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))
```

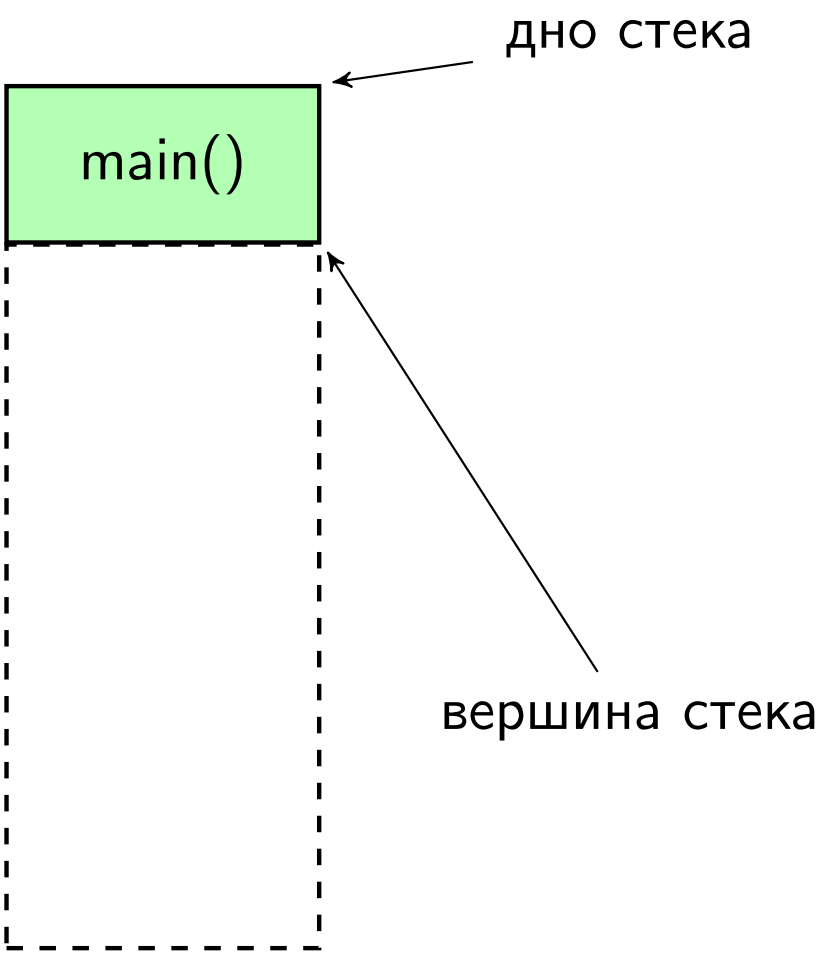
- Определять функции через макросы — плохая идея.
- Макросы можно использовать для условной компиляции:

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

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



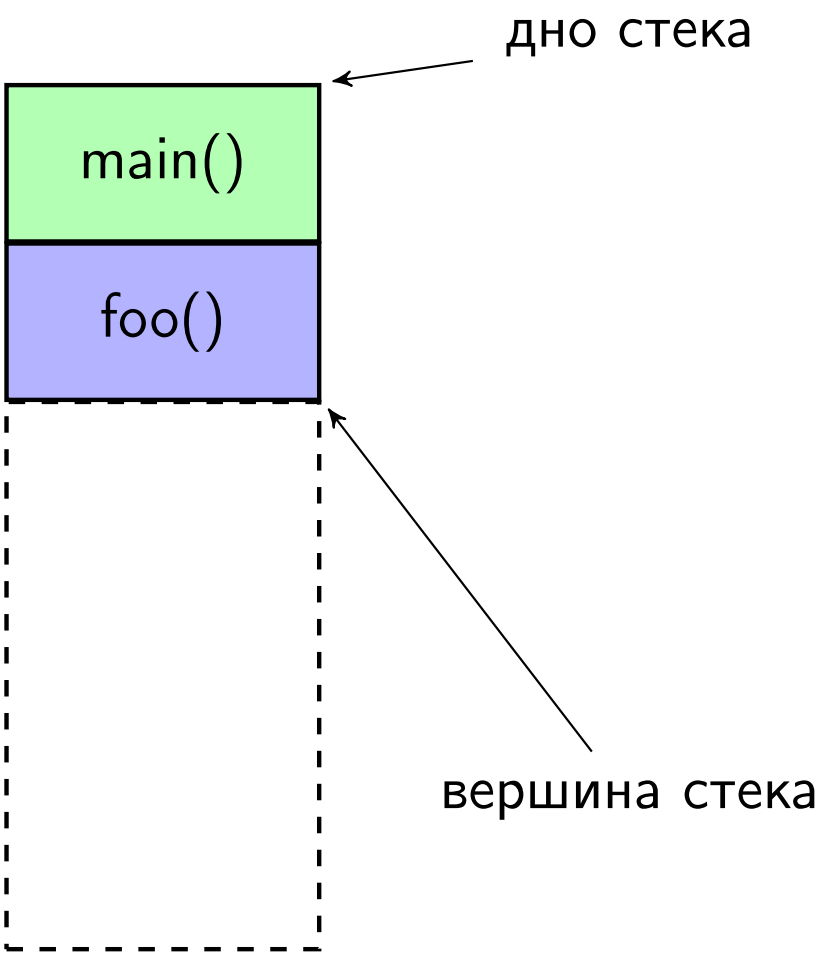

```
void bar( ) {
     int c;
}

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

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

     return 0;
}
```

Устройство стека



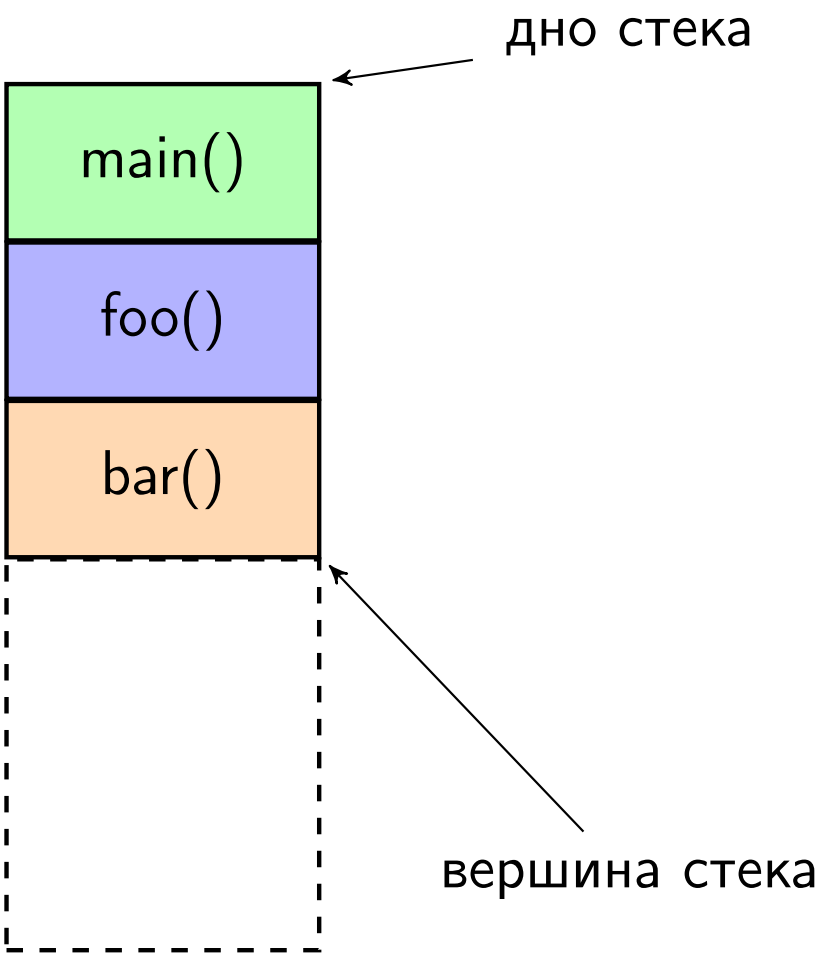

```
void bar( ) {
     int c;
}

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

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

     return 0;
}
```

Устройство стека



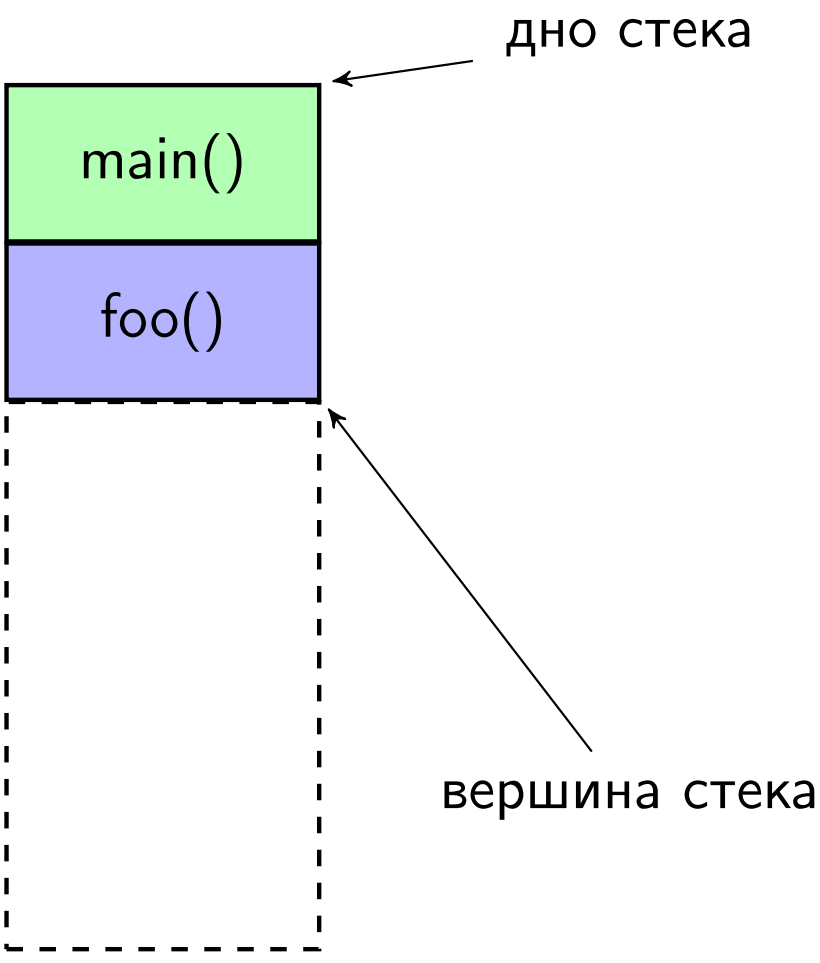

```
void bar( ) {
     int c;
}

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

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

     return 0;
}
```

Устройство стека



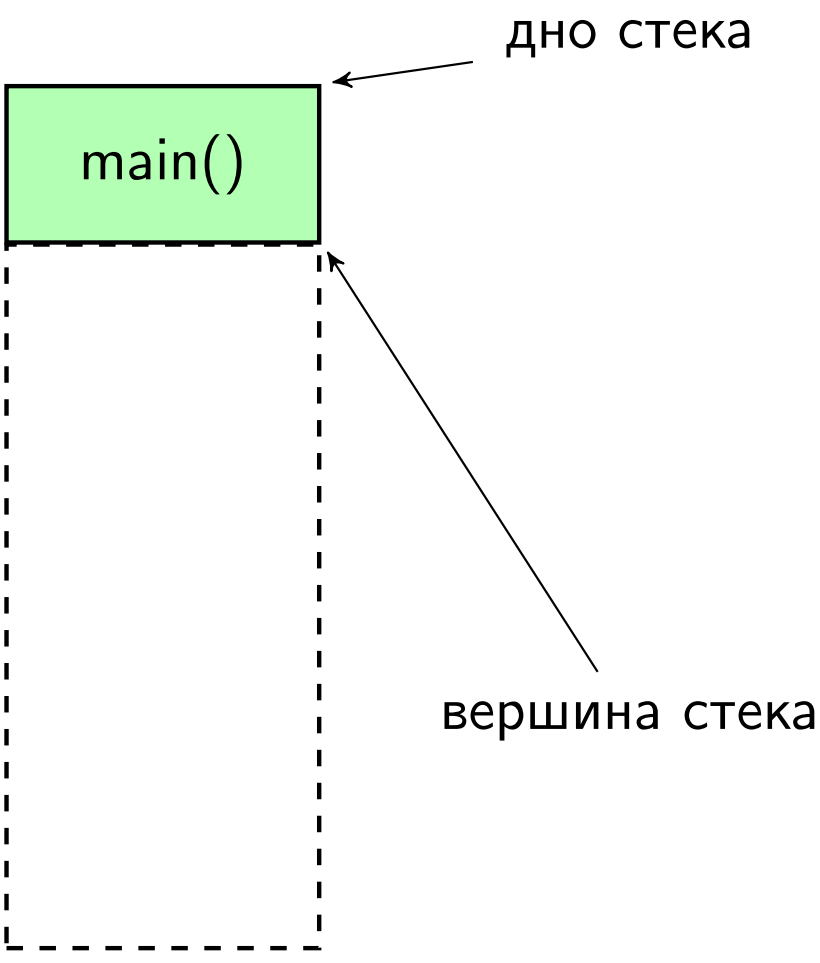

```
void bar( ) {
     int c;
}

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

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

     return 0;
}
```

Устройство стека



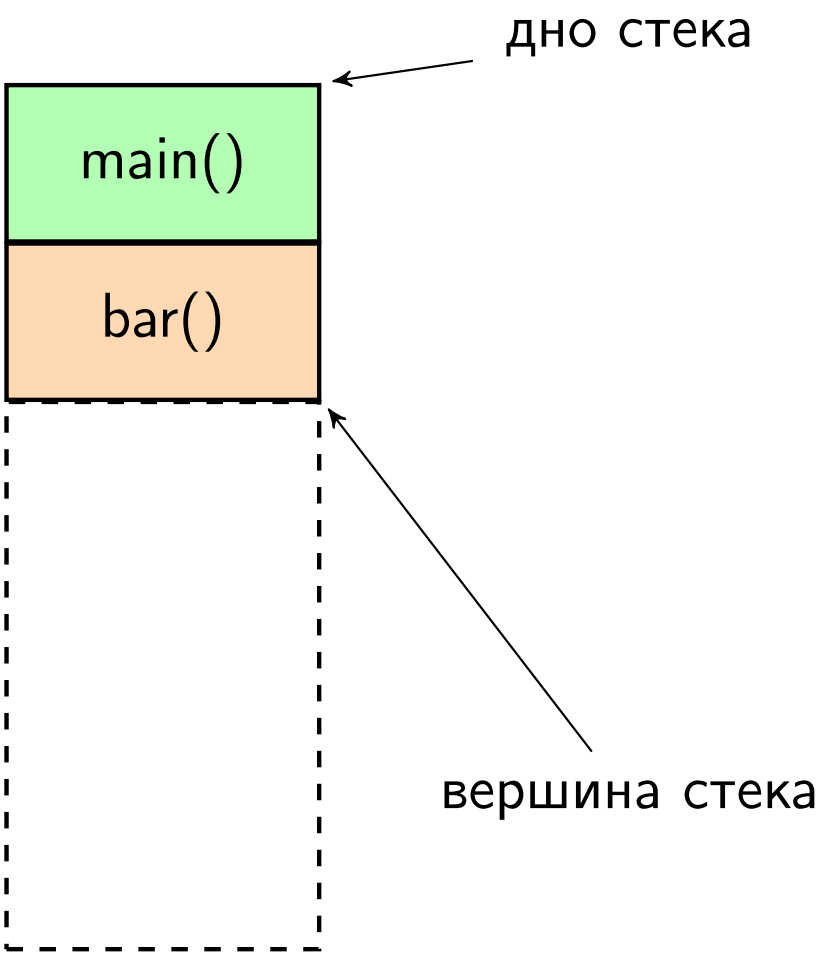

```
void bar( ) {
     int c;
}

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

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

     return 0;
}
```

Устройство стека



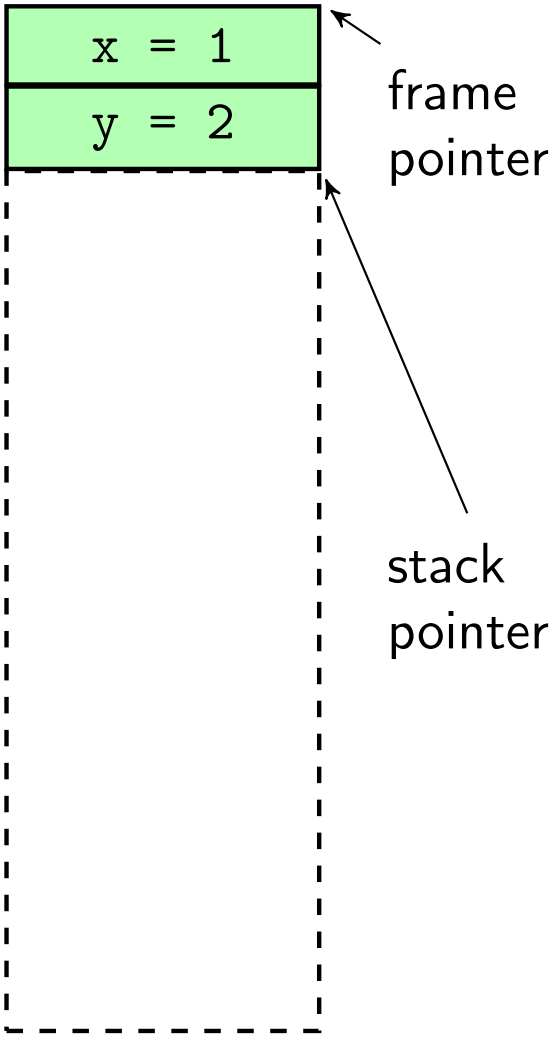

```
void bar( ) {
     int c;
}

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

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

     return 0;
}
```

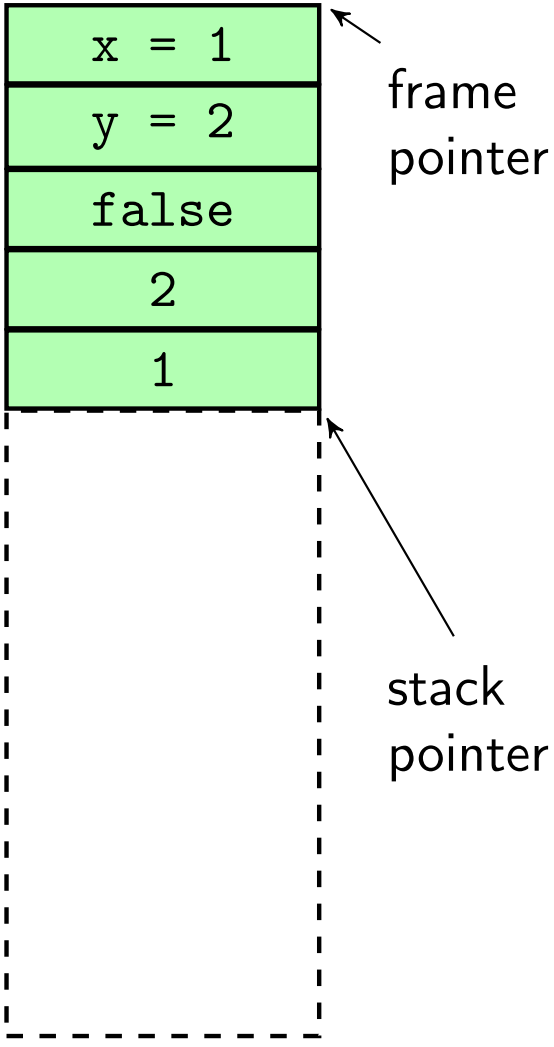
Вызов функции




```
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;
}
```

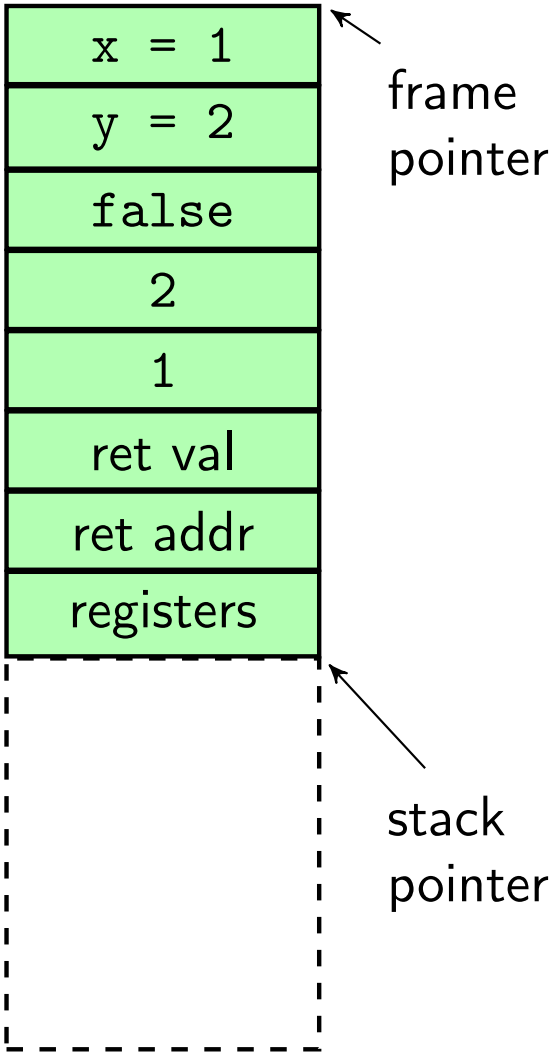
Вызов функции




```
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;
}
```

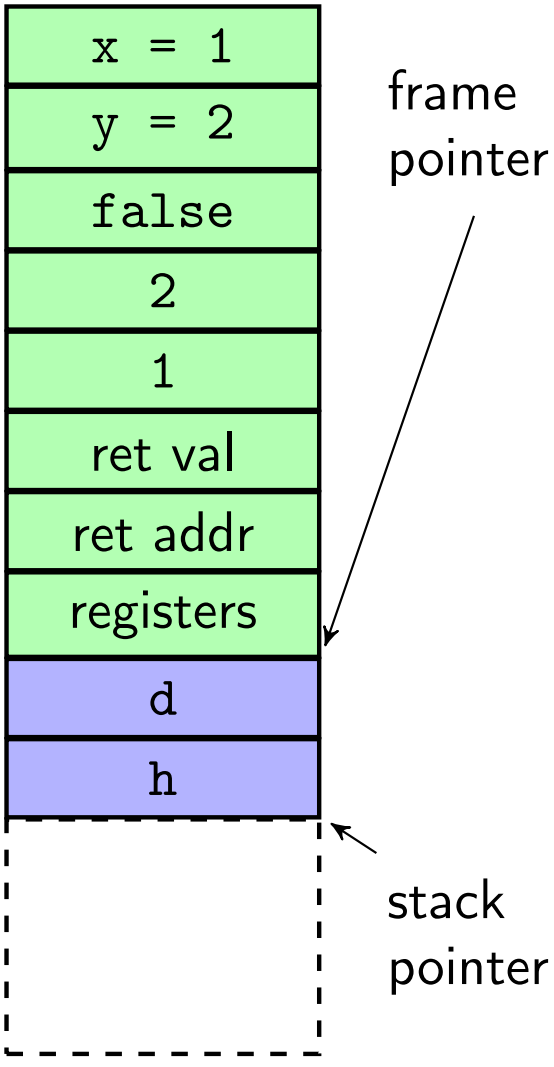
Вызов функции




```
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;
}
```

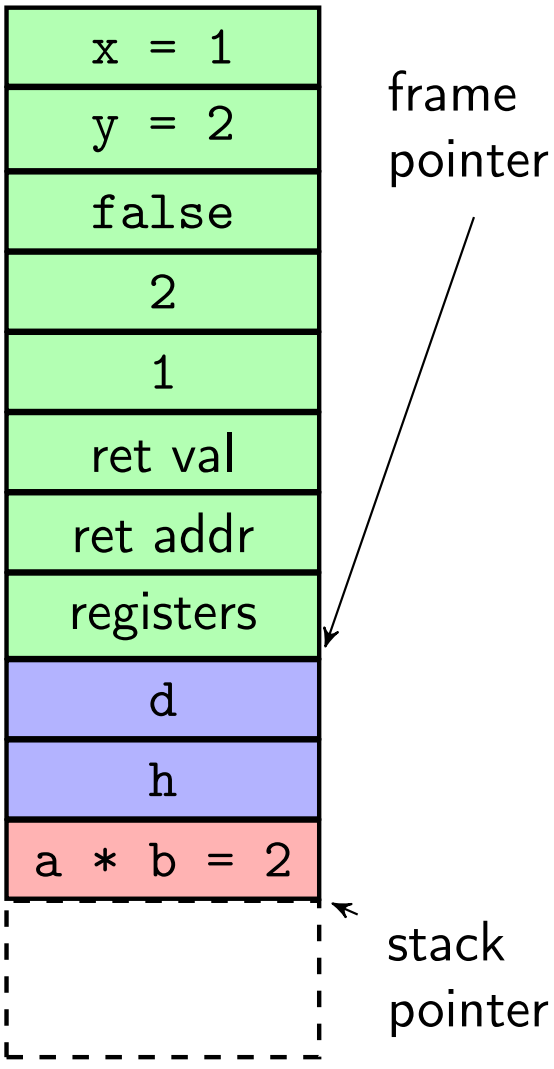
Вызов функции




```
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;
}
```

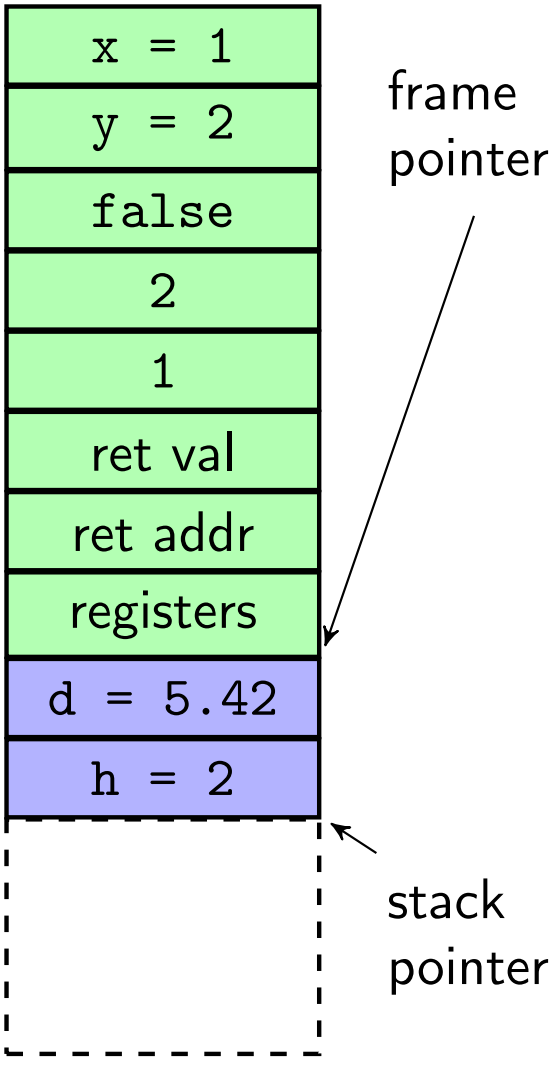
Вызов функции




```
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;
}
```

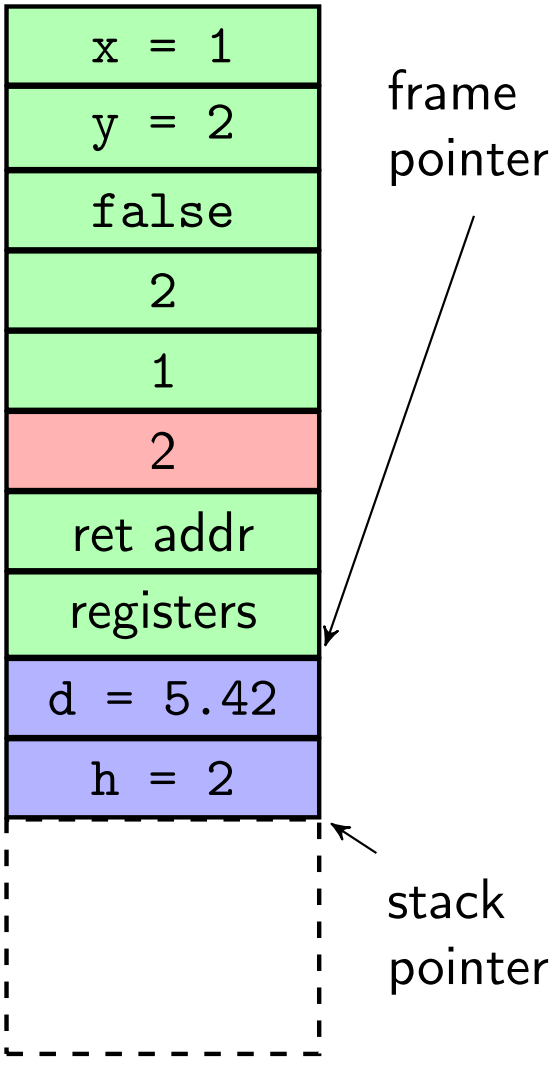
Вызов функции




```
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;
}
```

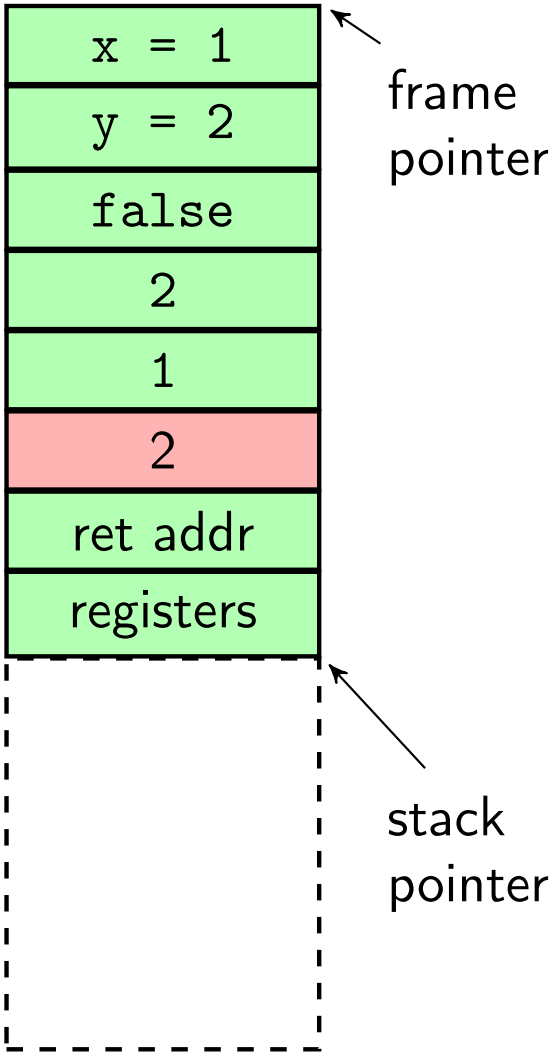
Вызов функции




```
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;
}
```

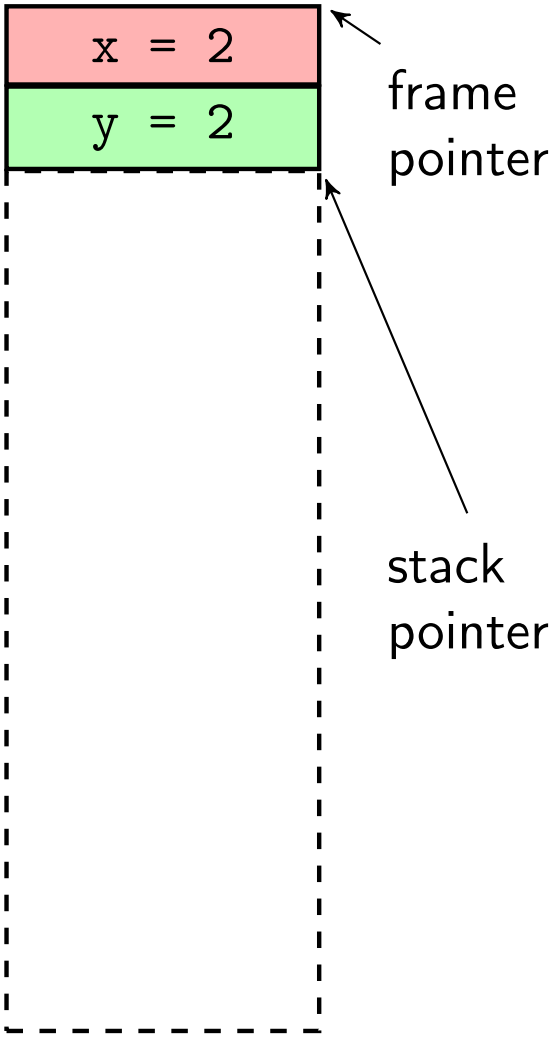
Вызов функции




```
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;
}
```

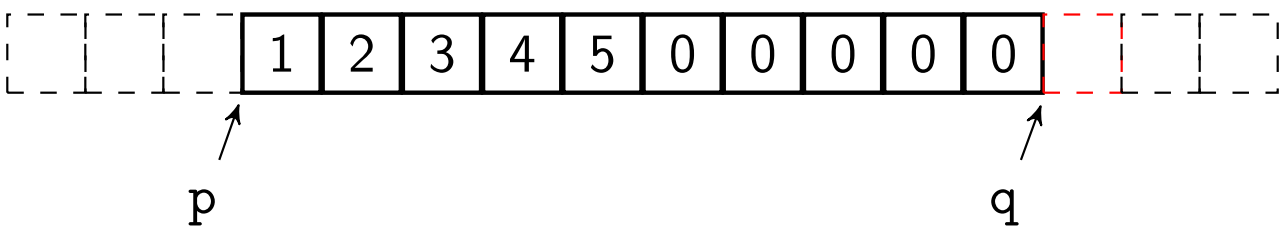
Вызов функции

- 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).

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;
}
```



Возрат указателя из функции

Функция для поиска максимума в массиве:

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

    return max;
}
```

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

Возрат указателя из функции

Функция для поиска максимума в массиве:

```
int * max_element (int * p, int * q) {
    int * pmax = p;
    for (; p != q; ++p)
        if (*p > *pmax)
            pmax = p;

    return pmax;
}
```

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

Возрат значения через указатель

Функция для поиска максимума в массиве:

```
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;
}
```

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

Возрат значения через указатель на указатель

Функция для поиска максимума в массиве:

```
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;
}
```

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

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.
- Динамическая память выделяется в сегменте данных.
- Структура, отвечающая за выделение дополнительной памяти, называется **кучей** (не нужно путать с одноимённой структурой данных).
- Выделение и освобождение памяти *управляется вручную*.

Выделение памяти в стиле C

- 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

- 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

- 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;
```

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 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:

```
// 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 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
```

Многомерные встроенные массивы

- C++ позволяет определять многомерные массивы:

```
int m2d[2][3] = { {1, 2, 3}, {4, 5, 6} };
for( size_t i = 0; i != 2; ++i ) {
    for( size_t j = 0; j != 3; ++j ) {
        cout << m2d[i][j] << ' ';
    }
    cout << endl;
}
```

- Элементы m2d располагаются в памяти “по строчкам”.
- Размерность массивов может быть любой, но на практике редко используют массивы размерности > 4.

```
int m4d[2][3][4][5] = {};
```

Динамические массивы

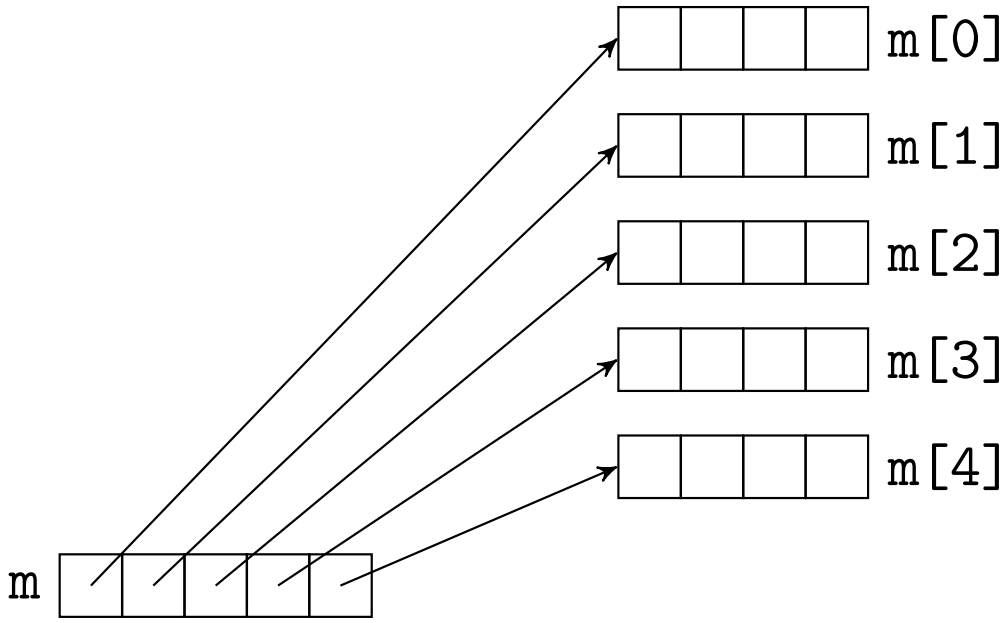
- Для выделения одномерных динамических массивов обычно используется оператор `new []`.

```
int * m1d = new int[100];
```

- Какой тип должен быть у указателя на двумерный динамический массив?
  - Пусть `m` — указатель на двумерный массив типа `int`.
  - Значит `m[i][j]` имеет тип `int` (точнее `int &`).
  - `m[i][j]` $\Leftrightarrow$ `*(m[i] + j)`, т.е. тип `m[i]` — `int *`.
  - аналогично, `m[i]` $\Leftrightarrow$ `*(m + i)`, т.е. тип `m` — `int **`.
- Чему соответствует значение `m[i]`?
  Это адрес строки с номером `i`.
- Чему соответствует значение `m`?
  Это адрес массива с указателями на строки.

Двумерные массивы

Давайте рассмотрим создание массива 5×4.




```
int ** m = new int * [5];
for (size_t i = 0; i != 5; ++i)
    m[i] = new int[4];
```

Двумерные массивы

Выделение и освобождение двумерного массива размера $a \times b$.

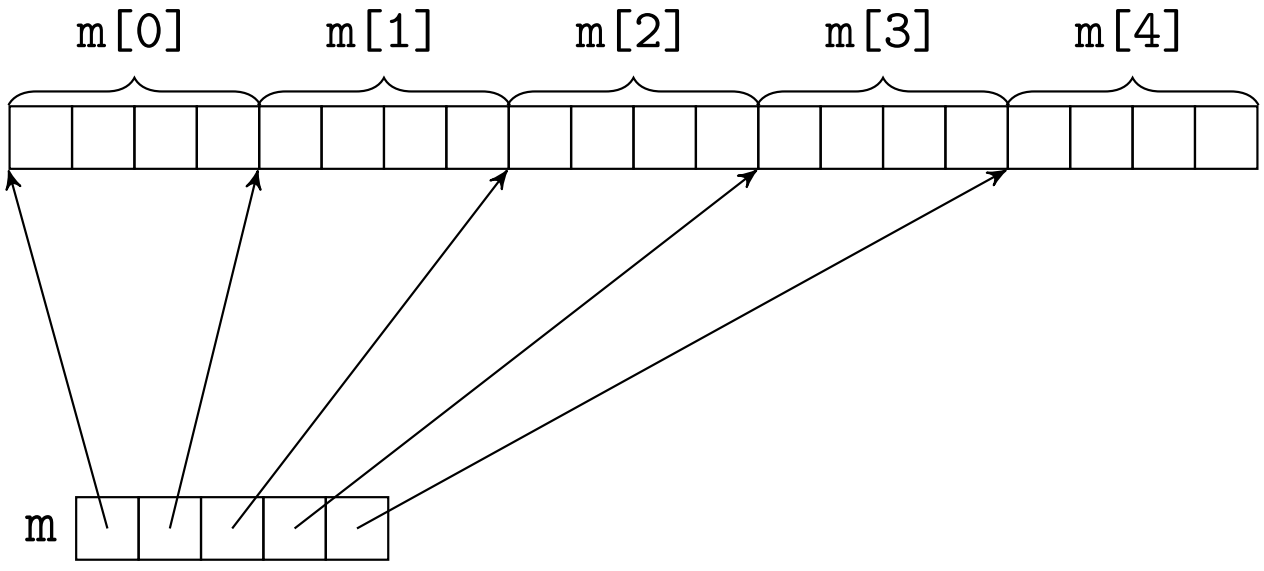
```
int ** create_array2d(size_t a, size_t b) {
    int ** m = new int *[a];
    for (size_t i = 0; i != a; ++i)
        m[i] = new int[b];
    return m;
}

void free_array2d(int ** m, size_t a, size_t b) {
    for (size_t i = 0; i != a; ++i)
        delete [] m[i];
    delete [] m;
}
```

При создании массива оператор `new` вызывается $(a + 1)$ раз.

Двумерные массивы: эффективная схема

Рассмотрим эффективное создание массива 5×4.




```
int ** m = new int * [5];
m[0] = new int[5 * 4];
for (size_t i = 1; i != 5; ++i)
    m[i] = m[i - 1] + 4;
```

Двумерные массивы: эффективная схема

Эффективное выделение и освобождение двумерного массива размера $a \times b$.

```
int ** create_array2d(size_t a, size_t b) {
    int ** m = new int *[a];
    m[0] = new int[a * b];
    for (size_t i = 1; i != a; ++i)
        m[i] = m[i - 1] + b;
    return m;
}

void free_array2d(int ** m, size_t a, size_t b) {
    delete [] m[0];
    delete [] m;
}
```

При создании массива оператор `new` вызывается 2 раза.