
Вычислительная техника и компьютерное моделирование в физике

Лекция 4

Зинчик Александр Адольфович

zinchik_alex@mail.ru

Оформлению кода на C++

- Рекомендации Стэнфордского университета
- <http://stanford.edu/class/archive/cs/cs106b/cs106b.1158/styleguide.shtml>

Отделяйте пробелами фигурные скобки:

```
1 // Плохая практика
2 int x = 3, y = 7; double z = 4.25; x++;
3 if (a == b) { foo(); }
4
5 // Хорошая практика
6 int x = 3;
7 int y = 7;
8 double z = 4.25;
9
10 x++;
11 if (a == b) {
12     foo();
13 }
```

Ставьте пробелы между операторами и операндами:

```
1 | int x = (a + b) * c / d + foo();
```

Когда строка становится длиннее 100 символов, разделите её на две.

```
1 | int result = reallyLongFunctionOne() + reallyLongFunctionTwo() +  
2 |     reallyLongFunctionThree() + reallyLongFunctionFour();  
3 |  
4 | int result2 = reallyLongFunction(parameterOne, parameterTwo, parameterThree,  
5 |     parameterFour, parameterFive, parameterSix);
```

Оставляйте пустые линии между функциями и между группами выражений:

```
1 | void foo() {  
2 |     ...  
3 | }  
4 |                                     // пустая линия  
5 | void bar() {  
6 |     ...  
7 | }
```

Используйте текстовую строку, стандартную для C++

```
1 // Плохая практика: текстовая строка в стиле Си
2 char* str = "Hello there";
3
4 // Хорошая практика: текстовая строка в стиле C++
5 string str = "Hello there";
```

Названия и переменные

- Давайте переменным описательные имена, такие как **firstName** или **homeworkScore**.
- Избегайте однобуквенных названий вроде **x** или **c**, за исключением итераторов вроде **i**.
- Называйте переменные и функции, используя **верблюжий Регистр**.
- Называйте классы **Паскальным Регистром**, а константы — в **ВЕРХНЕМ_РЕГИСТРЕ**.

Константы

- Если определенная константа часто используется в вашем коде, то обозначьте её как `const` и всегда ссылайтесь на данную константу, а не на её значение:

```
1 | const int VOTING_AGE = 18;
```

- Плохая практика

```
int i;
```

```
for(i = 0; i < 18; i++) {...}
```

- Хорошая практика

```
for(int age = 0; age < VOTING_AGE; age++) {...}
```

Глобальные переменные

- Никогда не объявляйте изменяемую глобальную переменную. Глобальными переменными должны быть только константы.
- Вместо того, чтобы делать значение глобальным, сделайте его параметром и возвращайте значение, когда необходимо:

Не используйте глобальные переменные!

```
// Плохая практика
int count; // Глобальная переменная

void func1() {
    count = 42;
}

void func2() {
    count++;
}

int main() {
    func1();
    func2();
}
```

```
// Хорошая практика!
int func1() {
    return 42;
}

void func2(int& count) {
    count++;
}

int main() {
    int count = func1();
    func2(count);
}
```

Комментарии

- **Заглавный комментарий.** Размещайте заглавный комментарий, который описывает назначение файла, вверху каждого файла. Предположите, что читатель вашего комментария является продвинутым программистом, но не кем-то, кто уже видел ваш код ранее.

- **Заголовок функции / конструктора.** Разместите заголовочный комментарий на каждом конструкторе и функции вашего файла. Заголовок должен описывать поведение и / или цель функции.

Комментарии

- **Параметры / возврат.** Если ваша функцию принимает параметры, то кратко опишите их цель и смысл. Если ваша функция возвращает значение — кратко опишите, что она возвращает.
- **Исключения.** Если ваша функция намеренно выдает какие-то исключения для определенных ошибочных случаев, то это требует упоминания.

Комментарии

- **Комментарии на одной строке.** Если внутри функции имеется секция кода, которая длинна, сложна или непонятна, то кратко опишите её назначение.
- **TODO.** Следует удалить все `// TODO` комментарии перед тем, как заканчивать и сдавать программу.

Использование namespace std

- //не рекомендуется

```
using namespace std;  
int main() {  
    cout<<“Hello World!”;  
}
```

- //рекомендуется

```
int main(){  
    std::cout<<“Hello World!”;  
}
```

Функции и процедурное проектирование

Хорошо спроектированная функция имеет следующие характеристики:

1. Полностью выполняет четко поставленную задачу;
2. Не берет на себя слишком много работы;
3. Не связана с другими функциями бесцельно;
4. Хранит данные максимально сжато;
5. Помогает распознать и разделить структуру программы;
6. Помогает избавиться от излишков, которые иначе присутствовали бы в программе.

Пример правильно оформленной функции

```
1  /*
2   * Решает квадратное уравнение  $ax^2 + bx + c = 0$ ,
3   * внося результаты в root1 и root2.
4   * Предполагается, что данные уравнения имеют два корня.
5   */
6  void quadratic(double a, double b, double c,
7                double& root1, double& root2) {
8      double d = sqrt(b * b - 4 * a * c);
9      root1 = (-b + d) / (2 * a);
10     root2 = (-b - d) / (2 * a);
11 }
```

Векторы.

- В классе **vector** поддерживаются динамические массивы, увеличивающие свои размеры по мере необходимости.
- Ниже представлена спецификация шаблона для класса **vector**:
- **template<class T, class Allocator = allocator<T>>class vector**

- Здесь **T** – это тип данных, предназначенных для хранения в контейнере, а ключевое слово **Allocator** задает распределитель памяти, который по умолчанию является стандартным распределителем памяти. В классе определены следующие конструкторы:
 - **explicit vector(const Allocator &a=Allocator());**
 - **explicit vector(size_type число, const T &значение = T(), const Allocator a=Allocator());**
 - **vector(const vector <T,Allocator> объект);**

- `template <class InIter> vector(InIter начало, InIter конец,`
- `const Allocator &a=Allocator());`
- Первая форма представляет собой конструктор пустого вектора.
- Во второй форме конструктора вектора число элементов это **число** , а каждый элемент равен значению **значение**. Параметр **значение** может быть значением по умолчанию.
- В третьей форме конструктора вектор предназначен для одинаковых элементов, каждый из которых – это **объект**.
- Четвертая форма – это конструктор вектора, содержащего диапазон элементов, заданный итераторами **начало** и **конец**.

- Ниже представлено несколько примеров:
- `vector<int> iv;` // создание вектора нулевой длины для целых
- `vector<char> cv(5);` // создание пятиэлементного вектора для символов
- `vector<char> cv(5, 'x');` // создание и инициализация пятиэлементного вектора для символов
- `vector<int> iv2d(iv);` // создание вектора для целых из вектора для целых (2D массив).

- Для любого объекта, который будет храниться в векторе, должен быть определен конструктор по умолчанию и операторы $<$ $>$ и $==$.
- Для класса **vector** определяются следующие операторы сравнения:
- $==$, $<=$, $<$, $!=$, $>$, $>=$.
- Кроме этого для класса **vector** определяется оператор индекса $[]$, что обеспечивает доступ к элементам вектора посредством обычной индексной нотации.

- Пример работы с вектором. В файле находится произвольное количество целых чисел. Программа считывает их в вектор и выводит на экран в том же порядке.

```
#include <fstream>
#include <vector>
//using namespace std;
int main(){
    std::ifstream in ("inpnum.txt");
    std::vector<int> v;
    int x;

    while ( in >> x, !in.eof())
        v.push_back(x);

    for (std::vector<int>::iterator i = v.begin(); i != v.end(); ++i)
        std::cout << *i << " ";
}
```

- Поскольку файл содержит целые числа, используется соответствующая специализация шаблона `vector` — `vector<int>`. Для создания вектора `v` используется конструктор по умолчанию. Организуется цикл до конца файла, в котором из него считывается очередное целое число. С помощью метода `push_back` оно заносится в вектор, размер которого увеличивается автоматически.
- Для прохода по всему вектору вводится переменная `i` как итератор соответствующего типа (напомню, что операция `::` обозначает доступ к области видимости, то есть здесь объявляется переменная `i` типа «итератор для конкретной специализации шаблона»). С помощью этого итератора осуществляется доступ ко всем по порядку элементам контейнера, начиная с первого.

- Метод `begin()` возвращает указатель на первый элемент, метод `end()` — на элемент, следующий за последним. Реализация гарантирует, что этот указатель определен.
- Сравнить текущее значение с граничным следует именно с помощью операции `!=`, так как операции `<` или `<=` могут быть для данного типа не определены. Операция инкремента (`i++`) реализована так, чтобы после нее итератор указывал на следующий элемент контейнера в порядке обхода. Доступ к элементу вектора выполняется с помощью операции разадресации, как для обычных указателей.
- ```
for (int i = 0; i < v.size(); i++) std::cout << v[i] << endl;
```

- В данном примере вместо вектора можно было использовать любой последовательный контейнер путем простой замены слова `vector` на `deque` или `list`. При этом изменилось бы внутреннее представление данных и набор доступных операций, а в поведении программы никаких изменений не произошло бы.
- Однако если вместо цикла `for` вставить фрагмент `for (int i = 0; i < v.size(); i++) cout << v[i] << " ";` в котором использована операция доступа по индексу `[ ]`, программа не будет работать для контейнера типа `list`, поскольку в нем эта операция не определена.

- Пример (с клавиатуры вводятся в вектор 10 значений 0 или 1, после чего они выводятся на экран).

```
#include <vector>
#include <iostream>
using namespace std;
vector <bool> v (10);
int main(){
 for(int i = 0; i<v.size(); i++)cin >> v[i];

 for (vector <bool>:: const_iterator p = v.begin(); p!=v.end();
 ++p) cout << *p;
}
```