Шаблоны функций, специализация

Степанюк Константин Сергеевич
24const@gmail.com

Шаблоны функций

```
template <class T> void sort(vector<T>&);

void f (vector<int>& vi, vector<string> &vs){
     sort< int >(vi);    //sort(vector<int>&)
      sort(vs);          //sort(vector<string>&)
}
```

- При вызове шаблона функции аргументы могут однозначно определять какая версия шаблона используется. В этом случае говорят что аргументы шаблона функции выводятся по аргументам функции
- Компилятор может вывести аргументы как являющиеся типами так и обычные при условии, что список аргументов функции однозначно идентифицирует набор аргументов шаблона

Пример

```
template <class T, int i> T& lookup(Buffer<T,i>&b, const char *p);
class Record {
        const char[12];
        //…
};
Record& f(Buffer<Record, 128> &buf, const char *p)
{
        // вызвать lookup(), где T-Record, i -128
        return lookup(buf,p);
}
//Иногда требуется указать аргумент явно:
template <class T> T* create();
void f() {
        vector<int> v;
        int *p = create<int>();
}
```

Перегрузка шаблонов функций

```
template <class T> T sqrt(T);
template <class T> complex<T> sqrt( complex<T>);
double sqrt(double);

void f(complex<double> z){
    sqrt(2);      //sqrt<int> (int)
    sqrt(2.0);    //sqrt(double)
    sqrt(z); //sqrt<double> (complex<double>)
}
```

Правила разрешения перегрузки

1. Ищется набор специализаций шаблонов функций, принимается решение какие аргументы были бы использованы, если бы в текущей области видимости не было других шаблонов функций и обычных функций с тем же именем (в примере –sqrt<double>(complex<double>) и sqrt<complex<double>>(complex<double>)

2. Если могут быть вызваны два шаблона функции и один из них более специализирован чем другой, на последующих этапах только он и рассматривается.(Для нашего примера предпочтение отдается sqrt<double>(complex<double>) по отношению к sqrt<complex>(complex))

Правила разрешения перегрузки

3. Разрешается перегрузка для данного набора функций, а также для любых обычных функций (в соответствии с правилами разрешения перегрузки для обычных функций). Если аргументы функции шаблона были определены путем выведения по фактическим аргументам шаблона, к ним нельзя применять «продвижение», стандартные и определяемые пользователем преобразования типа. (в нашем примере sqrt<int>(int) является точным соответствием и поэтому ей отдается предпочтение по отношению к sqrt(double)
4. Если и обычная функция и специализация подходят одинаково хорошо, то предпочтение отдается обычной функции (sqrt(double) а не sqrt<double>(double)

Правила разрешения перегрузки

5. Если ни одного соответствия не найдено или процесс оканчивается нахождением двух или более одинаково хорошо подходящими вариантами, то выдается соответствующая ошибка компиляции

```
template<class T> T max (T, T);
const int s = 7;
void doIt() {
        max(1,2);       // max<int>(1,2)
        max('a','b');   // max<char>('a','b')
        max(2.7,4.9);       // max<double> (2.7,4.9)
        max(s,7);       //max<int>(int(s), 7) –тривиальное преобр.
        max('a',1);     //error –неоднозначность, стандартные
                        //преобразования не применяются
        max(2.7,4);     //error –неоднозначность
}
```

Разрешение неоднозначности

- Явная квалификация:

```
max<int>(‘a’,1);            // max (int(‘a’), 1)
max<double>(2.7,4)          // max(2.7, double(4))
```

- Добавление подходящих объявлений

```
inline int max (int i, int j) {return max<int>(i, j);}
inline double max (double d1, double d2) {
        return max<double>(d1, d2);
}
inline double max (int i, double d) {
        return max<double>(i, d);}
inline double max (double d, int i) {
        return max<double>(d, i);}
void f () {
        max(‘a’,1);             //max (int ('a'), 1)
        max(2.7, 4);            //max (2.7, double (4))
}
```

Перегрузка и наследование

- Правила перегрузки гарантируют, что функции шаблоны корректно взаимодействуют с наследованием:

```
template <class T> class B {/*…*/};
template <class T> class D: public B<T> {/*…*/};
template <class T> void f( B<T>* );
void g (B<int> *pb, D<int> *pd) {
    f(pb);   //f <int>(pb)
    f(pd);   //f <int>(static_cast<B<int>*>(pd));
}
```

Дополнительные аспекты разрешения

- Аргумент функции, не использующийся при выведении параметра шаблона рассматривается точно также, как аргумент функции, не являющейся шаблоном, и к нему применяются обычные правила преобразования для аргумента функции при перегрузке обычных функций:

```
template <class C> int get_nth (C& p, int n);
class Index {
public:
      operator int();
};
void f (vector<int> &v, short s, Index i) {
      int i1 = get_nth (v, 2);  //точное соответсвие
      int i2 = get_nth (v, s);  //short в int
      int i3 = get_nth (v, i);   //Index в int
}
```

Использование аргументов шаблона для выведения алгоритма

```
template <class T, class C=Cmp<T> >
        int compare (const String<T>& s1, const String<T>& s2) {
            for (int i=0; i<s1.length() && i<s2.length(); i++)
                if(!C::eq(s1[i], s2[i]))
                    return C::lt(s1[i], s2[i]) ? -1 : 1;
        return s1.length() – s2.length();
}
//Определяем класс для правил сравнения
template <class T> class Cmp {
public:
        static int eq (T a, T b) { return a==b; }
        static int lt (T a, T b) { return a < b;}
};
String<char> str1, str2;
//compare <char, Cmp<char>> (str1, str2);
compare (str1,str2);
compare<wchar, MyCmp<wchar> > (wstr1, wstr2);
```

Специализация

- По умолчанию шаблон предоставляет единое определение генерируемого типа которое используется для всех возможных аргументов шаблона
- Но иногда может возникнуть необходимость в уточнении определения для определенных категорий аргументов, например:
  - «Если аргументом является указатель, то используй эту реализацию, если нет –используй ту»
  - «Выдай сообщение об ошибке если аргументом шаблона не является указатель на объект класса, производного от My_base»
- Многие подобные проблемы решаются обеспечением альтернативных определений шаблона , выбор которых при инстанцировании осуществляет компилятор на основании аргументов шаблона указанных при его использовании

Класс Vector – кандидат на специализацию

```
template <class T> class Vector {
        T* v;
        int sz;
public:
        Vector();
        Vector(int);
        T& elem(int i) {return v[i];}
        T& operator[] (int i);
        void swap(Vector&);
};
Vector<int> vi;
Vector<Shape*> vps;
Vector<string> vs;
Vector<char*> vpc;
Vector<Node*> vpn;
```

Специализация Vector<void*>

```
template <> class Vector <void*> {
        void **p;
        int sz
public:
        Vector();
        Vector(int);
        void* &elem(int i) {return p[i];}
        void* &operator[] (int i) {return elem(i);}
        void swap(Vector&);
        //…
};
```

- Полная специализация –отсутствует параметр шаблона, который бы следовало задавать или который бы выводился при инстанцировании при использовании специализации

Частичная специализация

```
template <class T> class Vector <T*> : private Vector<void*> {
public:
        typedef Vector<void*> Base;
        Vector():Base() {}
        explicit Vector(int): Base(i) {}
        T* &elem(int i) {
                return static_cast<T*&> (Base::elem());
        }
        T* &operator[] (int i) {
                return static_cast<T*&> (Base::operator[](i));
        }
        //…
};
Vector<Shape*> vps;       //<T*> -это <Shape*>, T –shape
Vector<int**> vppi; //<T*> -это <int**>, T –int*
```

Правила объявлений для специализаций

- Общий шаблон должен быть объявлен прежде любой специализации
- Если программист специализирует где-нибудь шаблон, то эта специализация должна быть в области видимости при каждом использовании шаблона с типом, для которого он был специализирован
- Все специализации шаблона должны быть объявлены в том же пространстве имен что и сам шаблон
- Одна специализация считается более специализированной чем другая если каждый список аргументов соответствующий образцу первой специализации соответствует и второй специализации, но не наоборот
- Более специализированной версии будет отдаваться предпочтение в объявлениях объектов, указателей, а также при разрешении перегрузки

Пример, специализация функций

```
//общий шаблон
template<class T> class Vector;
//частичная специализация
template <class T> class Vector<T*>;
//полная специализация
template <> class Vector<void*>;

template <class T> bool less (T a, T b) {return a<b;}

//Для функций поддерживается только полная специализация
template<> bool less<> (const char *a, const char *b) {
        return strcmp(a,b)<0;
}
//Вторые пустые <> можно опустить:
template<> bool less(const char *a, const char *b) {
        return strcmp(a,b)<0;
}
```

Наследование и шаблоны

- Наследование реализации и обеспечение типобезопасного использования, например:

```
template <class T> class Vector <T*> : private Vector<void*>{/*…*/};
```

- Уточнение существующего шаблона (класса) или обобщение набора операций, например:

```
template <class T> class CheckedVector : public Vector<T>{/*…*/};
template <class C> class Basic_ops {
public:
      bool operator==(const C&) const;
      bool operator!=(const C&) const;
    const C& derived() {return static_cast<C&>*this;}
};
```

Пример использования

```
template <class T> class Math_container :public Basic_ops<
    Math_container<T> > {
public:
      size_t size() const;
      T& operator[] (size_t);
      const T& operator[](size_t) const;
      //…
};
template <class C>
bool Basic_ops<C>::operator==(const C& a) const {
      if (derived().size() != a.size()) return false;
      for (int i = 0; i<derived().size(); ++i)
            if (derived()[i]!=a[i]) return false;
      return true;
}
```

Организация исходного кода

- Ранние компиляторы:
  - Объявления и определения шаблонов размещаются в заголовочном файле
  - Заголовочный файл включается во все единицы трансляции где используются данные шаблоны
- Современные компиляторы:
  - Заголовочный файл включает только объявления шаблона, определение иго структуры и его специализаций (без определения членов шаблона)
  - Определения членов шаблона размещаются в некоторой единице компиляции и предваряются ключевым словом export по аналогии со встраиваемыми (inline)функциями

Примеры кода

- Первый вариант

```
//file out.h :
#include <iostream>
template <class T> void out (const T &t) {std::err<<t;}
//file user1.cpp –включается и iostream
#include ”out.h”
```

- Второй вариант

```
//file out.h :
template <class T> void out (const T&);
//file out.cpp
#include <iostream>
export template <class T> void out(const T &t) {…}
//file user2.cpp – отсутствует зависимость от iostream
#include “out.h”
```

Вопросы

- Какая лекция\тема понравилась Вам больше всего?
- Удовлетворяет ли Вас качество организации и проведения занятий? Как вы считаете, что следует изменить для улучшения качества усвоения материала?
- Планируете ли Вы в дальнейшем заниматься в сфере информационных технологий?