Информатика: семестр 3

Программирование на C++

Конспекты лекций

Лектор: Хирьянов Т.Ф.

2 января 2018 г.

Оглавление

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

```
#include <iostream>

T absolute(T x)     // Will not work
{
    return (x > 0) ? x : -x;
}

int main()
{
    int a = -5;
    double b = -5.1;
    long c = -1L;

    std::cout << absolute(a) << std::endl;
    std::cout << absolute(b) << std::endl;  // different types
    std::cout << absolute(c) << std::endl;
}
```

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 под каждый из типов. В языке можно создать *шаблон* и компилятор будет знать, как под каждый из них делать функцию. Исправим код:

Код 6.4: Неопределенный тип с шаблоном

```
#include <iostream>

template<typename T>

T absolute(T x)     // That is fine
{
    return (x > 0) ? x : -x;
}

int main()
{
    int a = -5;
    double b = -5.1;
    long c = -1L;

    std::cout << absolute(a) << std::endl;
    std::cout << absolute(b) << std::endl;  // different types
    std::cout << absolute(c) << std::endl;
}
```

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 инстанцировать функцию:

Код 6.5: Явное инстанцирование функции

```
#include <iostream>

template<typename T>

T absolute(T x)     // That is fine
{
    return (x > 0) ? x : -x;
}

int main()
{
    std::cout << absolute<int32_t>(-8.0) << std::endl;
}
```

Здесь мы изменили вызов на **absolute<int32_t>**, тем самым указав тип, которым будет инстанцирована функция при компиляции.

Посмотрим, как ведет себя шаблон:

Код 6.6: Сложное инстанцирование

```
#include <iostream>

template<int number>
void self_counter()
{
    static int32_t counter = 0;
    counter++;
    std::cout << "Was called: " << counter << " times";
}

int main()
{
    //self_counter(); // --> No, it won't work
    self_counter<1>();
    self_counter<1>();
    self_counter<1>();
    self_counter<2>();
    self_counter<2>();
    return 0;
}
```

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). Обратите внимание, в записи **template<int number>** подразумевается, что инстанцирование будет по числам, то есть при указании разных чисел инстанцируются 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=0>  // Has default value 0
4  void self_counter()
5  {
6      static int32_t counter = 0;
7      counter++;
8      std::cout << "Was called: " << counter << " times" << std::endl;
9  }
10
11 int main()
12 {
13     self_counter<1>();
14     self_counter<1>();
15     self_counter<1>();
16     self_counter<2>();
17     self_counter<2>();
18     self_counter(); // --> Yeah, now without instansiate
19     return 0;
20 }
```

Здесь мы уже задали значение по-умолчанию в шаблон (строка **3**), поэтому вызов без инстанцирования (строка **18**) сработал.

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
```

```
foo(1)              //100
foo(1, 2)          // 120
foo(1, 2, 3)     // 123
```

6.4 Односвязный список для любого типа данных

Пользуясь полученными знаниями, мы теперь мы можем переписать **LinkedList** из прошлой лекции:

Код 6.9: LinkedList

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

template<typename DataType>
struct tNode
{
    DataType data;
    tNode* next = nullptr;
};

template<typename DataType>
tNode<DataType> * insert_node(tNode<DataType> *p_begin, DataType data)
{
    tNode<DataType> *p = new tNode<DataType>;
    p->data = data;
    p->next = p_begin;
    return p;
}

int main()
{
    tNode<std::string> *p_begin = nullptr;

    p_begin = insert_node(p_begin, std::string("Hello"));
    p_begin = insert_node(p_begin, std::string("World"));
    return 0;
}
```

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

```cpp
#include <iostream>
#include <string>
#include "linkedList.hpp"

int main()
{
    LinkedList<std::string> str_list;
    list_init(str_list);

    list_insert(str_list, std::string("World"));
    list_insert(str_list, std::string("Hello"));
    list_insert(str_list, std::string("First"));
    list_print(str_list);

    list_destroy(str_list);

    return 0;
}
```

Код 6.11: Хидер

```cpp
// Added module

template<typename DataType>
struct tNode
{
    DataType data;
    tNode* next;
};

void list_init(List<T>);

void list_insert(List<T>, const T &value);

void list_print(List<T>);

void list_destroy(List<T>);

template<typename DataType>
tNode<DataType> * insert_node(tNode<DataType> *p_begin, DataType data);
```

Исходные коды всех программ можно найти здесь