C++编程(11)

唐晓晟
北京邮电大学电信工程学院

第13章 模板

- 引言
- 一个简单的string模板
- 函数模板
- 用模板参数描述策略
- 专门化
- 派生和模板
- 源代码组织
- 忠告

13.1 引言

- 模板提供了一种很简单的方式来表述范围广泛的一般性概念，以及一些组合它们的简单方法，利用模板产生出来的类和函数，在运行时和空间效率方面，可以与手工写出的更特殊的代码媲美
- 模板直接支持采用类型作为参数的程序设计
- 这里对于模板的介绍将集中关注与标准库的设计、实现和使用有关的各种技术

引言

- 标准库要求的时更高水平的通用性、灵活性和效率
- 每个主要的标准库抽象都被表示为一个模板(string, ostream, complex, list和map等)，所有的关键操作(string比较、输出运算符<<、complex的加法、取得list的下一个元素、sort()等)也是如此
- 下面通过一些小例子阐述模板的各种基本技术

引言

- 13.2 定义和使用类模板的基本机制
- 13.3 函数模板，函数重载和类型推断
- 13.4 用模板参数刻画通用型算法的策略
- 13.5 通过多个定义为一个模板提供多种实现
- 13.6 派生和模板(运行时的和编译时的多态性)
- 13.7 源代码组织

13.2 一个简单的String模板

- 串是一个能够保存一些字符的类，而且提供了各种串的基本操作
- 实际应用中，我们可能希望为多种不同种类的字符(有符号字符、无符号字符、中文字符、希腊字符)提供这样的功能，因此希望能够以最不依赖于特定字符种类的方式给出串的概念
- 因此可以将以前的char类型的串抽象为一个更具普遍性的串类型

一个简单的String模板

```
template<class C> class String
{
    struct Srep;
    Srep* rep;
public:
    String();
    String(Const C*);
    String(const String&);

    C read(int i) const;
    //
};
```

template<class C>前缀说明当前正在声明的是一个模板，它有一个将在声明中使用的类型参数C

C的作用域将一直延伸到由这个template<class C>作为前缀的声明的结束处

C代表了一种类型，不必一定是某个类的名称

示例

```
String<char> cs;
String<unsigned char> us;
String<wchar_t> ws;
class Jchar{
    // …
};
String<Jchar> js;
```

```
// 统计输入各个单词出现的次数
int main()
{
    String<char> buf;
    map<String<char>, int> m;
    while(cin>>buf) m[buf]++;
}
// 日文版本
int main()
{
    String<Jchar> buf;
    map<String<Jchar>,int> m;
    while(cin>>buf) m[buf]++;
}
```

示例

```
// 标准库提供了一个模板类basic_string，其中，string被定义为
basic_string<char>的同义词
typedef basic_string<char> string;

int main()
{
    string buf;
    map<string, int> m;
    while(cin>>buf) m[buf]++;
}
```

13.2.1 定义一个模板

- 一个由类模板生成的类也是一个完全正常的类
- 方法：先做好特定的类，如String，并排除其中错误，然后再将它转化为String<C>一类的模板
- 模板类中成员的声明和定义与在非模板类里完全一样，模板类的成员也不一定非要在类内部定义，可以在外部定义，成员本身也是模板参数化的

定义一个模板示例

```
template<class C>
class String
{
    struct Srep;
    Srep* rep;
public:
    String();
    String(Const C*);
    String(const String&);

    C read(int i) const;
    //…
};
```

```
template<class C> struct
String<C>::Srep {
    C* s;
    int sz;
    int n;
};

template<class C> C
String<C>::read(int i) const {
    return rep->s[i];
}

template<class C> String<C>::String()
{
    rep = new Srep(0,C());
};
```

定义一个模板示例

类模板的名字不能重载

```
template<class T> class String{/*…*/};

class String{/*…*/}; // 错误，重复定义
```

13.2.2 模板实例化

- 从一个模板类和一个模板参数生成一个类声明的过程通常被称为模板实例化
- 该说法同样用于由模板函数加上模板参数产生一个函数的过程
- 针对一个特定模板参数的模板版本被称为是一个专门化

模板实例化

- 一般来说，具体实现负责保证对所用的每组模板参数能够生成模板函数的相应版本，程序员不必考虑该问题
- 下例中，实现应该生成String<char>和String<Jchar>的声明、与它们对应的Srep类型、构造和析构函数、以及赋值String<char>::operator=(char*)，其他的成员函数没有被使用，也就不应该生成

```
String<char> cs;
String<Jchar> js;
cs = " It's the implementation's job to figure out what
code needs to be generated.";
```

13.2.3 模板参数

- 模板的参数可以有常规类型参数如int，也可以有模板参数，一个模板也可以有多个参数，例如：一个模板参数可以用于定义跟随其后的模板参数

```
template<class T, T def_val> class Cont{/*…*/};
```

- 整数模板参数常常用于提供大小或者界限，必须为常量

模板参数示例

```
template<class T, int i> class Buffer
{
    T v[i];
    int sz;
public:
    Buffer():sz[i]{}
};
Buffer<char,127> cbuf;
Buffer<Record,9> rbuf;
```

// 当运行效率和紧凑性特别重要时，可以使用Buffer这类容器，将数组大小作为参数传递，可以使Buffer的实现避免使用自由存储(例如vector或者string)

模板参数可以使常量表达式，具有外部连接的对象或者函数的地址，或者非重载的指向成员的指针，用作模板参数的指针必须具有&of的形式，其中of是对象或者函数的名字；或者具有f的形式，f必须是一个函数名，到成员的指针必须具有&X::of的形式，这里的of是一个成员名

13.2.4 类型等价

- 给出一个模板，可以为它提供模板参数，以生成一些类型

```
String<char> s1;
String<unsigned char> s2;
Sring<int> s3;
```

对于同一模板使用同一组模板参数，将总是表示同一个生成出的类型

```
typedef unsigned char Uchar;
String<Uchar> s4;
String<char> s5;
```

typedef并不引入新的类型，所以String<Uchar>和String<unsigned char>是同样的类型(同一组模板参数)

```
Buffer<String<char>,10> b1;
Buffer<char,10> b2;
Buffer<char,20-10> b3;
```

编译器能对常量表达式求值，左边10和20-10是同样的类型

13.2.5 类型检查

- 模板定义时，要检查这个定义的语法错误，编译器可以帮助捕捉一些简单的语义错误
- 与模板参数的使用有关的错误在模板使用之前不可能检查出来
- 与模板参数相关的错误能被检查出来的最早位置，也就是在这个模板针对该特定模板参数的第一个使用点，被称为第一个实例化点，简称实例化点

类型检查示例

```
template<class T> class List {
    struct Link { Link* pre;  Link* suc; T val;
        Link(Link* p, Link* s, const T& v) : pre(p), suc(s), val(v) {}
    } // 错误
    Link* head;
public:
    List() : head(7) {} // 错误
    List(const T& t):head(new Link(0,o,t)) {} // 错误
    void print_all() const {
        for(Link* p=head;p;p=p->suc)
            cout << p->val << '\n';
    }};
class Rec{/*…*/};
void f(const List<int>& li, const List<Rec>& lr){
    li.print_all();    // ok            lr.print_all();
} // 错误，Rec没有定义<<运算符号
```

13.3 函数模板

- 一般在应用了容器类(basic_string, vector, vector, list, map)后，就会提出对于模板函数的要求
- 在调用模板函数时，函数参数的类型决定到底应使用模板的哪个版本

函数模板示例

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

template<class T> void sort(vector<T>& v)
{ // Shell sort(Knuth, Vol. 3, pg.84)
  const size_t n = v.size();
  for(int gap=n/2; 0<gap; gap/=2)
    for(int i=gap; i<n; i++)
      for(int j=i-gap; 0<=j; j-=gap)
          if(v[j+gap]<v[j]) swap(v[j], v[j+gap]);
          // 注意，用<做比较运算，并非所有类型都支持该运算符
}
```

13.3.1 函数模板的参数

- 模板函数对于写出通用型算法是至关重要的，这种算法可以应用于广泛且多样化的容器类型
- 对于一个调用，能从函数的参数推断出模板参数的能力是其中最关键的

函数模板的参数示例

- 编译器能够从一个调用推断出类型参数和非类型参数，条件是由这个调用的函数参数表能够唯一地标识出模板参数的一个集合

```
template<class T, int i> T& lookup(Buffer<T,i>& b, const char* p);
class Record{
  const char v[12];
  // ...
};
Record& f(Buffer<Record,128>& buf, const char* p)
{
  return lookup(buf,p); // 使用lookup()，其中T是Record, i是128
}
```

函数模板的参数示例

- 如果不能从模板函数的参数推断出某个模板参数，那么就必须显式地去描述它

```
template<class T> class vector{/*...*/};
template<class T> T* create();
// 返回一个指向T的指针
void f()
{
  vector<int> v;
  int* p = create<int>(); // 注意create后面的<int>参数
}
```

函数模板的参数示例

- 显式描述的最常见用途是为模板函数提供返回值类型
- 可以将implicit_cast函数理解为隐式转换的一个显式版本

```
template<class T, class U> T implicit_cast(U u) {return u;}
void g(int i)
{
  implicit_cast(i); // 错误：无法推断T
  implicit_cast<double>(i); // T是double，U是int
  implicit_cast<char,double>(i); // T是char，U是double，可以
  implicit_cast<char*,int>(i); // T是char*，U是int
                               // 错误，无法将int转换为char*
}
```

13.3.2 函数模板的重载

- 可以声明多个具有同样名字的函数模板，甚至可以声明具有同一个名字的多个函数模板和常规函数的组合

```
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>)
}
```

模板函数重载规则

- 找出能参与这个重载解析的一组函数模板的专门化。例如：sqrt(z)将产生候选函数sqrt<double>(complex<double>)和sqrt<complex<double>>(complex<double>)
- 如果两个模板函数都可用，而其中一个比另外一个更专门，在随后的步骤中就只考虑那个最专门的模板函数。例：sqrt(z)意味着sqrt<double>(complex<double>)而非sqrt<complex<double>>(complex<double>)

模板函数重载规则

- 在这组函数上做重载解析，包括那些常规函数，如果某个函数模板参数已经通过模板参数推断确定下来，这个参数就不能再同时应用提升、标准转换或者用户定义的转换。对于sqrt(2)，sqrt<int>(int)是确切匹配，优先于sqrt(double)
- 如果一个函数和一个专门化具有同样的匹配，那么就选用函数，因此，sqrt(2.0)将选用sqrt(double)而不是sqrt<double>(double)
- 如果找不到匹配或者找到多个匹配，都产生错误

模板函数重载示例

```
template<class T> T max(T,T);
const int s = 7;
void k(){
  max(1,2); // max<int>
  max('a','b'); // max<char>
  max(2.7,4.9); // max<double>
  max(s,7); // max<int>(int(s),7)
  max(‘a’,1); // 错误，歧义
  max(2.7,4); // 错误，歧义
}
void f() { // 显式限定，避免歧义
  max<int>(‘a’,1);
  max<double>(2.7,4);
}
```

模板函数重载示例

- 也可以增加适当的声明来避免歧义

```
int max(int i, int j) {return max<int>(i,j);}
double max(int i, double d){return max<double>(i,d);}
double max(double d, int i){return max<double>(d,i);}
double max(double d1, double d2)
                {return max<double>(d1,d2);}

void g()
{
  max('a',1); // max(int('a'),1)
  max(2.7,4); // max(2.7,double(4))
}
```

模板函数重载示例

- 重载解析规则保证模板函数能够正确地与继承机制相互作用

```
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))
}
```

模板函数重载示例

- 在对模板参数的推断中未涉及到的那些函数参数，就完全按照非模板函数的参数处理。特别是可以使用普通的转换规则

```
template<class T, class C> T get_nth(C& p, int n);
                    // 取容器C中的第n个元素并将其返回
struct Index{
  operator int();
};
void f(vector<int>& v, short s, Index i)
{
  int i1 = get_nth<int>(v,2); // 准确匹配
  int i2 = get_nth<int>(v,s); // 标准转换，short到int
  int i3 = get_nth<int>(v,Index); // 用户定义转换：Index到int
}
```

13.4 用模板参数描述策略

- 考虑串排序，涉及到三个概念：串(容器)、元素类型和排序算法
- 一般来说，无法将排序算法硬编码到容器中，而且也无法将排序算法硬编码到元素类型
- 通用的解决方案是以通用的方式表述排序算法，并且该算法可以用于各种类型的数据排序

```
template<class T, class C>
int compare(const String<T>& str1, const String<T>& str2)
{   // 使用C中的静态方法进行比较
  for(int i=0;i<str1.length()&&i<str2.length();i++)
    if(!C::eq(str1[i],str2[i])) return C::lt(str1[i],str2[2]) ? -1:1;
  return str1.length()-str2.length();
}
```

用模板参数描述策略示例

```
template<class T> class Cmp{ // 常规，默认比较
public:
  static int eq(T a, T b) {return a==b;}
  static int lt(T a, T b) {return a<b;}
};
class Literate{
public:
  static int eq(char a, char b){return a==b;}
  static int lt(char,char);
}; // 根据文化习惯比较瑞典人的名字

void f(String<char> swede1,
                String<char> swede2)  {
  compare<char,Cmp<char>>(swede1,swede2);
  compare<char,Literate>(swede1,swede2);
}
```

将比较操作作为模板传递有两个重要的优点：可以通过一个参数传递几个操作；没有任何运行时额外开销

13.4.1 默认模板参数

- 每次调用都需要显式地给出比较准则也会使人厌烦，为此，可以使用函数重载，也可以使用默认模板参数

```
// 函数重载
template<class T, class C> // 用C比较
int compare(const String<T>& str1, const String<T>& str2);
template<class T> // 用Cmp<T>比较
int compare(const String<T>& str1, const String<T>& str2);

template<class T, class C=Cmp<T>> // 默认模板参数
int compare(const String<T>& str1, const String<T>& str2) {
  for(int i=0;i<str1.length()&&i<str2.length();i++)
    if(!C::eq(str1[i],str2[i])) return C::lt(str1[i],str2[2]) ? -1:1;
  return str1.length()-str2.length();
}
```

默认模板参数示例

```
// 然后可以按照如下方式调用compare
void f(String<char> swede1, String<char> swede2)
{
  compare(swede1,swede2); // 用Cmp<char>
  compare<char,Literate>(swede1,swede2); // 用Literate
}
```

- 通过模板参数支持执行策略，以及在此基础上采用默认值支持最常用策略的技术在标准库中广泛使用
- 用于表示策略的模板参数常常被称为"特征"(traits)，例如：标准库string依赖于char_traits，标准算法依赖于迭代器特征类，所有标准库容器依赖于allocator

13.5 专门化

- 用户可以提供多个模板设计定义，编译器在实际应用中基于模板参数来进行选择
- 一个模板的多个可以互相替代的定义被称为用户的专门化，或者简称为用户专门化

```
template<class T> class Vector{
  T* v;
  int sz;
public:
  Vector();
  explicit Vector(int);
  T& elem(int i){return v[i];}
  T& operator[](int i);
};
// 大部分Vector中存储某种指针
// 基本原因是为了保持多态的行为方式
Vector<int> vi;
Vector<Shape*> vps;
Vector<string> vs;
Vector<char*> vpc;
Vector<Node*> vpn;
```

专门化

- 大部分C++实现的默认方式是对模板函数建立代码段的副本，这可能导致严重的代码膨胀
- 可以通过设计，使得所有指针容器都能共享同一份实现代码，这可以通过专门化做到
- 首先，定义一个void的指针的(专门化)Vector版本
- 而后这个专门化就会被用作所有指针的Vector的共同实现，因为所有特定的指针类型都可以被转化为void*

专门化示例

- template<>表示这是一个专门化，不需要模板参数，<void*>表示这个定义应该用在所有的T是void*的Vector实现中
- Vector<void*>将用于如下定义的Vector： Vector<void*> vpv;

```
template<> class Vector<void*>{
  void** p;
  void*& operator[](int i);
};
```

专门化示例

- 为了使得该专门化能够用于所有指针的Vector，且仅仅用于指针的Vector，需要一个部分专门化(partial specification)
- 名字后面的<T*>说明这个专门化将被用于每一个指针类型

```
template<class T> class Vector<T*> : private Vector<void*>{
public:
  typedef Vector<void*> Base;
  Vector() : Base() {}
  explicit Vector(int i) : Base(i) {}
  T*& elem(int i){return reinterpret_cast<T*&>(Base::elem(i));}
  T*& operator[](int i){return
                reinterpret_cast<T*&>(Base::operator[](i));}
}; // 部分专门化示例
```

专门化示例

- 上面的定义将用于每一个其模板参数能够表述为T*的Vector
- 现在可以使得所有指针的Vector共享了同一个实现，Vector<T*>实际上只是Vector<void*>的界面，完全通过派生和内联展开实现
- 实际应用表明，这种技术在抑制代码膨胀方面非常有效

```
Vector<Shape*> vps; //<T*>是<Shape*>所以T是Shape
// 注意，当运用到某个部分专门化时，模板参数是从专门化模式匹
// 配出来的，所以此处模板参数T不是Shape*，而是Shape
Vector<int**> vppi; //<T*>是<int**>所以T是int*
```

13.5.1 专门化的顺序

- 一个专门化比另一个更专门的意思是：能够与它匹配的每个实际参数表也能与另外的那个专门化匹配，但是反之则不行
- 更专门可以被理解为更特殊
- 在声明对象、指针等的时候，在做重载解析的时候，最专门化的那个版本都将比别的版本优先被选中

```
template<class T> class Vector; // 一般情况
template<class T> class Vector<T*>; // 对任何指针的专门化
template<> class Vector<void*>; // 对void*的专门化，最专门
```

13.5.2 模板函数的专门化

- 专门化对于模板函数也非常有意义，考虑前面提到过的shell排序，其中用<比较元素
- 一个更好的定义是使用模板函数less()替代<比较符号
- 可以通过less()对const char*的一个专门化做好这类比较

```
template<class T> bool less(T a, T b) {return a<b;}
// 对于T为char*时，无法正确比较
template<> bool less<const char*>(const char* a,
                    const char* b) {
  return strcmp(a,b)<0;
}
```

模板函数的专门化

```
// 模板参数可以从函数的实际参数表推断，不需要显式描述
template<> bool less<>(const char* a,
                       const char* b){
  return strcmp(a,b)<0;
}
// 给出了template<>前缀，后面的<>也就多余了
template<> bool less (const char* a,
                       const char* b){
  return strcmp(a,b)<0;
}
```

13.6 派生和模板

- 模板和派生都是从已有类型创建新类型的机制，通常都被用于写利用各种共性的代码
- 从一个非模板类派生出一个模板类，这是为一组模板提供一个公用实现的一个方法，例如：
  template<class T> class Vector<T*> : private Vector<void*>{};
- 从模板类派生模板类也很有用，如果某个基类的成员依赖于其派生类的模板参数，那么这个基类本身也必须参数化

```
template<class T> class vector{/*...*/};
template<class T> class Vec : public vector<T>
{/*...*/};
```

13.6.1 参数化和继承

- 模板是将一个类型或者一个函数用其他的类型进行参数化，该模板对于所有参数类型的代码实现都是相同的
- 抽象类定义了一个界面，抽象类的不同实现中的许多代码可以被该类层次结构所共享，同时，利用该抽象类的大部分代码不依赖于它的具体实现
- 两者在“声明一次，然后被应用于多种不同类型”方面非常相似，可用多态性来描述之

参数化和继承

- 为了区分，将虚函数提供的东西称作运行时多态性(run-time polymorphism)，而把模板提供的多态性称为编译时多态性(compile-time polymorphism)或者参数式多态性(parametric polymorphism)
- 选择虚函数或者模板的依据：若所操作的一组对象之间不需要层次性的关系，选择模板；若编译时无法得知这些对象的类型，选择抽象类；若对运行效率要求严格，选择模板可以将各种操作方便地内联化

13.6.2 成员模板

- 一个类或者模板的成员也可以是模板
- 但是成员模板不能是virtual

```
template<class Scalar> class complex{
  Scalar re, im;
public:
  template<class T> complex(const complex<T>& c) :
                    re(c.real()), im(c.imag()){}
};
complex<float> cf(0,0);
complex<double> cd = cf; // 可以，用float到double的转换
class Quad{
  // 没有到int的转换
};
complex<Quad> cq;
complex<int> ci = cq; // 错误，无从Quad到int的转换
```

13.6.3 继承关系

- 可以将类模板理解为一种有关如何生成特定类的规范
- 因此，类模板有时也被称为类型生成器(type generator)
- 如果只考虑C++的规则，由同一个类模板生成的两个类之间并不存在任何关系

继承关系示例

```
class Shape{/*...*/};
class Circle:public Shape{/*...*/};
```
基于上述声明，人们有时会想到把set<Circle*>当作set<Shape*>
但这是错误的，Circle是Shape，但是Circle的集合并非Shape的集合

```
class Triangle:public Shape{/*...*/};
void f(set<Shape*>& s) {
  s.insert(new Triangle()); // 可以
}
void g(set<Circle*>& s) {
  f(s); // 错误，类型不匹配，s是set<Circle*>不是set<Shape*>
}
// 不存在内部的从set<Circle*>到set<Shape*>的转换，也不应该
// 有这种转换，set<Circle*>应该能保证所有成员一定是Circle
```

13.6.3.1 模板转换

- 模板生成的各个类之间没有任何默认的关系，然而对于某些模板，我们可能希望能表述这种关系

```
template<class T> class Ptr{
  T* p;
public:
  Ptr(T*);
  Ptr(const Ptr&);
  template<class T2> operator Ptr<T2>();
  // 将Ptr<T>转换到Ptr<T2>
};
void f(Ptr<Circle> pc) {
  Ptr<Shape> ps = pc; // 应该能行
  Ptr<Circle> pc2 = ps; // 应该指出错误
}
```

模板转换

- 我们希望上例中，只有当Shape是Circle的直接或者间接基类时才能够进行指针的转化
- 如下方式，当且仅当p(其类型是T*)可以作为Ptr<T2>(T2*)构造函数的参数时，这个返回语句才能编译

```
template<class T> // 模板的模板参数表
  template<class T2> // 模板成员的模板参数表
    Ptr<T>::operator Ptr<T2>(){return Ptr<T2>(p);}

void f(Ptr<Circle> pc) {
  Ptr<Shape> ps = pc; // 可以：Circle*可以转换到Shape*
  Ptr<Circle> pc2 = ps; // 错误：Shape*不能转换到Circle*
}
```

13.7 源代码组织

- 使用了模板的代码有两种明显的组织方式
  - 在使用模板的编译单位最前面包含(include)有关模板的定义
  - 只在使用模板的编译单位最前面包含(include)有关模板的声明，并且分别编译它们的模板定义
- 此外，模板函数有时是首先声明，然后使用，最后在一个编译单位里定义

两种源代码组织方式示例(1)

```
//out.c
#include <iostream>
template<class T> void out(const T& t) { std::cerr<<t;}

//user1.c
#include “out.c”

//user2.c
#include “out.c”
```

out()的定义和它所依赖的所有声明都被#include在几个不同的编译单位里，(仅)在需要时才去生成代码，这会增大编译器必须处理的信息量同时也可能使得out.c中的代码和用户代码互相影响，此外，优化对冗余定义的读入过程等工作都是编译器的责任

两种源代码组织方式示例(2)

```
// out.h
template<class T> void out(const T& t);
// out.c
#include <iostream>
#include "out.h"
export template<class T> void out(const T& t){std::cerr<<t;}

//user1.c
#include “out.h”

//user2.c
#include “out.h”
```

这种策略对模板函数的处理与对非内联函数的处理一样，out()定义被单独进行编译，在需要该定义时，实现必须负责找到那个唯一的定义

源代码组织的理想方式

- 无论作为一个单位编译，还是分成一些独立的编译单元，这些代码都应能以同样的方式工作
- 接近这种理想的方式应该是限制模板定义对其环境的依赖性，而不是在进入实例化过程时为它的定义带上尽可能多的环境信息

13.8 忠告

- [1] 用模板描述需要使用到许多参数类型上去的算法
- [2] 用模板表述容器
- [3] 为指针的容器提供专门化，以减小代码规模
- [4] 总是在专门化之前声明模板的一般形式
- [5] 在专门化的使用之前先声明它
- [6] 尽量减少模板定义对于实例化环境的依赖性
- [7] 定义你所声明的每一个专门化
- [8] 考虑一个模板是否需要有针对C风格字符串和数组的专门化

忠告

- [9] 用表述策略的对象进行参数化
- [10] 用专门化和重载为同一概念的针对不同类型的实现提供统一界面
- [11] 为简单情况提供简单界面，用重载和默认参数去表述不常见的情况
- [12] 在修改为通用模板前，在具体实例上排除程序错误
- [13] 如果模板定义需要在其他编译单位里访问，请记住写***export***
- [14] 对大模板和带有非平凡环境依赖性的模板，应采用分开编译的方式
- [15] 用模板表示转换，但要非常小心地定义这些转换

忠告

- [16] 如果需要，用***constraint***( ) 成员函数给模板的实参增加限制
- [17] 通过显式实例化减少编译和连接时间
- [18] 如果运行时的效率非常重要，那么最好用模板而不是派生类
- [19] 如果增加各种变形而又不重新编译是很重要的，最好用派生类而不是模板
- [20] 如果无法定义公共的基类，最好用模板而不是派生类
- [21] 当有兼容性约束的内部类型和结构非常重要时，最好用模板而不是派生类