C++ 11/14/17(Session 4)

Gong xuan Jun.2021

课程内容

- 函数模板
  - 模板实例化
  - 函数模板特化
  - 重载函数模板
- 模板类
  - 模板类的实例化
  - 模板类的特化
  - 模板类的偏特化
  - 见session 5

函数模板(Function template)

函数模板

使用和实现通用算法

- 函数模板是通用的函数描述，使用泛型来定义函数，其中的泛型可用具体的类型替换。
- 函数模板允许以任意类型的方式定义函数。

示例：

```
template <class T>
void Swap(T*a,T* b){
 T temp;
 temp = *a;
 *a = *b;
 *b = temp;
}
```

当将同一算法用于不同类型时，请使用模板

函数模板实例化

显示实例化

- **显示实例化：可以直接命令编译器创建特定的实例，其语法举例：**

template void swap<int>(int,int*);// 编译器看到这种声明后，将使用Swap()模板生成一个使用int类型的实例，完整的代码如下：*

```
template <class T>
void Swap(T*a,T*b)
{
  T temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

```
template  void Swap<int>(int*,int*);
int main()
{
   int a=10;
   int b=20;

   Swap(&a , &b);
   return 0;
}
```

函数模板特化（一）

- 对于之前的Swap函数模板，如果交换两个字符串数组，如何做到?

```
const char* firstArray[] = {"one","two"};
const char* secondArray[] = {"three","four"};
```

- 提供一个特例化函数定义来解决上面的问题:
  - 当编译器找到与函数调用匹配的特例化定义时，将使用该定义，而不再寻找模板。
  - 特例化的原型和定义应以template<>打头，并通过名称来指出类型，例如:

```
template <>
void Swap(const char**first, const char**second){//...}
```

函数模板特化（二）

- 函数模板特化引入了重载和实参演绎两个概念，有别于类模板特化；
- 模板特化不能包含缺省的实参值，例如下面的声明编译时将发生错误

```
template <>
    void Swap<const char*>(const char**first, const char**second,int len=2) //
```

实参演绎即用实参类型演绎声明中给出的参数类型

函数模板特化（三）

- 对于基本模板指定的缺省实参，显示特化版本可使用这些缺省的实参值,例如，下面的代码输出结果是什么。

```
template <class T>
void Swap(T*a,T*b,int len =10){//...}

template <>
void Swap<const char*>(const char**first, const char**second,int len){
 //...
 std::cout<<"array length:"<<len<<std::endl;
}
const char* firstArray[] = {"one","two"};
const char* secondArray[] = {"three","four"};
Swap(firstArray,secondArray);
```

重载函数模板（一）

- 同名模板以及各自的实例化体可同时存在，即使这些实例化体具有相同的参数类型和返回类型，例如用int*和int分别替换下面两个模板。

```
template<typename T>
 int  fun(T){
 return 1;
 }
 template<typename T>
 int fun(T*){
  return 2;
 }
std::cout<<fun<int*>((int*)0)<<std::endl;
std::cout<<fun<int>((int*)0)<<std::endl;
```

重载函数模板（二）

- 同一作用域中声明重载的函数模板，实例化过程可能会导致二义性。

```
template<typename T1, typename T2>
void foo(T1,T2){
  std::cout<<"foo(T1,T2)"<<std::endl;
}
template<typename T1, typename T2>
void foo(T2,T1){
  std::cout<<"foo(T2,T1)"<<std::endl;
}
```

- 下面的几个实例化是否都正常?

```
foo<int ,char>(10,'a');
foo<int ,char>('a',10);
foo<char ,char>('a','b');
```

重载函数模板（三）

重载函数模板的局部排序

- 重载解析总是会选择一个最佳的函数并进行调用，例如下面的显示模板实例化（下面的代码基于上面的函数模板）；

```
std::cout<<fun<int*>((int*)0)<<std::endl;
std::cout<<fun<int>((int*)0)<<std::endl;
```

- 如果没有提供显示模板实参的情况下， 也会有一个函数被选中，此时模板实参演绎将会起作用，考虑下面的例子（下面的代码基于上面的函数模板）：

```
std::cout<<fun(0)<<std::endl;
std::cout<<fun((int*)0)<<std::endl;
```

模板解析的额外规则：选择“产生自更特殊的模板的函数”

重载函数模板（四）

函数模板与非模板函数重载

- 函数模板也可以和非模板函数同时重载，其他所有条件都一样的时候，实际的函数调用将会优先选择非模板函数;

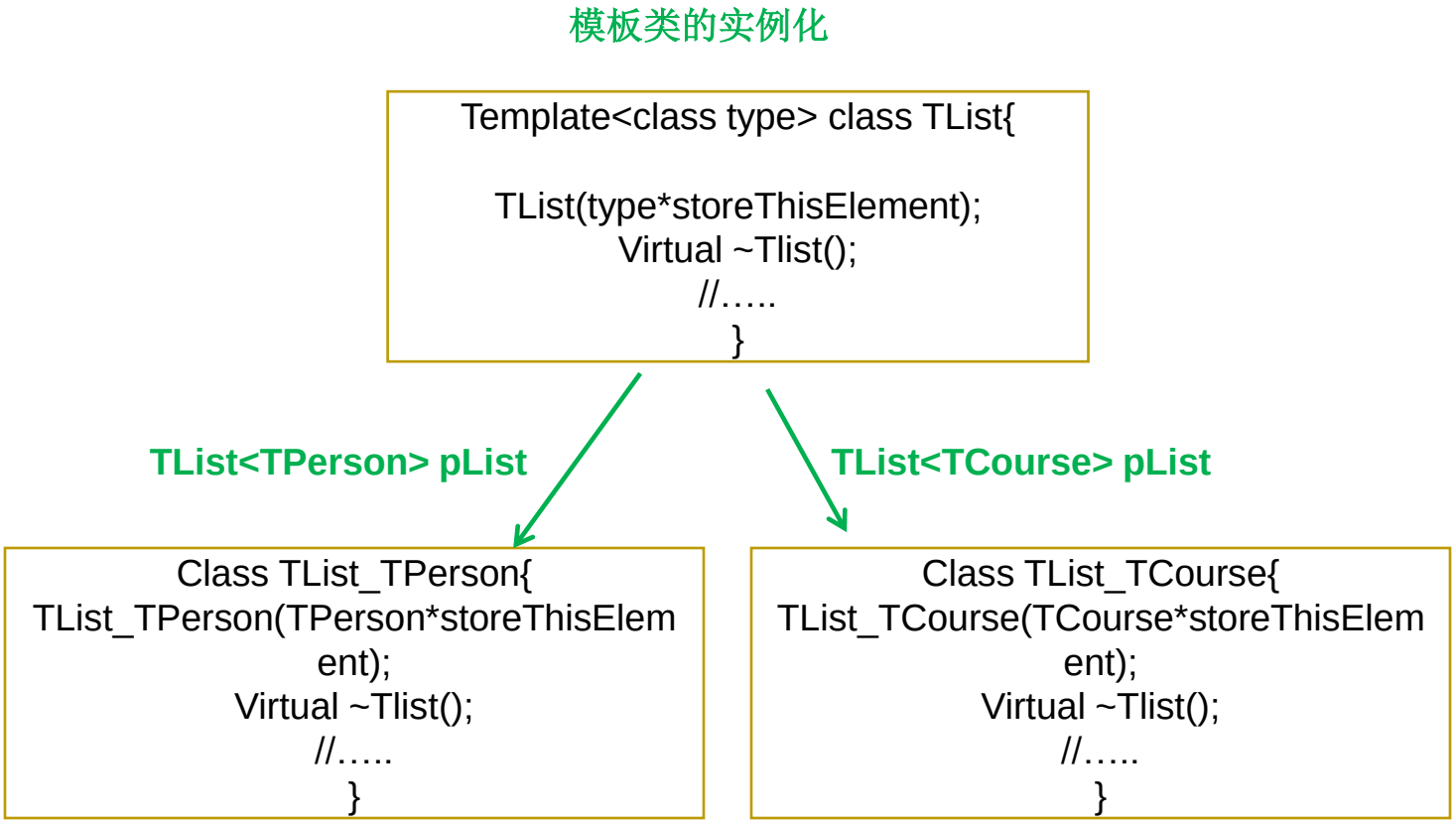
```
template <typename T>
 void Swap(T *a, T *b);

template <typename T>
void Swap(T *a, T *b，int n);  // template function

void Swap(const char**first, const char**second);  //non-template function
```

模板类(Template class)

模板类实例化

代码复制

- 当为模板形参提供实参而从模板类实例化新类时，模板类的所有代码（接口和实现代码）都将复制在生成的新类中，新类中，实参替换了形参。
- “代码复制”指代码生成、所生成代码的类型检查等整个过程。



模板类实例化

选择性实例化

- 编译器总是为下面两种情况生成代码
  - 泛型类的所有虚方法
  - 为实际某个类型调用的非虚方法

```
template <typename T>
  class A{
   public:
        A();
        ~A();
        void fun(int x, const T& y);
        T&  get();
    //...
  }
  A<int>  intA;
  intA.fun(1,10);
```

只会为int版本的 A生成无参构造函数、析构函数和方法fun()的代码

模板类常见问题

首先让我们看两个例子：

- 某个小组开发了一个模板类TList, 在它的实现中使用IsEqual成员函数调用，但是客户可能希望使用operator==,而不是IsEqual，而开发人员又没有时间修改整个实现，如何办？这个在商业软件的开发中非常常见。

- 如果客户希望使用TList<int>,但是，TList<int>::Exists的实现无法编译(因为int不是类)，显然问题来自Exists方法的生成，如何办?

对于这两个例子，如果能实现一个特殊的成员函数，则问题可以解决,例如， 如果我们能够阻止从 TList的通用代码中生成 TList<int>::Exists,我们的问题将会解决

模板类的特化

全局特化

- 要特化一个类模板，需要特化该类模板的所有成员函数，下面的例子使用int类型特化list模板

```
template <typename T>
class TList{
    public:
        TList(T* storeThisElement);
        virtual ~TList();
        //....
        bool exists(const  T& thisObject) const;
};
```

```
template <>
class TList<int>{
        public:
          TList(int* storeThisElement);
          virtual ~TList();
          //....
          bool exists(const int& thisObject) const;
};
```

无论全局特化还是部分特化都没有引入一个全新的模板或模板实例，它们只是对基础模板中已经隐式声明的实例提供了另一种定义

模板类的特化

部分特化（一）

- 部分特化允许特化部分模板参数，而不像普通的特化那样特化全部参数:

```
template <typename T, size_t WIDTH, size_t HEIGHT>
class Grid{
  //...
};
```

特例化这个模板类

```
template <size_t WIDTH, size_t HEIGHT>
class Grid<const char*, WIDTH, HEIGHT>  {
 //...
}
```

实例化模板时必须指定3个参数,下面两个实例化有什么不同?

```
Grid<int, 2, 2> grid1;
Grid<const char*, 2, 2> grid2
```

模板类的特化

部分特化（二）

- 部分特化可为一个类型子集编写特例化,而不需要为每个类型特例化，这是真正强大之处

```
templete<typename T>
class Grid<T*>
{
  //...
 }
```

上面的语法表明这个类是Grid模板对所有指针类型的特例化

模板类的特化

部分特化（三）

- 部分特化声明的参数列表和实参列表，有一些重要的约束
  - 部分特化的实参必须和基本模板的相应参数在种类上（类型，非类型或者模板）是匹配的
  - 部分特化的参数列表不能具有缺省实参；但部分特化仍然可以使用基本类模板的缺省实参
  - 部分特化的非类型实参只能是非类型值，或者是普通的非类型模板参数；不能是更复杂的依赖型表达式（如，2*T，T是模板参数）
  - 部分特化的模板实参列表不能和基本模板的参数列表完全等同

Q&A

/