模板和泛型

函数模板
类模板
泛型

模　板

• **Templates are a feature of the C++ programming language that allow functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one.**

• 代码要想重用，就得通用，通用的一大障碍就是类型限制。若代码能自动适应类型的变化，通用进而重用就成为可能，而模板可以简化这种设计。

•关于C++中加入模板有一段关于Alexander Stepnova和Bjarne Stroustrup的历史，前者是STL之父，后者则是C++创始人。

•直到1991年，C++才加入模板。

泛　型

- C语言中有泛型吗？看看下面这几个C库函数头:

  void *memcpy(void* to, const void* from, size_t count);

  int memcmp(const void* buff1, const void* buff2, size_t count);

  void *memmove(void* to, const void* from, size_t count);
- 看来C中是有泛型的，只不过很简单罢了。

- 那么什么是通俗意义上的泛型呢？
- Alexander Stepanov wrote: “Generic programming is about abstracting and classifying algorithms and data structures...
- Bjarne Stroustrup : Lift algorithms and data structures from concrete examples to theirmost general and abstract form…

- C++的模板分为函数模板和类模板；
- 模板代表了一个代码家族：
  函数模板代表了一个函数家族；
  类模板代表了一个类家族；
- 模板引入的类型的抽象，是泛型程序设计的基础，在STL和boost中得到广泛的应用。
- 泛型编程技术与面向对象的编程风格在编程思想上是相左的，面向对象风格表现了数据和操作被捆绑封装为一体；而泛型编程技术则极力将数据和操作分离、独立。
- 模板和泛型的使用使得Intel C++编译器比原来未用的版本源代码量减少了1/3

函数模板

函数

- 函数模板是一类相同功能但不同类型函数的制造模型，可用来创建多个不同类型的函数，这将大大降低源代码行数。
- 函数模板不再是一般意义上的函数，是函数模型。

模板

声明方法：

```
template <类型参数列表>
函数声明;
// 例如：
template <typename T1, typename T2> // typename或class
void fun(const T1& t1, const T2& t2);
```

求两个值中较小的一个

函数模板

```
inline int my_min(int a, int b) { return a < b ? a : b; }
inline double my_min(double a, double b)
{ return a < b ? a : b; }
inline const std::string& my_min(const std::string& a,
                                 const std::string& b)
{
  return a < b ? a : b;
}

... // 还有很多其它任何可以比较大小的数据类型
```

抽象一下就成了：

```
template <typename T>          // typename或class均可
inline const T& my_min(const T& x, const T& y)
{  return x < y ? x : y; }
```

// 是不是节约了很多源代码的编写呢？

求两个值中较小的一个

函数模板

```
#include <iostream>
#include <string>

int main()
{
  std::cout << my_min(10, 20) << '\n';          // my_min(int, int);
  std::cout << my_min(10.1, 10.09) << '\n';     // my_min(double, double);

  std::string str1 = "It is unnecesary!", str2("It is unnecessary.");
  std::cout << my_min(str1, str2) << '\n';
  std::cout << my_min("ok", "error") << '\n'; // 这个能匹配吗？
  std::cout << my_min(10, 10.1) << '\n';        // 你到底想调用哪个呀？

  // ...
}
```

- 由于前面的my_min函数只有一个模板参数，因此导致min(10, 10.1) 无法编译。显然这是一个态度问题，你为何不用my_min(10.0, 10.1)呢?!
- 一个模板版本的my_min使得原来写的那么多my_min都得删除吗？没有必要，因为编译器才懒得做类型实例化后再调用它们！

编译器进行函数匹配的顺序

- 先寻找有没有类型完全匹配的函数，有则调用，没有则下一步；若多于一个则出错。
- 然后寻找有没有匹配的模板函数，有则调用，没有则下一步；
- 最后寻找有没有将实参类型转换后可以调用得函数，有则调用，没有则出错。
- 由此看来编译器也很“懒”啊，能少干点就少干点。

  --难道这不正是我们希望的吗;-)

```
#include <string>

inline const std::string& my_min(const std::string& a,  const std::string& b)
{
  return a < b ? a : b;
}

template <typename T>
inline const T& my_min(const T& x, const T& y) {  return x < y ? x : y; }

#include <iostream>
int main()
{
  std::cout << my_min(10, 20) << '\n';            // my_min(int, int);
  std::cout << my_min(10.1, 10.09) << '\n';     // my_min(double, double);

  std::string str1 = "It is unnecesary!", str2("It is unnecessary.");
  std::cout << my_min(str1, str2) << '\n';        // 调用的哪个呀？
  std::cout << my_min("ok", "error") << '\n'; // 这个怎么突然就行啊？如果VC++8.0不行，
                                                  // 请用VC++10.0或g++4.5.2以上版本
  std::cout << my_min(10.0, 10.1) << '\n';
  // ...
}

// 这个程序留作课后练习。
```

类　模　板

类模板

- 同样，类模板不是类，是一类相同属性和功能但不同类型的类的制作模型。用户使用类模板可以为类声明一种模式，使得类中的某些数据成员的类型、某些成员函数的参数类型、某些成员函数的返回值类型，都能取任意类型(包括基本类型的和用户自定义类型)。
- 可用来创建成员布局相同的多个类，可大大简化代码的设计。

类模板的声明

- 类模板：

类模板

```
template <模板参数列表>
class 类名      // 或者struct 类名
{
  类成员声明
};
```

类 模 板 举 例

- 下面设计一个**cpair**

```
// cpair.hpp
#ifndef CPAIR_HPP
#define CPAIR_HPP

#include “compare.hpp“                          // 它定义了 !=, >, <=, >=

template <typename T1, typename T2>              // 前向声明
struct cpair;

template <typename T1, typename T2>              // 为了友元，这么前向声明
bool operator == (const cpair<T1, T2>&, const cpair<T1, T2>&);

template <typename T1, typename T2>              // 为了友元，这么前向声明
bool operator < (const cpair<T1, T2>&, const cpair<T1, T2>&);
```

```
template <typename T1, typename T2>
struct cpair
{
  typedef T1 first_type;
  typedef T2 second_type;

  T1 first;
  T2 second;

  cpair(const T1& a = T1(), const T2& b = T2());

  template <typename U1, typename U2>
  cpair(const cpair<U1, U2>& p);

  friend bool operator == <>(const cpair&, const cpair&);
  friend bool operator < <> (const cpair&, const cpair&);
}; // cpair
```

```
template <typename T1, typename T2>
cpair<T1, T2>::cpair(const T1& a, const T2& b) : first(a), second(b) { }

// 下面这个实现将考验你的编译器的编译能力，希望你的编译器不会趴下;-)
template <typename T1, typename T2>
  template <typename U1, typename U2>
cpair<T1, T2>::cpair(const cpair<U1, U2>& p) : first(p.first),
    second(p.second) { }

template <typename T1, typename T2>
inline bool operator == (const cpair<T1, T2>& x, const cpair<T1, T2>& y)
{
  return x.first == y.first && x.second == y.second;
}

template <typename T1, typename T2>
inline bool operator < (const cpair<T1, T2>& x, const cpair<T1, T2>& y)
{
  return x.first < y.first || (!(y.first < x.first) && x.second < y.second);
}
```

```
template <typename T1, typename T2>
inline cpair<T1, T2> make_cpair(const T1& x, const T2& y)
{
  return cpair<T1, T2>(x, y);
}

#endif /* CPAIR_HPP */

// compare.hpp 这个文件可使我们一劳永逸，以后凡需要之处包含它就是了
#ifndef COMPARE_HPP
#define COMPARE_HPP

template <typename T>
inline bool operator != (const T& x, const T& y) { return !(x == y); }

template <typename T>
inline bool operator > (const T& x, const T& y) { return y < x; }

template <typename T>
inline bool operator >= (const T& x, const T& y) { return !(x < y); }
```

```
template <typename T>
inline bool operator <= (const T& x, const T& y) { return !(y < x); }
#endif /* COMPARE_HPP */

#include <iostream>
// main.cpp
#include <iostream>
#include "cpair.hpp"

int main() // 编译运行一下，看一下结果。
{
    cpair<int, double> p0(123, 456.789);
    std::cout << p0.first << ", " << p0.second << '\n';
    cpair<short, char> p1 = make_cpair(97, 'b'), p2 = p1;
    std::cout << p2.first << ", " << p2.second << '\n';

    std::cout << (p1 <= p2 ? p1.first : p2.second);
    return 0;
}
```

使用类模板时编译器的工作

类模板

- 编译器对代码中的类模板记下了类框架；
- 扫描到创建对象时，按模板的实参类型生成模板类的代码，即此时才将类模板实例化；
- 运行时再用已生成的模板类创建对象；
- 若是用多种不同类型实例化对象，编译器将产生多段模板类代码。
- 此时的特化仅完成了类模板的定义，并未对其成员函数特化，直到某函数被调用时才特化它，也就是说特化是分两步做的，这不同于函数模板。

类模板不是类，仅是产生类的模型；模板类才是类，

使用类模板的要点

- 类成员函数模板只有被调用时才会被实例化。言外之意就是，那些尚未使用的成员函数可以只有函数原形，没有函数实现。——只要不调用到它就行。

其原因有三：

一是为了节省时间和空间；

二是对于“未能提供所有成员函数的全部操作”的类，也可以实例化成模板类。结果是“尽管类残缺，但照样不影响使用”。

三是成员函数可以是与模板类的模板参数不同的函数模板。

模板参数表中含有常量表达式

```
template<class T, int SZ=4>
class Ss
{
private:
   T m[SZ];                          //类的私有数据成员
public:
   Ss(T a,T b,T c,T d)       //构造函数，使用了模板
   { m[0]=a;m[1]=b;m[2]=c;m[3]=d; }
   T s()                             //一般函数成员，使用了模板
   { return (m[0]+m[1])+m[2]+m[3]; }
};
```

```
#include <iostream>

int main()
{
   Ss<int,4> i1(-23,5,38,-2);        //模板类定义对象
   Ss<float,4> f1(3.4,-44.5,69.3,-29.1);
   Ss<double,4> d1(355.4,256.2,98.23,-156.9);
   std::cout<<“int: "<<i1.s()<<std::endl;
   std::cout<<“float: "<<f1.s()<<std::endl;
   std::cout<<“double: "<<d1.s()<<std::endl;
}
```

类模板的参数

类模板

- 由于类型参数在类模板生成模板类时已经明确给出，勿须编译器在实例化类模板时再对那些类型实例化，于是生成模板类时它(们)往往扮演着“指导者”的角色。这一点不同于数值参数。
- 但是要求那些类型能提供被用到的全部操作。
- 于是，模板参数可以带默认值。所带的默认值可以是基本类型，也可以是已定义好的扩展类型，甚至可以是另一个模板。

类模板的参数可含有类型默认值

```
template<class T = int, unsigned long LEN =100>
class MyArray
{  T m_A[LEN];      //类的私有数据成员
    unsigneg long m_len;
 public:
    MyArray() : m_len(LEN)  { }
    T & operator[ ] (unsigneg long  index)
      {  return  m_A[index];   }
    T  sum () const;
};
MyArray < >  ai;          // < >可不要忘记哦
MyArray <long>  al;
MyArray<bool,32> ab;
```

模板参数的默认值是模板

```
template<class T, class Sequence = std::deque<T> >
class stack
{
private:
    Sequence container;      //类的私有数据成员
public:
    ...
};
stack<int>  mystack1;
stack<int,list<int> >  mystack2；// 最后两个>之间有空格
stack<int,stack<int> >  mystack3;
```

这叫类型默认值

这叫另一个模板充当模实参。

这会形成“二维栈”。

含有函数模板的无名类

问题：

比如：当你定义了**const int NULL = 0;** 之后，对于**void f(int x);**函数可用**f(0)**也可用**f(NULL)**调用，可对于**void f(string* p);**用**f(NULL)**可能会出现类型不匹配错误。

那么就定义成**void* const NULL = 0;** 这对于**f(0)**没影响，可对于**f(NULL)**又错了，类型不符。总没法兼顾两者。改用宏也没有起色：你总不能定义两个宏吧？(不要忘记宏是怎么被处理的)

 # define NULL 0

 # define NULL ((void *)0)

对于形形色色的类型以及它们的指针，我们常常要用到**NULL**，而这些**NULL**要兼顾各种类型。于是要有个能适应各种类型的类模板。由于该类产生的对象是唯一的**NULL** ，所以模板类不必有名。

NULL一种解决方案nullptr

```
const       //用来修饰对象是个常对象
class       //不必有名
{
public:
template < class T >
operator T*() const    {  return  0 ; }
//该函数能将NULL转换成指向T类型的指针
template < class C, class T> // 注意一下这个语法
operator T C::*() const    {  return  0 ; }
//能转换成指向C类的成员函数的指针，其返回类型是T类型
private:
   void operator &() const;  // 不允许取地址哦
}nullptr;       //对象名是唯一的
```

类模板带来的复杂性

•类模板的出现使“类名”和“类型名”不再统一。

一个普通类,如**class A{ ..... };**它的类名和类型名都是**A;**

而类模板**templete <typename T> class A{ ..... };**

中的**A**是类名，由构造、拷贝、析构等函数作函数名使用；

A<T>是类型名，由函数的形参和返回类型使用；

（如 **A (const A<模实参> &) ;** ）

•类中的成员函数在类外实现时，应使用类型名，因为类外实现的函数是实体，则必然是类实体而非类框架的成员。

- 类中的成员函数在实例化时，可能会发生“二次编译”（有的编译器是这样）。即由链接器的辅助工具重新激活编译器，将编译时未特化的函数完成特化。可见，类实例化和成员函数实例化实是分别进行的。
- 类模板中的成员函数仅是被调用的才会被实例化（节省时间）。更重要的是，对于未能提供完全操作的某些类型，也可以用作实例化的模板参数，只要那些尚不完善的成员函数没被调用就行。

类模板的特化

•原因：若有一个类模板 如

类中的成员函数的类外实现如下：

```
template<class T, unsigned long LEN>
T MyArray<T>:: sum () const
{
    usigned long count = LEN;
    T theSum = 0;
    while(count !=0)
    {
       theSum +=m_A[count-1];
       count--;
     }
    return theSum;
}
```

当我们用char*来实例化该类时，sum()函数将把LEN个地址累加，这显然不是我们要的结果。因此有必要针对char*类型手工写出该类及成员函数特定的实现代码。—— 这就叫特化。

```
template<unsigned long LEN> //或写成 template< >
class MyArray <char*, LEN>
{       char*  m_A[LEN];      //类的私有数据成员
        unsigneg long m_len;
 public:
        MyArray():m_len(LEN)  { }
        char * operator[ ] (unsigneg long  index)
           {  return  m_A[index];   }
        char*  sum (char *) const;
};
template<unsigned long LEN>
char* MyArray<char*, LEN>:: sum (char * buffer) const
{    assert(0 != buffer);
     for(unsigned long i =0;i<LEN;i++)
        strcat(buffer,m_A[i] );
     return buffer;
}
```

可见类模板的特化就是针对某种特定的类型写出该类及成员函数特定的实现代码。即“特定的实现方案”。一般是出于对时空效率的优化或满足特殊需要而为之。

STL中的vector<bool>类就是对vector<T>类的bool类型的特化。

使用时，模板特化的声明（定义）必须出现在基本模板之后；

特化版的优先级要高于基本模板。

如：**MyArray< char *，10> strArr;**

...

cout << strArr.sum (buffer) ；//调用的是特化版

类模板的局部特化

类模板

- 可以对已有的类模板中的特化稍加改造（不像特化那样给出确切类型），而是仍然使用抽象类型来特化原类模板，结果是要求用户在使用时再给出确切类型，如现有类模板：

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

可局部特化为：

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

类模板

或：

```
template<typename T>
class MyClass < T, int >
{     //   C
   ....
};
```

或：

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

于是可以：

```
MyClass <int , float>   mif;       //使用A
MyClass <float , float>   mff;     //使用B
MyClass <float , int >   mfi;      //使用C
MyClass <int , int >   mi;         //错
MyClass <int *, int* >   mpp;      //错
```

类模板

这后两句错都是二义性错：

MyClass <int , int > mi;

可以同等程度匹配 **MyClass < T, int >**和**MyClass < T, T > ；**

MyClass <int*, int* > mpp;

可以同等程度匹配 **MyClass < T, T >**和 **MyClass < T1 * , T2 * > ；**

当你再提供一个局部特化：

```
template<typename T>
class MyClass < T*, T* >
{       //  E
  ....
};
```

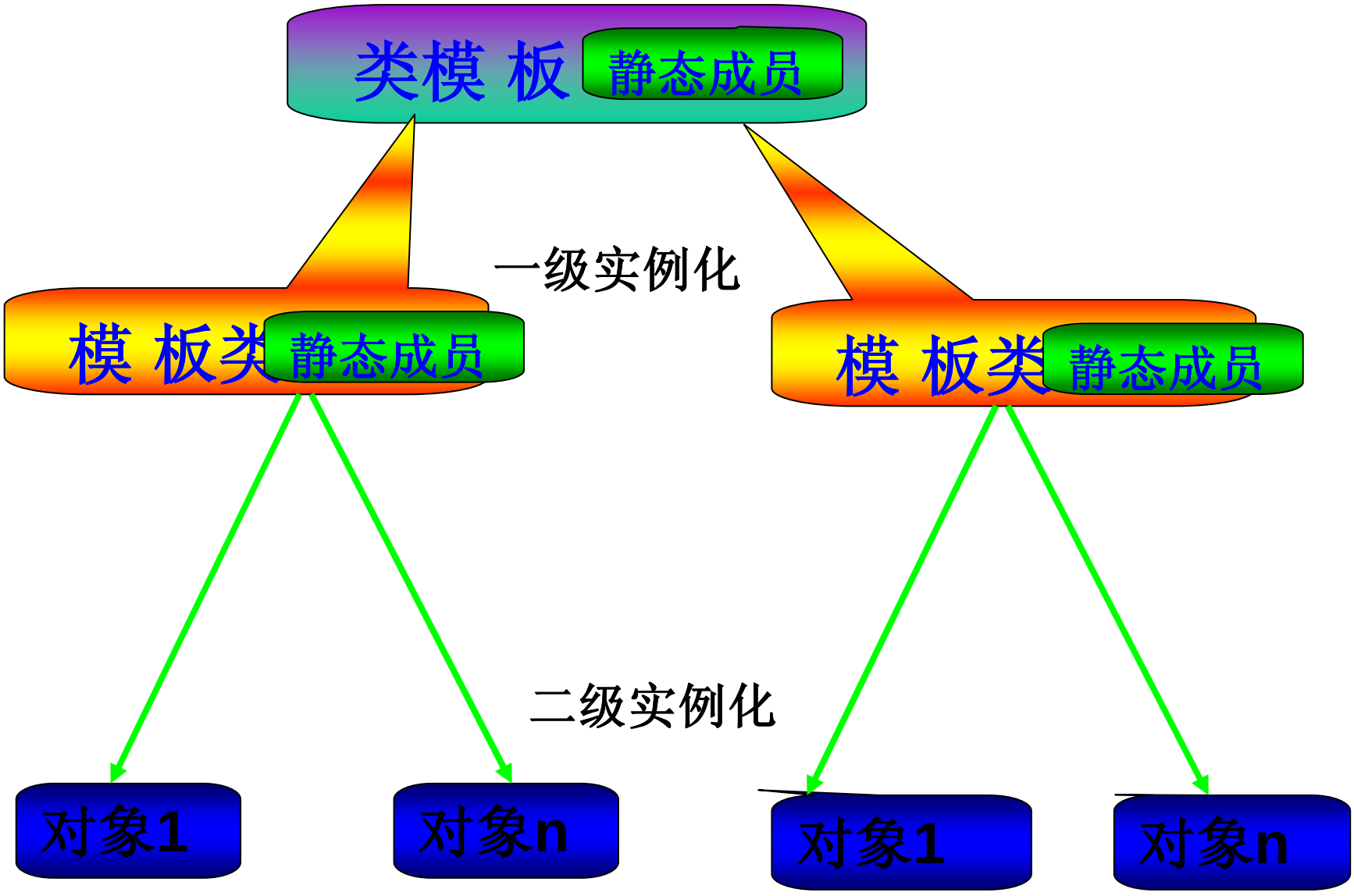
后，第二种错误就不存在了。

类模板与静态

类模板

- 类模板可以含有静态成员。这就使类模板的每个模板类的所有对象都共享静态成员；
- 类模板定义时不会创建静态成员，即静态成员在实例化前是不存在的。是由模板类创建、由模板类的对象使用的。

类模板与静态



类模板与友元

类模板

类模板可以含有友元（函数和类）。

1. 若友元函数是普通函数，则它将是这个类模板的所有模板类的友元函数；
2. 若友元函数是函数模板，但其模板参数与类模板的模板参数无关，则这个友元函数所有的模板函数都将是类模板的所有模板类的友元函数；
3. 若友元函数是函数模板，其模板参数与类模板的模板参数相关，则这个友元函数的模板函数有可能是该类模板的特例（不是所有的）类的友元；

类模板的继承

1.类模板基类派生出类模板子类

```
#include <iostream>
template <class T>
class Base
{
  public:
  void Showb(const T& b)  const  { std::cout << b << std::endl; }
};
template <class T1, class T2>
class Derived : public Base<T2>
{
  public:
  void Showd(const T1& obj1, const T2& obj2) const
  { std::cout << obj1 << " “ << obj2 << std::endl; }
};
```

```
int main()
{
    Derived<char*, double>obj1;
    obj1.Showb(55.5);       //只显示父类部分
    obj1.Showd(“The value is:”, 999.9);     //全显示
    Derived<int, int>obj2;
    obj2.Showd(32, 89);
    Derived<float, char*>obj3;
    obj3.Showd(5.8, "is ok.");
    Derived<char, char*>obj4;
    obj4.Showd('I', "and you");
}
// 调试出本运行结果，作为练习。
```

类模板的继承

2. 类模板基类派生出普通的子类

```
#include <iostream>
template <class T>
class Base
{
public:
  void Showb(const T& b) const
  { std::cout<<“Base: "<< b << std::endl; }
};
class Derived1 : public Base<int>
{
public:
  void Showd1(char* obj1, int obj2) const
  { std::cout << obj1 << " " << obj2 << std::endl; }
};
```

```
class Derived2 : public Base<double>
{
  public:
     void Showd2(char*obj1, double obj2) const
     { std::cout<<obj1<<" "<<obj2<<std::endl;}
};

int main()
{
    Derived1 obj1;                          //注意语法！
    obj1.Showb(58);
    obj1.Showd1(“obj1: ", 25);
    Derived2 obj2;
    obj2.Showd2("obj2: ", 89.76);
     std::cout<<std::endl;
}
```

//作为练习，调试出运行结果。

类模板的继承

3.普通基类派生出类模板子类

```
#include <iostream>
class Base       //普通的基类Base
{
  public:
     Base(int a) : x(a)  { }
     int Getx() const {   return x;  }
     void Showb()  const  {  std::cout << x << std::endl;  }
  private:
     int x;
};
```

```
template <class T>              //派生出一个模板类Derived
class Derived : public Base
{
public:
  Derived(const T& a, int b) : Base(b), y(a)  { }
  const T& Gety() const { return y; }
  void Showd() const  { std::cout<<y<<" "<<Getx()<<std::endl;}
private:
  T y;
};

int main()
{
  Base A(458);
  A.Showb();
  Derived<int>D1(5678,1234);
  Derived<float>D2(5.7,1234);
  Derived<char *>D3("It is ",1234);
  Derived<char >D4('=',1234);
  D1.Showd();      D2.Showd();
  D3.Showd();      D4.Showd();
}
```

// 调试一下，作为练习。

类模板作函数参数

- 函数的形参可以是类模板或类模板的引用。则实参应是该类模板实例化的模板类的对象，该函数一定是函数模板。

定义一个函数模板：

```
template<typename  T>
void TShow(const  Array<T>&x , int  index)
   { std::cout<<x.Entry(index)<< std::endl;  }
```

模板类创建对象：

```
Array<double>   DouArr(10);
```

以对象为实参调用函数：

```
TShow(DouArr , 4);  // 怎么调用的？
```

模板的模板参数（模板嵌套）

一个类若使用了(组合、聚集)另一个类模板，于是可能需要多次指定模板参数类型。如对于 **stack** 则要同时传递容器类型和所容纳的元素类型：

Stack<int,std::vector<int> > vStack;

这样分别指明，会造成了两个**int** 的无关联而留下隐患。若将第**2**个参数改为模板的模板参数，则不会造成类型的无关联，从而消除隐患。**stack**应声明如下：

```
template <typename T,
    template <typename ELEM> class CONT =
    std::deque >
class Stack {
  private:
      CONT<T> elems;        // elements
```

```
public:
    void push(T const&);   // push element
    void pop();                // pop element
    const T& top() const;  // return top element
    T& top();
    bool empty() const
   {
        // return whether the stack is empty
        return elems.empty();
   }
};
```

例中的第2参数是模板的模板参数，是个类模板，并且由第1参数 T(CONT<T> elems)传入的类型来实例化：template <typename ELEM> class CONT = std::deque >。缺省值也由std::deque<T>写成std::deque .

“类模板”与“继承”的选择

其分水岭是“类型”——

是否需要用类的类型来影响类的行为？

若不需要，则用模板；

若需要，则用继承和虚函数 。

C++对重载、模板、转换的匹配规则

1. 首先寻找参数类型完全匹配的函数,若找到且仅有一个，则调用之；若有多个，则出错。
2. 若没找到，再寻找是否有模板函数，且实例化后参数类型能匹配的,若找到，则将其实例化，以备调用；
3. 若还没找到，再寻找是否有函数，其参数类型经隐式转换后能匹配的,若找到，则调用之；
4. 经以上努力，仍然找不到合适的函数，则出错。

先重载，再模板，最后转换。

模板的多态性

```
 template <typename T>
void fun(const  T & a)  // 求面积函数
{
  std::cout << a.area() ;   // 调用类求面积成员函数
}
void g (const Circle&  x, const Rectangle&  y)
{
    fun( x) ;  // 这个函数能求圆面积
    fun ( y); // 这个函数能求长方形面积
}
// 请注意：这种多态属于静态多态，是编译时确定的 。
```

模板的多态性

下面的代码是使用虚函数来实现动态多态的 。（没用模板）

```
// house.hpp
#ifndef HEADER_HOUSE
#define HEADER_HOUSE
#include<iostream>
class House   // 房屋类（与轿车类无关）
{
public:
    void open() const
    {  std::cout<<"Open the house door.\n";  }
};
#endif  // HEADER_HOUSE
```

动态多态(通过虚函数)

```
// car.hpp
#ifndef HEADER_CAR
#define HEADER_CAR
class Car    // 轿车类 （是个抽象类）
{
public:
   virtual void open() const = 0;
};
#endif  // HEADER_CAR
// cari.hpp
#ifndef HEADER_CARI
#define HEADER_CARI
#include"car.h"
#include<iostream>
class Car1 : public Car //  公有继承了Car
{
public:
    void open() const { std::cout<<"Open the Car1 door.\n"; }
};
```

动态多态(通过虚函数)

```
class Car2 : public Car
{
public:
  void open() const { std::cout<<"Open the Car2 door.\n"; }
};
class Car21 : public Car2
{
public:
  void open() const { std::cout<<"Open the Car21 door.\n"; }
};
class Car22 : public Car2
{
public:
  void open() const { std::cout<<"Open the Car22 door.\n"; }
};
#endif  // HEADER_CARI
```

动态多态(通过虚函数)

```
#include"house.hpp"
#include“cari.hpp”  // 只包含cari.hpp就可以了
#include<iostream>
#include<vector>

void openHouse(const House& a)
{
    a.open();
}

void openCar(const std::vector<Car*>& bs)
{
   for(unsigned i=0; i<bs.size(); ++i)
      bs[i]->open();
}
```

动态多态(通过虚函数)

```
int main()
{
  House xa;
  openHouse(xa);

  Car21 b21;
  Car2 b2;
  Car1 b1;
  Car22 b22;
  std::vector<Car*>  vb;
  vb.push_back(&b21);
  vb.push_back(&b2);
  vb.push_back(&b1);
  vb.push_back(&b22);
  openCar(vb);
}
```

使用模板实现同样的效果，不仅节约内存，而且执行速度更快。

```
// house.hpp
#ifndef HEADER_HOUSE
#define HEADER_HOUSE
#include<iostream>
class House   // 房屋类（与轿车类无关）
{
public:
    void open() const
    {  std::cout<<"Open the house door.\n";  }
};
#endif  // HEADER_HOUSE
```

```
// cari.hpp
#ifndef HEADER_CARI
#define HEADER_CARI
#include<iostream>
class Car1                    // 是否继承不影响结果，只要能open就行。
{
public:
   void open() const { std::cout<<"Open the Car1 door.\n"; }
};
class Car2
{
public:
  void open() const { std::cout<<"Open the Car2 door.\n"; }
};
class Car21
{
public:
  void open() const { std::cout<<"Open the Car21 door.\n"; }
};
class Car22
{
public:
  void open() const { std::cout<<"Open the Car22 door.\n"; }
};
#endif  // HEADER_CARI
```

使用了模板，用静态多态实现同样的效果，代码更简洁。

```
#include"house.hpp"
#include"cari.hpp“
#include<iostream>

template<typename T>
void doOpen(const T& x)  // 能open是T的关键所在
{
  x.open();
}

int main()
{
  doOpen(House());
  doOpen(Car21());
  doOpen(Car2());
  doOpen(Car1());
  doOpen(Car22());
}
```

编程练习

1. 设计和编写一个双向循环链表dlist(用模板)，至少提供构造函数，拷贝构造函数，析构函数，赋值运算符成员函数，求表头和表尾元素(front, back)在链表的任何一端插入和删除的操作(push_front, pop_front, push_back, pop_back)，在指定位置插入和删除的操作(insert, erase)，判断是否链表空(empty)，求链表中元素个数(size)，反转整个链表(reverse)，清空整个链表(clear)，交换两个链表(swap)等成员函数。
2. 编写一个栈的类(用模板来做行为类似于STL中的stack)，测试时用单链表来配接(参阅第2次课课后练习slist)和上题中双链表来配接。
3. 调试运行本次课中给出的所有例子程序。
4. 编写一个环形双端队列(用模板来做)，至少提供至少提供构造函数，拷贝构造函数，析构函数，赋值运算符成员函数，求表头和表尾元素(front, back)，在任何一端插入和删除的操作(push_front, pop_front, push_back, pop_back)， 判断是否队空(empty)，判断是否队满(full)，求队中元素个数(size)，清空整个队列(clear)，交换两个队列(swap)，队列的最大容量(capacity)等成员函数。

Thanks!