Основы алгоритмизации и программирования

Пашук Александр Владимирович

pashuk@bsuir.by

Содержание лекции

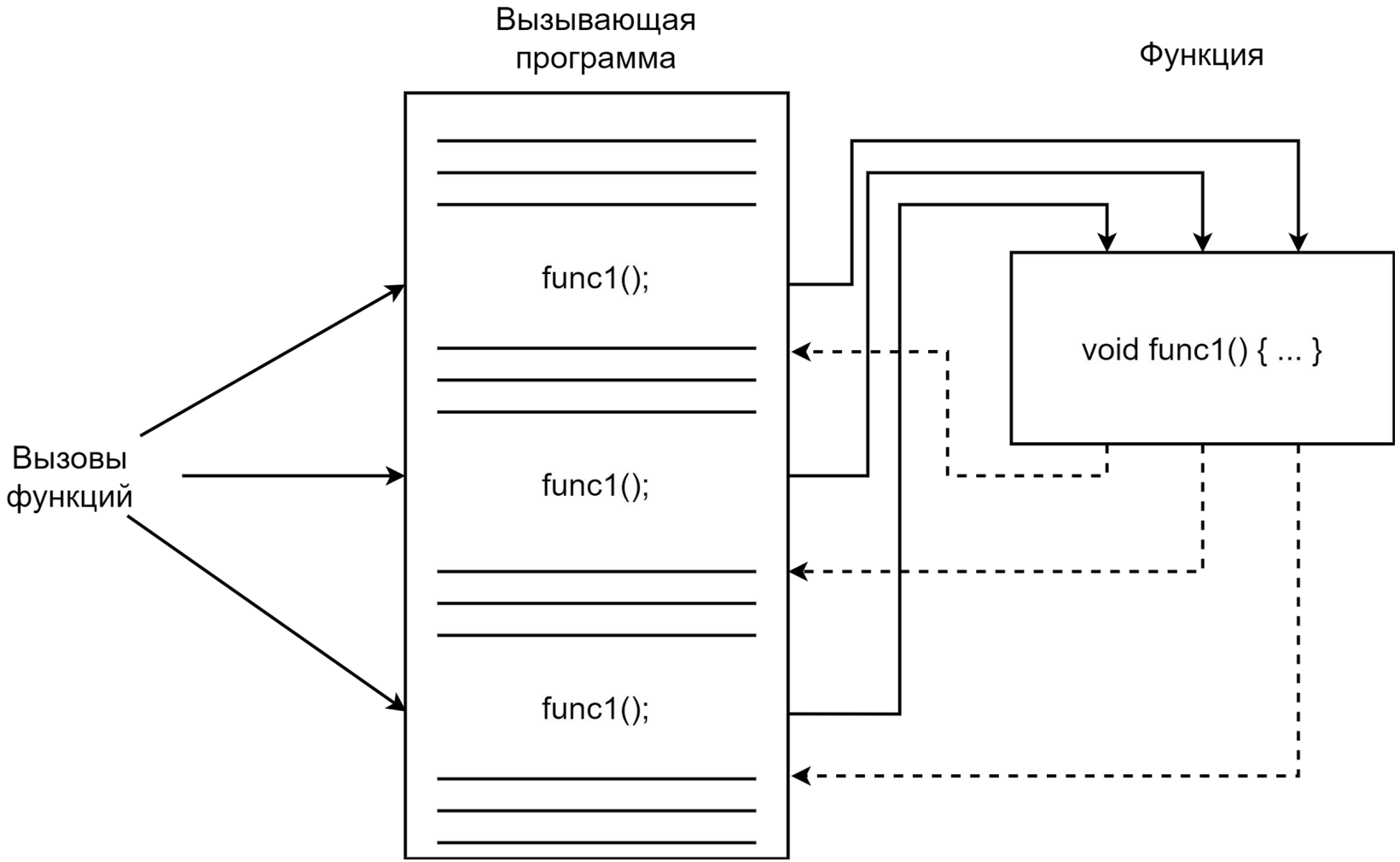
1. Функции. Простые функции
2. Передача аргументов в функции
3. Значения, возвращаемые функцией
4. Ссылки на аргументы
5. Перегруженные функции
6. Вопросы из теста

Что такое функция?

Функция – это именованное объединение группы операторов, которое может быть вызвано из других частей программы.

- Основная причина использования функций: **необходимость структурировать программу**.
- Концепция функций появилась из-за стремления разработчиков сократить размер программного кода.
- Код функции хранится только в одной области памяти

Что такое функция?



Простые функции

```
void starline() {
    for (int i=0; i<50; i++)
        cout << "*";
    cout << endl;
}

int main(){
    starline();
    cout << "Username: Test User" << endl;
    starline();
    cout << "Password: ******" << endl;
    starline();
}
```

Объявление функции

Как и переменные, функции нельзя использовать до указания необходимых атрибутов. Есть два способа описать функцию:

- Объявить функцию (прототип)
- Определить функцию

Синтаксис объявления функции в общем виде:

```
<return_value> <name>(<args>);
```

```
void starline();
```

Пример

```
void starline();

int main(){
    starline();
    cout << "Username: Test User" << endl;
    starline();
    cout << "Password: ******" << endl;
    starline();
}

void starline() {
    // ...
}
```

Пример

```
**************************************************

Username: Test User

**************************************************

Password: ******

**************************************************
```

Определение функции

Определение функции состоит из заголовка и тела функции.

```
void starline()
{
    for (int i=0; i<50; i++)
        cout << "*";
    cout << endl;
}
```

Заголовок функции

Тело функции

Заголовок функции должен соответствовать её прототипу.

Определение без объявления

- Объявление функции (прототип) может отсутствовать, если определение функции происходит раньше, чем первый её вызов.
- Такой подход удобен в небольших программах.
- Теряется гибкость работы с функциями: в крупных проектах используется множество функций и разработчик вынужден следить за их местоположением в коде.
- Неприменимо, если в проекте принято располагать функцию `main()` первой.

Передача аргументов

Аргумент – единица данных, передаваемая программой в функцию.

Аргументы позволяют функции оперировать различными значениями или выполнять различные действия в зависимости от переданных ей значений.

Передача констант в функцию

```
void repchar(char, int);

int main(){
    repchar('-', 30);
    cout << "Username: Test User" << endl;
    repchar('+', 30);
    cout << "Password: ******" << endl;
    repchar('-', 30);
}
void repchar(char symbol, int n) {
    for (int i=0; i<n; i++)
        cout << symbol;
    cout << endl;
}
```

Передача констант в функцию

```
----------------------------
Username: Test User
++++++++++++++++++++++++++++
Password: ******
----------------------------
```

Передача аргументов

Два правила:

- Важно соблюдать порядок следования аргументов
- Типы аргументов в объявлении и определении должны быть согласованы.

Переменные, используемые внутри функции для хранения значений аргументов, называются **параметрами**.

Передача значений переменных

```
void repchar(char symbol, int n);

int main(){
    char symbol_in;
    int n_in;

    cout << "Enter symbol: ";
    cin >> symbol_in;
    cout << "Enter n: ";
    cin >> n_in;

    repchar(symbol_in, n_in);
}
```

```
Enter symbol: !
Enter n: 10
!!!!!!!!!!
```

Передача аргументов по значению

```
void square(int x) {
    x = x*x;
}

int main(){
    int x = 100;
    cout << x << endl; // 100

    square(x);

    cout << x << endl; // ???
}
```

Прототипы

```
float calc_dist(int, int, int, int); // Not bad!
float calc_dist(int x1, int y1, int x2, int y2); // Better!

// You can use different names in definition
float calc_dist(int px1, int py1, int px2, int py2){
    // ...
}
```

Возвращаемое значение

Для того, чтобы вернуть вызывающей программе значение используется оператор `return`

```
float calc_dist(int x1, int y1, int x2, int y2) {
    float dist = sqrt(pow(x2 - x1, 2) + pow(y2 - y1,
2));
    // Return value to main program
    return dist;
}
```

Возвращаемое значение

```
int main(){
    int x1=1, y1=1;
    int x2=0, y2=-1;

    float dist1 = calc_dist(0, 0, x1, y1);
    float dist2 = calc_dist(0, 0, x2, y2);

    if (dist1 < dist2 )
        cout << "Point 1 closer to (0,0)\n";
    else
        cout << "Point 2 closer to (0,0)\n";
}
```

Возвращаемое значение

```
float calc_dist(int x1, int y1, int x2, int y2) {
    float dist = sqrt(pow(x2 - x1, 2) + pow(y2 - y1,
2));
    cout << &dist << endl; // 0x61fecc
    return dist;
}

int main(){
    // ...
    float dist1 = calc_dist(0, 0, x1, y1);
    cout << &dist1 << endl; // 0x61fefc
    // ...
}
```

Возвращаемое значение

- Количество аргументов функции может быть произвольным, но возвращаемое значение – всегда только одно.
- Есть способы, позволяющие возвращать несколько значений, один из них – передача аргументов по ссылке.
- Всегда нужно указывать тип значения, возвращаемого функцией или `void`, если функций ничего не возвращает.
- Функция без указания возвращаемого типа должна вернуть `int`.

Ссылки на аргументы

Ссылка является псевдонимом, или альтернативным именем переменной.

Наиболее важное применение ссылок – передача аргументов в функции.

- При передаче аргументов по значению функция не имеет доступа к переменным-аргументам, а работает с их копиями.
- При передаче аргументов по ссылке функция получает не копию значения переменной, а ссылку на эту переменную.

Ссылки на аргументы

```
void increment(int& number) {
    number = number + 1;
}

int main(){
    int a = 100;
    increment(a);
    increment(a);
    increment(a);

    cout << "a = " << a << endl; // a = 103
}
```

Пример

```
void swap(int&, int&);
int main(){
    int a = 101, b = 202;

    swap(a, b); // No &!

    cout << "a = " << a << "; b = " << b << endl;
    // a = 202; b = 101
}
void swap(int& x, int& y) {
    int temp;
    temp = x;     x = y;     y = temp;
    return;
}
```

Передача по указателю

```
void swap(int* x, int* y) {
    int temp = *x;    *x = *y;    *y = temp;
    return;
}

int main(){
    int a = 101, b = 202;

    swap(&a, &b); // With &!

    cout << "a = " << a << "; b = " << b << endl;
    // a = 202; b = 101
}
```

Что получим?

```
void swap(int* x, int* y) {
    int* temp;    temp = x;    x = y;    y = temp;
    return;
}

int main(){
    int a = 101, b = 202;

    swap(&a, &b);

    cout << "a = " << a << "; b = " << b << endl; // ???
}
```

Передача указатель/по ссылке

- Передача указателя в функцию в качестве аргумента похожа на передачу по ссылке.

- Но механизмы различны: **ссылка – это псевдоним переменной**, а **указатель – адрес переменной**.

Передача массивов

```
void degrees2radians(int length, float arr[]) {
    for (int i = 0; i < length; i++)
        arr[i] *= 0.0174533;
}
int main(){
    float degrees[] = {0, 90, 180, 270};
    int length = sizeof(degrees) / sizeof(float);

    degrees2radians(length, degrees); // No &!

    for (int i = 0; i < length; i++)
        cout << degrees[i] << " ";
    // 0 1.5708 3.14159 4.71239
}
```

Передача массивов

```
void degrees2radians(int length, float arr[]) {
    cout << arr << endl; // 0x61fef4
    // ...
}

int main(){
    float degrees[] = {0, 90, 180, 270};

    cout << degrees << endl; // 0x61fef4
    degrees2radians(length, degrees);
    // ...
}
```

Немного сложнее

```
void print_matrix(int matrix[][], int m_size) {
    for (int i = 0; i < m_size; i++)
        for (int j = 0; j < m_size; j++)
            cout << setw(4) << matrix[i][j] << " ";
        cout << endl;
}

int main() {
    int matrix_size = 3;
    int matrix[matrix_size][matrix_size] = {
         {1, 2, 3}, {4, 5, 6}, {7, 8, 9}
    };
    print_matrix(matrix, matrix_size);
}
```

Немного сложнее

```
void print_matrix(int matrix[3][3], int m_size) {
    for (int i = 0; i < m_size; i++)
        for (int j = 0; j < m_size; j++)
            cout << setw(4) << matrix[i][j] << " ";
        cout << endl;
}

int main() {
    const int matrix_size = 3;
    int matrix[matrix_size][matrix_size] = {
        {1, 2, 3}, {4, 5, 6}, {7, 8, 9}
    };
    print_matrix(matrix, matrix_size);
}
```

Объявление функции с аргументами в виде массива

Можно использовать следующие форматы объявления:

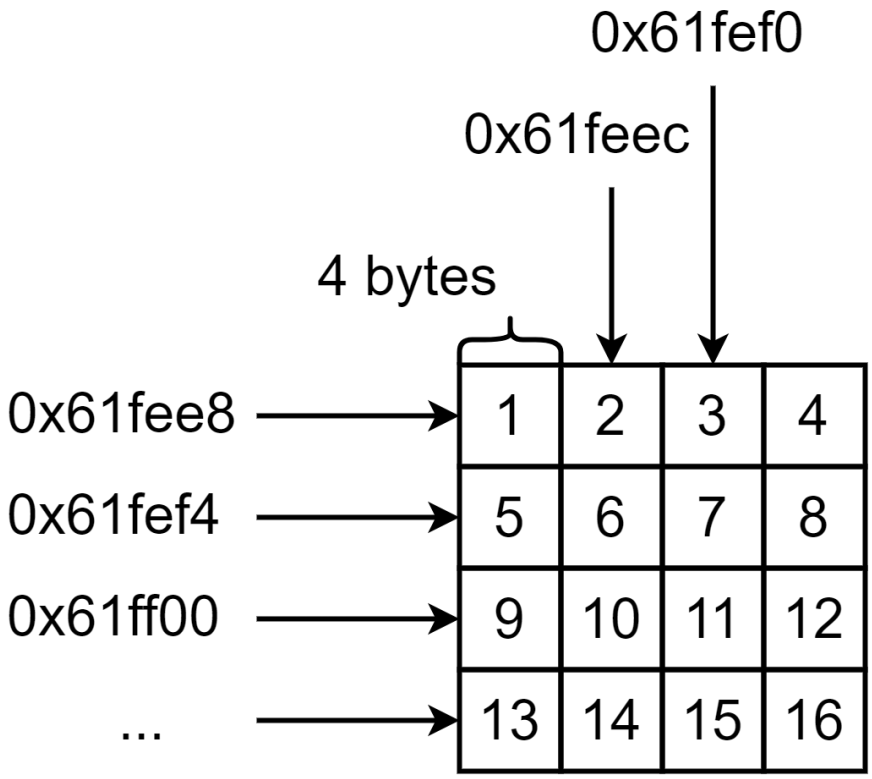
```
void print_matrix(int matrix[3][3], int m_size)
void print_matrix(int matrix[][3], int m_size)
void print_vector(int vector[3], int v_size)
void print_vector(int vector[], int v_size)
```

Но не:

```
void print_matrix(int matrix[3][], int m_size)
void print_matrix(int matrix[][], int m_size)
```

Почему?

```
int matrix[N][M] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12},
    {13, 14, 15, 16},
};
```



Передача массивов: указатели

```
void degrees2radians(int length, float* arr) {
    for (int i = 0; i < length; i++)
        arr[i] *= 0.0174533;
}
int main(){
    float degrees[] = {0, 90, 180, 270};
    int length = sizeof(degrees) / sizeof(float);

    degrees2radians(length, degrees); // No &!

    for (int i = 0; i < length; i++)
        cout << degrees[i] << " ";
    // 0 1.5708 3.14159 4.71239
}
```

Передача массивов: указатели

```
void degrees2radians(int length, float* arr) {
    for (int i = 0; i < length; i++)
        arr[i] *= 0.0174533;
}
int main() {
    // ... Get number of items
    float* degrees = new float[length];
    // ... Fill array

    degrees2radians(length, degrees);

    for (int i = 0; i < length; i++)
        cout << degrees[i] << " ";
    // 0 1.5708 3.14159 4.71239
}
```

Немного сложнее

```
void print_matrix(int* matrix[], int m_size) {
    for (int i = 0; i < m_size; i++)
        for (int j = 0; j < m_size; j++)
            cout << setw(4) << matrix[i][j] << " ";
        cout << endl;
}

int main() {
    int m_size = 3;
    int** matrix = new int*[m_size];
    for (int i = 0; i < m_size; i++)
        matrix[i] = new int[m_size];

    print_matrix(matrix, m_size);

    // ... Clear memory
}
```

Немного сложнее

```
void print_matrix(int** matrix, int m_size) {
    for (int i = 0; i < m_size; i++)
        for (int j = 0; j < m_size; j++)
            cout << setw(4) << matrix[i][j] << " ";
        cout << endl;
}

int main() {
    int m_size = 3;
    int** matrix = new int*[m_size];
    for (int i = 0; i < m_size; i++)
        matrix[i] = new int[m_size];

    print_matrix(matrix, m_size);

    // ... Clear memory
}
```

Перегрузка функций

Перегруженная функция выполняет различные действия, зависящие от типов данных, передаваемых ей в качестве аргументов.

Популярный пример-сравнение: анекдот про термос.

Пример

```
void starline() {
    for (int i=0; i<50; i++)
        cout << "*";
    cout << endl;
}
void repchar(char symbol, int n) {
    for (int i=0; i<n; i++)
        cout << symbol;
    cout << endl;
}
void charline(char symbol) {
    for (int i=0; i<50; i++)
        cout << symbol;
    cout << endl;
}
```

Перегрузка функций

Очевидный недостаток: разработчику нужно запомнить все имена функций и различия между действиями, выполняемыми функциями.

Решение: использование перегрузки. Очевидно, что использовать одно и то же имя было бы гораздо удобнее.

Улучшенная версия примера

```
void repchar() {
    for (int i=0; i<50; i++)
        cout << "*";
    cout << endl;
}
void repchar(char symbol) {
    for (int i=0; i<50; i++)
        cout << symbol;
    cout << endl;
}
void repchar(char symbol, int n) {
    for (int i=0; i<n; i++)
        cout << symbol;
    cout << endl;
}
```

Улучшенная версия примера

```
int main() {
    repchar();
    repchar('#');
    repchar('$', 15);

    // **********************************************
    // ##############################################
    // $$$$$$$$$$$$$$

    return 0;
}
```

Как это работает?

Довольно просто: с помощью сигнатуры функции, которая позволяет различать между собой функции по количеству аргументов и их типам.

```
repchar();
repchar('#');
repchar('$', 15);
```

Компилятор, обнаружив несколько функций с одинаковыми именами, позволяет корректно обработать все определения функций и их вызовы.

Различные типы аргументов

```
void print(int i) {
    cout << "It is an int: " << i << endl;
}

void print(double f) {
    cout << "It is a float: " << f << endl;
}

void print(char const *c) {
    cout << "It is a char*: " << c << endl;
}
```

Пример

```
int main() {
    print(10);
    print(101.11);
    print("Hello World");

    // It is an int: 10
    // It is a float: 101.11
    // It is a char*: Hello World

    return 0;
}
```

Рекурсия

Рекурсия – это средство программирование, позволяющее функции вызывать саму себя на выполнение.

Самые популярные примеры: вычисление факториала, нахождение чисел Фибоначчи, вычисление N-го элемента какой-либо последовательности.

Простейший пример

```
unsigned long fact(unsigned long n) {
    if (n > 1)
        return n * fact(n - 1);
    else
        return 1;
}

int main() {
    cout << fact(10);
    // 3628800
}
```

Простейший пример

Версия	Действие	Значение
1	Вызов	5
2	Вызов	4
3	Вызов	3
4	Вызов	2
5	Вызов	1
5	Возврат	1
4	Возврат	2
3	Возврат	6
2	Возврат	24
1	Возврат	120

```
int fact(int n) {
    if (n > 1)
        return n * fact(n-1);
    else
        return 1;
}
```

Рекурсия

SAFELY ENDANGERED
SWEET JESUS, POOH! THAT'S NOT HONEY
YOU'RE EATING RECURSION
SAFELY ENDANGERED
SWEET JESUS, POOH! THAT'S NOT HONEY
YOU'RE EATING RECURSION

Рекурсия

Каждая рекурсия должна включать в себя условие окончания рекурсии.

В противном случае рекурсия будет происходить бесконечно, что приведет к аварийному завершению программы.

В некоторых случаях рекурсия может привести к исчерпанию оперативной памяти, особенно, если речь идет о большом количестве вложенных вызовов.

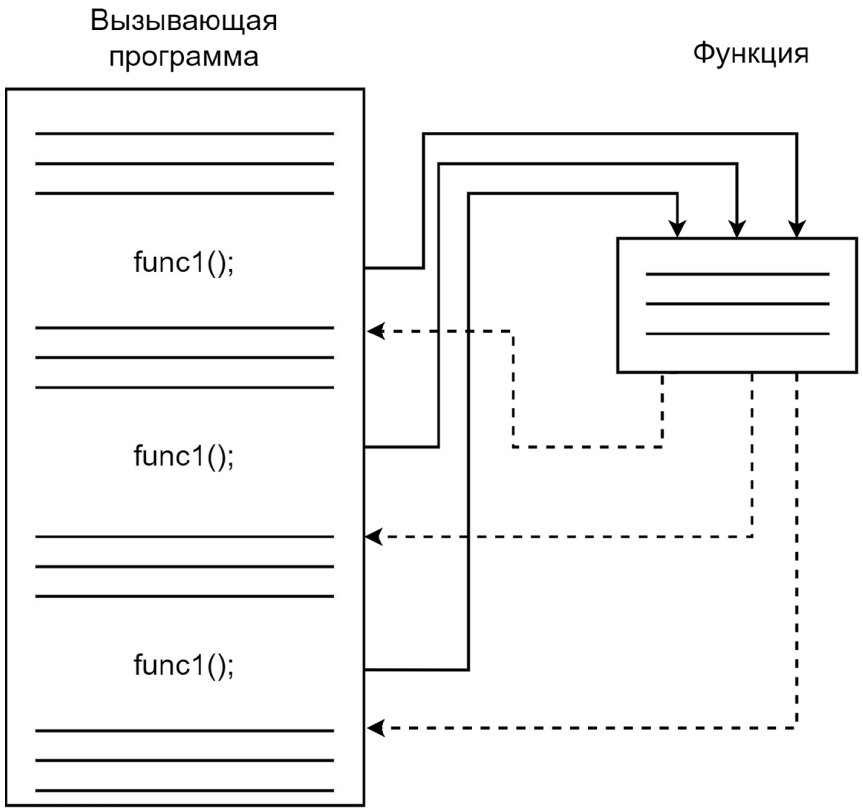
Еще один пример

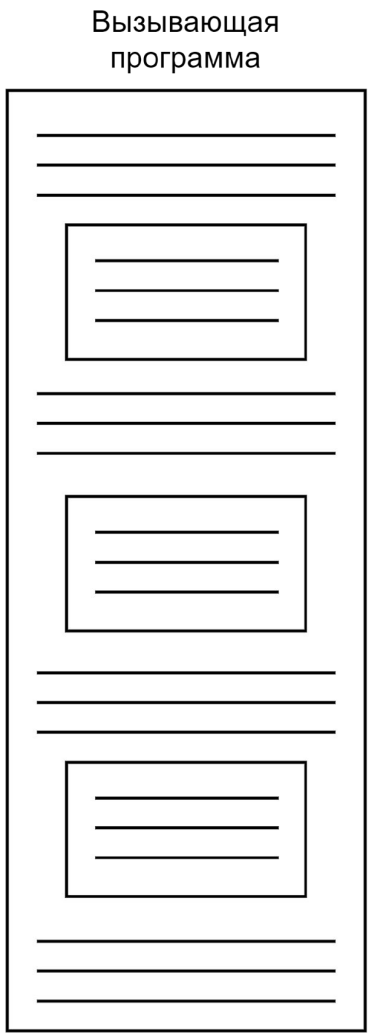
```
// 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144
int fib(int n) {
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}

int main() {
    cout << fib(15);
    // 610
}
```

Встраиваемые функции

- Использование функций является экономичным с точки зрения использования памяти, т.к. вместо дублирования используется механизм вызовов функции.
- Однако, кроме экономии памяти, использование функций увеличивает время выполнения программ: необходимо время на выполнение команды перехода в функцию и команду перехода на оператор, следующий после вызова функции.

Встраиваемые функции





Когда использовать?

Обычно встраиваемые функции представляют собой небольшой фрагмент кода.

В этом случае дополнительные инструкции, необходимые для вызова функции, могут занять столько же памяти, сколько занимает код самой функции. Экономия памяти превращается в дополнительный расход.

Очевидное решение: вставлять повторяющиеся последовательности операторов (тело функции) там, где это необходимо.

Когда использовать?

Неочевидное решение: использовать встраиваемые функции (inline functions).

Отличие встраиваемых функций от обычных заключается в том, что исполняется код таких функций вставляется (встраивается) в исполняемый код программы.

Главное преимущество: код программы остается организованным, при этом производительность не «проседает».

Пример

```
inline void starline() {
    for (int i=0; i<50; i++)
    cout << "*";
    cout << endl;
}

int main(){
    starline();
    cout << "Username: Test User" << endl;
    starline();
    cout << "Password: ******" << endl;
    starline();
}
```

Встраиваемые функции

Важно: ключевое слово inline является рекомендацией компилятору, которая может быть проигнорирована.

В случае игнорирования, функция будет скомпилирована как обычная. Например, если компилятор посчитает функцию слишком длинной для того, чтобы делать ее встраиваемой.

Аргументы по умолчанию

- В C++ можно организовать функцию так, чтобы ее можно было вызвать вообще не указывая при этом никаких аргументов.

- Для этого используются значения аргументов по умолчанию.

Улучшенная версия примера

```
void repchar() {
    for (int i=0; i<50; i++)
        cout << "*";
    cout << endl;
}
void repchar(char symbol) {
    for (int i=0; i<50; i++)
        cout << symbol;
    cout << endl;
}
void repchar(char symbol, int n) {
    for (int i=0; i<n; i++)
        cout << symbol;
    cout << endl;
}
```

Улучшенная версия примера

```
void repchar(char symbol='*', int n=50) {
    for (int i=0; i<n; i++)
        cout << symbol;
    cout << endl;
}

int main() {
    repchar();
    repchar('#');
    repchar('$', 10);
    // **************************************************
    // ##################################################
    // $$$$$$$$$$
}
```

Улучшенная версия примера

```
void repchar(char symbol='*', int n=50);

int main() {
    repchar();
    repchar('#');
    repchar('$', 10);
}

void repchar(char symbol, int n) {
    for (int i=0; i<n; i++)
        cout << symbol;
    cout << endl;
}
```

Аргументы по умолчанию

- Опускать при вызове можно только аргументы, стоящие в конце списка при объявлении функции.

Например: можно не указать три последних аргумента, но нельзя одновременно пропустить предпоследний аргумент и указать последний.

- Нельзя пропускать аргументы, для которых не указано значение аргумента по умолчанию.

Область видимости и класс памяти

Два аспекта, касающихся взаимодействия переменных и функций:

- Область видимости
- Класс памяти.

Область видимости определяет из каких частей программы возможен доступ к переменной, а **класс памяти** – время, в течение которого переменная существует в памяти компьютера.

Типы области видимости

Три типа области видимости:

- Локальная область видимости
- Область видимости файла
- (Область видимости класса)

Переменные, имеющие локальную область видимости, доступны внутри блока (`{ }`), в котором они определены.

Переменные, имеющие область видимости файла, доступны из любого места файла.

Классы памяти

Существует два класса памяти: automatic (автоматический) и static (статический).

- У переменных первого класса время жизни равно времени жизни функции, внутри которой они определены.
- У переменных второго класса время жизни равно времени жизни всей программы.

Локальные переменные

Переменные, определяемые внутри функции (включая функцию `main`), называются **локальными**, поскольку их область видимости ограничивается этой функцией.

- Такие переменные также иногда называются автоматическими, поскольку они имеют класс памяти static.

Локальные переменные

- Создаются и уничтожаются при входе и выходе из функции соответственно.
- Компилятор не инициализирует локальные переменные. Они имеют неопределенное значение.
- Использование таких переменных позволяет обеспечить модульность и организованность программы.

Глобальные переменные

Глобальные переменные определяются вне каких-либо функций (а также вне классов).

- Глобальные переменные видимы из всех функций данного файла (определенных позже, чем сама переменная) и, потенциально, из других файлов.
- Иногда глобальные пременные также называют внешними.
- Если нет явной инициализации, компилятор во время создания переменной присвоит ей значение 0.

Глобальные переменные

- Глобальные переменные имеют статический класс памяти, что означение их существование в течение всего времени выполнения программы.
- Память под эти переменные выделяется в начале выполнения программы и закрепляется до завершения программы.
- Не обязательно использовать ключевое слово `static`, т.к. они и имеют статический класс памяти.

Статические переменные

Существует два вида статических переменных:

- Статические локальные переменные
- Статические глобальные переменные

Статическая локальная переменная имеет такую же область видимости, как и автоматическая: функция, к которой принадлежит переменная.

Время жизни такой переменной совпадает со временем жизни глобальной переменной, но существование начинается при первом вызове функции.

Пример

```
float get_avg(float new_number) {
    static float total = 0;
    static int count = 0;
    count++;        total += new_number;
    return total/count;
}
int main() {
    float number = 1;
    while(number) {
        cout << "Enter the number: ";
        cin >> number;
        cout << "Average: " << get_avg(number) << endl;}
}
```

Область видимости/Класс памяти

	Локальная	**Статическая (Л)**	**Глобальная**
Области видимости	Функция	Функция	Программа
Время жизни	Функция	Программа	Программа
Начальное значение	Случайное	0	0
Область памяти	Стек	Динамическая	Динамическая
Назначение			

Возвращение значения по ссылке

В функции можно не только передавать аргументы с помощью ссылок, но также можно возвращать значение функции по ссылке.

Одна из причин использования такого ссылочного механизма – необходимость избежать копирования объектов большого размера.

Другая причина – использование функции в качестве левого операнда операции присваивания.

Пример

```
int x;
int& setx();

int main() {
  setx() = 92;
  cout << "x = " << x << endl;
  // x = 92
  return 0;
}

int& setx() {
  return x;
}
```

Возвращение значения по ссылке

- Вызов функции интерпретируется как значение, получаемое при его выполнении:

```
y = squareroot(x);
```

- Вызов функции интерпретируется как переменная (возврат ссылки = возврату псевдонима для переменной в `return`):

```
setx() = 92;
```

Еще примеры

```
int x;
int& setx();

int main() {
  setx() = 92;
  cout << "x = " << x << endl;
  return 0;
}

int& setx() {
  return 33;
}
```

```
int x;
int& setx();

int main() {
  setx() = 92;
  cout << "x = " << x << endl;
  return 0;
}

int& setx() {
  int y = 33;
  return y;
}
```

Зачем всё это?

- В процедурном программирование существует очень мало задач, в которых может понадобиться возвращать значение по ссылке.
- Иногда используется при перегрузке операций.

Константные аргументы функции

```
void aFunc(int& a, const int& b);

int main() {
  int alpha = 7, beta = 11;

  aFunc(alpha, beta);
}

void test_func(int& a, const int& b) {
  a = 107;
  b = 111; // Error!
}
```

Пример вопроса на экзамене

Значение аргумента по умолчанию:

- Может использоваться вызывающей программой
- Может использоваться функцией
- Должно быть константой
- Должно быть значением переменной

Пример вопроса на экзамене

Каков результат работы этой программы?

```
#include <iostream>

int main() {
    int x = 9;
    {
        int x = 7;
        std::cout << x << ' ';
    }
    std::cout<<x;
}
```

Пример задачи на экзамене

Написать функцию, которая принимает количество секунд, переводит их формат `<сутки> <часы> <минуты> <секунды>` **и выводит на экран.**

- Предусмотреть валидацию входных данных.
- Программа должна запрашивать новое количество секунд до тех пор, пока пользователь не введет N.
- Программа должна пропускать пустые единицы, например: 5 часов (а не 0 суток 5 часов…)

Когда 2 часа искал ошибку в коде,
а оказалось, что в цикле for
вместо j написал i