Специализация шаблонов

17 апреля 2024 года

1 Специализация шаблонных классов

Пусть имеется шаблонный класс `S`:

```
template <typename T>
struct S
{
        T x = 1;
};
```

Будем считать, что представленное определение шаблонного класса справедливо для всех типов `T`, кроме некоторых типов, для которых мы дадим другое определение. `C++` позволяет выразить данную идею с помощью *специализации шаблонов*, пример которой представлен в листинге ниже:

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

template <typename T>
struct S
{
        T x = 1;
};

```

```
10 template <>
11 struct S<int>
12 {
13         int y = 2;
14 };
15
16 template <>
17 struct S<double>
18 {
19         int z = 3;
20 };
21
22 int main()
23 {
24         S<long int> s1;
25         S<int> s2;
26         S<double> s3;
27         cout << s1.x << endl;
28         cout << s2.y << endl;
29         cout << s3.z << endl;
30         return 0;
31 }
```

Вывод

```
1
2
3
```

Заметим, что все три типа, используемых в вышеприведенном примере, являются различными. Попытка выполнить инструкцию типа `s3=s2;` приведет к ошибке компиляции, т.к. все три типа с точки зрения `C++` никак друг с другом не связаны. Логика за таким рассуждением кроется в том, что получаемые посредством специализации типы могут принципиально отличаться от базового типа.

В качестве прикладного примера использования специализации рассмотрим хэш-функцию, которая реализована в стандартной библиотеке в виде объекта `std::hash`. У данного объекта есть функция `operator()`, которая хэширует входные данные. Тип хэшируемых данных является шаблонным параметром. Сам объект `std::hash` представляет собой только декларацию и не содержит объявления. Объявление содержится в специализированных классах. Схематически это можно выразить в следующем коде:

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

template<typename T>
struct MyHash;

template<>
struct MyHash<int>
{
        int operator()(int x)
        {
                return x;
        }
};

int main()
{
        MyHash<int> hsh;
        cout << hsh(2) << endl;
        return 0;
}
```

Понимая, как определяется объект `std::hash`, мы можем самостоятельно доопределить стандартную библиотеку так, чтобы объект-функция `std::hash` работал бы и для созданных нами типов данных:

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

```
3
4  struct MyType{};
5
6  namespace std
7  {
8          template<>
9          struct hash<MyType>
10         {
11                 int operator()(MyType x){return 1;}
12         };
13 }
14
15 int main()
16 {
17         hash<MyType> hsh;
18         cout << hsh(MyType()) << endl;
19         return 0;
20 }
```

Вывод

1

Если класс зависит от нескольких шаблонных параметров, то возможна *частичная специализация*, в рамках которой мы фиксируем только часть параметров, либо накладываем какие-либо ограничения на параметры. При выборе конкретной реализации класса компилятор руководствуется принципом «частное лучше, чем общее». Следующий пример проясняет эту идею:

```
1  #include <iostream>
2  using namespace std;
3
4  template <typename T, typename U>
5  struct S
6  {
7          void f()
```

```
8           {
9                   cout << "General\n";
10          }
11  };
12
13  template <typename T>
14  struct S<T, T>
15  {
16          void f()
17          {
18                  cout << "Coincident\n";
19          }
20  };
21
22  template <typename T, typename U>
23  struct S<T &, U &>
24  {
25          void f()
26          {
27                  cout << "Reference\n";
28          }
29  };
30
31  int main()
32  {
33          S<int, double> s1;
34          s1.f();
35          S<int, int> s2;
36          s2.f();
37
38          //S<int&, int&> s3;
39          //s3.f();
40
41          S<int&, double&> s4;
42          s4.f();
```

```
43          S<int&, double> s5;
44          s5.f();
45          return 0;
46  }
```

Результат работы программы

```
General
Coincident
Reference
General
```

Пояснения

Если раскомментировать строки 38 и 39, то возникнет ошибка компиляции: компилятор не сможет выбрать, какой из шаблонов `S<T&, U&>` или `S<T, T>` использовать, какой из них более «специализированный».

Что же касается объекта `s5`, то используется общий шаблон по причине того, что один из шаблонных аргументов не является ссылкой, следовательно, использовать последнюю частичную специализацию со ссылками не получится.

2 Специализация и перегрузка шаблонных функций

Функции можно специализировать также, как и классы, с тем исключением, что стандарт не разрешает частичную специализацию функций.

Рассмотрим ситуацию перегрузки шаблонных функций, имеющих к тому же специализации. При выборе конкретной функции для выполнения компилятор прежде всего руководствуется следующим правилом: выбирать наиболее подходящую функцию, и только затем смотреть на ее специализации (если они есть) с целью найти самую

подходящую. Иными словами, если у нас три перегруженные функции, у каждой из которых по три специализации, то компилятор на первом этапе выберет наиболее подходящую функцию из трех (не из 9!), а на втором этапе рассмотрит все три специализации выбранной функции, найди среди них самую подходящую.

Под словом «подходящая» понимается выполнение следующих двух правил:

1. Чем меньше кастов, тем лучше. Точное соответствие лучше, чем приведение типов.

2. Частное лучше, чем общее.

Данные правила проверяются «сверху вниз», то есть сначала проверяется выполнение первого правила, и только если несколько функций одинаково хорошо удовлетворяют ему, то компилятор смотрит на второе правило.

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

template <typename T, typename U>
void f(T x, U y)
{
        cout << "General\n";
}

template<>
void f(int x, int y)
{
        cout << "Specialization\n";
}

template<typename T>
void f(T x, T y)
{
```

```
19          cout << "Not so general\n";
20  }
21
22  int main()
23  {
24          f(0, 0);
25          return 0;
26  }
```

Результат работы программы

```
Not so general
```

Логика компилятора

В программе фигурируют две функции, одна из которых обладает к тому же специализацией. Сперва нужно выяснить, какая из двух функций (в строке 16 или в строке 4) нам подходит. По первому правилу обе функции нас полностью устраивают, т.к. не требуют приведений типов. Поэтому переходим к пункту 2. Шаблонная функция в строке 16 более частная по сравнению с шаблонной функцией в строке 4. Следовательно, ее и выберем.

Изменим немного рассмотренный пример: поставим специализацию первой шаблонной функции после определения второй шаблонной функции, т.е.:

```
1   template <typename T, typename U>
2   void f(T x, U y)
3   {
4           cout << "General\n";
5   }
6
7   template <typename T>
8   void f(T x, T y)
9   {
10          cout << "Not so general\n";
```

```
11 }
12
13 template <>
14 void f(int x, int y)
15 {
16         cout << "Specialization\n";
17 }
```

Тогда программа выведет слово `Specialization`. Дело в том, что компилятор рассматривает специализацию в строке 13 как специализацию функции, которая окажется наиболее подходящей для данного вызова, из числа тех, что объявлены выше объявления специализации. Наиболее подходящей по-прежнему является функция `f(T x, T y)`, но теперь специализация помещена ниже ее объявления, и поэтому расценивается как специализация именно это функции.