基于 Linux 的 C++

第十一讲　泛型编程

提　纲

泛型编程概览

- 什么是泛型编程？
- 为什么需要泛型编程？
- 怎样进行泛型编程？

泛型编程实践

- 标准模板库
- 泛型算法：函子与完美转发
- 泛型数据结构：队列
- 泛型元编程：Fibonacci数列、素数枚举
- 工程实践：事件机制

泛型编程概览

什么是泛型编程？

- 泛型就是通用型式
- 编写不依赖数据对象型式的代码就是泛型编程

为什么需要泛型编程？

- 动机三问：函数重载、相似类定义与型式兼容性

怎样进行泛型编程？

- 泛型编程技术手段：模板与型式参数化

函数重载问题

设计函数，求两个数据对象的较小者

```
//  未明确规定参数型式，因C/C++的强型式检查特性
//  必须为不同型式的参数分别实现，没完没了……
//  函数重载数目巨大，并且永不完备……
int  Min( int a, int b );
double  Min( int a, double b );
double  Min( double a, int b );
double  Min( double a, double b );
class A;
const A &  Min( const A & a, const A & b );
class B;
const B &  Min( const B & a, const B & b );
……
```

函数重载问题

解决方案：使用C的含参宏

- #define Min(x, y) ((x) < (y) ? (x) : (y))
- a = Min(b, c) * Min(b+c, b); ==>
  a = ((b) < (c) ? (b) : (c)) * ((b+c) < (b) ? (b+c) : (b));

缺　点

- 无型式检查，无法在编译期检查程序错误
- 宏文本替换时，要注意操作符优先级，错误的宏文本有可能导致问题

结　论

- 需要一种机制，能够在语法层面解决宏的问题

相似类定义问题

动态数组类

- 定义存储整数的动态数组类
- 定义存储浮点数的动态数组类
- 定义存储某类对象的动态数组类
- 定义存储某类对象指针的动态数组类
- ……

结　论

- 需要一种机制，能够在语法层面解决相似类的重复定义问题，降低编程工作量

型式兼容性问题

C型式转换：(T)x

- 不安全
- 内建型式（int、double）对象转换安全性基本保证
- 类对象转换可能导致无法控制的严重后果

C++型式转换：T(x)

- 可能需要单参数构造函数和重载的型式转换操作符
- 不安全
- 如果未实现，转换就不存在

型式兼容性问题

类库架构

- 类的继承和多态频繁要求能够通过基类的指针或引用访问派生类的对象
- 需要沿着类的继承层次，频繁进行对象的型式转换

存在的问题

- C/C++已有型式转换均为静态转换，不能适应指针或引用的多态性
- 型式转换必须适应全部型式，并能自如操作；然而很不幸，型式无穷尽，程序员无法编写完备的型式转换代码

型式兼容性问题

保证型式兼容性的机制

- 确保型式转换操作合法有效，并在失败时通知用户
- 需要维持对象的运行期型式信息（run-time type information, RTTI）
- 转换结果确认：通过转换操作的返回值确认结果，或者在失败时触发特定信号；后者需要使用异常处理机制

实现策略：模板与型式参数化

异常处理机制

异常处理机制基础

异常的引发

异常的捕获

异常类与异常对象

异常处理策略

异常描述规范

异常处理机制基础

异常的定义

- 程序中可以检测的运行不正常的情况
- 示例：被0除、数组越界、存储空间不足等

异常处理的基本流程

- 某段程序代码在执行操作时发生特殊情况，引发一个特定的异常
- 另一段程序代码捕获该异常并处理它

栈

```
class JuStack
{
public:
  JuStack( int cap ) : _stk(new int[cap+1]), _cap(cap), _cnt(0), _top(0)  {  }
  virtual ~JuStack()  {  if( _stk )  delete _stk, _stk = NULL;   }
public:
  int Pop();
  void Push( int value );
  bool IsFull() const  {  return _cap == _cnt;  }
  bool IsEmpty() const  {  return _cnt == 0;  }
  int GetCapacity() const  {  return _cap;  }
  int GetCount() const  {  return _cnt;  }
private:
  int * _stk;
  int   _cap,  _cnt,  _top;
};
```

栈

存在的问题

- 调用成员函数JuStack::Pop()时，栈是空的
- 调用成员函数JuStack::Push()时，栈是满的

解决方案

- 定义异常类
- 修改成员函数，在出现异常情况时引发之
- 在需要的位置处理该异常

异常的引发

```
// 异常类：空栈异常类与满栈异常类
class EStackEmpty  {  };
class EStackFull  {  };

int JuStack::Pop()
{
  if( IsEmpty() )   //  引发空栈异常，构造该异常类的一个对象并抛出
    throw EStackEmpty();
  --_top,  --_cnt;
  return _stk[_top];
}

void JuStack::Push( int value )
{
  if( IsFull() )   //  引发满栈异常，构造该异常类的一个对象并抛出
    throw EStackFull();
  _stk[_top] = value;
  _top++,  _cnt++;
}
```

异常的捕获

```
const int err_stack_full = 1;

int main()
{
  JuStack stack( 17 );
  try
  {
    for( int i = 0; i < 32; i++ )
      stack.Push( i );    //  可能引发异常的函数调用
  }
  catch( const EStackFull & )    //  捕获抛出的异常，执行相应处理
  {
    std::cerr << "Error: Stack full" << std::endl;
    return err_stack_full;
  }
  return 0;
}
```

异常类与异常对象

```
// 精心设计异常类，提供必要的异常信息
class EStackFull
{
public:
  EStackFull( int i ) : _value( i )  {  }
  int GetValue()  {  return _value;  }
private:
  int _value;
};

void JuStack::Push( int value )
{
  if( IsFull() )
    throw EStackFull( value );    //  使用value构造异常类对象并抛出
  _stk[_top] = value;
  _top++,  _cnt++;
}
```

异常类与异常对象

```
const int err_stack_full = 1;

int main()
{
  JuStack stack( 17 );
  try
  {
    for( int i = 0; i < 32; i++ )
      stack.push( i );
  }
  catch( const EStackFull & e )   // 使用异常对象获取详细信息
  {
    std::cerr << "Stack full when trying to push " << e.GetValue() << std::endl;
    return err_stack_full;
  }
  return 0;
}
```

异常处理策略

异常类可以派生和继承，形成类库架构

可捕获的异常对象的型式

- **普通型式（包括类）：异常对象需要拷贝**
- **对某型式对象的引用：没有额外的拷贝动作**
- **指向某型式对象的指针：要求对象动态构造或者在catch子句中可访问**

异常处理策略

catch子句

- 可以有多个catch子句，每个负责捕获一种、一类或全部异常
- 捕获一种：catch(int)、catch(const char *)
- 捕获一类（该类或其派生类异常）：catch(const EStackFull &)
- 捕获全部：catch(...)
- 所有catch子句按照定义顺序执行，因此派生异常类处理必须定义在基类之前，否则不会被执行

异常处理策略

异常再引发

- 可以在基本任务完成后重新引发所处理的异常
- 主要用于在程序终止前写入日志和实施特殊清除任务

```
try
{
  throw AnException();
}
catch(...)
{
  // ...
  throw;
}
```

异常处理策略

栈展开

- 异常引发代码和异常处理代码可能属于不同函数
- 当异常发生时，沿着异常处理块的嵌套顺序逆向查找能够处理该异常的catch子句
- 如找到对应catch子句，处理该异常
- 异常处理完毕后，程序保持catch子句所在的函数栈框架，不会返回引发异常的函数栈框架
- 函数栈框架消失时，局部对象被析构，但如果未执行delete操作，动态分配的目标对象未析构

异常处理策略

未处理异常

- 所有未处理的异常由预定义的std::terminate()函数处理
- 可以使用std::set_terminate()函数设置std::terminate()函数的处理例程

```
void term_func()  {  exit( -1 );  }
int main()
{
  try
  {
    set_terminate( term_func );
    throw "Out of memory!";
  }
  catch( int ){  /* ... */  }
  return 0;
}
```

异常处理策略

描述函数是否引发异常

- 否：throw()
- 是，引发任意型式的异常：throw(...)
- 是，引发某类异常：throw(T)，部分编译器将其作为throw(...)

C++11规范

- 否：noexcept，等价于noexcept(true)
- 是：noexcept(false)
- 可能：noexcept(noexcept(expr))，expr为可转换为true或false的常数表达式
- C++11下，建议使用noexcept代替throw

异常描述规范

```
class JuStack
{
public:
  JuStack( int cap ) : _stk(new int[cap+1]), _cap(cap), _cnt(0), _top(0)  { }
  virtual ~JuStack()  {  if( _stk )  delete _stk, _stk = NULL;  }
public:
  int Pop()  throw( EStackEmpty );
  void Push( int value )  throw( EStackFull );
  bool IsFull() const  {  return _cap == _cnt;  }
  bool IsEmpty() const  {  return _cnt == 0;  }
  int GetCapacity() const  {  return _cap;  }
  int GetCount() const  {  return _cnt;  }
private:
  int * _stk;
  int _cap, _cnt, _top;
};
```

运行期型式信息

RTTI

- 运行期标识对象的型式信息
- 优势：允许使用指向基类的指针或引用自如地操纵派生类对象
- typeid：获取表达式的型式；type_info：型式信息类
- 头文件：“typeinfo”

对象转型模板

- dynamic_cast：动态转型
- static_cast：静态转型
- reinterpret_cast：复诠转型
- const_cast：常量转型

typeid操作符与type_info类

type_info 类

- 编译器实现的动态型式信息型式
- 用于在程序运行时保存数据对象的型式信息
- 不能直接使用该类，只能通过typeid操作符
- 调用成员函数name()可以获得类的名称

typeid 操作符

```
#include <typeinfo>
Programmer p;
Employee & e = p;
// 输出p实际类名的字符串“Programmer”
cout << typeid(e).name() << endl;
```

dynamic_cast

动态转型的三种方式

- 向上转型：沿着类继承层次向基类转型
- 向下转型：沿着类继承层次向派生类转型
- 交叉转型：沿着类多重继承层次横向转型

指针的动态转型

- 正确执行时，结果为指向目标类对象的指针
- 错误执行时，结果为0/NULL（C++11：nullptr）

引用的动态转型

- 正确执行时，结果为目标类对象的引用
- 错误执行时，引发bad_cast异常

dynamic_cast

假设软件公司包括程序员和经理两类职员，需要按照不同规则支付薪水和奖金。如何实现？

```
class Employee
{
public:
  virtual void PaySalary();
  virtual void PayBonus();
};
class Manager: public Employee
{
public:
  virtual void PaySalary();
  virtual void PayBonus();
};
```

```
class Programmer: public Employee
{
public:
  virtual void PaySalary();
  virtual void PayBonus();
};
class Company
{
public:
  virtual void PayRoll( Employee * e );
  virtual void PayRoll( Employee & e );
private:
  vector<Employee*> _employees;
};
```

dynamic_cast

```
void Company::PayRoll( Employee * e )          //  版本一
{    //  调用哪个成员函数？如何区分程序员和经理？
  e->PaySalary();
  e->PayBonus();
};

void Company::PayRoll( Employee * e )          //  版本二
{
  Programmer * p = dynamic_cast<Programmer*>( e );
  if( p )    //  p确实指向程序员对象
  {
    p->PaySalary();
    p->PayBonus();
  }
  else     //  p不指向程序员，不发奖金
    e->PaySalary();
};
```

dynamic_cast

```
void Company::PayRoll( Employee & e )     //  版本三
{
  try
  {
    Programmer & p = dynamic_cast<Programmer&>( e );
    p.PaySalary();
    p.PayBonus();
  }
  catch( std::bad_cast )
  {
    e.PaySalary();
  }
};
```

static_cast

静态转型的用途

- 与dynamic_cast不同，static_cast不仅可用于指针和引用，还可用于其他型式
- 一般用于非类型式的普通数据对象转型
- 可以在类继承层次上进行向上或向下转型

静态转型的问题

- 不进行运行期型式检查，不安全
- 若转型失败，结果无定义

const_cast

常量转型的目的

- 用于取消或设置量的const状态

常量转型的问题

- 如果原始数据对象不能写入，则取消常量修饰可能会导致未知结果

const_cast

```
#include <iostream>

class ConstCastTest
{
public:
  void SetNum( int num ){ _num = num; }
  void PrintNum() const;
private:
  int _num;
};

void ConstCastTest::PrintNum() const
{
  // 临时取消常量约束，修改目标对象的内容
  const_cast<ConstCastTest*>( this )->num--;
  std::cout << _num;
}
```

reinterpret_cast

复诠转型的目的

- 将任意型式的数据对象转型为目标型式，即重新解释其位序列的意义
- 可以用于整型与指针型的互转

复诠转型的问题

- 由程序员保证重新解释的数据对象是否有意义，编译器简单按照目标型式理解该存储区的内容
- 注意：在64位操作系统中，指针型可能为64位，而整型可能为32位，复诠转型有可能丢失数据或得到错误结果

reinterpret_cast

```
#include <iostream>
using namespace std;

int f( void* p )
{
  unsigned int n = reinterpret_cast<unsigned int>( p );
  return n;
}

int main()
{
  int a[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
  int n = f( a );
  cout << n << endl;
}
```

模板与型式参数化

转型操作

- 接受目标型式作为模板参数
- Programmer * p = dynamic_cast<Programmer*>(e)

模板工作原理

- 使用template<typename T>定义函数模板或类模板
- 体化（instantiation）：使用特定的模板实际参数，生成真正的模板函数和模板类
- 编译模板函数和模板类，生成最终程序代码

模板代码

- 一般放在头文件中：编译器需要看到模板源代码

模板与型式参数化

模板特点

- 抽象性：模板代码高度抽象，是函数和类的模范
- 安全性：型式检查能够发现大多数型式失配问题
- 通用性：函数和类模板定义一次，按需生成函数和类的实体
- 易用性：接口相对直观且高度一致
- 效　率：减少冗余代码，提升编程效率；通过编译优化，提升程序执行效率

模板用途

- 函数模板：构造函数集，实现不依赖特定数据结构的抽象算法
- 类模板：构造类集，实现抽象数据结构
- 元编程：构造在编译期执行的运算，提升程序执行效率

题外话：术语翻译

dereference：引领，好于“解引用”

- reference：引用；英语本义：*something such as a number or a name that tells you where you can obtain the information you want*
- 引领：伸颈远望，带领

type：型式，好于“类型”

- int：整数型式，简称整型
- class：类型式，简称类型

constructor：建构函数或构造函数

- 与destructor协调，建构函数更佳

destructor：析构函数或解构函数

题外话：术语翻译

instance：定体，好于“实例”

- 英语本义：*a particular example or occurrence of something*
- 例：可以做依据的事物；调查或统计时指合于某种条件的具有代表性的事情，如事例、案例。根据类型生成的具体对象，根据模板生成的具体函数或根据类型生成的对象算什么“例”？需要调查和统计吗？它们需要代表什么吗？
- 体：事物本身或全部；物质存在的状态或形状；文章或书法的样式或风格；事物的格局与规矩
- 定体：固定不变的形态、性质、体例或体式；尽量不用“实体”，以区分entity
- 象体：按照类型构造的对象定体或对象实体的简称，好于“对象实例”
- 函体：根据函数模板生成的函数定体或函数实体的简称
- 类体：根据类模板生成的类型定体或类型实体的简称；不使用“型体”，因为型并不仅仅只有类
- instantiation：定体化，简称体化，好于“实例化”
- specialization：特体化，简称特化

泛型编程实践

标准模板库

函数模板

- 泛型算法：函子与完美转发

类模板

- 泛型数据结构：队列

元编程

- 编译期计算：Fibonacci数列、素数枚举

工程实践

- 事件机制

标准模板库

标准模板库的内容

- 标准模板类：复数、序偶
- 迭代器
- 标准容器：向量、表、栈、队列、集合、映射等
- 标准算法：查找、排序等

标准模板库型式的使用方法

- “<>”：模板名称<数据对象基型式> 数据对象名称;
- 示例一：complex<double> a(1.0, 2.0);
- 示例二：pair<string, string> name("Zhang", "San");
- 示例三：vector<int> v(8);

复　数

一般说明

- 头文件：“complex”
- 模板名：complex<>
- 基型式：float、double、long double
- 首选double，float精度太低，long double已废弃

实部与虚部

- 成员函数real()与imag()

复数操作

- 复数全部操作均可以按照数学格式进行
- cout、cin均已重载：格式为(real,imag)

序　偶

一般说明

- 头文件：“utility”
- 模板名：pair<>
- 用于表示总是成对出现的两个对象
- 示例一：pair<int, double> a(1, 1.0);
- 示例二：pair<string, string> name("Zhang", "San");

使用规则

- 公开的数据成员：first、second
- 示例：cout << name.first << ", " << name.second;
- 序偶比较：先比较first大小，相同时比较second大小
- make_pair：构造序偶的辅助函数
- 示例：pair<int, double> a; a = make_pair(1, 1.0);

向　量

向量的目的

- 替代数组，可以像数组一样使用向量

向量的使用

- 定义格式：vector<int> v(8); // 包含8个整数元素
- operator[]：已重载，使用格式v[i]访问第i个元素
- 向量可以整体赋值
- size()：返回向量中元素数目
- capacity()：返回向量当前可存储的最多元素数目
- clear()：删除向量所有元素，但不释放向量本身
- resize(int newsize)：重新设置向量容量

迭代器

迭代器的性质

- 通过迭代器访问容器中的数据对象
- 类似指针、数组索引的功能：通过指针加减与数组下标运算获得下一数据对象
- 迭代器可以是指针，但并不必须是指针，也不必总是使用数据对象的地址

迭代器

迭代器的典型使用方法

- 声明迭代器变量
- 使用引领操作符访问迭代器指向的当前目标对象
- 使用递增操作符获得下一对象的访问权
- 若迭代器新值超出容器的元素范围，类似指针值变成NULL，目标对象不可引用

迭代器

迭代器的分类

- 输入迭代器：提供对象的只读访问
- 输出迭代器：提供对象的只写访问
- 前向迭代器：提供对象的正向（递增）读写访问
- 双向迭代器：提供对象的正向与反向（递增递减）读写访问
- 随机访问迭代器：提供对象的随机读写访问

指针作为迭代器

调用标准模板库的find()函数查找数组元素

```
#include <iostream>
#include <algorithm>
using namespace std;
const int size = 16;
int main()
{
  int a[size];
  for( int i = 0; i < size; ++i )  a[i] = i;
  int key = 7;
  int * ip = find( a, a + size, key );
  if( ip == a + size )   //  不要使用NULL做指针测试，直接使用过尾值
    cout << key << " not found." << endl;
  else
    cout << key << " found." << endl;
  return 0;
}
```

向量迭代器

使用迭代器操作向量

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main()
{
  int key = 7;
  vector<int> iv( 10 );
  for( int i = 0; i < 10; i++ )  iv[i] = i;
  vector<int>::iterator it, head = iv.begin(), tail = iv.end();
  it = find( head, tail, key );
  if( it != tail )
    cout << "Vector contains the value " << key << endl;
  else
    cout << "Vector does NOT contain the value " << key << endl;
  return 0;
};
```

常迭代器

常迭代器

- 若不想通过迭代器修改目标对象值，定义迭代器常量

示　例

- const vector<int>::iterator it;
- 非法操作：*it = 10; // 不能修改常迭代器指向的对象

流迭代器

使用迭代器访问流

- 将输入输出流作为容器

使用方式：定义流迭代器对象

- 示例一：ostream_iterator<int> oit(cout, " ");
- 示例二（从cin获取数据）：istream_iterator<int> iit(cin);
- 示例三（使用空指针创建流结束迭代器）：istream_iterator<int> iit;
- 凡是可以出现迭代器参数的标准算法都可以使用

输出流迭代器

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include <vector>
#include "random.h"
using namespace std;
const int size = 8;
const int lower_bound = 10;
const int upper_bound = 99;

void Display( vector<int> & v, const char * s )
{
  cout << endl << s << endl;
  vector<int>::iterator head = v.begin(), tail = v.end();
  ostream_iterator<int> oit( cout, "; " );
  copy( head, tail, oit );
  cout << endl;
}
```

输出流迭代器

```
int main()
{
  vector<int> a( size );
  for( int i = 0; i < size; ++i )
    a[i] = GenerateRandomNumber( 10, 99 );
  Display( a, "Array generated:" );
  vector<int>::iterator head = a.begin(), tail = a.end();
  sort( head, tail );
  Display( a, "Array sorted:" );
  reverse( head, tail );
  Display( a, "Array reversed:" );
  return 0;
}
```

输入流迭代器

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include <vector>
using namespace std;
int main()
{
  vector<int> v( 4 );
  vector<int>::iterator it = v.begin();
  cout << "Enter 4 ints separated by spaces & a char:\n";
  istream_iterator<int> head( cin ), tail;
  copy( head, tail, it );
  cin.clear();
  cout << "vector = ";
  for( it = v.begin(); it != v.end(); it++ )
    cout << *it << " ";
  cout << endl;
  return 0;
}
```

表

表：标准模板库中的为双向链表

表的使用

- 定义包含Point对象的容器：list<Point> pts(8);
- 插入：pts.insert(pts.begin(), Point(1,2));
- 表头插入：pts.push_front(Point(1,2));
- 插入：pts.insert(pts.end(), Point(1,2));
- 表尾插入：pts.push_back(Point(1,2));
- 定义包含Point指针的容器：list<Point*> ppts(8);
- 插入：ppts.insert(ppts.begin(), new Point(1,2));
- 插入：ppts.insert(ppts.end(), new Point(1,2));
- 删除：delete ppts.front(); ppts.remove(ppts.front());
- 判断表是否为空：if(pts.empty()) cout << "Empty!";

表

表与迭代器

- 迭代器可以和表协同工作，方式与向量相同

```
list<int> a(8);
list<int>::iterator it;
for( it = a.begin(); it != a.end(); it++ )
  *it = GenerateRandomNumber( 10, 99 );
```

表排序

- 直接使用表的成员函数：a.sort(); // 默认升序
- 降序排序之一：升序排序后调用成员函数reverse()
- 降序排序之二（传入函子greater_equal<int>()）：

```
a.sort( greater_equal<int>() );
```

- 对于自定义对象，需要重载operator<以进行比较

标准算法

- 查找算法
- 排序算法
- 删除和替换算法
- 排列组合算法
- 算术算法
- 关系算法
- 集合算法
- 生成和变异算法
- 堆算法

标准算法

adjacent_find：查找两个相等或满足特定条件的相邻元素

all_of：当给定区间内全部元素均满足条件时返回true

any_of：当给定区间内至少一个元素满足条件时返回true

binary_search：折半查找，原始数据集已序

copy：复制给定区间内全部元素

copy_backward：反向复制给定区间内全部元素

copy_if：复制给定区间内满足特定条件的元素

copy_n：复制特定位置处开始的指定数目的元素

count：返回给定区间内匹配特定值的元素个数

count_if：返回给定区间内匹配特定条件的元素个数

标准算法

equal：比较两个区间是否相等，逐元素比较

equal_range：给定一个已序区间，返回与特定值相等的子区间

fill：使用特定值填充给定区间内全部元素

fill_n：使用特定值填充从特定位置处开始的指定数目的元素

find：查找给定区间内特定元素，返回其首次出现位置

find_end：查找给定区间内特定序列，返回其末次出现位置

find_first_of：查找给定区间内多个特定值中任意值，返回其首次出现位置

find_if：查找给定区间内满足特定条件的元素，返回其首次出现位置

find_if_not：查找给定区间内不满足特定条件的元素，返回其首次出现位置

for_each：正向遍历给定区间，实施函子指定的操作，并返回该函子

标准算法

generate：将函子生成的值复制给给定区间内全部元素

generate_n：将函子生成的值复制给从指定位置开始的指定数目的元素，返回最后一次赋值的位置

includes：测试第一已序区间是否包含第二已序区间全部元素

inplace_merge：在位归并两个已序区间

is_heap：返回给定区间内元素是否形成堆

is_heap_until：返回给定区间直到最后一个元素是否形成堆

is_partitioned：给定区间是否已划分，当区间内满足条件的元素全部出现在不满足条件的元素之前时，返回true

is_sorted：给定区间的元素是否已排序

is_sorted_until： 给定区间的元素是否已排序，返回最后一个已序元素的前向迭代器

iter_swap：互换两个迭代器引用的值

标准算法

lexicographical_compare：逐元素按词典序比较两个序列，第一序列较小时返回true

lower_bound：下界，返回给定已序区间大于或等于特定值的首个元素位置

make_checked_array_iterator：创建一个checked_array_iterator

make_heap：将给定区间内元素转换为堆

max：比较两个对象，返回较大者

max_element：查找给定区间内最大元的首次出现位置

merge：归并两个已序区间

min：比较两个对象，返回较小者

min_element：查找给定区间内最小元的首次出现位置

minmax：比较两个参数，按前小后大的顺序返回数偶

标准算法

minmax_element：同时实施min_element和max_element操作

mismatch：比较两个给定区间元素，返回失配元素的首次出现位置

move：移动给定区间内元素

move_backward：逆向移动给定区间内元素

next_permutation：按词典序返回元素序列的下一排列

none_of：给定区间内元素如果不满足条件返回true

nth_element：划分给定区间，返回第n个元素位置，该元素不小于其前全部元素，不大于其后全部元素

partial_sort：部分排序

partial_sort_copy：部分排序拷贝

partition：按特定条件将元素划分为不相交的两个集合

标准算法

partition_copy：划分拷贝

partition_point：在给定区间内查找不满足特定条件的首个元素（划分点），其前元素满足条件，该元素及其后元素不满足该条件

pop_heap：从堆中删除最大元素

prev_permutation：按词典序返回元素序列的前一排列

push_heap：向堆中添加元素

random_shuffle：随机生成元素序列的新排列

remove：删除给定区间内某个元素

remove_copy：仅拷贝给定区间内非特定值的元素

remove_copy_if：仅拷贝给定区间内不满足特定条件的元素

remove_if：删除给定区间内满足特定条件的元素

标准算法

replace：将具有特定值的元素替换为新值

replace_copy：拷贝时替换

replace_copy_if：拷贝时替换满足特定条件的元素

replace_if：满足特定条件时替换

reverse：逆序

reverse_copy：拷贝时逆序

rotate：旋转，互换两个相邻区间的元素

rotate_copy：拷贝时旋转

search：查找

search_n：查找具有特定值的特定个数元素构成的子序列

标准算法

set_difference：集合差运算

set_intersection：集合交运算

set_symmetric_difference：集合对称差运算

set_union：集合并运算

sort：排序

sort_heap：将堆转换为已序区间

stable_partition：稳定划分

stable_sort：稳定排序

swap：元素交换

swap_ranges：区间内全部元素交换

标准算法

transform：变换

unique：删除重复元素

unique_copy：拷贝时删除重复元素

upper_bound：上界，返回给定已序区间大于特定值的首个元素位置

标准函子

算术函子

- plus<T>、minus<T>、multiplies<T>、divides<T>、modulus<T>、negate<T>

关系函子

- equal_to<T>、not_equal_to<T>、greater<T>、greater_equal<T>、less<T>、less_equal<T>

逻辑函子

- logical_and<T>、logical_or<T>、logical_not<T>

函数模板

函数模板的目的

- 设计通用的函数，以适应广泛的数据型式

函数模板的定义格式

- template<模板型式参数列表> 返回值型式 函数名称(参数列表);
- 原型： template<class T> void Swap(T& a, T& b) ;
- 实现： template<class T> void Swap(T& a, T& b) { … }

函数模板的体化与特化

- 针对特定型参数，在声明或第一次调用该函数模板时体化
- 每次体化都形成针对特定型参数的重载函数版本
- 文件最终只保留特定型参数的一份体化后的函体
- 显式体化主要用于库设计；显式特化覆盖体化的同型函体

函数模板

```
// 函数模板
template< class T >  void f( T t ) { /*……*/ }

// 显式体化：使用显式的长整型模板参数
template  void f<long> ( long n );

// 显式体化：使用d的型式推导模板参数型式
template  void f( double d );

// 显式特化：使用显式的整型参数
template<>  void f<int> ( int n );

// 显式特化：使用c的型式推导模板参数型式
template<>  void f( char c );
```

交换函数

```
template< class T >  void Swap( T & a, T & b )
{
  T t;  t = a,  a = b,  b = t;
}
int main()
{
  int m = 11, n = 7;  char a = 'A', b = 'B';   double c = 1.0, d = 2.0;
  //  正确调用，体化Swap( int &, int & )
  Swap( m, n );
  //  正确调用，体化Swap( char &, char & )
  Swap<char>( m, n );
  //  正确调用，体化Swap( double &, double & )
  Swap<double>( c, d );
  return 0;
}
```

函　子

编写函数，求某个数据集的最小元，元素型式为T

- 实现策略：使用函数指针作为回调函数参数
- 实现策略：使用函子（function object，functor）作为回调函数参数

函数指针实现

```
template< typename T >
const T & Min( const T * a, int n, bool (*comparer)(const T&, const T&) )
{
  int index = 0;
  for( int i = 1; i < n; i++)
  {
    if( comparer( a[i], a[index] ) )
      index = i;
  }
  return a[index];
}
```

函　子

函子的目的

- 功能上：类似函数指针
- 实现上：重载函数调用操作符，必要时重载小于比较操作符

函子的优点

- 函数指针不能内联，而函子可以，效率更高
- 函子可以拥有任意数量的额外数据，可以保存结果和状态，提高代码灵活性
- 编译时可对函子进行型式检查

函子实现

```
// 使用方法
int a[8] = { 9, 2, 3, 4, 5, 6, 7, 8 };
int min = Min( a, 8, Comparer<int>() );    // 构造匿名函子作为函数参数
```

函　子

```
template< typename T >  class Comparer
{
public:
  //  确保型式T已存在或重载operator<
  bool operator()( const T & a, const T & b )  {  return a < b;  }
};

template< typename T, typename Comparer >
const T & Min( const T * a, int n, Comparer comparer )
{
  int index = 0;
  for( int i = 1; i < n; i++)
  {
    if( comparer( a[i], a[index] ) )    index = i;
  }
  return a[index];
}
```

完美转发

完美转发的意义

- 库的设计者需要设计一个通用函数，将接收到的参数转发给其他函数
- 转发过程中，所有参数保持原先语义不变

完美转发的实现策略

- 当需要同时提供移动语义与拷贝语义时，要求重载大量建构函数，编程工作量巨大，易出错
- 右值引用与函数模板相互配合，可以实现完美转发，极大降低代码编写量

完美转发

```
class A
{
public:
  A( const string & s, const string & t ) : _s(s), _t(t)  {  }
  A( const string & s, string && t ) : _s(s), _t(move(t))  {  }
  A( string && s, const string & t ) : _s(move(s)), _t(t)  {  }
  A( string && s, string && t ) : _s(move(s)), _t(move(t))  {  }
private:
  string _s, _t;
};

int main()
{
  string s1("Hello");                    const string s2("World");
  A a1( s1, s2 );                        A a2( s1, string("Bingo") );
  A a3( string("Good"), s2 );            A a4( string("Good"), string("Bingo") );
  return 0;
}
```

完美转发

```
class A
{
public:
  //  根据实际参数型式生成不同的左值或右值引用的建构函数版本
  //  T1或T2可以为不同型，此处相同仅为示例
  //  实参推演时，使用引用折叠机制
  //  当形式参数为T&&型时，当且仅当实际参数为右值或右值引用时，
  //  实际参数型式才为右值引用
  //  引用折叠机制与const/volatile无关，保持其参数性质不变
  //  std::forward<T>(t)转发参数的右值引用T&&
  template<typename T1, typename T2>  A( T1 && s, T2 && t )
    : _s(std::forward<T1>(s)), _t(std::forward<T2>(t))  {  }
private:
  std::string _s, _t;
};
```

类模板

类模板的目的

- 设计通用的类型式，以适应广泛的成员数据型式

类模板的定义格式

- template<模板形式参数列表> class 类名称{ … };
- 原型：template<typename T> class A;

类模板的成员

- 像普通类的成员一样定义
- 定义在类中或类外均可，后者需要在类名后列些模板参数，以区别非模板类的成员函数
- template<typename T> T A<T>::f(T & u) { }

类模板

类成员函数的模板

- 成员函数可以使用其他模板

```
template< typename T >  class A
{
public:
  template<typename U>  T f( const U & u );
};

template<typename T>  template<typename U>
T A<T>::f( const U & u )
{
}
```

类模板

类模板的体化

- 与函数模板不同，类模板体化时必须给定模板实际参数，如：
  A<T> a;
- 类模板体化时，编译器生成模板类或成员函数的代码；成员函数在调用时体化，虚函数在类构造时体化

类模板的显式体化

- template class A<int>;
- 解决模板库的创建问题，库的使用者可能没有体化的机会，而未体化的模板定义不会出现在目标文件中
- 显式体化类模板后，显式体化其构造函数
- 其他成员函数可显式体化，也可不显式体化

类模板

类模板的显式特化

- 使用特定的型或值显式特化类模板，以定制类模板代码，如：
  `template<> class A<char> { ... };`
- 显式特化版本覆盖体化版本
- 显式特化并不要求与原始模板相同，特化版本可以具有不同的数据成员或成员函数
- 类模板可以部分特化，结果仍是类模板，以支持类模板的部分定制

类模板的缺省模板参数

- 与函数模板相同，类模板可以具有缺省模板参数

队　列

```
#include <iostream>
#include <cstdlib>
// 空队列异常类
class EQueueEmpty { };
// 队列项类前置声明
template< typename T > class JuQueueItem;
// 队列类
template< typename T > class JuQueue
{
public:
  JuQueue(): _head(NULLL), _tail(NULL) { }
  virtual ~JuQueue();
  virtual void Enter( const T & item );
  virtual T Leave();
  bool IsEmpty() const { return _head == 0; }
private:
  JuQueueItem<T> *_head, *_tail;
};
```

队　列

```
// 队列项类，单向链表结构
template< typename T >  class JuQueueItem
{
  friend class JuQueue<T>;
public:
  JuQueueItem( const T & item ) : _item(item), _next(0)  {  }
private:
  T _item;
  JuQueueItem<T> * _next;
};

// 队列类析构函数
template< typename T >  JuQueue<T>::~JuQueue()
{
  while( !IsEmpty() )
    Leave();
}
```

队　列

```
// 入队
template< typename T >  void JuQueue<T>::Enter( const T & item )
{
  JuQueueItem<T> * p = new JuQueueItem<T>( item );
  if( IsEmpty() )    _head = _tail = p;
  else    _tail->_next = p,  _tail = p;
}

// 出列
template< typename T >  T JuQueue<T>::Leave()
{
  if( IsEmpty() )    throw EQueueEmpty();
  JuQueueItem<T> * p = _head;
  T _retval = p->_item;
  _head = _head->_next;
  delete p;
  return _retval;
}
```

队　列

```
int main()
{
  JuQueue<int> * p = new JuQueue<int>;
  for( int i =0; i < 10; i++ )
    p->Enter( i );
  std::cout << p->Leave() << std::endl;

  int * r = new int(10), * q = new int(20);
  JuQueue<int*> * t = new JuQueue<int*>;
  t->Enter( r );
  t->Enter( q );
  int * s = t->Leave();
  std::cout << *s << std::endl;

  return 0;
}
```

元编程

什么是元编程（metaprogramming）？

- 利用模板可以进行编译期计算（数值计算、型式计算和代码计算）的特点进行程序设计

为什么可以进行元编程？

- C++是两层语言：执行编译期计算的代码称为静态代码，执行运行期计算的代码称为动态代码
- 模板可用于函数式编程（functional programming）：强调抽象计算，重视模块化，使用递归控制流程
- 模板是图灵完备的：理论上，模板可以执行任何计算任务

元编程

为什么需要元编程？

- 编译期计算可以使代码更通用，更易用，提升程序执行性能

元编程的缺点

- 相对结构化编程，编译效率极低
- 代码丑陋不堪，阅读难、调试难、维护难，易导致代码膨胀

元编程可以做什么？

- 数值序列计算、素性判定、控制结构、循环展开、型式判定、表达式、编译期多态、特性、策略、标签、元容器、……
- 注：对操作系统编程而言，元编程意义不大

Fibonacci数列

```
#include <iostream>
// 类模板，计算Fibonacci数列的第i项
template< int i = 1 > class Fibonacci
{
public: enum { value = Fibonacci<i-1>::value + Fibonacci<i-2>::value };
};
// 类模板特化，递归终止条件
template<> class Fibonacci<2> { public: enum { value = 1 }; };
template<> class Fibonacci<1> { public: enum { value = 1 }; };

int main()
{
  std::cout << "Fib(" << 1 << ") = " << Fibonacci<1>::value << std::endl;
  std::cout << "Fib(" << 2 << ") = " << Fibonacci<2>::value << std::endl;
  std::cout << "Fib(" << 3 << ") = " << Fibonacci<3>::value << std::endl;
  std::cout << "Fib(" << 4 << ") = " << Fibonacci<4>::value << std::endl;
  return 0;
}
```

素数枚举

```
#include <iostream>
#include <iomanip>
// 递归计算p是否为素数；若是，素性判定结论answer为1，否则为0
template< int p, int i > struct PrimeMagicCube {
  enum { answer = p % i && PrimeMagicCube<p,i-1>::answer };
};
// 素数魔方类模板部分特化，递归终止条件，除数为1，没有找到因子
template< int p > struct PrimeMagicCube<p,1> { enum { answer = 1 }; };
// 数值类模板，输出不大于i的全部素数
template< int i > struct Number {
  Number<i-1> a;    // 递归定义数值对象
  enum { answer = PrimeMagicCube<i,i-1>::answer };
  void IsPrime()
  {     // 先降序输出全部素数，后升序输出全部数值素性序列
    if( answer )    std::cout << std::setw(4) << std::right << i;
    a.IsPrime();    // 递归调用，计算下一数值的素性
    std::cout << std::setw(2) << answer;
  }
};
```

素数枚举

```
// 数值类模板特化，终止于2
template<>  struct Number<2>
{
  enum  { answer = 1  };
  void IsPrime()
  {
    std::cout << std::setw(4) << std::right << 2 << std::endl;
    std::cout << std::setw(2) << answer;
  }
};

int main()
{
  Number<100> a;
  a.IsPrime();
  std::cout << std::endl;
}
```

事件机制

事件基本概念

- 操作系统或应用程序内部发生某件事，程序的某个组件需要响应该事件，并进行特定处理

面向对象架构中，事件响应函数最可能为成员函数

- 问题：指向类成员函数的指针不能转换为哑型指针void *，也不能随意转换为指向另一个类的成员函数的指针
- 解决方案：使用指向指向类成员函数的指针的指针

实现策略：事件委托模型

- Event类模板：管理事件响应者对象，实现事件多播
- EventResponsor类模板：响应者对象与响应者行为配对
- Empty类：委托模型和指针转换

事件机制

```
#include <iostream>
#include <vector>
using namespace std;

// 空类，用于指代响应者对象
class Empty { };

// 事件响应者类模板，保存特定事件的响应者与响应行为
template< typename EventAction > class EventResponsor
{
public:
  EventResponsor() : actor(NULL), action(NULL) { }
  EventResponsor( Empty * actor, EventAction * action ) : actor(actor), action(action) { }
  friend bool operator==( const EventResponsor & lhs, const EventResponsor & rhs )
  { return lhs.actor == rhs.actor && *lhs.action == *rhs.action; }
public:   // 公开的数据成员，以方便使用者
  Empty * actor;
  EventAction * action;
};   // template<typename EventAction> class EventResponsor
```

事件机制

```
// 事件类模板，用于管理特定事件的所有响应者
template< typename EventAction >  class Event
{
public:
  typedef vector<EventResponsor<EventAction> > EventResponsors;
  typedef typename vector<EventResponsor<EventAction> >::iterator EventIterator;

public:
  virtual ~Event()
  {
    for( EventIterator it = this->_ers.begin();  it != this->_ers.end(); ++it )
    {
      delete it->action,  it->action = NULL;
    }
  }

  EventResponsors & GetResponsors()  {  return this->_ers;  }
```

事件机制

```
// 事件绑定，将实际响应者和响应行为挂接到事件响应者对象上
template< typename Responsor, typename Action >
void Bind( Responsor * actor, Action action )
{
  Action * act = new Action( action );
  EventResponsor<EventAction> er( (Empty*)actor, (EventAction*)act );
  bool unbound = true;
  for( EventIterator it = this->_ers.begin(); it != this->_ers.end(); ++it )
  {
    if( *it == er )    // 发现重复的事件响应者，说明已绑定
    {
      unbound = false;  break;
    }
  }
  if( unbound )
    this->_ers.push_back( er );
  else
    delete er.action,  er.action = NULL;
}
```

事件机制

```
// 解除事件绑定，删除事件响应者对象
template< typename Responsor, typename Action >
void Unbind( Responsor * actor, Action action )
{
  Action * act = new Action( action );
  EventResponsor<EventAction> er( (Empty*)actor, (EventAction*)act );
  for( EventIterator it = this->_ers.begin(); it != this->_ers.end(); ++it )
  {
    if( *it == er )    // 找到待删除的事件响应者对象
    {
      delete it->action, this->_ers.erase( it );  break;
    }
  }
  delete er.action,  er.action = NULL;
}

private:
  EventResponsors _ers;
};    // template<typename EventAction> class Event
```

事件机制

```
// 定义事件委托模型、指向类成员函数的指针
typedef Empty EventDelegator;
typedef void ( EventDelegator::*ValueChanged )( int value, void * tag );

// 触发者
class Trigger
{
public:
  Trigger() : _value(0) { }
  void SetValue( int value, void * tag );
  int GetValue() { return _value; }
public:
  // 值变化事件，公开属性，方便在类外设定
  Event<ValueChanged> value_changed;
private:
  int _value;
};
```

事件机制

```
// 设定值，遍历特定事件的响应对象列表，逐一触发值变更事件
void Trigger::SetValue( int value, void * tag )
{
  if( _value == value )
    return;
  _value = value;
  Event<ValueChanged>::EventResponsors  ers;
  ers = this->value_changed.GetResponsors();
  if( !ers.empty() )
  {
    Event<ValueChanged>::EventIterator  it;
    for( it = ers.begin(); it != ers.end(); ++it )
    {
      ( ( it->actor )->*( *( it->action ) ) )( value, tag );    // 响应事件
    }
  }
}
```

事件机制

```
// 行动者
class Actor
{
public:
 // 侦听事件，绑定本对象的事件响应函数到侦听的事件
 void Listen( Trigger * trigger )
 { trigger->value_changed.Bind( this, &Actor::OnValueChanged ); }

 // 停止侦听，从侦听的事件中取消绑定本对象的事件响应活动
 void Unlisten( Trigger * trigger )
 { trigger->value_changed.Unbind( this, &Actor::OnValueChanged ); }

 // 值变更事件的响应函数
 void OnValueChanged( int value, void * tag )
 { cout << reinterpret_cast<char*>(tag) << value << "." << endl; }
};
```

事件机制

```
int main()
{
  const char * s = "Now the value is ";
  Trigger t;
  Actor a1, a2;

  a1.Listen( &t );
  a2.Listen( &t );

  cout << "Listening..."  << endl;
  t.SetValue( 10, reinterpret_cast<void*>( const_cast<char*>(s) ) );

  a2.Unlisten( &t );
  cout << "Listening again..."  << endl;
  t.SetValue( 20, reinterpret_cast<void*>( const_cast<char*>(s) ) );

  return 0;
}
```

编程实践

11.1 使用类模板实现自己的抽象链表类。

11.2 按照Black-Scholes期权定价模型，衍生品的价值与标的证券的价格有关。当标的证券的价格发生变化时，其对应的所有衍生品的价值，理论上都应随之发生变化。股票也是一种证券。因此当股票价格波动时，以该股票为标的的所有股票期权的价值自然也应随之发生变化。通过发现股票期权的交易价格与通过期权定价模型测算的“实际价值”的差异，可以实施套利或对冲交易。编写程序，应用事件机制实现上述策略。