7 模板 Template

董洪伟

http://hwdong.com

templates （模板）-为什么

- 泛型编程的需要。

  设想你实现了一个整数数组排序的算法：

  ```
  void sort(int *ints, int n);
  ```

  用该函数可以对实、复数或工资单排序吗？
- 模板可以复用源代码-泛型编程

交换两个对象

```
inline void swap(int &x, int &y){
  int t = x; x = y; y  =t;
}
inline void swap(string &x, string &y){
  string t = x; x = y; y  =t;
}
inline void swap(double &x, double &y){
  double t = x; x = y; y  =t;
}
```

交换两个对象

```
struct  student
{
   string name;
   int age;
};
```

```
void f(){
   int a=3,b = 5;
   string s1(“Li”),s2(“Wang”);
   student stu1,stu2;
   swap(a,b);
   swap(s1,s2);
   swap(stu1,stu2); //错 !
}
```

交换两个对象

```
struct  student
{
    string name;
    int age;
};
template<class T>
void swap(T &x,T &y){
   T t = x; x = y; y  =t;
}
```

```
void f(){
    int a=3,b = 5;
    string s1(“Li”),s2(“Wang”);
    student stu1,stu2;
    swap(a,b);
    swap(s1,s2);
    swap(stu1,stu2);  //OKay !
}
```

```
struct  student{
    string name;
    int age;
};
inline void swap(int &x,int &y){
   int t = x; x = y; y  =t;
}
inline void swap(string &x,string &y)
{
   string t = x; x = y; y  =t;
}
inline void swap(double &x, double &y)
{
   double t = x; x = y; y  =t;
}
```

```
void f(){
    int a=3,b = 5;
    string s1(“Li”),s2(“Wang”);
    student stu1,stu2;
    swap(a,b);
    swap(s1,s2);
    swap(stu1,stu2); //错 !
 }
```

```
struct  student{
    string name;
    int age;
};

template <class T>
void swap(T &x,T &y){
   T t = x; x = y; y  =t;
}
```

```
void f(){
    int a=3,b = 5;
    string s1(“Li”),s2(“Wang”);
    student stu1,stu2;
    swap(a,b);
    swap(s1,s2);
    swap(stu1,stu2); //错 !
 }
```

定义模板

- 在函数和类的前面加上模板头：关键字template和<模板参数表>，即：template <模板参数表>
- 简单的方法是先写出普通函数或类，然后将其转换为模板：1）加模板头 2）代码中的元素类型换成模板中的参数类型。
- 注意在类模板体外成员函数定义时，一定要说明模板类型。类模板也称为模板类。

定义模板-函数模板

```
int  swap(int& x, int& y) {
      int t = x; x =y ; y = t;
}
```

定义模板-函数模板

```
template <typename T>   //增加模板头
T  swap(T& x, T& y) {    //将参数类型换成模板类型参数
     int t = x; x =y ; y = t;
}
```

定义模板-类模板

```
class Vector{
public:
    Vector( int size);
    ~Vector() { delete[]  data;  }
    int& operator[]( int i );
private:
     int *_data;
    int _size;
};
```

```
Vector:: Vector(int size){
   _data = new int[size];
   _size = size;
}
int& Vector:: operator[]( i ){
   if(i >=0&&i <_size)
      return _data[i];
}
```

定义模板-类模板

```
int main(){
   Vector v( 10 );       //整数向量（数组）
   for( int i = 0; i < 10; ++i )
       v[i] = i;
   v[2] = “LiPing”;  //错：Vector只能存放整数
   return v[0];
}
```

定义模板-类模板

- 整数向量->向量模板

```
class Vector{
public:
    Vector( int size);
    ~Vector() { delete[]  data;  }
    int& operator[]( int i );
private:
     int *_data;
    int _size;
};
```

定义模板-类模板

- 整数向量->向量模板

```
template<typename T>
class Vector{
public:
    Vector( int size);
    ~Vector() { delete[]  data;  }
    T& operator[]( int i );
private:
     T*_data;
    int _size;
};
```

定义模板-类模板

- 整数向量->向量模板

```
Vector :: Vector( int size )
{
   _data= new int[size];
   _size = size;
}
```

定义模板-类模板

- 整数向量->向量模板

```
template <class T>
Vector<T>::Vector( int size )
{
   _data= new T[size];
   _size = size;
}
```

定义模板-类模板

- 整数向量->向量模板

```
int& Vector   :: operator[]( int nSubscript ){
   if(nSubscript >= 0&&nSubscript<iUpperBound )
         return _iElements[nSubscript];
   else{
         throw std::out_of_range("out_of_range in
              Vector  :: operator[]");
   }
}
```

定义模板-类模板

- 整数向量->向量模板

```
int&   Vector   :: operator[]( int i ){
   if(i >= 0 && i<_size )
         return _data[i];
   else{
         throw std::out_of_range("out_of_range in
              Vector<T>::operator[]");
   }
}
```

定义模板-类模板

- 整数向量->向量模板

```
template <typename T>
T& Vector<T> ::operator[]( int i ){
    if(i >= 0 && i<_size )
        return _data[i];
    else{
        throw std::out_of_range("out_of_range in
            Vector<T>::operator[]");
    }
}
```

定义模板-类模板

```
#include <string>
int main(){
    Vector<int> v( 10 );      //实例化模板
    for( int i = 0; i < 10; ++i )
            v[i] = i;

   Vector<string> s( 10 );
   s[2] = “hello!”;
   s[4] = s[2];
   return v[0];
}
```

练习

```
class calc {
  public:
    int multiply(int x, int y);
    int add(int x, int y);
 };
int calc::multiply(int x, int y)
{
  return x*y;
}
int calc::add(int x, int y)
{
  return x+y;
}
```

```
template <class T>
class calc{
  public:
    T multiply(T x, T y);
    T add(T x, T y);
};
template <class T> T
calc<T>::multiply(T x,T y){
  return x*y;
}
template <class T> T
calc<T>::add(T x, T y){
  return x+y;
}
```

模板实例化/ Template Instantiation

- 由类模板（模板类）通过指定模板参数类型，得到一个具体的类的过程。如

  Vector<int>
- 模板实例化是由编译器完成的，与程序员无关
- 如果**模板实例化**过程中，发现某个模板实参不能提供模板所要的语义时，将产生编译错误。如：

```
template<class T> T max(T a, T b) { return a>b?a:b; }
```

max(stu1,stu2); //如果stu1,stu2对应类型没有重载>运算符，则编译出错

模板参数/ template parameters

- 模板参数分为类型模板参数和非类型模板参数。

  如上2个例子中的T.

- 类型模板参数用来参数化一个类型，非类型模板参数用来参数化一个常量。类型模板参数由关键字typename或class和参数名构成，非类型模板参数与一般函数参数一样，由普通类型和参数名构成

模板参数/ template parameters

```
template <typename T，  int size=20>
class Vector{
public:
    Vector(): _size(size){};
    T & operator[]( int i );
private:
     T  _data[size];
     int  _size;
};
```

非类型模板参数必须是常量表达式

- 因为模板实例化是编译期行为。

```
int i= 6;
Vector<int , 45> intVec; //OK
Vector<int , i> intVec2;   //Error
```

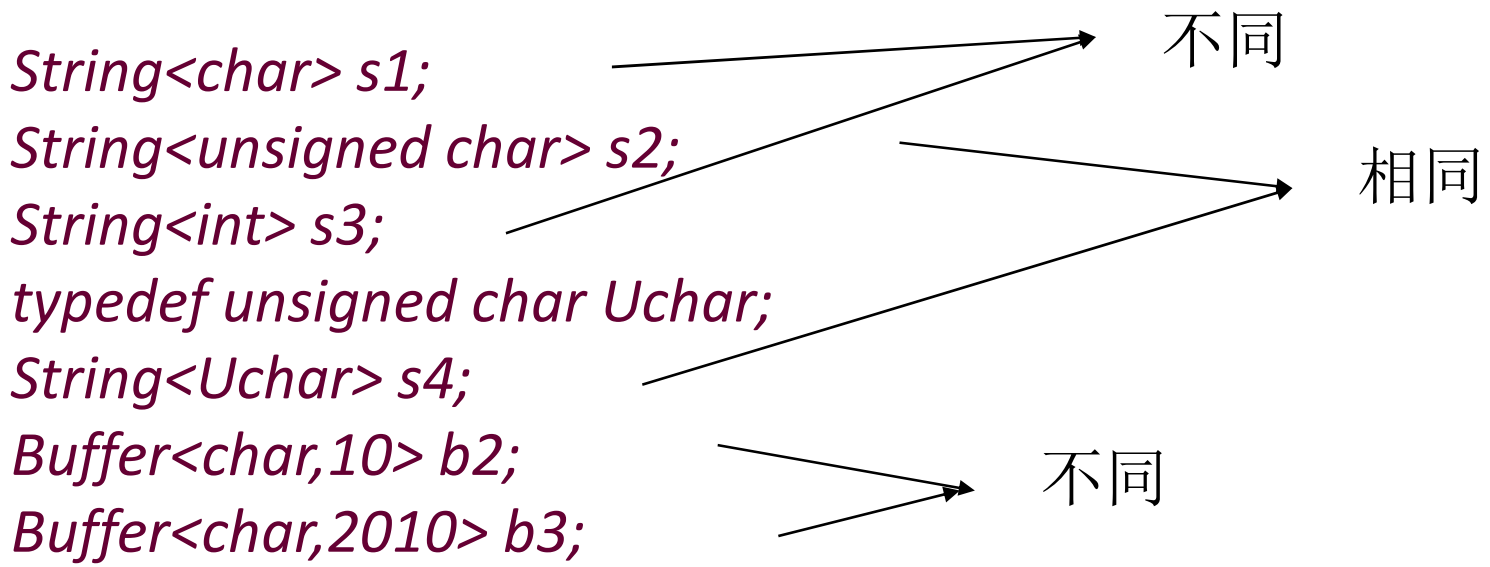
缺省模板参数

- 象普通函数参数有缺省参数一样，模板也可以有缺省模板参数

```
template <typename T = int>
class Vector{
public:
        Vector( int size );
        ~Vector() { delete[] _data; }
        T & operator[]( int i );
private:
        T *_data;
        int _size;
};
```

类型等价/ Type Equivalence

- 对于一个模板，给它不同的模板参数，将产生不同的模板实例化类型。
- 同样的模板参数将生成同样的类型。



类型等价/ Type Equivalence

- 同一模板生成的不同类型的变量当然不能相互赋值.

```
String<unsigned char> s2;
String<Uchar> s4;
Buffer<char,10> b2;
Buffer<char,2010> b3;

s4 = s2 ; //可以
b3 = b2 ;  //错!
```

类型检查/ Type Checking

- 对模板错误的检查分为：定义时的检查和使用时（实例化）的检查。
- 定义时的检查：语法错误。如：丢失分号（语法错误）、定义中遇到的名字要么在作用域要么以合理的方式依赖于某个模版参数。
- 使用时（实例化）的检查：少数简单的语义错误。如：指针不能用整数初始化（语义错误）、与模板参数有关的错误

模板参数推断

- 隐式推断：根据实际参数推断模板参数类型

```
template<typename T>
T max(T a,T b){
   return a>b?a:b;
}
```

```
void f(){
   int i = 3,j = 4;
   double f=12.5,g = 25.8;
   i = max(i,j);  // max(int,int)
    double s = max(f,g); // max(double,double)
}
```

模板参数推断

- 显式指定参数类型

```
template<class T> class Vector;
template<class T> T* create();
 void f(){
    Vector<int> v;              //显式指定
     int *p = create<int>();   //显式指定
 }
```

模板专门化/Template Specialization

- 模板的专门化或特化：重载模板的默认(default)实现。-即通过提供一个模板的不同定义,并由编译器在使用时根据提供的模板参数,选择一个合适的定义.

```
template<typename T>
class Stack{
 //.....
};
```

专门化(特化)

```
template<>
class Stack<std::string>{
    //.....
};
```

```
template <class T>
T max(T a, T b) {    return a > b ? a : b ;  }


// Specialization of max for char*
template <>
char* max(char* a, char* b){
    return strcmp(a, b) > 0 ? a : b ;
}

int main(){
    cout << "max(10, 15) = " << max(10, 15) << endl ;
    cout << "max('k', 's') = " << max('k', 's') << endl ;
    cout << "max(10.1, 15.2) = " << max(10.1, 15.2) << endl ;
    cout << "max(\"Aladdin\", \"Jasmine\") = "
     << max("Aladdin", "Jasmine") <<endl ;
    return 0 ;
}
```

局部专门化

```
template<typename T1, typename T2>
class myclass{
  //.....
};
```

可以有下面的局部专门化:

1)两个模板参数具有相同的类型

```
template< typename T>
class myclass<T ,T> {
   //.....
};
```

２)第２个模板参数类型是int

```
template< typename T>
class myclass<T, int>{//第2个参数是int
  //.....
};
```

３)两个模板参数都是指针类型

```
template<typename T1, typename T2>
class myclass<T1*,T2*>{
  //.....
};
```

一个比另一个更专门

- template<typename T1, typename T2>
  class myclass；

- template<typename T1, typename T2>
  class myclass<T1*,T2*>；

- template<typename T>
  class myclass<T*,T*>；

模板的重载解析--函数模板的重载解析

- 可以声明多个函数模板,甚至可以声明函数模板与常规函数的组合。当调用被重载了的函数时,将通过重载解析找出最合适的函数。

```
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>)
}
```

重载解析的过程

1）找出能参与重载解析的一组模板函数的专门化。如*sqrt*(*z*)可产生侯选函数:
sqrt<double>(*complex<double>*)
sqrt< complex<double> >(*complex<double>*)

2）如果有两个模板函数可用，使用其中更专门化的那一个。*sqrt<double>*(*complex<double>*) 比*sqrt< complex<double> >*(*complex<double>*)更专业。因为能与*sqrt<T>(complex<T>)*匹配的一定也匹配sqrt<T>(T)。

重载解析的过程

3）如果一般函数和一个专门化有同样好的匹配，那么选用一般函数。*sqrt*(*double*) 比 *sqrt<double>*(*double*) 更适合 *sqrt*(*2.0*).

4）重载当然包含常规函数。如果某个模板函数参数已经通过模板参数确定了,这个参数就不能再同时进行提升、标准或用户定义等转换了。如对sqrt(2) , sqrt<int>(int) 是精确匹配, 所以优先于sqrt(double).

重载解析的过程

5）如果找不到匹配，则错误；或者如果得到2个以上同样好的匹配，就是歧义，也是错误。

```
template<class T> T max(T,T) ;
const int s = 7;
void k(){
    max(1,2) ; // max<int>(1,2)
    max(´a´,´b´) ; // max<char>('a','b')
    max(s,7) ; // max<int>(int(s),7)
    max(´a´,1) ; // 歧义错误
    max(2.7,4) ; //歧义错误
}
```

重载解析-消除歧义

- 显式限定

```
void f(){
    max<int>('a',1);
}
```

- 增加声明

```
inline int max(int,int);
inline double max(double,int);
```

C++标准模板库-STL

- 算法（函数模板）

- 容器类模板及迭代器模板

  容器类模板：数组(vector)、链表(list)、集合(set)等

  而迭代器(iterator)则类似于指针，用于遍历容器中的元素。

- 每个容器类有若干与之对应的迭代器类。对迭代器可以用++、--修改它，可以通过*或->访问容器的元素

vector/向量(数组)

- 初始化、插入、擦除、大小、遍历、改变大小等

```
vector<int>  ints;
ints.push_back(23); //insert
ints.erase(ints.begin()+0);
ints.resize(20);
for(int i = 0  ;i<ints.size();i++){
      ints[i] = i+2;
}
```

list(链表) 及迭代器iterator

- 初始化、插入、擦除、大小、遍历、改变大小等

```
list<int>  ints;
ints.push_back(23); //insert
ints.resize(20);
list<int>::iterator it = ints.begin();
for(; it !=ints.end();it++){
        *it= 2;
}
```

泛型算法:sort

```
template<class RanIt>
void sort(RanIt first, RanIt last);

template<class RanIt, class Pred>
void sort(RanIt first, RanIt last, Pred pr);
```

比较谓词

泛型算法:sort

```
#include<algorithm>
#include<functional>    //greater需要
 int main(){
    vector<int>  ints;
     ints.resize(20);
     for(int i = 0  ;i<ints.size();i++){
           ints[i] = i+2;
    }
    sort(ints.begin(),ints.end());
    sort(ints.begin(),ints.end(),greater<int>());
}
```

泛型算法:sort

```
#include<algorithm>
bool my_greater(int i,int j){return i>j;}
 int main(){
    vector<int>  ints;
     ints.resize(20);
     for(int i = 0  ;i<ints.size();i++){
           ints[i] = i+2;
    }

    sort(ints.begin(),ints.end(), my_greater);
}
```

泛型算法:sort

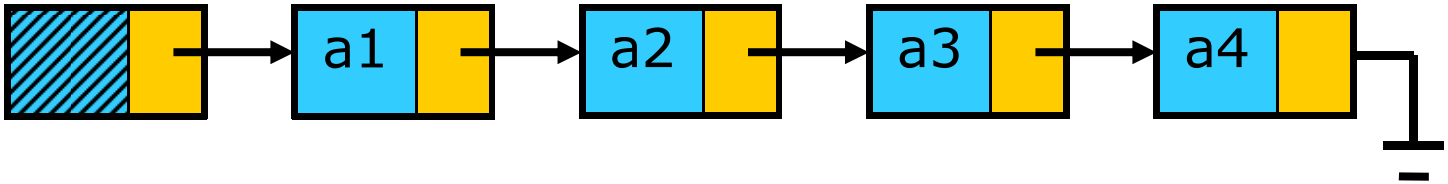
```
#include<algorithm>
 struct myLess{
    bool operator() (int i,int j){return i>j;}
 };
 int main(){
    vector<int>  ints;
     ints.resize(20);
     for(int i = 0  ;i<ints.size();i++){
           ints[i] = i+2;
    }
    sort(ints.begin(),ints.end(), myLess());
}
```

泛型算法: find

```
template<class InIt, class T>
InIt  find (InIt first, InIt last, const T& val);
```

练习

- 实现一个冒泡排序的函数模板
- 实现一个较实用的线性表类Vector模板（a1,a2,….an）
- 实现一个单链表模板



- 使用find泛型算法在一个 list链表中查找数据