КОНСТРУИРОВАНИЕ ПРОГРАММ

Лекция № 15 — Перегрузка и шаблоны

Преподаватель: Поденок Леонид Петрович, 505a-5

+375 17 293 8039 (505a-5)

+375 17 320 7402 (ОИПИ НАНБ)

prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by/

Кафедра ЭВМ, 2021

2021.11.24

Оглавление

Передача аргументов по значению и по ссылке....3
Вопросы эффективности и постоянные ссылки....6
Перегрузка и шаблоны....8
Перегрузка функций....8
Шаблоны функций....11

Передача аргументов по значению и по ссылке

В языке C аргументы всегда передаются по значению. Это означает, что при вызове функции в функцию передаются копии значений аргументов, представленных параметрами функции в момент вызова.

```
int addition(int k, int m) {
    return k++ + m--;
}
...
int x = 5, y = 3, z;
z = addition(x, y);
```

В этом случае функции сложения передаются значения 5 и 3, которые являются копиями значений **x** и **y**, соответственно.

Эти значения (5 и 3) используются для инициализации переменных **k** и **m**, установленных в качестве параметров в определении функции, однако, любая модификация этих переменных внутри функции не влияет на значения переменных **x** и **y** вне ее, потому что **x** и **y** не передавались в функцию при ее вызове, а передавались их копии.

В некоторых случаях, однако, может быть полезно получить доступ к внешней переменной изнутри функции (превратив ее в процедуру).

Как вариант, можно передавать указатель на переменную, однако в C++ есть другой способ — аргументы могут быть переданы не по значению, а по ссылке.

Например, функция **duplicate** удваивает значение своих трех аргументов, в результате чего переменные, используемые в качестве аргументов, будут изменены:

```
// passing parameters by reference
#include <iostream>
using namespace std;

void duplicate(int& a, int& b, int& c) {

    a *= 2;
    b *= 2;
    c *= 2;
}

int main () {

    int x = 1, y = 3, z = 7;
    duplicate(x, y, z);
    cout << "x = " << x << ", y = " << y << ", z = " << z;
    return 0;
}
```

Вывод

```
x = 2, y = 6, z = 14
```

Чтобы получить доступ к своим аргументам, функция объявляет свои параметры как *ссылки*.

В C++ ссылки обозначаются амперсандом **&** после типа параметра, как в примере выше.

Когда переменная передается по ссылке, то та сущность, которая передается, не является копией, а самой переменной.

В результате любая модификация соответствующих локальных переменных внутри функция отражается в переменных, передаваемых в качестве аргументов в вызове.

Фактически, **a**, **b** и **c** становятся псевдонимами аргументов, передаваемых при вызове функции (**x**, **y**, **z**), и любое изменение их внутри функции фактически модифицирует переменные вне функции.

Любое изменение в переменных **a**, **b**, **c** приводит, соответственно, к изменению **x**, **y**, **z**.

Если бы вместо определения **duplicate** как:

```
void duplicate (int& a, int& b, int& c)
```

она была бы определена без знаков амперсанда как:

```
void duplicate (int a, int b, int c)
```

то переменные передавались бы не по ссылке, а по значению, при этом создавались бы копии их значений. В результате на выходе программы значения **x**, **y** и **z** остались бы без изменения (т.е. 1, 3, 7).

Вопросы эффективности и постоянные ссылки

Вызов функции с параметрами, взятыми по значению, приводит к созданию копий значений. Это относительно недорогая операция для фундаментальных типов, таких как **int**, но если параметр имеет большой составной тип, это может привести к определенным накладным расходам.

Например, рассмотрим следующую функцию:

```
string concatenate(string a, string b) { // <string> из STL
    return a + b;  //
}
```

Эта функция принимает в качестве параметров две строки (по значению) и возвращает результат их объединения. При передаче аргументов по значению, функция имеет дело с копиями строк **a** и **b**, которые передаются в функцию при ее вызове. И если это длинные строки, это может означать копирование большого количества данных только для вызова функции.

Но этой копии можно полностью избежать, если оба параметра являются ссылками:

```
string concatenate(string& a, string& b) { // <string> из STL
    return a + b;
}
```

Передача аргументов по ссылке не требуют копирования. Функция работает непосредственно со строками (с псевдонимами строк), передаваемыми в качестве аргументов, и, самое большее, это может означать передачу определенных указателей в функцию. В этом отношении версия **concatenate** со ссылками более эффективна, чем версия со значениями, поскольку ей не нужно копировать дорогостоящие для копирования строки.

С другой стороны, функции со параметрами, которые передаются по ссылке, обычно воспринимаются как функции, которые изменяют передаваемые аргументы, потому что именно для этого такие параметры фактически предназначены.

Здесь же совершенно другая ситуация — мы просто пытаемся быть эффективнее и возможность изменения аргументов нас беспокоит.

Решение проблемы состоит в том, чтобы функция гарантировала, что ее параметры, которые она ожидает получить по ссылке, не будут изменены этой функцией. Это можно сделать, указав параметры как **const**:

```
string concatenate (const string& a, const string& b) {
    return a + b;
}
```

Квалифицируя их как **const**, функции запрещается изменять значения как **a**, так и **b**, но функция может иметь доступ к их значениям в качестве ссылок (псевдонимов аргументов), не делая фактических копий строк.

Следовательно, **const** ссылки предоставляют функциональность, аналогичную передаче аргументов по значению, но с повышенной эффективностью в случае параметров больших по размеру типов. Поэтому они чрезвычайно популярны для определения аргументов составных типов.

Следует отметить, что для большинства основных типов заметной разницы в эффективности нет, а в некоторых случаях константные ссылки могут быть даже менее эффективными.

Перегрузка и шаблоны

Перегрузка функций

В C++ две разные функции могут иметь одно и то же имя, если их параметры различны. Различие состоит в том что, либо они имеют разное количество параметров, либо хотя бы один из них имеет другой тип.

Например:

```
#include <iostream>

int operate(int a, int b) {
  return (a * b);
}

double operate(double a, double b) {
  return (a / b);
}

int main () {

  int x = 5, y = 2;
  double n = 5.0, m = 2.0;
  std::cout << operate(x, y) << '\n'; // std::endl
  std::cout << operate(n, m) << '\n'; // std::endl
  return 0;
}
```

В этом примере есть две функции с именем **operate**, но у одной из них есть два параметра типа **int**, а у другой — два параметра типа **double**.

Компилятор знает, какой из них вызывать в каждом случае, изучая типы, передаваемые в качестве аргументов при вызове функции.

Если функция вызывается с двумя аргументами типа **int**, он вызывает функцию с двумя параметрами типа **int**, а если он вызывается с двумя значениями типа **double**, он вызывает функцию с двумя значениями типа **double**.

```
#include <iostream>

int operate (int a, int b) {
  return (a * b);
}

double operate (double a, double b) {
  return (a / b);
}

int main () {

  int x = 5, y = 2;
  double n = 5.0, m = 2.0;

  std::cout << operate(x, y) << '\n'; // std::endl
  std::cout << operate(n, m) << '\n'; // std::endl
  return 0;
}
```

В вышеприведенном примере обе функции ведут себя по-разному, версия **`int`** перемножает свои аргументы, а версия **`double`** их делит.

Это вообще не очень хорошая идея.

Обычно ожидается, что две функции с одинаковыми именами будут иметь, по крайней мере, похожее поведение, но этот пример демонстрирует, что вполне возможно этого не делать.

Две перегруженные функции (т.е. две функции с одинаковым именем) имеют совершенно разные определения, поэтому это разные функции, которые имеют одно и то же имя.

Следует обратить внимание, что функция не может быть перегружена только своим типом возврата. По крайней мере, один из ее параметров должен иметь другой тип.

Шаблоны функций

Перегруженные (overloaded) функции могут иметь одно и то же определение. Например:

```
#include <iostream>
using namespace std;

int sum(int a, int b) {
  return a + b;
}

double sum(double a, double b) {
  return a + b;
}

int main () {

  cout << sum(10, 20)   << '\n';
  cout << sum(1.0, 1.5) << '\n';

  return 0;
}
```

В данном случае функция **sum** перегружена разными типами параметров, но с одинаковым телом.

Функция **sum** может быть перегружена для большого количества типов, и для всех из них может быть смысл иметь одинаковое тело.

В таких случаях C++ предоставляет возможность определять функции с универсальными типами. Такие определения называются *шаблонами функций*.

При определении шаблона функции используется тот же синтаксис, что в случае определения обычной функции, за исключением того, что ему предшествует ключевое слово **template** и ряд параметров шаблона, заключенных в угловые скобки **<>**:

template < *template-parameters* > *function-declaration*

Параметры шаблона «*template-parameters*» представляют собой последовательность параметров, разделенных запятыми. Эти параметры могут быть обобщенными шаблонными типами, указанными с помощью ключевое слово **class** или **typename**, за которым следует идентификатор. Этот идентификатор после такого включения можно использовать в объявлении функции, как если бы он был обычным типом.

Например, общая функция **sum** может быть определена как:

```
template <class T>
T sum (T a, T b) {
  return a + b;
}
```

Не имеет значения, указан ли универсальный тип с ключевым словом **class** или ключевое слово в списке аргументов шаблона **typename**.

В приведенном выше коде объявление **T** (универсальный тип в параметрах шаблона, заключенных в угловые скобки) позволяет использовать **T** в любом месте определения функции, как и любой другой тип.

T может использоваться в качестве типа для параметров, в качестве типа возвращаемого значения или для объявления новых переменных этого типа.

Во всех случаях он представляет обобщенный тип, который будет определен в момент «воплощения» шаблона (instantiation).

Вызов нужного «воплощения»

«Воплощение» шаблона — это применение шаблона для создания функции с использованием определенных типов или значений для параметров шаблона.

Это делается путем вызова шаблона функции с тем же синтаксисом, что и при вызове обычной функции, но с указанием аргументов шаблона, заключенных в угловые скобки:

```
name <template-arguments> (function-arguments)
```

Например, шаблон функции суммы, определенный выше, может быть вызван с помощью:

```
x = sum<int>(10,20);
y = sum<double>(10,20);
```

Функция **sum<int>** является лишь одним из возможных воплощений шаблона функции **sum**. В этом случае, используя **int** в качестве аргумента шаблона в вызове, компилятор автоматически создает версию суммы, где каждое вхождение **T** заменяется на **int**, как если бы оно было определено как:

```
int sum(int a, int b) {
  return a + b;
}
```