C++ 11/14/17(Session 5)

Gong xuan Jun.2021

课程内容

- 模板类
  - 模板类型参数
  - 模板非类型参数
  - 模板参数模板
  - 可变参数模板
  - 类模板类型推导
- 模板元编程
  - 元编程介绍
  - 模板元数据
  - 模板元函数
  - Type_traits

模板类(Template class)

模板参数

模板类型参数

- 模板参数:类型参数、非类型参数和模板参数模板
- 模板的类型参数是模板的精髓

下面的例子,用户可通过后一个参数类型指定底层容器是vector 或deque

```
template<typename T, typename Container>
 class Grid {
  //...
  private:
    std::vector<Container> mCells
 };
```

Container并不意味着其一定是个容器

可按以下方式使用Grid对象

```
Grid<int,vector<int>> vectorGrid;
Grid<int,deque<int>> dequeGrid;
Grid<int,int> intGrid; //what's problem?
```

也可给模板参数指定默认值

```
template<typename T, typename Container=std::vector<T>>
```

模板参数

非类型的类模板参数

- 非类型参数表示的是在编译期或链接期可以确定的常值;
- 非类型模板参数只能是右值;
- 对于函数模板和类模板，普通值也可以作为模板参数，例如:

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

Grid<int,10,20> grid;
Grid<int,int,int> grid; //it's ok?
```

模板参数

非类型的函数模板参数

- 非类型参数也可用作函数模板定义，例如:

```
template <typename T, size_t VAL>
T add(T const& r){
  return r+VAL;
}
```

- 如果把函数或操作用作参数，那么这类函数会很有用（示例）

模板参数

非类型模板参数的限制(一)

- 非类型的模板参数只能是int,char,..long、枚举、或者指向外部链接对象的指针和引用;
- 浮点数（double，float）和类对象是不允许作为非类型模板参数的;

```
template<double  VAL>
double process(doulbe v) {
   return v*VAL;
 }
template<std::string name>
class MyClass{
...
};
```

- 不能使用全局指针或字符串作为模板参数, 例如:

```
template<char const* name>
class MyClass{
...
};
char const* s =  “hello”;
MyClass<s> x ;  // MyClass<“hello”> x
```

模板参数

使用auto声明非类型模板参数

- C++17之前,必须明确非类型模板参数的具体类型，C++17后，能够使用auto关键字在模板中声明非类型模板参数，即让编译器自动推导类型，例如:

```
template <typename T, size_t WIDTH, size_t HEIGHT>//pre-C++17
class Grid{//…};

template <typename T, auto WIDTH, auto HEIGHT> //c++17
 class Grid{//…};

// function templete
template <typename T, size_t VAL> //pre-C++17
template <typename T, auto VAL> // c++17
T add(T const& r)
{
  return r+VAL;
}
```

模板参数

模板参数模板

- 将模板作为模板参数，通用的语法规则：

templete<...,template<TemplateTypeParams> class ParameterName,...>

例如：

```
template <typename T,template <typename E, typename Allocator = std::allocator<E>> class Container = std::vector>
class Grid
{
//...
 private:
   std::vector< Container<T> > mCells;
 }
```

可以这样使用上面的例子

```
Grid<int, vector> vecGrid;
Grid<int, deque>  dequeGrid;
```

类模板参数推导

- 在C++17之前，类模板构造器的模板参数是不能被自动推导的，例如：

```
std::pair a{1, 2.2}; // C++17
std::pair<int, double> a{1, 2.2}; // C++14
```

- 用户自定义的模板类

```
template<typename X, typename Y>
struct Struct{
    Struct(X x, Y y) : x(x),y(y){}
   //...
    X x;
    Y y;
};
```

```
Struct<int, std::string> myStruct(integer, str); // pre-C++17
Struct myStruct(integer, str); // C++17
```

编译器将会根据初始化器的类型自动推导出模板参数类型

可变参数模板

可变参数模板

- 可变参数模板即模板函数和模板类可接受可变数量的参数，例如下面的函数可接受下面的参数输入：

  fun(2, 10.0);

  fun(2,10.0,'c',"hello");

- 要创建可变参数模板，需要理解几个要点：
  - 模板参数包（parameter pack）
  - 函数参数包
  - 展开（unpack）参数包
  - 递归

可变参数模板

模板和函数参数包

- 模板参数包是一个**类型列表**，C++11提供了用省略号表示的元运算符用来表示模板参数包标识，同时，它也能表示函数参数包，函数参数包基本上是一个**值列表 ，基本语法：**

```
template<typename ... Args>
void fun(Args... args)
{
  //...
}
```

Args是一个模板参数包，与T的差别是什么？

args是一个函数参数包

对于fun(2,10.0,’c’,”hello”)来说，参数包Args包含与函数调用中的参数匹配的类型：

int,double,char, char*

可变参数模板fun()与下面的函数调用都匹配：

```
fun();
fun(10);
fun(10,'c');
```

可变参数模板

展开参数包

- 函数如何访问这些包的内容呢？索引功能在这里不适用，即不能使用Args[2]来访问包中的第三个类型。

```
void fun(){}
template<typename T, typename… Args>
void fun(T value, Args… args)
{
   cout<<value<<“,”;
   fun(args…);
}
int main(){
 int a=10;
 double b=1.41;
 string str=“hellow”;
 fun(b,’c’,8,str);
}
```

定义无参的情况，用来终止调用

定义一个或更多参数的情况

第一个实参导致T为double,其余的三种类型将放入Args，而其他三个值放入args中，依次类推，采用递归调用，最后args为空时，将调用不接受任何参数的fun(),导致处理结束

模板元编程(Template meta programming)

模板元编程

模板元编程概念(一)-一个计算阶乘的例子

- 模板元编程本质上是泛型编程的一个子集，目的是在编译时执行一些计算，而不是在运行时执行；

```
template<unsigned char f>
class Factorial
{
 public:
  static const unsigned long long value = (f * Factorial<f - 1>::value);
 };

template<>
class Factorial<0>
{
 public:
        static const unsigned long long value = 1;
};
```

```
int main(){
        cout << Factorial<6>::value << endl;
        return 0;
}
```

运行时可通过 ::val访问编译时计算出来的值，这是一个静态常量

模板元编程

模板元编程概念(二)

- 模板元编程是一种可产生程序的程序；
- 模板元编程是许多boost库组件的基础；遵循C++语法，但编程思想与C++有很大的不同；
- 编译期执行，操作的对象不是普通的变量，不能使用C++运行时关键字（如if,else,for等），常用语法元素有：
  - enum,static　　定义编译期的整数常量
  - typedef,using　　定义元数据
  - templete　　定义元函数
  - ::　　域运算符

模板元编程

元数据（meta data）

- 元数据是C++编译器在编译阶段可操作的数据
- 模板元编程中的元数据更多的是以类型的方式出现，如int，class这样的抽象数据类型
- 元数据类型
  - 整数元数据
  - 值型元数据
  - 函数元数据
  - 类元数据
- 声明元数据举例

typedef int meta_data //元数据meta_data,值为int

typedef std::vector<int> meta_data1 //元数据meta_data1,值为vector<int>

*typedef void(*meta_data2)(int,int)*

模板元编程

元函数(meta function)-规则

- 元函数用于操作元数据，编译期被调用，功能和形式类似运行时的函数
- 元函数实际上表现为C++中的一个类或模板类

- 元函数规则：

```
templete<typename T1>
struct meta_function
{
 typedef T1 type;  // 返回T1,等价于using type = T1
 type value;  //返回值
}
```

模板元编程

元函数(meta function)-值元函数

- 下面的值元函数在编译期比较两个int类型的整数大小

```
#include <iostream>

template<int firstValue,int secondValue>
struct get_min_value
{
  static const int value = (firstValue < secondValue)?firstValue:secondValue;
};
int main()
{
 std::cout<<get_min_value<10,20>::value<<std::endl;
 return 0;
}
```

模板元编程

元函数(meta function)-操作类型的元函数

- 下面的元函数对类型进行操作，如果输入的元数据T是指针类型，则返回const T,否则返回const T*

```
template<typename T>
struct return_type{
  typedef const T* type;  //return const T*
};
template<typename T>
struct return_type<T*> { //template specilization for T*

  typedef const T type; //return const T
};
int main(){
  std::cout<<std::is_same<return_type<int>::type  , const int*>::value<<std::endl;
  std::cout<<std::is_same<return_type<int*>::type , const int>::value<<std::endl;
   return 0;
}
```

type_traits

- type_traits以库的方式实现了类型特征提取功能，是泛型编程和模板元编程所必须的基础设施

- type_traits已经成为了C++11标准的一部分(头文件<type_traits>),但boost.type_traits并不完全与标准一致
- type_traits位于名字空间boost(为了使用type_traits组件，需要包含头文件<boost/type_traits.hpp>)，像下面这样：

```
#include <boost/type_traits.hpp>
using namespace boost;
```

通过类型traits，可在编译时根据类型做出决策

模板元编程

type_traits 分类

原始类型类别
- is_void<T>
- is_integral<T>
- is_pointer<T>
- …

复合类型类别
- is_reference
- is_object
- …

类型属性
- is_const
- is_unsigned
- is_move_constructible
- …

类型关系
- is_same
- is_convertible
- …

Const_volatile 修改
- remove_const
- add_const
- …

引用修改
- remove_reference
- add_lvalue_reference
- add_rvalue_reference
- …

符号性修改
- make_signed
- make_unsigned
- …

其他转换
- enable_if
- conditional
- …

type_traits-使用类型分类（以is_integral为例）

- 以is_integral为例了解这种类的工作方式，is_integral继承自integral_constant，其实现：

```
template<class T, T v>
struct integral_constant {
  static constexpr T value = v;
  typedef T value_type;
  typedef integral_constant type;
  constexpr operator value_type() const noexcept { return value; }   constexpr value_type operator()()
 const noexcept { return value; } //since c++14
 };
typedef integral_constant<bool,true> true_type;
typedef integral_constant<bool,false> false_type;
```

type_traits-使用类型分类（以is_integral为例）

- 结合模板根据类型的某些属性生成代码时，type traits才更有用（代码示例）
- 有三种类型关系:is_same,is_base_of和is_convertible（代码示例）

SFINAE

- SFINAE（替换失败不是错误， Substitution Failure Is Not An Error）主要用于解决重载歧义的情况，即无法使用其他技术（如特化）解析歧义时使用。

- 如果有选择地使用SFINAE和enable_if禁用重载集中的错误重载，有时会得到奇怪的编译错误，这些错误很难跟踪

代码示例

Q&A

/