目　　录

第一章　类、接口 …… 7
1.1　Handle-Body与接口、抽象接口 ……7
1.2　多继承、与菱形缺陷、this跳转等…… …… 13
1.3　C++多态的两种多态形式和区别…… 18
第二章　重载 ……18
2.1　函数重载 ……19
2.2　运算符重载 …… 20
第三章　模板 ……29
3.1　模块函数 ……29
3.2　模块类……31
3.3　STL标准模板库……34
附录：参考资料 …… 39

前言

我们在C++基础课程中已经了解了C++的一些基本概念，知道了什么是类什么是对象。也了解了继承、封装、多态等C++面向对象的基本特征，本课程主要是更进一步探讨一下C++一些基本模型的应用，加深对概念的理解，由于课程时间有限，C++，模型和内容又如此之多，对任何一个模型都无法深入进去，所以只能泛泛而谈。

第一章　　类、接口

学习要求：

1、了解类的继承、封装等概念之间的关系

2、了解什么是接口，什么是虚函数，它有什么样的特点。学会使用接口编程的思想

本章节主要介绍C++中的类、接口。类，包涵了一组数据和一组基于数据上的一组方法。它描述了一个对象的属性、状态和行为；接口，它只是描述了一个对象的简单的行为。

有关类的基本概念：

Class names
Class members
Member Functions
Static Member Functions
Unions
C++ Bit Fields
Nested Class Declarations
Type Names in Class Scope
Multiple Base Classes
Virtual Functions
Abstract Classes
Controlling Access to Class Members
　　p**rivate** Members
　　protected Members

public Members

Access Specifiers for Base Classes，priavte，public、protected

Friends

Constructors

Destructors

Conversion Functions

the new operator and the delete operator

Copying Constructor Functions

Interface

1.1 Handle-Body与接口、抽象接口

在C++中封装的概念是把一个对象的外观接口同实际工作方式（实现）分离开来，但是C++的封装是不完全的，编译器必须知道一个对象的所有部分的声明，以便创建和管理它。我们可以想象一种只需声明一个对象的公共接口部分的编程语言，而将私有的实现部分隐藏起来。C + +在编译期间要尽可能多地做静态类型检查。这意味着尽早捕获错误，也意味着程序具有更高的效率。然而这对私有的实现部分来说带来两个影响：一是即使程序员不能轻易地访问实现部分，但他可以看到它；二是造成一些不必要的重复编译。

然而C++并没有将这个原则应用到二进制层次上，这是因为C++的类既是描述了一个接口同时也描述了实现的过程,示例如下：

```
class CMyString
{
private:
  const int m_cch;
  char *m_psz;
public:
  CMyString(const char *psz);
  ~CMyString();
  int Length() const;
  int Index(const char *psz) const;
}
```

CMyStirng对外过多的暴露了内存布局实现的细节，这些信息过度的依赖于这些成员变量的大小和顺序，从而导致了客户过度依赖于可执行代码之间的二进制耦合关系，这样的接口不利于跨语言跨平台的软件开发和移植。

1.1.1 Handle-Body模式

解决这个问题的技术有时叫句柄类（ handle classes）或叫“Cheshire Cat”[1]。有关实现的任何东西都消失了，只剩一个单一的指针“**m_pThis**”。该指针指向一个结构，该结构的定义与其所有的成员函数的定义一样出现在实现文件中。这样，只要接口部分不改变，头文件就不需变动。而实现部分可以按需要任意更动，完成后只要对实现文件进行重新编译，然后再连接到项目中。

这里有这项技术的简单例子。头文件中只包含公共的接口和一个简单的没有完全指定的类指针。

```
class CMyStringHandle
{
private:
  class CMyString;
  CMyString *m_pThis;
public:
  CMyStringHandle (const char *psz);
  ~ CMyStringHandle ();
  int Length() const;
  int Index(const char *psz) const;
};
CMyStringHandle:: CMyStringHandle(const char *psz)
:m_pThis(new CMyString(psz));
{
}
CMyStringHandle::~ CMyStringHandle()
{
   delete m_pThis;
```

```
}
int CMyStringHandle::Length()
{
 return m_pThis->Length();
}
int CMyStringHandle::Index(const char *psz)
{
 return m_pThis->Index(psz);
}
```

这是所有客户程序员都能看到的。这行

class CMyString;

是一个没有完全指定的类型说明或类声明（一个类的定义包含类的主体）。它告诉编译器，cheshire 是一个结构的名字，但没有提供有关该结构的任何东西。这对产生一个指向结构的指针来说已经足够了。但我们在提供一个结构的主体部分之前不能创建一个对象。在这种技术里，包含具体实现的结构主体被隐藏在实现文件中。

在设计模式中，这就叫做Handle-Body 模式，**Handle-Body**只含有一个实体指针，服务的数据成员永远被封闭在服务系统中。

Handle-Body模式如下：

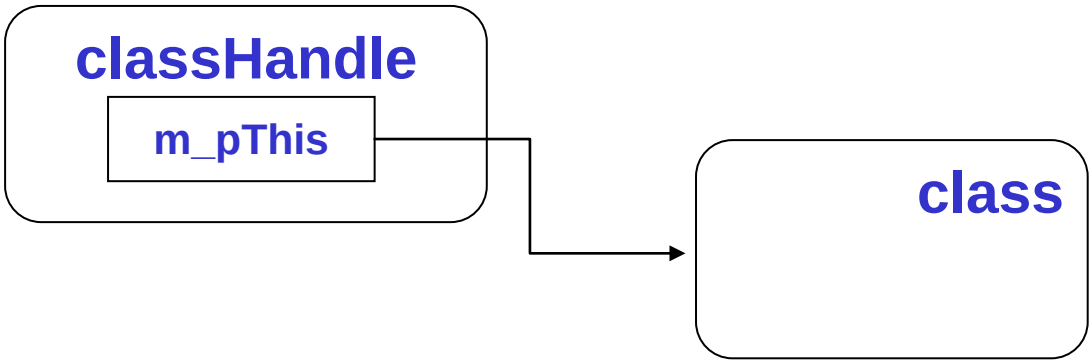


图1 Handle-Body模式（句柄类做为接口）

Handle-Body的布局结构永远不会随着实现类数据成员的加入或者删除或者修改而导致Handle-Body的修改，即Handle-Body协议不依赖于C++实现类的任何细节。这就有效的

对用户的编译器隐藏了这些斜街，用户在使用对这项技术时候，Handle-Body 接口成了它唯一的入口。

然而**Handle-Body**模式也有自己的弱点：

1、接口类必须把每一个方法调用显示的传递给实现类，这在一个只有一个构造和一个析构的类来说显然不构成负担，但是如果一个庞大的类库，它有上百上千个方法时候，光是编写这些方法传递就有可能非常冗长，这也增加了出错的可能性。

2、对于关注于性能的应用每一个方法都得有两层的函数调用，嵌套的开销也不理想

3、由于句柄的存在依然存在编译连接器兼容性问题。

接口和实现分离的Handle-Body。

1.1.2　抽象接口

使用了“接口与实现的分离”技术的 **Handle-Body** 解决了编译器/链接器的大部分问题，而C++面向对象编程中的抽象接口同样是运用了“接口与实现分离”的思想，而采用抽象接口对于解决这类问题是一个极其完美的解决方案。

1、抽象接口的语言描述：

```
class IMyString
{
 virtual int Length() const = 0;   //这表示是一个纯虚函数，具有纯虚函数的接口
 virtual int Index(const char *psz) const = 0;
};
```

2、抽象接口的内存结构：

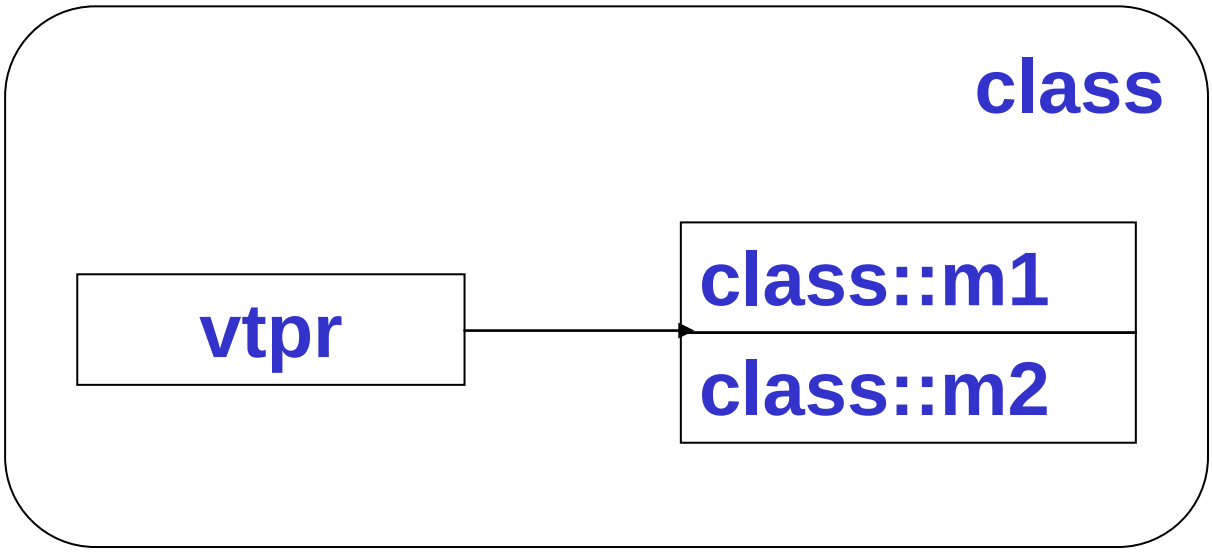


图2 抽象接口的内存布局

3、　抽象接口的实现代码：

接口：

```
class IMyString
{
    virtual int Length() const = 0;   //这表示是一个纯虚函数，具有纯虚
                                      //函数的接口
    virtual int Index(const char *psz) const = 0;
};
```

实现：

```
class CMyString：public IMyString
{
private:
  const int m_cch;
  char *m_psz;
public:
  CMyString(const char *psz);
  virtual ~CMyString();
  int Length() const;
  int Index(const char *psz) const;
```

}

从上面采用抽象接口的实例来看，抽象接口解决了Handle-Body所遗留下来的全部缺陷。

抽象接口的一个典型应用：

抽象工厂（AbstractFactroy）

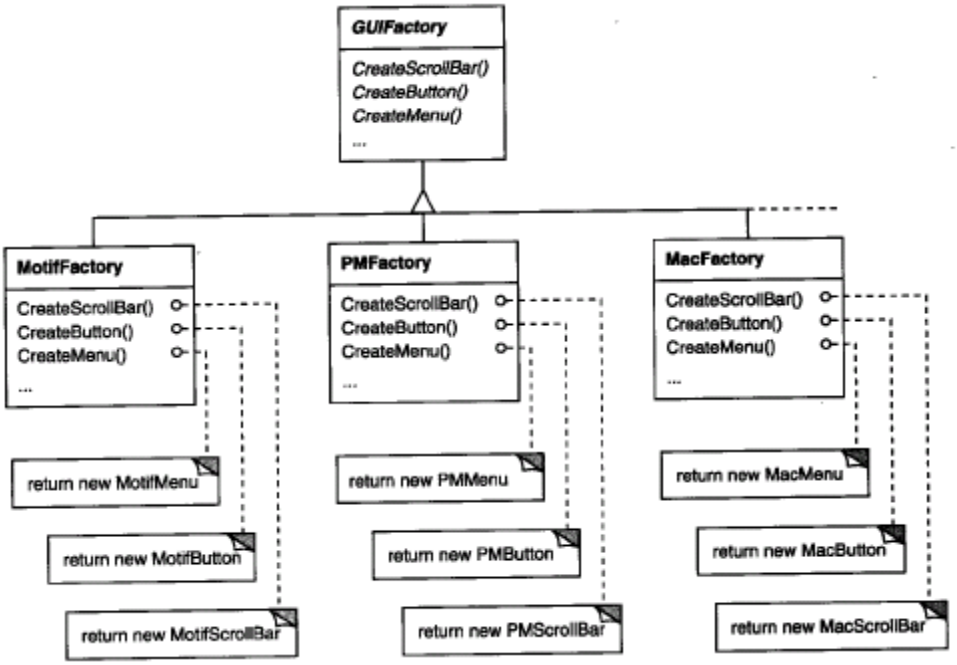


图3 抽象工厂模式

1.2　多继承与菱形缺陷、this跳转等

多重继承是C++语言独有的继承方式，其它几乎所有语言都秉承了单一继承的思想。这是因为多重继承致命的缺陷导致的：

1.2.1　菱形缺陷

当继承基类时，在派生类中就获得了基类所有的数据成员副本。假如类B 从A1和A2两个类多重继承而来，这样B类就包含A1、A2类的数据成员副本。

考虑如果A1、A2都从某基类派生，该基类称为Base，现在继承关系如下：

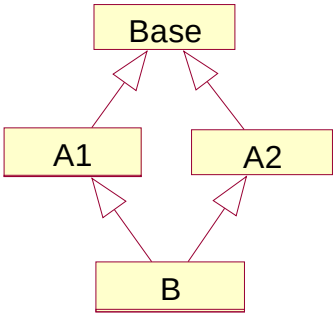


图4 菱形继承关系

我们C++语言来描述这种继承关系：

```
class Base{... ... };
class A1 :public Base {... ... };
class A2 :public Base {... ... };
class B :public A1,public A2 {... ... };
```

那么A1、A2都具有Base的副本。这样B就包含了Base的两个副本，副本发生了重叠，不但增加了存储空间，同时也引入了二义性。这就是菱形缺陷,菱形缺陷时间是两个缺陷：

1、子对象重叠

2、向上映射的二义性。

菱形缺陷的其中一种解决办法将

在C++世界里最广泛的使用虚拟继承解决菱形缺陷的应用便是标准C++的输入/输出iostream;

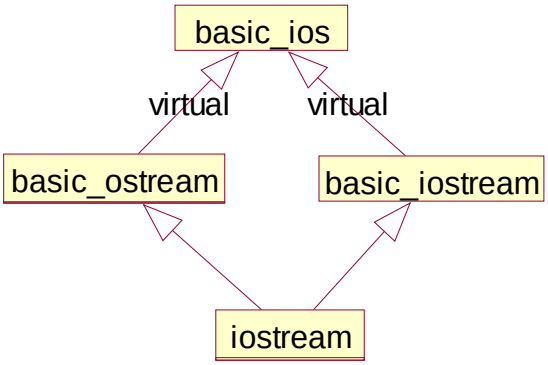


图5 标准C++的输入/输出

1.2.2　多重接口与方法名冲突问题（Siamese twins）

对继承而来的虚函数改写很容易，但是如果是在改写一个“在两个基类都有相同原型”的虚函数情况就不那么容易了。

提出问题：

假设汽车最大速度的接口为ICar，潜艇最大速度的接口为 IBoat，有一个两栖类的交通工具它可以奔跑在马路上，也可以航行在大海中，那么它就同时拥有ICar、IBoat两种交通工具的最大速度特性，我们定义它的接口为ICarBoat；

```
class    ICar
{
    virtual int GetMaxSpeed（）= 0；
};
class    IBoat
{
virtual int GetMaxSpeed（）= 0；
};
```

我们先对ICarBoat的接口做一个尝试：

```
class CCarBoat
{
    virtual int GetMaxSpeed（）；//既完成ICar的GetMaxSpeed（）接口方法又
                                 //完成IBoat的接口方法？显然不能够
};
```

解决问题：

显然上面这个尝试根本就无法成功，只用一个实现方法，怎么能够求出这个ICarBoat交通工具奔跑在马路上的最高时速，同时也能够求出航行在大海上的最大航行速度呢。

上面这一问题矛盾就在一一个方法，却需要两个答案。看来ICarBoat要返回两个答案就必须有两个方法了，我们假设一个方法是求在陆地上奔跑的速度，名称为GetCarMaxSpeed（）；另一个方法是求在大海上航行的最大速度，名称为GetBoatMaxSpeed（）；那这两个方法又怎么和GetMaxSpeed（）接口方法联系起来呢；

幸运的是，我们找到了解决办法，而且解决办法有很多种，下面介绍一下继承法。

```
class IXCar :public ICar
{
    virtual int GetMaxSpeed（）
    {
        GetCarMaxSpeed();
    }
    virtual int GetCarMaxSpeed（）  = 0;
};
class IXBoat:public IBoat
{
    virtual int GetMaxSpeed（）
    {
        GetBoatMaxSpeed();
    }
    virtual int GetBoatMaxSpeed（）  = 0;
};
classCCarBoat: public IXCar , public IXBoat
{
    virtual int GetCarMaxSpeed（）
    {
        …… 
    }
    virtual int GetBoatMaxSpeed（）
```

```
        {
            ……
        }
    };
```

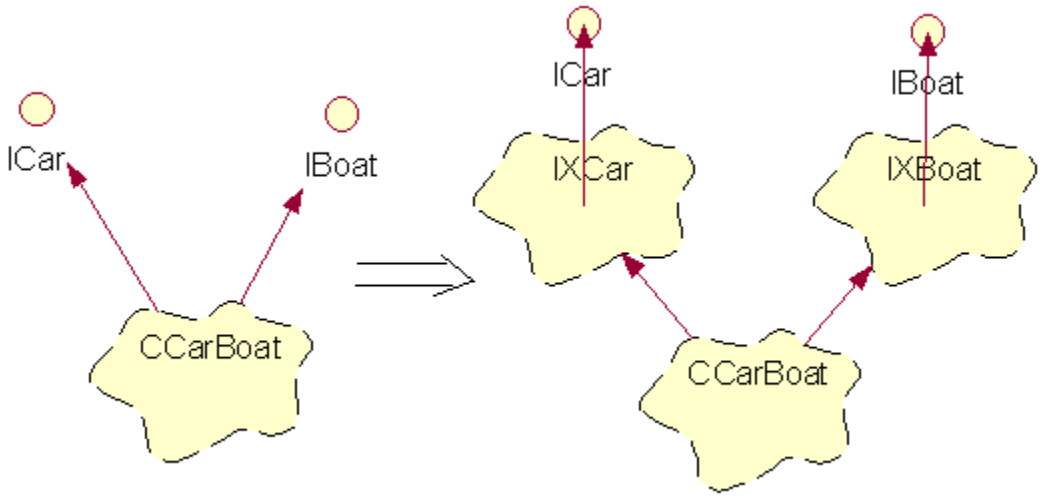


图6 多重接口与方法名冲突问题

1.2.3　this跳转

this跳转是指的“对象同一性”问题。

在单一继承的世界内，无论继承关系怎么复杂，针对于同一对象，无论它的子类或者父类的this指针永远相等。即如果有下面的模型：

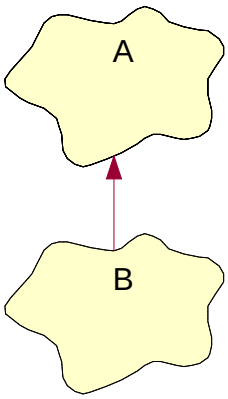


图7 B从A继承的关系图

那么 对于一个已经实例化B类的对象 bObject，永远有（B*）&bObject ==(A*)&bObject 成立。

但是在多继承的世界内，上面的等式就不能恒成立，对象的同一性受到了挑战。

特别的是，在多继承世界内如果**图四**的菱形关系存在情况下，如果对于已经实例化B类的对象bObject; （Base*）（A1*）&bObject != （Base*）（A2*）&bObject 成立，当这种事情发生的时候我们就只能特殊处理了。这种情况在COM应用中处处都会发生。

1.3　C++多态的两种多态形式和区别

C++有两种多态多态形式：

1、编译时刻多态，编译时刻多态依靠函数重载或者模板实现

2、运行时刻多态。运行时刻多态依靠需函数虚接口实现

第二章　　重载

学习要求：

1、了解什么是函数重载，什么是运算符重载

2、学会运用智能指针，仿函数

在 C++的世界里，有两种重载：函数重载和运算符重载，函数重载就采用采用参数匹配的原则，进行重载的，它是一种编译时刻的多态。而运算符重载，使采用改写或者说重新定义 C++的内嵌运算符的方法。

有关重载的基本概念：

Overloaded Functions

Overloaded Operators

Declaration Matching

Argument Matching

Argument Types Matching

Argument Counts Matching

C++ Unary Operators

Binary Operators

Smart Pointer

Function objects

1.1 函数重载

函数重载方法是在当前范围内选择一个**最佳匹配**的函数声明供调用该方法者使用。如果一个**适合**的函数被找到后，这个函数将会被调用，在这里“适合的”是指按下列顺序匹配的符合下面条件的：

1、一个精确匹配的函数被找到
2、一个参数只有细微的差别，几乎可以忽略不计的 。
3、象类似通过子类向父类转化达到参数匹配的
4、通过正常转化函数进行类型转换，能够达到参数匹配到的。
5、通过用户自定义的转化函数（如转化运算符或者构造函数）达到参数匹配的
6、参数是采用省略符号

函数重载的方法基本上有：

1、根据函数参数数据类型的不同进行的重载；
2、根据参数个数的不同进行的重载；
3、缺省参数上的重载

我们在这里把缺省参数也称为一种函数重载，实际上它并不是严格意义上的重载。在使用缺省参数时必须记住两条规则。第一，只有参数列表的后部参数才可是缺省的，也就是说，我们不可以在一个缺省参数后面又跟一个非缺省的参数。第二，一旦我们开始使用缺省参数，那么这个参数后面的所有参数都必须是缺省的。第三，缺省参数只能放在函数声明中。第四，缺省参数可以让声明的参数没有标识符。

4、返回值重载

特别注意，在C++中并没有根据返回返回值的不同进行重载的，即我们不能定义

这样的函数：

```
void f();
int f();
```

在C++中这样的函数声明方法是被禁止的，但是我们有时间可能又需要这样的重载方法，我们又怎么实现呢，其实很简单，jiang函数的参数进行扩展，将这个函数返回值的数据类型，做为扩展参数的数据类型来。如下：

```
void f（void）；
void f（int）；
```

此时这个例子中的参数列表的数据，只在编译时刻起到分练函数的作用，在运行时刻并不起到传值作用，模板中经常都应用到了这种方法。

1.2 运算符重载

你可以重新定义C++绝大多数内嵌运算符的实现方法和功能，这些重定义的或者说重载的运算符，有可能全局作用的，也有可能作用在类基础之上的，运算符重载的实现可能以类的成员函数的形式出现，也有可能以全局性的函数的身份出现。

在 C++中重载运算符的名字为 operator*x*, 在这里 x 是一个可重载的运算符，如：重载 加法运算符，你需要定义一个名为 operator+ 的函数，然后实现他，其它的类似定义就可以了，例如：

```
Class complex //very simplified complex
{
    doublere,im;
public:
    complex(doubler,doublei):re(r),im(i){};
    complex operator+(complex);
    complex operator*(complex);
};
```

定义了 complex 这个复数的一个简单的实现概念模型。一个复数是由一对 double 类型的数据组成，并定义了这个复数的两个方法，加法运算 complex::operartor+（）和乘法运算 complex::operator*（）.现在我们就能够实现下面的复数表达式了：

```
void f()
{
    complex a = complex(1 , 3.1);
    complex b = complex(1.2 , 2);
    complex c = b;
    a = b + c;
    b = b + c * a;
    c= a * b + complex(1 , 2);
}
```

1.3.1　C++可重载的和C++不可重载的运算符

可重载运算符表：

Operator	Name	Type	Operator	Name	Type
,	Comma	Binary	->*	Pointer-to-member selection	Binary
!	Logical NOT	Unary	/	Division	Binary
!=	Inequality	Binary	/=	Division/assignment	Binary
%	Modulus	Binary	<	Less than	Binary

%=	Modulus/assignment	Binary	<<	Left shift	Binary
&	Bitwise AND	Binary	<<=	Left shift/assignment	Binary
&	Address-of	Unary	<=	Less than or equal to	Binary
&&	Logical AND	Binary	=	Assignment	Binary
&=	Bitwise AND/assign	Binary	==	Equality	Binary
( )	Function call	—	>	Greater than	Binary
*	Multiplication	Binary	>=	Greater than or equal to	Binary
*	Pointer dereference	Unary	>>	Right shift	Binary
*=	Multiplication/assign	Binary	>>=	Right shift/assignment	Binary
+	Addition	Binary	[ ]	Array subscript	—
+	Unary Plus	Unary	^	Exclusive OR	Binary
++	Increment[1]	Unary	^=	Exclusive OR/assignment	Binary
+=	Addition/assignment	Binary	|	Bitwise inclusive OR	Binary
-	Subtraction	Binary	|=	Bitwise inclusive OR/assignment	Binary
-	Unary negation	Unary	||	Logical OR	Binary
- -	Decrement[1]	Unary	~	One’ s complement	Unary
-=	Subtraction/assign	Binary	delete	delete	—
->	Member selection	Binary	new		

不可重载运算符表：

Operator	Name
.	Member selection

.*	Pointer-to-member selection
::	Scope resolution
? :	Conditional
#	Preprocessor symbol
##	Preprocessor symbol

在上面可重载的运算符可以看出运算符重载共分为两类:一元运算符重载和二元运算符重载

一元运算符重载：

在声明一个类的非静态的一元运算符重载函数时，你必须声明的形式如下：

ret-type operator*op*()　　(1)

在这里 *ret-type 是指返回数据类型 op 是指一元运算符*

在声明一个全局的一元运算符重载函数时，你必须声明的形式日下：

ret-type operator*op*(*arg*)　(2)

在这里 *ret-type 与 op 和上面的意思一样，arg 是指这个运算符所作用的数据类型*

二元运算符重载：

在声明一个类的非静态的二元运算符重载函数时，你必须声明的形式如下：

ret-type operator*op*(*arg*) (3)

(3)式和二式基本相同 *arg 可以是任何一个*

在声明一个全局的二元运算符重载函数时，你必须声明的形式日下：

ret-type operator*op*(*arg1*, *arg2*) (4)

在这里 *ret-type 与 op 和上面的意思一样，arg1，arg2, 是指这个运算符所作用两个数据类型*

1.3.2　几类特殊的运算符重载

1、类型转换运算符

所有的数据类型均可以定义构造函数，包括系统定义的数据类型和用户自定义的数据类型，如：

```
class CString
{
 … …
 operator LPCSTR() const;
 … …
};
```

应用：

```
CString str = “12345”;
LPCSTR lpsz = str;//此处会进行LPCSTR运算
```

这只是一个简单的应用的示例，其实有时间类型转换具有无比强大的功能。我曾经就是用类型装换运算符重载解决一个跨平台通信的问题。

2、bool运算符重载

int、float、bool等运算符也是可以重载的，例如重载bool运算符，但是重载运算符bool 时候，需要注意有很多麻烦和臆想不到的东西

```
template<class T>
class testbool
{
 … …
 operator bool() const throw()
 {
     return m_ pT != 0;
 }
private:
 T *m_pT;
}
```

下面结果均通过编译

```
testbool<int> sp1;
testbool<std::string> sp2;
if (sp1 == sp2)
if (sp1 != sp2)
 bool b = sp1
 int I = sp1 * 10;
```

从上面可以看得出 bool 的表现已经远远超过 bool 本身了，所以建议大家不要轻易对 bool 进行重载操作。

3、地址运算符重载

在DCOM应用中，我们有一个重载运算符的例子：

```
STDAPI CoCreateInstance(
        REFCLSID rclsid,
        LPUNKNOWN pUnkOuter,
        DWORD dwClsContext,
        REFIID riid,
    LPVOID *ppv);
```

我们看最后一个参数 LPVOID 指针的指针，这里是一个输出参数，返回一个接口的指针。

一般情况下我们应用如下

```
IUnknown *pUn;
CoCreateInstance(…,…,…,…,(void **)& pUn);
```
（5）

然而我们也可以这样写：

```
IUnknown *pUn;
CComPtr comPtr(pUn);
CoCreateInstance(…,…,…,…,(void **)& comPtr);
```
（6）

之所以能够这么写这是因为CComPtr 重载了 “&” 运算符，如下：

```
template <class T>
class CComPtr
{
public:
    …
    CComPtr(T* lp)
    {
        if ((p = lp) != NULL)
        p->AddRef();
    }
```

```
        …

        T** operator&()
        {
            ATLASSERT(p==NULL);
            return &p;
        }
    private:
    T* p;
    };
```

&comPtr 实际上是得到了

一般的情况下，我们并不能对pUn的地址，所以 （5） 式和 （6）式其实传入的参数是一样当都是传入了 pUn 的地址。

虽然我们能够对运算符进行重载，但一般情况下我们并不是很提倡这种操作，这是因为：

A、暴露了封装对象的地址，如上面 CComPtr 对 pUn 的封装其实不起任何作用，任何时候我都可以直接访问和修改 pUn指针，这就意味着所有权的完全丧失，封装不起任何意义

B、对于 unary operator& 的重载使得重载对方永远无法与STL容器进行任何融合，甚至无法参与任何泛型编程。

一个对象的地址是一个对象最基本的概念，在一般情况下，我们并不提倡，也请大家慎用 地址运算符的重载。

4、指针运算符重载

指针运算符，有一个及其特殊且及其重要的机制：

当你对某个型别实施operator-〉而这个型别并非原生指针时候：编译器会从这个型别中找出用户自定义的 operator-〉，并实施后，编译器将继续对这个operator-〉返回的结果实施 operator-〉直到找到一个原生指针。

这种机制导致了一个特有的技术：（pre and post function calls ），“前调用”及后调用技术。应用如下：

```
class CallDoSomething
{
public:
    void DoCall()
    {
        TRACE("DoCall\n");
    }
```

```
};
template<class T>
class CallInMutiThread
{
    class LockProxy
    {
    public:
        LockProxy(T*pT)
        :m_pT(pT)
        {
            TRACE("Lock \n");
        }
        ~LockProxy()
        {
            TRACE("UnLock \n");
        }
        T *operator->()
        {
            return m_pT;
        }
    private:
        T *m_pT;
};
public:
    CallInMutiThread(T* pT)
        :m_pT(pT)
    {
    }
    LockProxy   operator->()
    {
        return LockProxy(m_pT);
    }
private:
    T *m_pT;
  };
```

上面 CallDoSomething 是函数调用，假设这个类原来是在单线程中运行的，但是现在已经移植到了多环境中，所以我们就增加了 CallInMutiThread 对 原始类进行配接使之适应与多线程环境，调用过程如下：

```
CallDoSomething DoSomthing;
CallInMutiThread<CallDoSomething> MutiThread(&DoSomthing);
MutiThread->DoCall();
```

调用结果如下：

```
Lock

DoCall
```

UnLock

从上面可以看出在调用 CallDoSomething 的成员函数 DoCall 之前调用了Lock方法，在调用结束后有调用了UnLock。这就是所谓的“前调用”和“后调用”，其实并不仅仅是多线程问题可以采用此办法，所有的“前调用”和“后调用”模式均可由此解。

重载“->”运算符，同时引出了智能指针的概念，参见下页。

5、括号运算符重载

语法特征：

primary-expression **(** *expression-list*$_{opt}$ **)**

括号运算符是一个同“->”运算符一样也是一个及其重要的运算符

在MSDN上说 括号运算符 是一个二元运算符，我觉得这个说法是完全错误的，在所有C++运算符重载中，括号运算符，应该是唯一没有规定参数元的个数的。它的参数可以从 0 个到 N个。

示例：

```
class Point
{
public:
        Point() { _x = _y = 0; }
        Point &operator()( int dx, int dy )
                { _x += dx; _y += dy; return *this; }
private:
        int _x, _y;
};
```

调用如下：

```
Point pt;
pt( 3, 2 );
```

从上面可以看出，括号运算符，调用形式如下：

object(*parameter list*);

看起来和函数的形式是完全一样的：

function(*parameter list*);

所以根据这一特点我们称之为仿函数。

第三章　　模板

学习要求：

1、了解什么是模板

2、学会运用模板函数，模版类和STL

模板（templates），以及以模版为基础的泛型编程和泛型模式，是当今C++中最活跃的一项编程技术，模版的第一个革命性的应用就是Standard Template Library（简称STL）。STL将templates 技术广泛应用于STL容器和STL算法上，在这一领域template技术发挥到了极致。

本章介绍C++ templates 的基本概念和语言特性

1.1 认识模板

1、模板的基本语法是：

template < [*typelist*] [**,** [*arglist*]] > *declaration*

这个**template**描述了一个参数化的类（模板类）或者是一个参数化的函数（模板函数），这个模板参数列表是用逗号分隔的类型列表（在这个表单忠使用 **class** 或者是 **typename** 来标识这个数据类型）。在某些情况下这个模板体内可能不存在任何的数据类型。*declaration* 域必须是一个函数或者类的声明。

1.4　模板函数

语法定义：

```
template< comma-separated-list-of-parameters >
function-name(parameter list)
{
}
```

例如：

```
template <typename T>
inline T const& max (T const& a, T const& b)
{
    // if a < b then use b else use a
```

```
        return a<b?b:a;
    }
```

调用形式 ：

1 ：通过调用的参数来识别模板的各参数类型

MAX (4,4.2)；　// OK, but type of first argument defines return type

2：明确指定参数的类型：

MAX<**int**,**float**>(4,4.2); //OK

在我们的例子中这个参数列表是 typename T, 其实在这里 typename 是可以用 class 替换的，typename 是在C++演化过程中逐渐形成的，而class是一个历史性的概念，typename 表达了一个比 class 更抽象意义上的概念。

有如下定义如：

```
class typenamedef
{
    typedef int INT_TYPE;
};
```

如果这样表达是正确的：

```
template<class T>
class testtypename:public typenamedef
{
public:
    typename T::INT_TYPE;
    INT_TYPE m_int;
};
```

但是如果把 此处的 **typename** 换成 class 就会报错

1.4.1 重载模板函数（Overloading Function Templates）

和普通的函数一样，模板函数也可以被重载，也就是说对象同的函数名，你能够具有不同的函数定义，在调用的时候再由C++编译器决定，那一个候选函数更有资格被匹配调用。

下面这个简单的例子说明了重载模板函数的方法和过程:

```
// maximum of two int values
inline int const& max (int const& a, int const& b)
{
    return a<b?b:a;
}

// maximum of two values of any type
template <typename T>
inline T const& max (T const& a, T const& b)
```

```
{
    return a<b?b:a;
}

// maximum of three values of any type
template <typename T>
inline T const& max (T const& a, T const& b, T const& c)
{
    return max (max(a,b), c);
}

int main()
{
    ::max(7, 42, 68);        // calls the template for three arguments
    ::max(7.0, 42.0);        // calls max<double> (by argument deduction)
    ::max('a', 'b');         // calls max<char> (by argument deduction)
    ::max(7, 42);            // calls the nontemplate for two ints
    ::max<>(7, 42);          // calls max<int> (by argument deduction)
    ::max<double>(7, 42); // calls max<double> (no argument deduction)
}
```

上面这个例子也说明了普通的函数与模板函数可以拥有同一个名字，而且可以被初始化为同一类型，如：

max(7, 42)

调用匹配非模板函数也匹配模板函数。

1.5　模板类

基本的**语法定义：**

```
template< comma-separated-list-of-parameters >
class class-name
{
    … …
};
```

具有缺省参数的模板定义形式

```
template <typename T, typename Alloc = alloc >
class class-name
{
    … …
};
```

在模板中用到了大量非习惯性思维方法，大家在学习模板之前需要了解这些模板设计的思维方法：

申明并不一定需要定义：

1、申明一个函数，并不实现

在C++中我们可能因为禁止某个缺省函数的调用操作而申明该缺省函数，但不定以它，例如：

```
class testDeclare
{
public:
    testDeclare();
    };
```

我们对上面的 testDeclare 的缺省构造函数进行了声明，但是我们并没有构造函数的的定义，当我们执行

```
testDeclare declare;
```

上面这个申请创建一个对象的操作会被编译系统所禁止

当然，我们也可以对缺省的重载运算符实施同样的手段

2、申明一个函数而不实现可能是为了模板函数的泛化

泛化：

```
template<typename T>
T testFun();
```

特化：

```
template<>
int testFun()
{
    return 10;
    }
```

3、申明一个函数可能仅仅为了获得特殊某一项功能

例如：

```
T MarkT();
char Test(T);
int    Test(...);
sizeof(MarkT());
```

上面的例子其实就是求 T 类的的字节数，其实在一般情况下，我们直接写 sizeof (T) 就可以了，然而有的时候系统并不允许我们这样做，所以我们就可以通过上面的例子 MarkT（）函数，其实上面的MarkT（），char Test(T) 函数 int　Test(...) 都是没有定义的，但是由于sizeof　是编译时刻的运算，所以它并不需要关心 这些函数是否实现。

- 申明一个类而不实现

例如我们在禁止模板类的泛化过程中就可以实现

```
template<typename T>
class    testClass; //泛化只申明

template<>
class    testClass<int> //特化进行实现
{
    };
```

如果我们有

```
testClass<char> test; //error
        //系统会调用泛化时发现没有没有实现二产生编译错误
testClass<int> test；//OK
        //系统调用特化故OK
```

1.5.1 模板设计基本方法

A、 编译器断言

```
template<class T,class U>
{
    typedef char small;
    class big {char dummy[2]};
    static small test(U);
    static big test(… );
    static T markT(); //函数定义只是为了得到一个返回类型
public;
    enum { value = sizeof(test(makT()))== sizeof(Small) };
};
```

B、模板特化

```
template<typename I,typename O>
struct testClass
{
    testClass(){ count << "I , O" << endl ;}
};
template<typename T>
struct testClass<T*,T*>
{
    testClass(){ count << "T*    , T*" << endl; }
};
```

C、常数映射型别

```
template <int v>
    struct testClass
```

```
{
    enum { value = v };
    };
```

D、 型别映射型别

```
template <typename T>
struct testClass
{
    typedef T OriginalType;
};
```

1.6 STL 标准模板库

容器

- 序列容器
  - vector,list、deque、stack(没有迭代器)、queue(没有迭代器)、即stack、queue不允许遍历行为
- 关联容器
  - set(标准)、map(标准)、hash_table、 RB-tree
- 通用算法
- begin()、end()、size()、empty()、erase(iterator __position)、clear()

迭代器

迭代器的基本算法

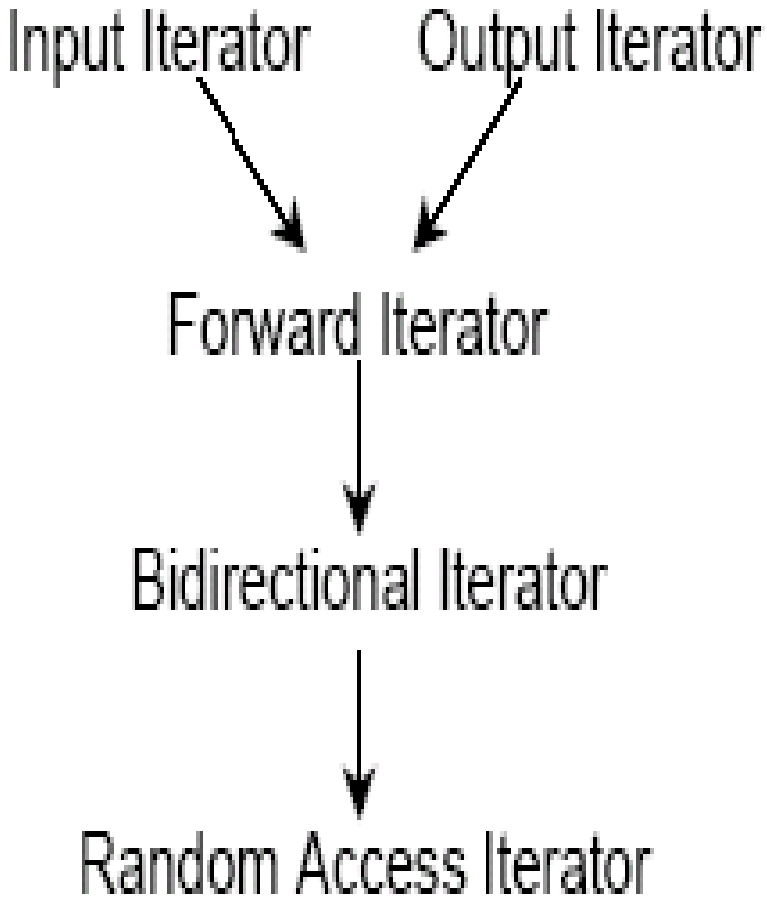
- 能够进行+、—、++、--、+=、—=、= = 、!=等运算
- 是一种智能性指针，实现operator * operator ->的重载
- 根据迭代器的特点，迭代器又称循环子

迭代器前闭后开区间 [first, last)

型别

- 单向迭代器
- ◊可逆迭代器
- ◊随机迭代器

- 迭代器的继承关系



型别萃取

- value_type
- difference_type
- refrence_type
- pointer_type
- iterator_category

- 型别萃取机
  - iterator_traits

算法

- **数值运算**
  - **power、itoa、accumulate**
- 基本运算
  - fill、fill_n、swap、max、min、iter_swap、copy
- 集合运算
  - set_union、set_interseion、set_diffrence
- 数据整理(以循环子为参数)
  - count、count_if、find、find_if、for_each、merge、sort、upper、search、search_n

仿函数

- 一元仿函数
- 二元仿函数
- 二元化一元仿函数
- 仿函数应用

一元仿函数

```
 template <class _Arg, class _Result>
struct unary_function {
  typedef _Arg argument_type;
  typedef _Result result_type;
};
template< class _Tp>
struct negate:public unary_function<T,T>{
_Tp operator()(const T&x) const {return -x;}
};
```

二元仿函数

```
 template <class _Tp>
struct less : public binary_function<_Tp,_Tp,bool>
{
```

- bool operator()(const _Tp& __x, const _Tp& __y) const { return __x < __y; }
- };

仿函数应用

```
template <class _InputIter, class Operation>
   inline _InputIter __find_if(_InputIter __first, _InputIter __last,
Operation __op,
                              const input_iterator_tag &)
{
  while (__first != __last && !__op(*__first))
    ++__first;
  return __first;
}
```

配接器

- 容器配接器
  - stack
  - Queue
- 迭代配接器
  - Reverse Iterators
  - IOStream Iterators
  - Inert_Iterator
- 仿函数配接器

容器配接器

- 容器配接器
  - Stack

```
   template <class _Tp, class _Sequence = deque<_Tp> >
  class stack {
   }
```

  - Queue

```
   queue< int , list < int > > myQueue;
```

仿函数配接器

- 绑定
  - bind1st、 bind2nd◊op(x,param) or op(param,x)
- 修饰
  - not1、not2◊!op(param) or !op(param1,param2)
- 组合
  - compose1、compose2◊op1(op2(param));
- 函数配接
  - ptr_fun1、ptr_fun2◊fp(param) or fp(param1,param2)
- 对象配接
  - mem_fun、mem_fun_ref、mem_fun1、mem_fun1_ref ◊(param->*f)() or (param.*f)() or (param->*f)(x) or (param.*f)(x)
- 举例
  - 不小于 x 表达式
    - not1(bind2nd(less<int>(),x))
  - f (g (x)) 的表达式
    - Compose1(f,g)
  - find_if(first , end, mem_fun(&clsss::fun));

STL库引用

- 标准引用办法
  - #include<vector>
  - #include<list>
  - using namespace std;
- 错误引用办法
  - #include<vector.h>
  - #include<list.h>

参考资料

- C++ Programming Language
- C++ Primer
- thinking in c++
- effective c++
- C++ Templates
- STL源码剖析
- Modern C++ Design
- 设计模式