

# Информатика: семестр 3

## Программирование на C++

### Конспекты лекций

Лектор: Хирьянов Т.Ф.

24 октября 2017 г.

# Оглавление

<b>6</b>	<b>Продвинутая работа с функциями</b>	<b>3</b>
6.1	Перегрузка (overload) функций . . . . .	3
6.2	Шаблоны . . . . .	4
6.3	Аргументы функции по умолчанию . . . . .	7
6.4	Односвязный список для любого типа данных . . . . .	8

# Лекция 6

## Продвинутая работа с функциями

### 6.1 Перегрузка (overload) функций

Полиморфизм — много разных функций с одним именем. При вызове будет выбираться только одна: та, которая лучше всего подходит по входным параметрам. Это означает, что у функции один *интерфейс* и много *реализаций*. Рассмотрим пример:

Код 6.1: Пример полиморфизма

```
1 #include <iostream>
2
3 int absolute(int x) // for int
4 {
5     std::cout << "Call for int" << "\n";
6     if(x < 0)
7         return -x;
8     else
9         return x;
10 }
11
12 double absolute(double x) // for double
13 {
14     std::cout << "Call for double" << "\n";
15     return ((x > 0) ? x : -x);
16 }
17
18 long absolute(long x) // for long
19 {
20     std::cout << "Call for long" << "\n";
21     return (x > 0) ? x : -x;
22 }
23
24 int main()
25 {
26     int a = -5;
```

```
27     double b = -5.1;
28     long c = -1L;
29
30     std::cout << absolute(a) << std::endl;
31     std::cout << absolute(b) << std::endl;    // different types
32     std::cout << absolute(c) << std::endl;
33 }
```

В примере 6.1 мы пишем одну функцию **absolute** три раза: по одному на каждый тип входного (и, соответственно, возвращаемого) объекта. В случае, когда для заданного типа данных нет подходящей функции, C++ отбирает несколько функций как кандидатов на вызов, а затем выбирает их в следующем порядке:

1. Точное совпадение типов, если оно есть.
2. Приведение путем повышения разрядности числа и вызов кандидата с полученным типом (например, **int8** → **int16**)
3. Стандартное приведение типов (например, **float** → **int**). Оно уже менее надежно, поэтому и находится ниже по приоритету.
4. Пользовательское приведение типов, т.е. приведение к типу, объявленному пользователем (программистом, написавшим тип).
5. Эллипсис. В C++ существуют функции, у которых на вход поступает неопределенное число параметров – они называются эллипсис.

Код 6.2: Пример эллипсиса

```
1 void printf(char *s, ...)
```

Если на какой-то стадии проверки будет несколько кандидатов, равноправно удовлетворяющих условиям, то компилятор выдаст ошибку (он не поймет, какую вызывать).

## 6.2 Шаблоны

Хотелось бы, чтобы можно было делать функцию с неопределенным типом, как здесь:

Код 6.3: Неопределенный тип

```
1 #include <iostream>
2
3 T absolute(T x)    // Will not work
4 {
5     return (x > 0) ? x : -x;
6 }
7
8 int main()
9 {
10     int a = -5;
11     double b = -5.1;
12     long c = -1L;
13
14     std::cout << absolute(a) << std::endl;
15     std::cout << absolute(b) << std::endl;    // different types
16     std::cout << absolute(c) << std::endl;
17 }
```

Однако C++ не позволит этого сделать. Дело в том, что при компиляции подобного кода необходимо под каждый тип переменной, для которой вызывается функция, иметь функцию в соответствии с *шаблоном*. Исправим код:

Код 6.4: Неопределенный тип с шаблоном

```
1 #include <iostream>
2
3 template<typename T>
4
5 T absolute(T x)    // That is fine
6 {
7     return (x > 0) ? x : -x;
8 }
9
10 int main()
11 {
12     int a = -5;
13     double b = -5.1;
14     long c = -1L;
15
16     std::cout << absolute(a) << std::endl;
17     std::cout << absolute(b) << std::endl;    // different types
18     std::cout << absolute(c) << std::endl;
19 }
```

Тут может возникнуть проблема. Если бы мы вызвали **absolute** от строки, то во время компиляции, при *инстанцировании* функции для типа **string**, возникнет ошибка

– у **string** нет оператора унарного минуса (т.е. нельзя для строки *s* делать операцию ‘-s’). Эту проблему можно решить, если явно указать, каким типом инстанциировать функцию:

Код 6.5: Явное instantiation функции

```
1 #include <iostream>
2
3 template<typename T>
4
5 T absolute(T x)    // That is fine
6 {
7     return (x > 0) ? x : -x;
8 }
9
10 int main()
11 {
12     std::cout << absolute<int32_t>(-8.0) << std::endl;
13 }
```

Здесь мы изменили вызов на **absolute<int32\_t>**, тем самым указав тип, которым будет инстанциирована функция при компиляции.

Посмотрим, как ведет себя шаблон:

Код 6.6: Сложное instantiation

```
1 #include <iostream>
2
3 template<int number>
4 void self_counter()
5 {
6     static int32_t counter = 0;
7     counter++;
8     std::cout << "Was called: " << counter << " times";
9 }
10
11 int main()
12 {
13     //self_counter(); // won't work
14     self_counter<1>();
15     self_counter<1>();
16     self_counter<1>();
17     self_counter<2>();
18     self_counter<2>();
19     return 0;
20 }
```

В коде функция `self_counter<1>` будет вызвана 3 раза, а `self_counter<2>` – только 2. Дело в том, что мы функцию инстанцируем по-разному: сначала с 1, потом с 2 (как будто это разные функции). Простой код `self_counter()` в этом случае уже не работает, так как для инстанцирования есть несколько подходящих кандидатов (см. 6.1).

Код 6.7: Инстанцирование с параметром по умолчанию

```
1 #include <iostream>
2
3 template<int number>
4 void self_counter ()
5 {
6     static int32_t counter = 0;
7     counter++;
8     std::cout << "Was called: " << counter << " times";
9 }
10
11 template<>
12 void self_counter<0>;
13
14 int main()
15 {
16     //self_counter(); // won't work
17     self_counter<1>();
18     self_counter<1>();
19     self_counter<1>();
20     self_counter<2>();
21     self_counter<2>();
22     self_counter(); // Now without instansiate
23     return 0;
24 }
```

. Пример выше может не работать, Т.Ф. обещает рассказать об этом через несколько лекций.

## 6.3 Аргументы функции по умолчанию

В C++ можно создавать функции, у которых будут значения по умолчанию:

Код 6.8: Функция с параметрами по умолчанию

```
1 void foo(int x = 0, int y = 0, int z = 0)
2 {
3     std::cout << x << y << z << std::endl;
4 }
5 // Can be called in these ways
6 foo() // 000
```

```
7 | foo(1)                //100
8 | foo(1, 2)             // 120
9 | foo(1, 2, 3)          // 123
```

Единственная тонкость: значение по умолчанию пишется в *реализации* функции, а не в ее объявлении.

## 6.4 Односвязный список для любого типа данных

Теперь мы можем переписать **LinkedList** из прошлой лекции:

Код 6.9: LinkedList

```
1 | #include <iostream>
2 | #include <string>
3 |
4 | template<typename DataType>
5 | struct tNode
6 | {
7 |     DataType data;
8 |     tNode* next = nullptr;
9 | };
10 |
11 | template<typename DataType>
12 | tNode<DataType> * insert_node(tNode<DataType> *p_begin, DataType data)
13 | {
14 |     tNode<DataType> *p = new tNode<DataType>;
15 |     p->data = data;
16 |     p->next = p_begin;
17 |     return p;
18 | }
19 |
20 | int main()
21 | {
22 |     tNode<std::string> *p_begin = nullptr;
23 |
24 |     p_begin = insert_node(p_begin, std::string("Hello"));
25 |     p_begin = insert_node(p_begin, std::string("World"));
26 |     return 0;
27 | }
```

Эту реализацию LinkedList можно несколько усложнить, разделив на файлы: один будет хранить лишь LinkedList, а другой будем пользоваться готовым кодом из первого. Получится что-то вроде библиотеки для Linked List. Часть ее реализации представлена ниже. Код далее достаточно сложен, и многие его аспекты будут объяснены в следующих лекциях.



Код 6.10: Основная часть

```
1 #include <iostream>
2 #include <string>
3 #include "LinkedList.hpp"
4
5 int main()
6 {
7     LinkedList<std::string> str_list;
8     // LinkedList<int32_t> int_list; // Not used now
9     list_init(str_list);
10
11     list_insert(str_list, std::string("World"));
12     list_insert(str_list, std::string("Hello"));
13     list_insert(str_list, std::string("First"));
14     list_print(str_list);
15
16     list_destroy(str_list);
17
18     return 0;
19 }
```

Код 6.11: Хидер

```
1 // Added module
2
3 template<typename DataType>
4 struct tNode
5 {
6     DataType data;
7     tNode* next;
8 };
9
10 void list_init(List<T>);
11
12 void list_insert(List<T>, const T &value);
13
14 void list_print(List<T>);
15
16 void list_destroy(List<T>);
17
18 template<typename DataType>
19 tNode<DataType> * insert_node(tNode<DataType> *p_begin, DataType data);
```

Исходные коды всех программ можно найти [здесь](#)