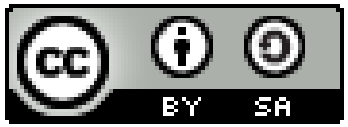
Шаблоны C++

Абстрактное программирование



May 24, 2016

Абстрактное программирование

Абстрактные алгоритмы

Существуют алгоритмы, действия которых не зависят от конкретного типа данных, есть только требования на возможные действия над объектами. Например, любой алгоритм сортировки требует, что бы объекты можно было сравнивать (<) и менять местами (копировать).

Абстрактные типы данных

Существуют типы данных (чаще всего контейнеры), интерфейс которых не зависит от конкретного вложенного типа данных. Например, и у списка собак, и у списка астероидов одинаковый набор действий: добавить объект, удалить объект, выдать первый объект, выдать следующий ...

Необходимость абстрактного программирования

Без поддержки языком методов абстрактного программирования программисту пришлось бы:

- Для каждого используемого типа данных дублировать код алгоритма
- Писать множество классов, которые отличаются только типом содержащихся данных.
- Как вариант – потребовать, что бы каждый тип данных, содержащийся в контейнере, был наследником какого-то “всеобщего предка”.
- В случае обнаружения ошибки в коде, её надо исправлять сразу в нескольких местах.
- Писать код для всех типов данных, для которых **может** потребоваться алгоритм, а не только для тех, для которых он в самом деле нужен.

Использование макросов как в языке C

Древний подход к абстрактному программированию

```
#define MMIN(x,y) (((x)<(y))?(x):(y))
int getint()
{
  int a, rc;
  rc = scanf( "%d", &a );
  return ( rc == 1 ) ? a : 0;
}
int main()
{
  int a = 5, b = 3, c = MMIN(a,b); // c == 3
  double t = 4.7, u = -5.1, v = MMIN(t,v);
  c = MMIN(getint(), getint()); // вызвана 3 раза
  c = MMIN( a++, b++ );  // a == 6, b == 5
  Tank w("T-80"); Student s("Petya", 321 );
  MMIN( w, s );  MMIN( s, 12 );
  return 0;
}
```

Достоинства и недостатки макросов

Достоинства

- Код не дублируется.
- Нет затрат на вызов функции.

Недостатки

- Дублирование побочных действий.
- Нет контроля типов.
- Нет возможности корректно использовать локальные объекты для алгоритма – можно написать только очень простые алгоритмы.
- Компилятор не видит исходного кода – у него нет возможности дать понятное описание ошибки.

Метод старого C++ (без шаблонов)

```
inline int mmin( int x, int y )
  { return (x<y) ? x : y; }
inline double mmin( double x, double y )
  { return (x<y) ? x : y; }
inline const Stud& mmin( const Stud &x, const Stud &y )
   { return (x<y) ? x : y; }
int main()
{
  int a = 5, b = 3, c = mmin(a,b); // c == 3
  double t = 4.7, u = -5.1, v = mmin(t,v);
  c = mmin(getint(), getint()); // вызвана 2 раза
  c = mmin( a++, b++ );  // a == 6, b == 4
  Student s("Petya", 321 ), w("Vasya",54 );
  mmin( w, s ); /* ok */  mmin( a, w ); /* error */
  return 0;
}
```

Достоинства и недостатки функций

Достоинства

- Нет дублирование побочных действий.
- Полный контроль типов.
- Можно использовать любые локальные объекты – можно написать любые простые алгоритмы.
- Компилятор видит исходный кода – у него есть возможность дать понятное описание ошибки.
- Если функция простая, её можно объявить inline – нет затрат на вызов простых функций.

Недостатки

- **Код дублируется**.

```
// шаблон функции (или шаблонная функция)
template<typename T>      // T - параметр шаблона
inline const T& mmin( const T &x, const T &y )
   { return (x<y)? x : y; }

int main()
{
  int a = 5, b = 3, c = mmin(a,b);
  //                      ^– точка инстанцирования
  //                       шаблона mmin при T - int
  double t = 4.7, u = –5.1, v = mmin(t,v);
  //                              ^ – точка
  //   инстанцирования шаблона mmin при T - double
  c = mmin( a++, b++ ); //используется уже созданная
  Student s("Petya", 321 ), w("Vasya",54 );
  mmin( w, s ); // инстанцирование при T - Stud
  mmin( a, w ); // error: T - int или T - Stud ?
  return 0;
}
```

Особенности использования шаблонов функций

- Создание функции из шаблона (**инстанцирование**) происходит только в том случае, если не нашлось подходящей обычной функции (уровень 6 при выборе перегруженной функции).
- Шаблон функции – не функция, его самого невозможно откомпилировать в объектный код. При инстанцировании и вызове полученной функции компилятор должен видеть определение шаблона. Поэтому шаблоны функций обычно целиком находятся в заголовочных файлах.
- Параметрами шаблона могут быть любые типы (фундаментальные, пользовательские, производные), если тип поддерживает все необходимые для шаблона действия.
- Если это имеет смысл, то параметром шаблона может быть целочисленная константа, указатель на объекты с внешним связыванием или другой шаблон.

Вывод параметров шаблонов функций

При инстанцировании шаблонов функций преобразование типов не производится, совпадение должно быть **точным**.

```
int a = 5; double b = 1.7, c;
c = mmin( a, b ); // ошибка, T - int или T - double
c = mmin( (double(a), b ); // можно, T - double
```

При необходимости, параметр шаблона можно указать вручную, используя **явное инстанцирование**.

```
c = mmin<double>( a, b );
int d = mmin<int>( a, b );
```

Перегрузка шаблонов функций

Шаблоны функций могут быть перегружены, и соседствовать с просто перегруженными функциями.

```
inline int mmax( int a, int b )
  { return ( a > b ) ? a : b; }
template<typename T>
inline const T& mmax( const T& a, const T& b )
  { return ( a > b ) ? a : b; }
inline const T& mmax( const T& a )
  { return a; }

int main()
{
  cout << "mmax(␣4,␣7␣)="      << mmax( 4,   7   ) << endl;
  cout << "mmax(␣1.5,␣0.9␣)=" << mmax( 1.5, 0.9 ) << endl;
  cout << "mmax(␣7␣)=" << mmax( 7 ) << endl;
  return 0;
}
```

Отложенное определение типа возвращаемого значения

Если в момент начале описания функции тип возвращаемого значения неизвестен, то его определения можно отложить с помощью **auto** и **decltype**:

```
template< class T, class U >
auto sum( T x, U y ) -> decltype(x+y)
{
  return x+y;
}
```

Эта возможность появилась в C++11.

Шаблоны классов
Синтаксис объявления

Создадим описание шаблонного класса (или шаблона класса Stack). Этот контейнер будет содержать объекты типа T.

```
template <typename T>
class Stack {
  public:
    Stack();
    Stack( const Stack<T> &r );
    ~Stack();
    void push( const T& obj );
    T pop();
    bool is_empty() const { return top == 0 };
  ...
};
```

При написании функций-членов и функций-статических членов внутри шаблона не надо повторять ключевое слово **typename**.

Определение функций шаблона класса вне его определения

При написании функций-членов и функций-статических членов снаружи шаблона надо использовать ключевое слово **typename** для указания того, что это не просто функция, а шаблон, зависящий от параметров.

```
template <typename T>
Stack<T>::Stack()
{ /* тело конструктора */ }

template <typename T>
Stack<T>::~Stack()
{ /* тело деструктора */ }

template <typename T>
void Stack<T>::push( const T& obj )
{ /* тело функции push */ }
```

Дружественные функции шаблона класса

Если дружественная функция шаблона класса зависит от того же параметра, что и сам шаблон, то для её описания и определения используется такой синтаксис:

```
template<typename T> class Tc; // предв. декл.
template<typename T> int N( Tc<T> z );

template<typename T> class Tc {
  public:
   Tc(T xx) : l(12), x(xx) { };
   friend int N<>( Tc<T> z ); // пустые скобки <>
     // обозначают, что N зависит от того же параметра
  private:
   int l;
   T x;
};

// определение шаблона дружественной функции шаблона
template<typename T> int N( Tc<T> z ) { return z.l; };
```

Инстанцирование шаблонов класса

В отличие от функций, инстанцирование шаблонов классов происходит только явно:

```
int main()
{
  Stack<int> si;// точка инстанцирования
                // Stack<T> для int
  Stack<double> sd; // ... для double
  Stack<Stud> sstd; // ... для Stud
  Stack<Stack<int> > xx; // ... для Stack<int>

  si.push(54);  sd.push(0.5);
  sstd.push( Stud("Katya", 17 ) );
  cout << si.pop() << ' ' << sd.pop << endl;
  return 0;
}
```

Инстанцирование происходит только для тех типов, и для тех функций, которые используются.

Специализация шаблонов класса

Если для какого-то типа T надо задать другое поведение, то можно использовать **специализацию шаблона**.

```
template<>
class Stack<int> {
   ...
};  //   полная специализация класса Stack
   //    для случая, если T - int
```

Для полных специализаций функции-члены пишутся как обычные функции-члены, заменяя T на тип специализации.

```
void Stack<int>::push( int& obj )
{
   ...
}
```

Частичная (неполная) специализация

Если у шаблона класса несколько параметров, или необходимо использовать неполную специализацию

```
template<typename T>
class Stack<T*> {
   ...
}; // неполная специализация класса Stack
   //   для случая, если параметр - указатель
int main()
{
   Stack<int> a; // используется полная спец.
   Stack<int*> xa; // ... частичная
   Stack<Stud*> spa; // ..частичная
   Stack<Stud> z; // общий случай
   return 0;
}
```

Если одинаково хорошо подходят две неполные специализации – ошибка.

Аргументы по умолчанию для шаблонов

Несколько последних параметров шаблонов могут иметь значение по умолчанию.

```
template< typename T, typename L = int >
class Vector {
  ...
};

int main()
{
  Vector<double> a; // используется
                    // Vector<double,int>

  Vector<int,long> z; // задано явно
  return 0;
}
```

Variadic templates

В C++11 можно задавать неопределённое кол-во параметров шаблона

```
template<class T>
void print_lst( T x )
{
  cout << x << endl;
}

template<class First, class ...Rest>
void print_lst( First f, Rest ...r )
{
  cout << f << ',';
  print_lst( r... );
}

print_lst( 38, "ptnhlo", 'Z', 3.14 );
```

Значительная часть стандартной библиотеки языка C++ – STL (Standard **Template** Library) основана на шаблонах. Это используется как явно,

```
vector<int> a;
map<double,int> mp;
```

так и неявно:

```
string s;
// на самом деле 'string' может быть определён так:
typedef basic_string<char,char_traits<char>,
                     Allocator<char> > string;
```