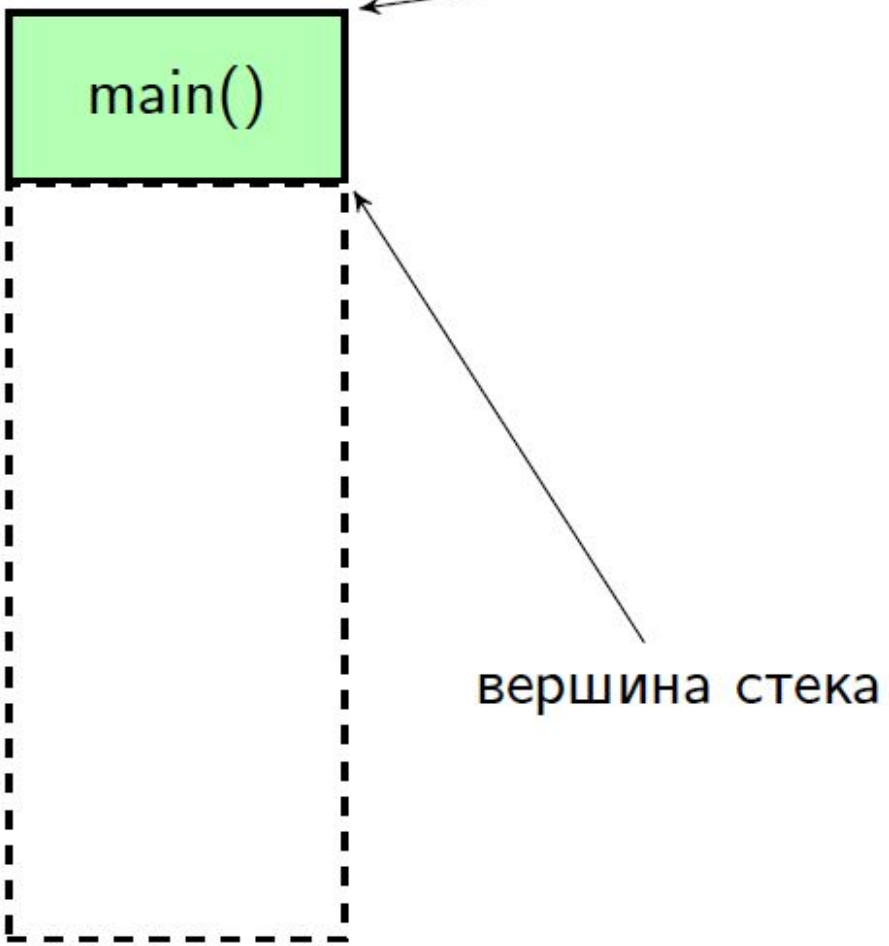

```
void bar( ) {
    int c;
}

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

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

    return 0;
}
```

Устройство стека



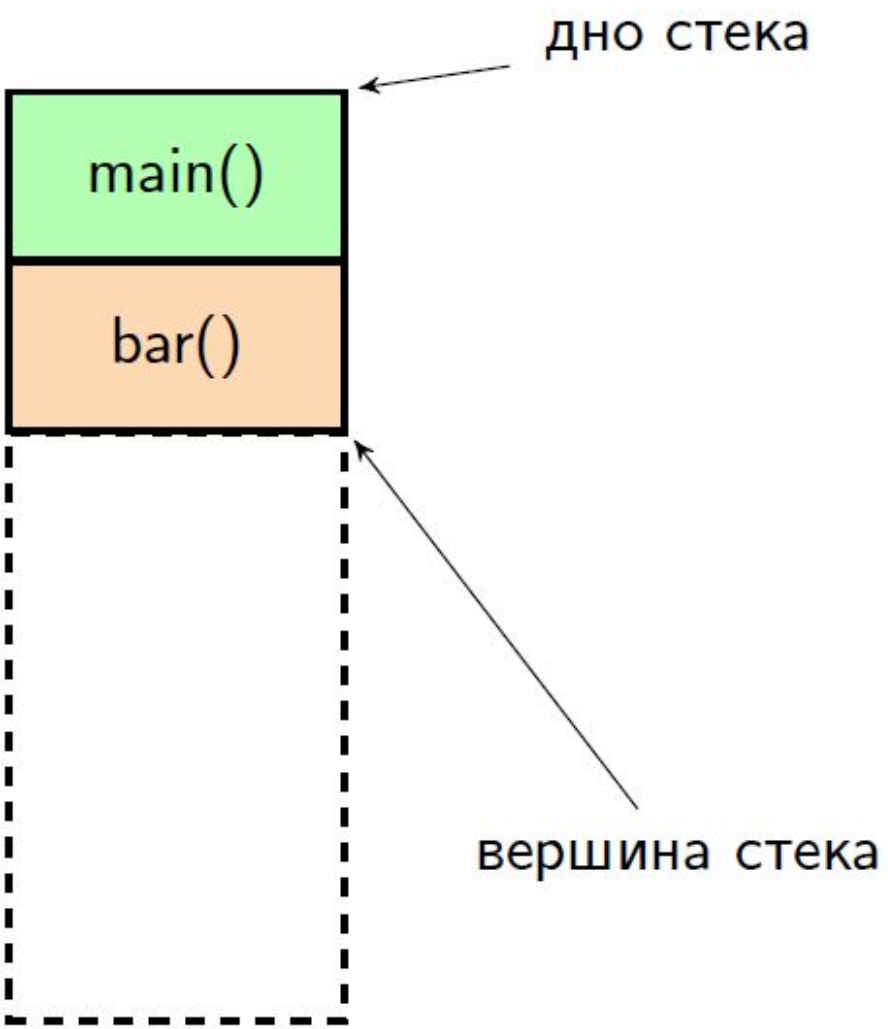

```
void bar( ) {
    int c;
}

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

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

    return 0;
}
```

Устройство стека



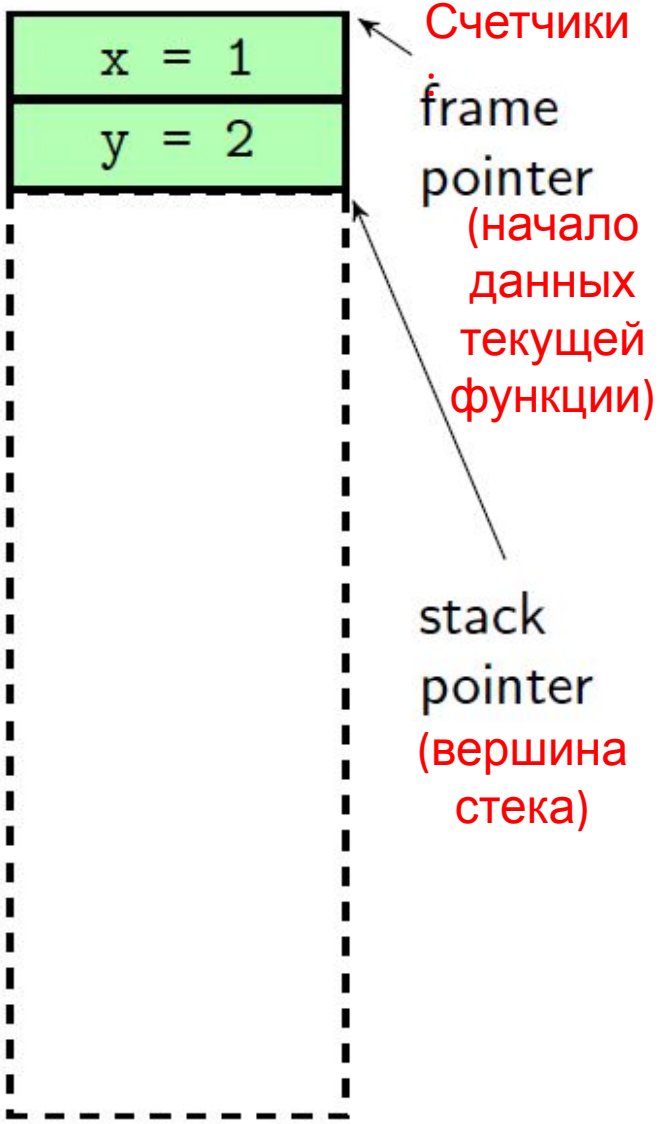

```
void bar( ) {
    int c;
}

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

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

    return 0;
}
```

Устройство стека



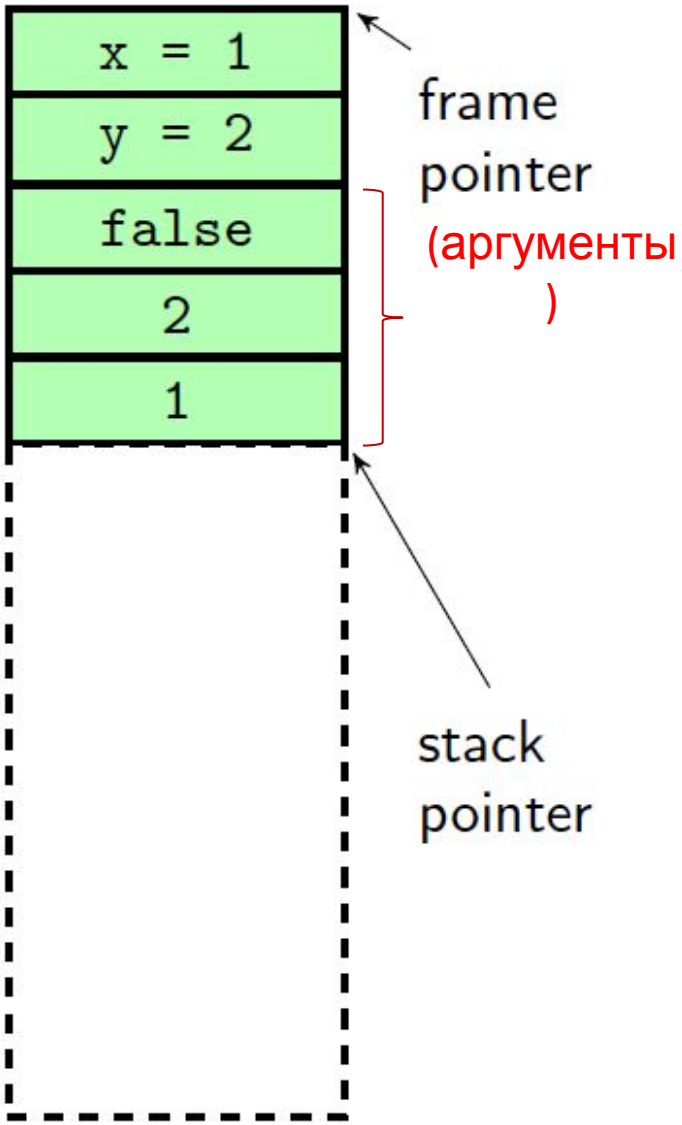

```
void bar( ) {
    int c;
}

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

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

    return 0;
}
```

Вызов функции



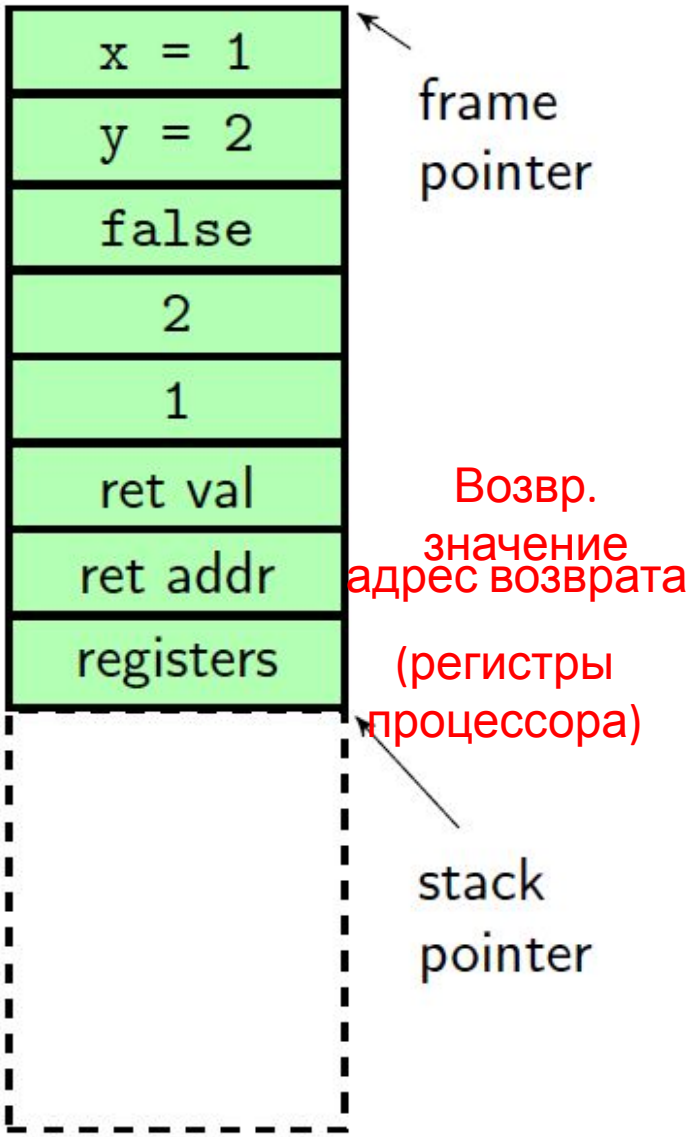

```
int foo(int a, int b, bool c)
{
    double d = a * b * 2.71;
    int    h = c ? d : d / 2;
    return h;
}

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;
}
```

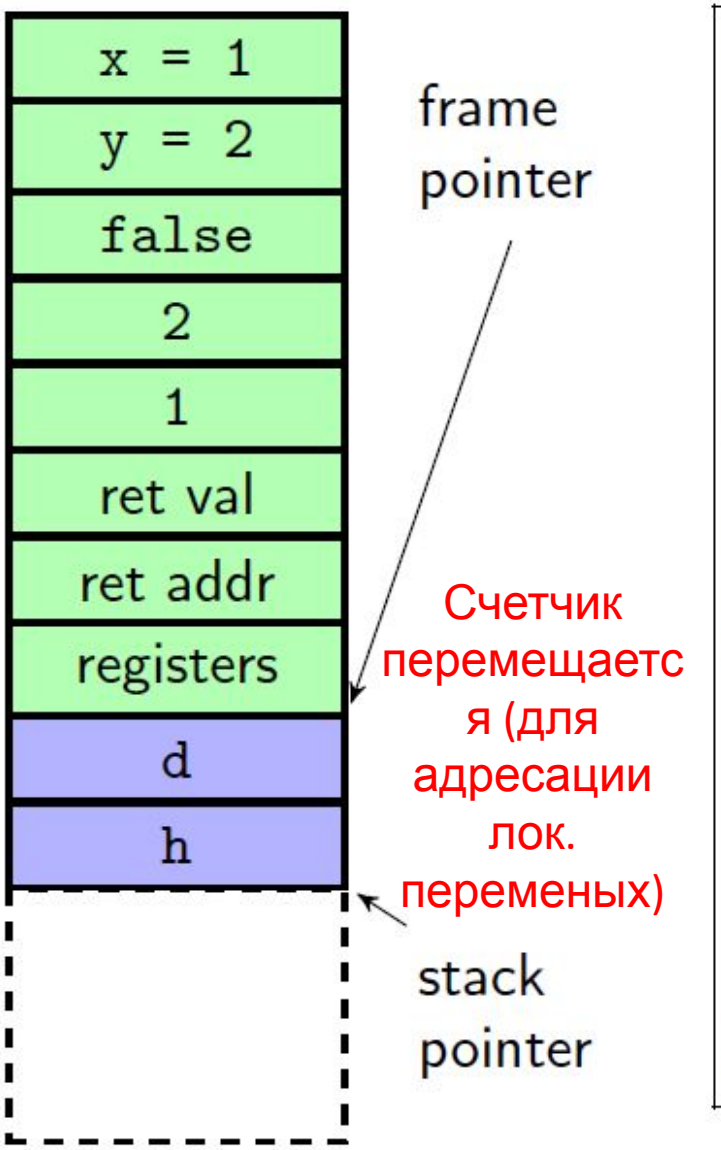
Вызов функции




```
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;
}
```

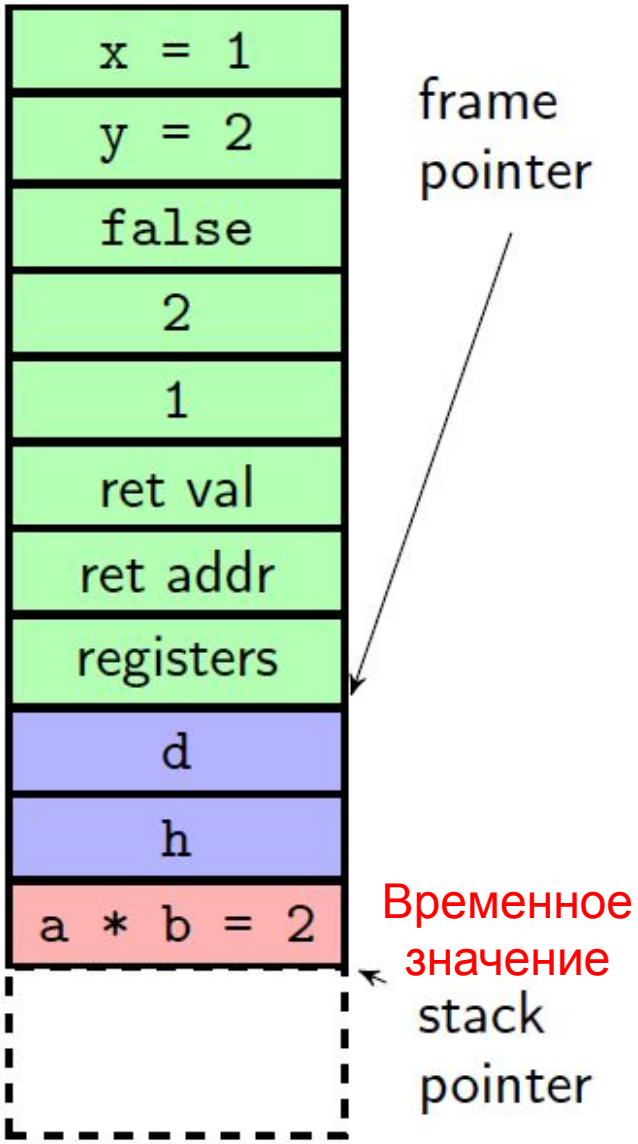
Вызов функции

Управление передается функции foo()



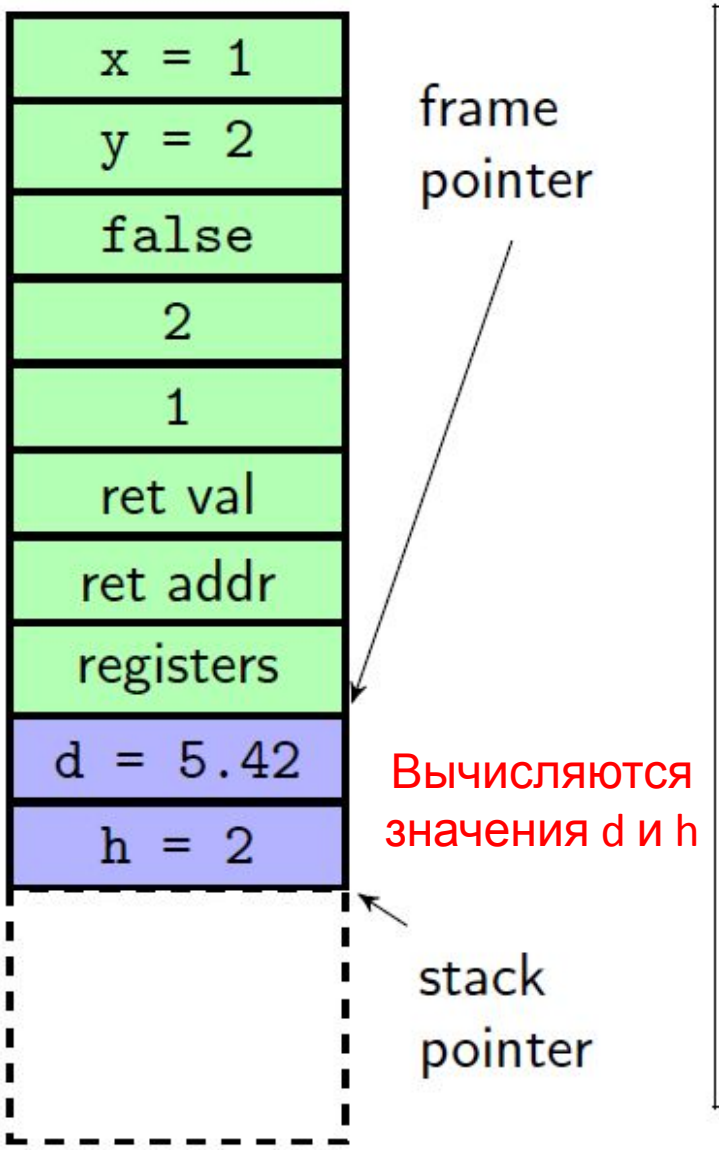

```
int foo(int a, int b, bool c)
{
    double d = a * b * 2.71;
    int    h = c ? d : d / 2;
    return h;
}

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

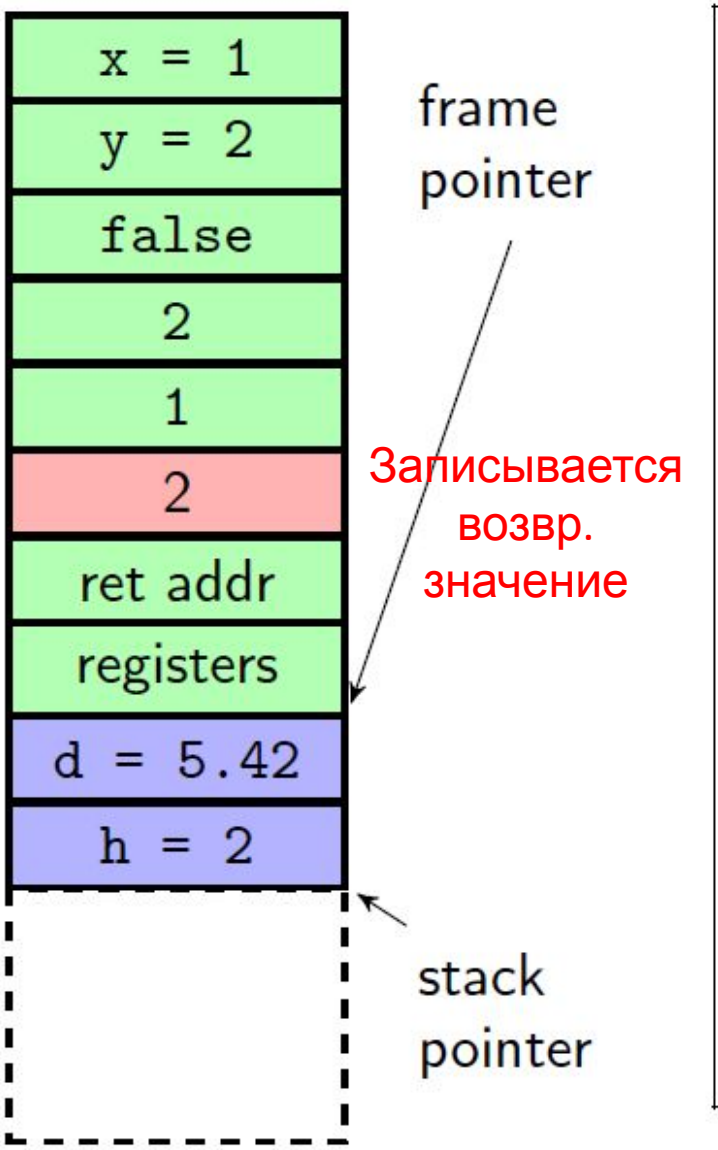


Вычисляются значения d и h

```
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;
}
```

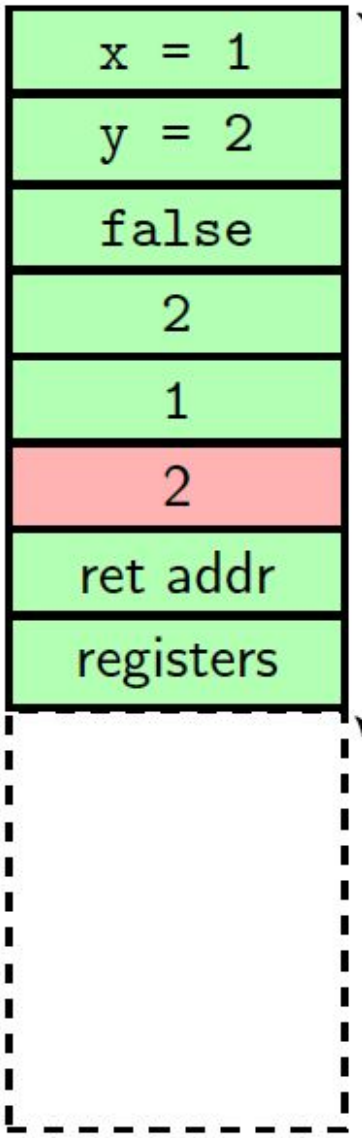
Условное выражение вида "условие" ? "выражение 1" : "выражение 2"

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;
}
```

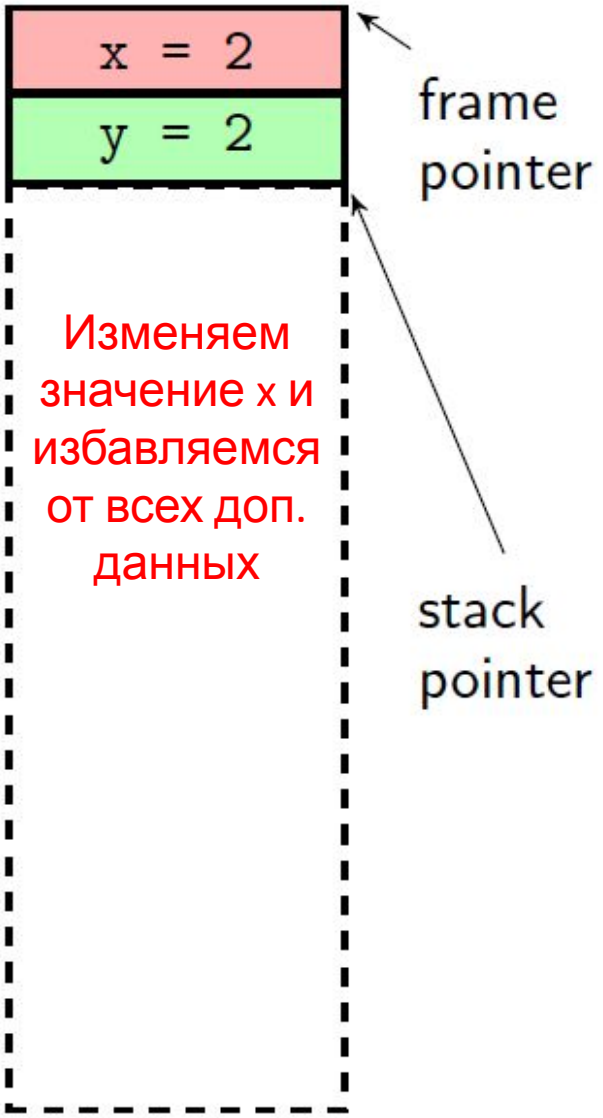
Вызов функции



Функция foo () закончила свое выполнение, лок. переменные удалены

```
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.
- Адресация локальных переменных функции и аргументов функции происходит относительно frame pointer.
- Конкретный процесс вызова зависит от используемых соглашений (cdecl, stdcall, fastcall, thiscall).

Зависит от компилятора

Динамическая память

- Это способ выделения дополнительных областей памяти для хранения данных.

Зачем нужна динамическая память?

- Стек программы ограничен. Он не предназначен для хранения больших объемов данных.

```
// Не умещается на стек
double m[10000000] = {}; // 80 Mb
```

- Время жизни локальных переменных ограничено временем работы функции.

Массив будет уничтожен при выходе из функции

- Выделение и освобождение памяти *управляется вручную*.

Выделение памяти в стиле C (самостоятельно)

- Стандартная библиотека `cstdlib` предоставляет четыре функции для управления памятью:

```
void * malloc (size_t size);
void   free   (void * ptr);
void * calloc (size_t nmemb, size_t size);
void * realloc(void * ptr, size_t size);
```

- `size_t` — специальный целочисленный беззнаковый тип, может вместить в себя размер любого типа в байтах.
- Тип `size_t` используется для указания размеров типов данных, для индексации массивов и пр.
- `void *` — это указатель на нетипизированную память (раньше для этого использовалось `char *`).

Выделение памяти в стиле C (самостоятельно)

- Функции для управления памятью в стиле C:

```
void * malloc (size_t size);
void * calloc (size_t nmemb, size_t size);
void * realloc(void * ptr, size_t size);
void   free   (void * ptr);
```

- `malloc` — выделяет область памяти размера $\geq$ `size`. Данные не инициализируются.
- `calloc` — выделяет массив из `nmemb` размера `size`. Данные инициализируются нулём.
- `realloc` — изменяет размер области памяти по указателю `ptr` на `size` (если возможно, то это делается на месте).
- `free` — освобождает область памяти, ранее выделенную одной из функций `malloc`/`calloc`/`realloc`.

Выделение памяти в стиле C (самостоятельно)

- Для указания размера типа используется оператор `sizeof`.

```
// создание массива из 1000 int
int * m = (int *)malloc(1000 * sizeof(int));
m[10] = 10;

// изменение размера массива до 2000
m = (int *)realloc(m, 2000 * sizeof(int));

// освобождение массива
free(m);

// создание массива нулей
m = (int *)calloc(3000, sizeof(int));

free(m);
m = 0;
```

Чтобы указатель m не указывал на какую-то область памяти

Выделение памяти в стиле C++

- Язык C++ предоставляет два набора операторов для выделения памяти:
  1. `new` и `delete` — для одиночных значений,
  2. `new []` и `delete []` — для массивов.
- Версия оператора `delete` должна соответствовать версии оператора `new`.

```
// выделение памяти под один int со значением 5
int * m = new int(5);
delete m; // освобождение памяти

// создание массива значений типа int
m = new int[1000];
delete [] m; // освобождение памяти
```

Типичные проблемы при работе с динамической памятью

- Проблемы производительности: создание переменной на стеке намного "дешевле" выделения для неё динамической памяти.
- Проблема фрагментации: выделение большого количества небольших сегментов способствует фрагментации памяти.
- Утечки памяти:

Память занята неравномерно

```
// создание массива из 1000 int
int * m = new int[1000];

// создание массива из 2000 int
m = new int[2000]; // утечка памяти

// Не вызван delete [] m, утечка памяти
```

Типичные проблемы при работе с динамической памятью

Неправильное освобождение памяти.

```
int * m1 = new int[1000];
delete m1; // должно быть delete [] m1

int * p = new int(0);
free(p); // совмещение функций C++ и C

int * q1 = (int *)malloc(sizeof(int));
free(q1);
free(q1); // двойное удаление

int * q2 = (int *)malloc(sizeof(int));
free(q2);
q2 = 0;    // обнуляем указатель
free(q2); // правильно работает для q2 = 0
```

Программа отработает нормально

Вопросы

1) В чем разница между статической памятью и динамической?
2) Назначение указателя.
3) Что означает символ & перед переменной.
4) Что такое двойной указатель
5) Приведите пример инициализации двухмерного массива.

Задание

```
#include
using namespace std;
int main()
{
int a = 5;
int *p = &a; //объявляем указатель.
cout << a << ' ' << *p << endl; //выведет ?
a = 6;
cout << a << ' ' << *p << endl; //выведет ?
*p = 7;
cout << a << ' ' << *p << endl; //выведет ?

int b = 8;
int *p = &b;
cout << b << ' ' << *p << endl; //выведет ?
a = 9;
cout << b << ' ' << *p << endl; //выведет ?
*p = 10;
cout << b << ' ' << *p << endl; //выведет ?
return 0;
}
```

Решение

```
#include
using namespace std;
int main()
{
int a = 5; //объявляем переменную №1
int *p = &a; /объявляем указатель.
cout << a << ' ' << *p << endl; //выведет 5 5
a = 6;
cout << a << ' ' << *p << endl; //выведет 6 6
*p = 7;
cout << a << ' ' << *p << endl; //выведет 7 7

int b = 8; //объявляем переменную №2
int *p = &b; /объявляем указатель. внимание - указатель остался тотже, но мы меняем адрес
на который он указывает
cout << b << ' ' << *p << endl; //выведет 8 8
a = 9;
cout << b << ' ' << *p << endl; //выведет 9 9
*p = 10;
cout << b << ' ' << *p << endl; //выведет 10 10
return 0;
}
```