

1. 导读

1.1 命名习惯

1. 指针命名: 指向一个T型对象

```
1 | widget* pw; //pw means "ptr to widget"
```

2. 引用命名: rw:reference to widget

2. 条款

2.1 让自己习惯C++

2.1.1 条款1：视C++为一个语言联邦

1. Object-Oriented C++

- classes
- 封装
- 继承
- 多态
- virtual函数 (动态绑定)

2. Template C++

3. STL: 容器、迭代器、算法、函数对象、适配器

4. 内置数据类型 (C-like) 通过值传递比通过引用传递更高效，而用户自定义数据类型则是因为构造函数和析构函数的存在，使得通过引用传递的方式更高效。

不然需要在去创建临时对象

2.1.2 尽量以const, enum, inline替换#define

1. 对于单纯变量，最好以const对象或者enum替换#define

2. 对于形似函数的宏，最好改用inline函数来替换#define

```
1 | template<typename T>
2 | inline void callWithMax(const T& a, const T& b)
3 | {
4 |     f(a > b ? a : b);
5 | }
```

也就是不要通过`#define ASPECT 1.653`这样的宏定义形式去定义一个常量。可以通过`const double AspectRatio = 1.653;`的形式定义常量

cpp宏定义：

常量指针的定义

由于常量定义式通常被放在头文件内，因此在定义常量指针的时候，有必要将指针也声明为`const`即

```
1 const char* const authorName = "Junney";
2 //string对象往往又比`char*`更好用，所以可以定义如下
3 const std::string const authorName = "Junney";
```

2.1.2.1 class专属常量

为了将常量的作用域限制在class内，需要将其设置为成员变量，并且为了确保此变量只有一份实体，需要声明为`static`

```
1 class GamePlayer{
2 private:
3     static const int NumTurns = 5; //常量声明式
4     int scores[NumTurns];
5 };
```

2.1.2.2 enum枚举类型

当上述的`GamePlayer::scores[Numturns];`中，编译器不支持常量在声明中定义的时候，可以考虑使用the enum hack的补偿做法

一个属于枚举类型(enumerated type)的数值可权充ints来使用。于是

```
1 class GamePlayer{
2 private:
3     enum {NumTurns = 5}; //令NumTurns称为5的一个记号名称
4     int scores[NumTurns];
5 };
```

1. enum hack的行为某方面比较像#define而不像const
2. 对一个const取地址是合法的，而取一个enum和#define的地址是不合法的
3. 作用：可以通过enum来限制获取某个整型常量的地址或者引用

2.1.3 尽可能使用 `const`

`const` 可以用来修饰

1. `classes` 外部 `global` 或 `namespace` 作用域中的常量
2. 修饰文件、函数或区块作用域中被声明为 `static` 的对象
3. `classes` 内部的 `static` 和 `non-static` 成员变量

1. 传入参数的过程中修饰参数

```
1 void f1(const widget* pw); //f1获得一个指针，指向一个不变的widget对象
```

2. STL 迭代器的两种 `const` 方式

```
1 std::vector<int> vec;
2 const std::vector<int>::iterator iter = vec.begin();
//iter等价于T* const指针常量，指针指向不可修改，指针指向的内容可以修改
3 *iter = 10; //可以
4 iter++; //错误，iter是T* const
5
6 std::vector<int>::const_iterator cIter = vec.begin();
//cIter等价于cont T*
```

3. `const` 在函数声明时的应用

- 令函数返回一个常数值，往往可以降低错误，而不至于放弃安全性和高效性

```
1 class Rational {};
2 const Rational operator* (const Rational& lhs,
const Rational& rhs);
3 //能够避免(a*b = c)这样的情形出现
```

2.1.3.1 `const` 成员函数

将成员函数声明为 `const` 的目的：

1. 使得该成员函数可以作用于 `const 对象` 个人理解：对于 `const Class& obj`, 提供的专用接口
2. 使得 `class` 接口容易被理解，明确哪个函数可以改变对象内容，而哪个函数不行

```

5   class TextBlock{
6     public:
7       TextBlock(string str)
8       { text = str; }
9       const char& operator[](size_t position) const //operator[] for const对象
10      { return text[position]; }
11      char& operator[](size_t position) //operator[] for non-const对象
12      { return text[position]; }
13
14     private:
15       string text;
16   };
17
18 int main()
19 {
20   TextBlock tb("hello");
21   cout << tb[0];
22   tb[0] = 'x'; //这是可改的
23   const TextBlock ctb("hello"); //声明一个const对象
24   ctb[0] = 'x'; //错误，不能修改const对象
25
26   //const对象只能调用重载后的const成员函数
27   //，否则会提示
28   //并且，无法通过const成员函数进行修改操作
29
30   int main()
31   {
32     const TextBlock ctb
33     //声明一个const对象
34     //表达式必须是可修改的左值 C/C++(137)
35     View Problem (Alt+F8) No quick fixes available
36     ctb[0] = 'x'; //错误，不能修改const对象
37
38   }
39
40   View Problem (Alt+F8) No quick fixes available
41   View Problem (Alt+F8) No quick fixes available
42
43   [2021/4/6 下午10:04:13] 对于 C++ 源文件, cppStandard 已根据编译器参数和查询 compilerPath 从 "c++11" 改为 "c++14": "C:/Program Files (x86)/Micro

```

存在的问题：以operator[]为例，如果需要实现多种功能，那么分别实现non-const和const版本的接口会使得代码非常冗余

解决办法：让non-const方法调用其const兄弟（涉及到转型）

原理：non-const成员函数本来就可以对对象做任何操作，所以调用其中一个成员函数不会带来任何风险

```

1 char& operator[](size_t position)
2 {
3     return const_cast<char&>(
4         static_cast<const TextBlock&>(*this)
5     [position]);
6 //1.static_cast 先将*this从原始类型TextBlock& 转换为
7 //const TextBlock&
8 //2.const_cast 然后将const operator[]的返回值中const移除

```

相关结论：

1. const成员函数不可以改变对象内任何non-static成员变量
2. 通过**mutable**实现non-static也能被const成员函数修改

2.1.4 确定对象被使用前已先被初始化

永远在使用对象之前先将它初始化

1. 对于无任何成员的内置类型，通过手动完成初始化过程
2. 对于内置类型外的其他自定义数据类型，通过**构造函数将对象的每一个成员进行初始化**
3. 如果成员变量是**const**或者**reference**就一定需要**初始化**，而不是赋值

2.1.4.1 赋值和初始化的区别

```
1 //执行步骤: 先调用default构造函数再调用copy assignment操作符
2 ABEntry::ABEntry (const std::string& name, const
3 std::string& address)
4 {
5     theName = name;
6     theAddress = address;
7 } //非赋值操作
8
9 //只执行一次拷贝构造
10 ABEntry::ABEntry (const std::string& name, const
11 std::string& address)
12         :theName(name), theAddress(address)
13 {} //这些也都是初始化操作, 而非赋值操作
```

C++规定，对象的成员变量的初始化动作发生在进入构造函数本体之前

2.1.4.2 如果classes有多个构造函数或许多成员变量或许多基类

这种情况可以选择定义一个private成员函数来进行赋值操作，并提供给构造函数使用

2.2 构造、析构、赋值运算

2.2.1 了解C++默认编写并调用哪些函数

c++默认提供的copy构造函数，default构造函数都是public且inline

当在类中声明了一个构造函数，那么编译器就不会再提供一个default构造函数

```

template<class T>
class NamedObject {
public:
    //以下构造函数如今不再接受一个 const 名称，因为 nameValue
    //如今是个 reference-to-non-const string。先前那个 char* 构造函数
    //已经过去了，因为必须有个 string 可供指涉。
    NamedObject(std::string& name, const T& value);
    ...
    //如前，假设并未声明 operator=
private:
    std::string& nameValue; //这如今是个 reference
    const T objectValue;   //这如今是个 const
};

现在考虑下面会发生什么事：

```

内部成员变量已经是一个引用，不能通过 operator= 的方式更改引用的指向。
因此，当 class 没有提供重载的 operator= 的时候，这时候无法进行任何的 = 操作

```

std::string newDog("Persephone");
std::string oldDog("Satch");
NamedObject<int> p(newDog, 2); //当初撰写至此，我们的狗 Persephone
//即将度过其第二个生日。
NamedObject<int> s(oldDog, 36); //我小时候养的狗 Satch 则是 36 岁,
//-- 如果她还活着。
p = s; //现在 p 的成员变量该发生什么事？

```

赋值之前，不论 p.nameValue 和 s.nameValue 都指向 string 对象（当然不是同一个）。赋值动作该如何影响 p.nameValue 呢？赋值之后 p.nameValue 应该

2.2.2 若不想使用编译器自动生成的函数，就该明确拒绝

通常情况下，如果不希望 class 支持某一特定技能，只要不声明对应函数即可。

但是，如果不声明 copy 构造函数 和 copy assignment 操作符，在尝试调用的时候编译器就会声明他们

结论：

1. 当不想实现 copy 构造函数和 operator= 功能的时候，将其声明为 **private** 并且不实现其定义
2. 通过 **private** 继承一个 base class (copy 构造函数 operator= 为 private)

在 C++11 以上中，支持 = delete 的方式拒绝自动生成的函数

2.2.3 为多态基类声明 virtual 析构函数

当父类指针指向子类对象的时候，由于要求子类对象存在在堆区，即通过 new 的方式开辟的内存块。因此，在适当的时机应该 delete 掉这部分内存。但是，如果父类的析构函数为 **non-virtual**，那么当子类对象的父类指针删除时，可能子类对象并不会被销毁。

这就是形成资源泄露，影响数据结构，程序进程的主要原因

解决办法：为父类提供一个 virtual 析构函数

总结：

1. 带多态性质（带有 virtual 成员函数）的父类应该声明一个虚析构函数

2. 并不是所有的class都应该将其析构函数声明为虚析构函数

- base classes
- 多态才需要

2.2.4 别让异常逃离析构函数

因为析构函数吐出异常可能会带来过早结束程序或发生不明确行为的风险

在析构函数体中，如果要执行一些收尾工作，

```
class DBConn {           //这个 class 用来管理 DBConnection 对象
public:
    ...
~DBConn()           //确保数据库连接总是会被关闭
{
    db.close();
}

private:
    DBConnection db;
};
```

- 如果 close 抛出异常就结束程序。通常通过调用 abort 完成：

```
DBConn::~DBConn()
{
    try { db.close(); }
    catch (...) {
        制作运转记录，记下对 close 的调用失败；
        std::abort();
    }
}
```

解决方案1

如果程序遭遇一个“于析构期间发生的错误”后无法继续执行，“强迫结束程序”是个合理选项。毕竟它可以阻止异常从析构函数传播出去（那会导致不明确的行为）。也就是说调用 abort 可以抢先制“不明确行为”于死地。

- 吞下因调用 close 而发生的异常：

```
DBConn::~DBConn()
{
    try { db.close(); }
    catch (...) {
        制作运转记录，记下对 close 的调用失败；
    }
}
```

解决方案2

一般而言，将异常吞掉是个坏主意，因为它压制了“某些动作失败”的重要信息！然而有时候吞下异常也比负担“草率结束程序”或“不明确行为带来的风险”好。为了让这成为一个可行方案，程序必须能够继续可靠地执行，即使在遭遇并忽略一个错误之后。

一个较佳策略是重新设计 DBConn 接口，使其客户有机会对可能出现的问题作出反应。例如 DBConn 自己可以提供一个 close 函数，因而赋予客户一个机会得以处理“因该操作而发生的异常”。DBConn 也可以追踪其所管理之 DBConnection 是否已被关闭，并在答案为否的情况下由其析构函数关闭之。这可防止遗失数据库连接。然而如果 DBConnection 析构函数调用 close 失败，我们又将退回“强迫结束程序”或“吞下异常”的老路：

```
class DBConn {  
public:  
    ...  
    void close() {  
        db.close();  
        closed = true;  
    }  
    ~DBConn()  
    {  
        if (!closed) {  
            try {  
                //关闭连接（如果客户不那么做的话）  
            }  
        }  
    }  
};
```

解决方案3：重新设计关闭接口

- 析构函数绝对不要吐出异常。如果一个被析构函数调用的函数可能抛出异常，析 构函数应该 **捕捉任何异常，然后吞下它们（不传播）或结束程序。**
- 如果客户需要对某个操作函数运行期间抛出的异常做出反应，那么 class 应该 提供一个**普通函数（而非在析构函数中）**执行该操作。

2.2.5 绝不在构造和析构过程中调用virtual函数

首先，在创建子类对象的过程中，父类的构造函数会被更早地调用。假如在父类的构造函数中调用 **virtual** 函数，那么这时候是不会产生多态，即：**在父类构造函数作用期间，所有函数都是父类自身的成员函数实现**

因为如果在子类对象构造过程中，父类构造的时候就能通过 **virtual** 虚函数调用子类的成员函数，那么就会出现问题。**要求使用对象内部未初始化的成分**

同样地，在析构过程中，子类的析构函数先调用，进而相关成员变量就为 **未定义值**，这时候父类析构函数如果再能够通过析构函数中的 **virtual** 函数调用子类的相关成员变量，就会出现问题

2.2.5.1 解决办法

1. 在父类中将该函数声明为 **non-virtual** 并且要求派生类的构造函数传递必要的信息给基类的构造函数

```

class Transaction {
public:
    explicit Transaction(const std::string& logInfo);
    void logTransaction(const std::string& logInfo) const; //如今是个
                                                               //non-virtual 函数
    ...
};

Transaction::Transaction(const std::string& logInfo)
{
    ...
    logTransaction(logInfo); //如今是个
                           //non-virtual 调用
}

class BuyTransaction: public Transaction {
public:
    BuyTransaction( parameters )
        : Transaction(createLogString( parameters )) //将 log 信息
        { ... } //传给 base class 构造函数
    ...
private:
    static std::string createLogString( parameters );
};

```

换句话说由于你无法使用 virtual 函数从 base classes 向下调用，在构造期间，你可以藉由“令 derived classes 将必要的构造信息向上传递至 base class 构造函数”替换之而加以弥补。

2.2.6 令operator=返回一个reference to *this

为了实现“连锁赋值”，赋值操作符必须返回一个reference指向操作符的左侧实参，

```

1 class widget{
2 public:
3     widget& operator=(const widget& rhs)
4         //返回类型是一个reference, 指向当前对象*this
5     {
6         ...
7         return *this;
8     }
9 };

```

2.2.7 在operator=中处理自我赋值？？

如果通过对象来管理资源，那么可能会出现在停止使用资源之前就意外释放了它

```

1 class Bitmap{...};
2 class widget{

```

```
3 ...  
4 private:  
5     Bitmap* pb; //指针，指向一个从heap上分配得到的数据  
6 };  
7  
8 //widget的operator=实现代码  
9 Widget& Widget::operator=(const Widget& rhs)  
10 {  
11     if(*this == rhs) return *this; //identity test  
12     delete pb; //停止使用当前的pb;  
13     pb = new Bimap(*rhs.pb);  
14     return *this;  
15 }  
16 //在自我赋值的时候，会出现问题  
17 //如果*bthis和rhs为同一个对象，即自我赋值，那么delete的时候就删除掉了rhs和*bthis的  
18 //Bitmap，进而通过pb = new Bimap(*rhs.pb);会使得当前指针指向一个已被删除的对象
```

解决办法：添加一个**identity test**

```
1 //widget的operator=实现代码，添加了identity test  
2 Widget& Widget::operator=(const Widget& rhs)  
3 {  
4     if(*this == rhs) return *this; //identity test  
5     delete pb; //停止使用当前的pb;  
6     pb = new Bimap(*rhs.pb);  
7     return *this;  
8 }  
9  
10 //在复制之前不要删除pb  
11 //widget的operator=实现代码  
12 Widget& Widget::operator=(const Widget& rhs)  
13 {  
14     Bitmap* pOrig = pb; //原来的pb  
15     pb = new Bitmap(*rhs.pb); //令pb指向一个*pb的副本  
16     delete pOrig; //删除掉原来的副本  
17     return *this;  
18 }  
19 //这样做的原理是因为：如果new发生了问题，会抛出异常
```

通过try, catch来捕获异常

2.2.8 复制对象时，勿忘其每一个成分

设计良好的面向对象系统会将对象的成员属性封装起来，只留两个函数负责对象拷贝复制操作。

1. copy构造函数
2. copy assignment操作符

如果所写的copy函数没有做到每个成分的复制，那么一旦发生继承，那么派生类的copy函数需要实现

1. 复制所有local成员变量
2. 调用所有base classes内的适当的copy函数来复制其中的base classes成分

```
PriorityCustomer::PriorityCustomer(const PriorityCustomer& rhs)
    : Customer(rhs), ← //调用 base class 的 copy 构造函数
    priority(rhs.priority)
{
    logCall("PriorityCustomer copy constructor");
}

PriorityCustomer&
PriorityCustomer::operator=(const PriorityCustomer& rhs)
{
    logCall("PriorityCustomer copy assignment operator");
    Customer::operator=(rhs); //对 base class 成分进行赋值动作
    priority = rhs.priority;
    return *this;
}
```

在同一个类中，可能copy构造函数和copy assignment操作符的函数实现有很大程度的相似，但是不要一个copy函数调用另一个copy函数

- copy assignment操作符调用copy构造函数是不合理的，因为这就像试图构造一个已经存在的对象
- copy构造函数调用copy assignment操作符也毫无意义，因为构造函数是用来初始化新对象，而operator=是作用于已初始化对象上
- 如果copy函数之间有相近的代码，消除重复代码的方法是，建立一个 private 成员函数init()

2.3 资源管理

2.3.1 以对象(智能指针对象)管理资源

2.3.1.1 常用的资源

内存，文件描述器，互斥锁，图形界面中的字型和笔刷，数据库连接，网络 sockets

通过new和delete关键字进行管理资源可能会使delete被忽略。因此，单独的依靠 delete语句时行不通的

需要将资源放进对象中去，利用对象在离开作用域的时候自动调用析构函数的机制去释放资源

RAII: Resource Acquisition is initialization

```
1 void f()
2 {
3     std::auto_ptr<Investment> pInv(createInvestment());
4     //调用factory函数
5
6     //老版本的智能指针
7     std::auto_ptr<std::string> aps(new std::string[10]);
8     //可以通过编译，但是auto_ptr和shared_ptr两者内部是delete而不是delete[]
9     std::shared_ptr //auto_ptr赋值会使右值为null
```

2.3.2 在资源管理类中小心copy行为

有时候需要自定义资源 管理类RAII

但是在应对RAII对象复制的过程中的两种应对策略

1. 禁止复制

- private copy function
- =delete

2. 引用计数法: 有时候我们需要保有资源直到最后一个使用者对象被销毁时。 (浅拷贝，但是计数)

3. 深拷贝

4. 转移底部资源的拥有权

当只需要一个RAII对象指向一个资源的时候，赋值操作采用这种实现。

`auto_ptr`

2.3.3 在资源管理类中提供对原始资源的访问

举个例子，条款 13 导入一个观念：使用智能指针如 `auto_ptr` 或 `tr1::shared_ptr` 保存 **factory 函数如 `createInvestment`** 的调用结果：

```
std::tr1::shared_ptr<Investment> pInv(createInvestment()); //见条款 13
```

假设你希望以某个函数处理 `Investment` 对象，像这样：

```
int daysHeld(const Investment* pi); //返回投资天数
```

你想要这么调用它：

```
int days = daysHeld(pInv); //错误！
```

却通不过编译，因为 `daysHeld` 需要的是 `Investment*` 指针，你传给它的却是个类型为 `tr1::shared_ptr<Investment>` 的对象。

这时候你需要一个函数可将 RAII class 对象（本例为 `tr1::shared_ptr`）转换为其所内含之**原始资源**（本例为底部之 `Investment*`）。有两个做法可以达成目标：**显式转换和隐式转换**。

```
class Font {  
public:  
    FontHandle get() const { return f; } //显式转换函数  
    operator FontHandle() const //隐式转换函数  
    { return f; }  
    ...  
};
```

```
Font f(getFont());  
int newFontSize;  
...  
changeFontSize(f, newFontSize); //将 Font 隐式转换为 FontHandle
```

隐式转换存在的问题：增加发生错误的机会，

但是这个隐式转换会增加错误发生机会。例如客户可能会在需要 `Font` 时意外创建一个 `FontHandle`：

```
Font f1(getFont());  
...  
FontHandle f2 = f1; //喔欧！原意是要拷贝一个 Font 对象，  
//原本应该是Font //却反而将 f1 隐式转换为其底部的 FontHandle  
//然后才复制它。  
隐式拷贝赋值
```

你的内心也可能认为，RAII class 内的那个返回原始资源的函数，与“封装”发生矛盾。那是真的，但一般而言它谈不上是什么设计灾难。RAII classes 并不是为了封装某物而存在；它们的存在是为了确保一个特殊行为——资源释放——会发生。如果一定要，当然也可以在这基本功能之上再加一层资源封装，但那并非必要。

- APIs 往往要求访问原始资源（raw resources），所以每一个 RAII class 应该提供一个“取得其所管理之资源”的办法。
- 对原始资源的访问可能经由显式转换或隐式转换。一般而言显式转换比较安全，但隐式转换对客户比较方便。

2.3.4 成对使用new和delete时要采取相同形式

delete的最大问题在于：即将被删除的内存之内究竟存有多少对象，进而决定了有多少个析构函数必须被调用起来

当你对着一个指针使用 delete，唯一能够让 delete 知道内存中是否存在一个“数组大小记录”的办法就是：由你来告诉它。如果你使用 delete 时加上中括号（方括号），delete 便认定指针指向一个数组，否则它便认定指针指向单一对象：

```
std::string* stringPtr1 = new std::string;
std::string* stringPtr2 = new std::string[100];
...
delete stringPtr1;           //删除一个对象
delete [] stringPtr2;        //删除一个由对象组成的数组
```

游戏规则很简单：如果你调用 new 时使用 []，你必须在对应调用 delete 时也使用 []。如果你调用 new 时没有使用 []，那么也不该在对应调用 delete 时使用 []。

对于typedef的考虑：

如果通过 **typedef** 将数组声明重命名，那么通过new关键字开辟的heap内存，应该通过 **delete[]** 去释放

最好不要对数组形式做typedefs

```
1 typedef std::string AddressLines[4]; //每个人的地址有四行，每行是一个string
2 std::string* pa1 = new AddressLines;
3 //new AddressLines返回一个string* 相当于new string[4]
4 //因此，需要通过delete[] pa1;来释放内存
```

2.3.5 以独立语句将newed对象置入智能指针

```
int priority();
void processWidget(std::tr1::shared_ptr<Widget> pw, int priority);
```

由于谨记“以对象管理资源”（条款 13）的智慧铭言，processWidget 决定对
其动态分配得来的 widget 运用智能指针（这里采用 tr1::shared_ptr）。

现在考虑调用 processWidget:

```
processWidget(new Widget, priority());
```

需要进行隐式转换

等等，不要考虑这个调用形式。它不能通过编译。tr1::shared_ptr 构造函数
需要一个原始指针（raw pointer），但该构造函数是个 explicit 构造函数，无法进
行隐式转换，将得自“newWidget”的原始指针转换为 processWidget 所要求的 tr1::
shared_ptr。如果写成这样就可以通过编译：

```
processWidget(std::tr1::shared_ptr<Widget>(new Widget), priority());
```

令人惊讶的是，虽然我们在此使用“对象管理式资源”（object-managing
resources），上述调用却可能泄漏资源。稍后我再详加解释。

1. 执行 "new Widget"
2. 调用 priority
3. 调用 tr1::shared_ptr 构造函数

现在请你想想，万一对 priority 的调用导致异常，会发生什么事？在此情况
下 "new Widget" 返回的指针将会遗失，因为它尚未被置入 tr1::shared_ptr 内，
后者是我们期盼用来防卫资源泄漏的武器。是的，在对 processWidget 的调用过程
中可能引发资源泄漏，因为在“资源被创建（经由 "new Widget"）”和“资源被

```
std::tr1::shared_ptr<Widget> pw(new Widget); //在单独语句内以
processWidget(pw, priority()); //智能指针存储
// newed 所得对象。
//这个调用动作绝不至于造成泄漏。
```

解决办法：以独立语句将 newed 对象置入智能指针

2.4 设计与声明

2.4.1 让接口容器被正确使用，不易被误用

```

struct Day {
    explicit Day(int d)
        : val(d) { }
    int val;
};

struct Month {
    explicit Month(int m)
        : val(m) { }
    int val;
};

struct Year {
    explicit Year(int y)
        : val(y) { }
    int val;
};

class Date {
public:
    Date(const Month& m, const Day& d, const Year& y);
    ...
};

Date d(30, 3, 1995); //错误!不正确的类型 声明为explicit禁止隐式转换
Date d(Day(30), Month(3), Year(1995)); //错误!不正确的类型 传入参数类型不匹配
Date d(Month(3), Day(30), Year(1995)); //OK, 类型正确

```

通过定义三个struct来接受数据并转换为可区分的自定义类型

令 Day, Month 和 Year 成为成熟且经充分锻炼的 classes 并封装其内数据，比简单使用上述的 structs 好（见条款 22）。但即使 structs 也已经足够示范：明智而审慎地导入新类型对预防“接口被误用”有神奇疗效。

实际上，返回 `tr1::shared_ptr` 让接口设计者得以阻止一大群客户犯下资源泄漏的错误，因为就如条款 14 所言，`tr1::shared_ptr` 允许当智能指针被建立起来时指定一个资源释放函数（所谓删除器，“deleter”）绑定于智能指针身上（`auto_ptr` 就没有这种能耐）。

- 好的接口很容易被正确使用，不容易被误用。你应该在你的所有接口中努力达成这些性质。
比如：`size()` 返回大小，`length()` 就尽可能返回常数。
- “促进正确使用”的办法包括接口的一致性，以及与内置类型的行为兼容。
- “阻止误用”的办法包括建立新类型、限制类型上的操作，束缚对象值，以及消除客户的资源管理责任。
- `tr1::shared_ptr` 支持定制型删除器（custom deleter）。这可防范 DLL 问题，可被用来自动解除互斥锁（mutexes；见条款 14）等等。

2.4.2 设计class犹如设计type

定义一个新的class相当于定义了一个新的type

2.4.2.1 设计class应该考虑的设计规范

- 新 type 的对象应该如何被创建和销毁？这会影响到你的 class 的构造函数和析构函

BIG THREE
数以及内存分配函数和释放函数 (operator new, operator new[], operator delete 和 operator delete[] ——见第 8 章) 的设计，当然前提是如果你打算撰写它们。

- 对象的初始化和对象的赋值该有什么样的差别？这个答案决定你的构造函数和赋值 (assignment) 操作符的行为，以及其间的差异。很重要的是别混淆了“初始化”

const 常量
引用和指针必须初始化：参数列表初始化

注意
1. 深拷贝和浅拷贝的区别
2. 数据是否只允许保持一份 (MCTOR)

- 新 type 的对象如果被 passed by value (以值传递)，意味着什么？记住，copy 构造函数用来定义一个 type 的 pass-by-value 该如何实现。

值传递：意味着对传入参数的影响不会作用到原有对象
并且，如果传入的是一个自定义数据类型，会调用 copy 构造

- 什么是新 type 的“合法值”？对 class 的成员变量而言，通常只有某些数值集是有效的。那些数值集决定了你的 class 必须维护的约束条件 (invariants)，也就决定了你的成员函数（特别是构造函数、赋值操作符和所谓 “setter” 函数）必须进行的错误检查工作。它也影响函数抛出的异常、以及（极少被使用的）函数异常明细列 (exception specifications)。

- 你的新 type 需要配合某个继承图系 (inheritance graph) 吗？如果你继承自某些既

有的 classes，你就受到那些 classes 的设计的束缚，特别是受到“它们的函数是 virtual 或 non-virtual”的影响（见条款 34 和条款 36）。如果你允许其他 classes 继承你的 class，那会影响你所声明的函数——尤其是析构函数——是否为 virtual（见条款 7）。

对于派生类：需要拷贝父类的相关设计，virtual 的影响
对于可能充当基类的 class，考虑声明的函数是否为 virtual

- 你的新 type 需要什么样的转换？你的 type 生存于其他一些 types 之间，因而彼此该有转换行为吗？如果你希望允许类型 T1 之物被隐式转换为类型 T2 之物，就必须在 class T1 内写一个类型转换函数 (operator T2) 或在 class T2 内写一个 non-explicit-one-argument (可被单一实参调用) 的构造函数。如果你只允许 explicit 构造函数存在，就得写出专门负责执行转换的函数，且不得为类型转换操作符 (type conversion operators) 或 non-explicit-one-argument 构造函数。（条款 15 有隐式和显式转换函数的范例。）

- 什么样的操作符和函数对此新 type 而言是合理的？这个问题的答案决定你将为你的 class 声明哪些函数。其中某些该是 member 函数，某些则否（见条款 23, 24, 46）。

比如：是否支持 copy 操作

- 什么样的标准函数应该驳回？那些正是你必须声明为 private 者（见条款 6）。

- 谁该取用新 type 的成员？这个提问可以帮助你决定哪个成员为 public，哪个为 protected，哪个为 private。它也帮助你决定哪一个 classes 和/或 functions 应该是 friends，以及将它们嵌套于另一个之内是否合理。

成员的作用域约束

- 什么是新 type 的“未声明接口”（undeclared interface）？它对效率、异常安全性（见条款 29）以及资源运用（例如多任务锁定和动态内存）提供何种保证？你在这些方面提供的保证将为你的 class 实现代码加上相应的约束条件。

- 你的新 type 有多么一般化？或许你其实并非定义一个新 type，而是定义一个整个 types 家族。果真如此你就不该定义一个新 class，而是应该定义一个新的 class template。

比如传入的参数类型可能多种，vector 容器这些，应该定义类模板

- 你真的需要一个新 type 吗？如果只是定义新的 derived class 以便为既有的 class 添加机能，那么说不定单纯定义一或多个 non-member 函数或 templates，更能够达到目标。

有时候定义一个新的普通函数或者一个函数模板就能取代 class 的定义

2.4.3 通过pass-by-reference-to-const替换pass-by-value

当C++通过值传递的时候，函数参数是以实际参数的副本为初值，这个副本是通过对象的copy构造函数得到。因此，通过值传递的时候，可能会使得pass-by-value成为一个昂贵的操作

（*by value*）开返回它是否有效：

```
bool validateStudent(Student s);           //函数以 by value 方式接受学生  
Student plato;                          //柏拉图,苏格拉底的学生  
bool platoIsOK = validateStudent(plato); //调用函数
```

当上述函数被调用时，发生什么事？

student s = plato

无疑地 Student 的 *copy* 构造函数会被调用，以 plato 为蓝本将 s 初始化。同样明显地，当 validateStudent 退回 s 会被销毁。因此，对此函数而言，参数的传递成本是“一次 Student *copy* 构造函数调用，加上一次 student 析构函数调用”。

但那还不是整个故事喔。Student 对象内有两个 string 对象，所以每次构造一个 Student 对象也就构造了两个 string 对象。此外 Student 对象继承自 Person 对象，所以每次构造 Student 对象也必须构造出一个 Person 对象。一个 Person 对象又有两个 string 对象在其中，因此每一次 Person 构造动作又需承担两个 string 构造动作。最终结果是，以 *by value* 方式传递一个 Student 对象会导致调用一次 Student *copy* 构造函数、一次 Person *copy* 构造函数、四次 string *copy* 构造函数。当函数内的那个 Student 复件被销毁，每一个构造函数调用动作都需要一个对应的析构函数调用动作。因此，以 *by value* 方式传递一个 Student 对象，总体成本是“六次构造函数和六次析构函数”！

```
bool validateStudent(const Student& s);    直接pass-by-value，不会对原本进行修改，因为函数实际获得的是一个实参的副本，通过copy构造获得
```

这种传递方式的效率高得多：没有任何构造函数或析构函数被调用，因为没有任何新对象被创建。修订后的这个参数声明中的 *const* 是重要的。原先的 validateStudent 以 *by value* 方式接受一个 Student 参数，因此调用者知道他们受到保护，函数内绝不会对传入的 Student 作任何改变；validateStudent 只能够对其复印件（副本）做修改。现在 Student 以 *by reference* 方式传递，将它声明为 *const* 是必要的，因为不这样做的话调用者会忧虑 validateStudent 会不会改变他们传入的那个 Student。

- 尽量以 *pass-by-reference-to-const* 替换 *pass-by-value*。前者通常比较高效，并可避免切割问题（slicing problem）。
- 以上规则并不适用于内置类型，以及 STL 的迭代器和函数对象。对它们而言，*pass-by-value* 往往比较适当。

2.4.4 必须返回对象时，别妄想返回其reference

```
const Rational& operator* (const Rational& lhs,
                           const Rational& rhs)
{
    Rational result(lhs.n * rhs.n, lhs.d * rhs.d); //警告！糟糕的代码！
    return result;
}
```

返回值做任何一点点运用，都将立刻坠入“无定义行为”的恶地。事情的真相是，任何函数如果返回一个 reference 指向某个 local 对象，都将一败涂地。（如果函数返回指针指向一个 local 对象，也是一样。）

即使调用者诚实谨慎，并且出于良好意识，他们还是不太能够在这样合情合理的用法下阻止内存泄漏：

```
Rational w, x, y, z;
w = x * y * z; //与 operator*(operator*(x, y), z) 相同
```

这里，同一个语句内调用了两次 operator*，因而两次使用 new，也就需要两次 delete。但却没有合理的办法让 operator* 使用者进行那些 delete 调用，因为没有合理的办法让他们取得 operator* 返回的 references 背后隐藏的那个指针。这绝对导致资源泄漏。

这是在函数operator*内部开辟出来的内存

一个“必须返回新对象”的函数的正确写法是：就让那个函数返回一个新对象呗。对 Rational 的 operator* 而言意味以下写法（或其他本质上等价的代码）：

```
inline const Rational operator * (const Rational& lhs, const Rational& rhs)
{
    return Rational(lhs.n * rhs.n, lhs.d * rhs.d);
} 等价于 const Rational res = Rational(lhs.n &* rhs.n, ~);
```

请记住

当连续调用该函数时，就会出现多个new而出现内存泄漏 ✗

- 绝不要返回 pointer 或 reference 指向一个 local stack 对象，或返回 reference 指向一个 heap-allocated 对象，或返回 pointer 或 reference 指向一个 local static 对象而有可能同时需要多个这样的对象。条款 4 已经为“在单线程环境中合理返回 reference 指向一个 local static 对象”提供了一份设计实例。

2.4.5 将成员变量声明为private

结论：成员变量应该是private

因为在public中的成员变量能够被无数的函数访问，一旦对该变量做了一些改动。就要涉及到许多潜在的改动。

2.4.6 宁以non-member、non-friend替换member函数

C++的特性之一是封装，当定义更多的成员函数的时候，就会使得封装性变得很差。`private`变量的意义就变得轻微起来

最好的是将一些操作封装成头文件，放在一个namespace之中去

将所有便利函数放在多个头文件内但隶属同一个命名空间，意味客户可以轻松扩展这一组便利函数。他们需要做的就是添加更多 non-member non-friend 函数到此命名空间内。举个例子，如果某个 `WebBrowser` 客户决定写些与影像下载相关的便利函数，他只需要在 `WebBrowserStuff` 命名空间内建立一个头文件，内含那些函数的声明即可。新函数就像其他旧有的便利函数那样可用且整合为一体。这是 class 无法提供的另一

请注意

- 宁可拿 non-member non-friend 函数替换 member 函数。这样做可以增加封装性、包裹弹性（packaging flexibility）和机能扩充性。

2.4.7 若所有参数皆需类型转换，请为此采用non-member函数

我在导读中提过，令 classes 支持隐式类型转换通常是个糟糕的主意。当然这条规则有其例外，最常见的例外是在建立数值类型时。假设你设计一个 class 用来表现有理数。

```
class Rational {  
public:  
    ...  
    const Rational operator* (const Rational& rhs) const;  
};  
//不返回引用是因为：可能是一个临时对象  
//返回const是因为防止返回值被修改  
Rational oneEighth(1, 8);  
Rational oneHalf(1, 2);  
Rational result = oneHalf * oneEighth;      //很好  
result = result * oneEighth;                  //很好
```

但你还不满足。你希望支持混合式运算，也就是拿 `Rationals` 和……嗯……例如 `ints` 相乘。毕竟很少有什么东西会比两个数值相乘更自然的了——即使是两个不同类型的数值。

然而当你尝试混合式算术，你发现只有一半行得通：

```
result = oneHalf * 2;           //发生隐式转换  
result = 2 * oneHalf;          //放在operator*无法发生隐式转换  
                                //错误！
```

这不是好兆头。乘法应该满足交换律，不是吗？

让以上语句通过编译。这把我们带回到上述两个语句，为什么即使 Rational 构造函数不是 explicit，仍然只有一个可通过编译，另一个不可以：

```
result = oneHalf * 2;           //没问题（在 non-explicit 构造函数的情况下）
result = 2 * oneHalf;          //错误！（甚至在 non-explicit 构造函数的情况下）
```

结论是，只有当参数被列于参数列 (parameter list) 内，这个参数才是隐式类型转换的合格参与者。地位相当于“被调用之成员函数所隶属的那个对象”——即 this 对象——的那个隐喻参数，绝不是隐式转换的合格参与者。这就是为什么上述第一次调用可通过编译，第二次调用则否，因为第一次调用伴随一个放在参数列内的参数，第二次调用则否。

麻烦往往多过其价值。当然有时候 friend 有其正当性，但这个事实依然存在：不能够只因函数不该成为 member，就自动让它成为 friend。

如果你需要为某个函数的所有参数（包括被 this 指针所指的那个隐喻参数）进行类型转换，那么这个函数必须是个 non-member。

2.4.8 考虑写出一个不抛出异常的swap函数

```
namespace std {
    template<typename T>           //std::swap 的典型实现;
    void swap( T& a, T& b)         //置换 a 和 b 的值.
    {
        T temp(a);
        a = b;
        b = temp;
    }
}
```

内置的swap函数模板，
需要创建一个临时对象temp

我们希望能够告诉 std::swap：当 Widgets 被置换时真正该做的是置换其内部的 pImpl 指针。确切实践这个思路的一个做法是：将 std::swap 针对 Widget 特化。下面是基本构想，但目前这个形式无法通过编译：

```
namespace std {
    template<>
    void swap<Widget>( Widget& a,           //这是 std::swap 针对
                        Widget& b )           //“T 是 Widget”的特化版本。
    {
        swap(a.pImpl, b.pImpl);           //目前还不能通过编译。
    }
}
```

这个函数一开始的 "`template<>`" 表示它是 std::swap 的一个全特化 (*total template specialization*) 版本，函数名称之后的 "`<Widget>`" 表示这一特化版本系针对 "T 是 Widget" 而设计。换句话说当一般的 swap template 施行于 Widgets 身上便会启用这个版本。通常我们不能够（不被允许）改变 std 命名空间内的任何东西，但可以（被允许）为标准 templates（如 swap）制造特化版本，使它专属于我们自己的 classes（例如 Widget）。以上作为正是如此。

2.5 实现

2.5.1 尽可能延后变量定义式的出现时间

原因：

1. 对于自定义数据类型，在定义的过程中，会有构造和析构的消耗
2. 并且，对于开辟在堆区的数据内存，若相应的delete被以某种形式跳过，则可能会造成内存泄漏

如果 classes 的一个赋值成本低于一组构造+析构成本，做法 A 大体而言比较高效。尤其当 n 值很大的时候。否则做法 B 或许较好。此外做法 A 造成名称 w 的作用域（覆盖整个循环）比做法 B 更大，有时那对程序的可理解和易维护性造成冲突。因此除非（1）你知道赋值成本比“构造+析构”成本低，（2）你正在处理代码中效率高度敏感（performance-sensitive）的部分，否则你应该使用做法 B。

请记住

- 尽可能延后变量定义式的出现。这样做可增加程序的清晰度并改善程序效率。

2.5.2 尽量少做转型动作

2.5.2.1 C风格转型

```
1 //旧式转型  
2 (T)expression; //将expression转型为T  
3 //函数风格的转型看起来像：通过拷贝构造创建了一个新的对象  
4 T(expression);
```

2.5.2.2 C++提供的四种新式转型

const_cast<T> (expression)
dynamic_cast<T> (expression)
reinterpret_cast<T> (expression)
static_cast<T> (expression)

- const_cast 通常被用来将对象的常量性转除（cast away the constness）。它也是唯一有此能力的 C++-style 转型操作符。
- dynamic_cast 主要用来执行“安全向下转型”（safe downcasting），也就是用来决定某对象是否归属继承体系中的某个类型。它是唯一无法由旧式语法执行的动作，也是唯一可能耗费重大运行成本的转型动作（稍后细谈）。尽可能避免使用dynamic_cast
- reinterpret_cast 意图执行低级转型，实际动作（及结果）可能取决于编译器，这也就表示它不可移植。例如将一个 pointer to int 转型为一个 int。这一类转型在低级代码以外很少见。本书只使用一次，那是在讨论如何针对原始内存（raw memory）写出一个调试用的分配器（debugging allocator）时，见条款 50。
- static_cast 用来强迫隐式转换（implicit conversions），例如将 non-const 对象转为 const 对象（就像条款 3 所为），或将 int 转为 double 等等。它也可以用来执行上述多种转换的反向转换，例如将 void* 指针转为 typed 指针，将 pointer-to-base 转为 pointer-to-derived。但它无法将 const 转为 non-const——这个只有 const_cast 才办得到。

使用新式转型的好处：

1. 容易在代码中被辨识出来
2. 各转型动作的目标越明确，编译器越可能诊断出错误的运用

请记住

- 如果可以，尽量避免转型，特别是在注重效率的代码中避免 `dynamic_casts`。
如果有设计需要转型动作，试着发展无需转型的替代设计。
- 如果转型是必要的，试着将它隐藏于某个函数背后。客户随后可以调用该函数，而不需将转型放进他们自己的代码内。
- 宁可使用 C++-style (新式) 转型，不要使用旧式转型。前者很容易辨识出来，而且也比较有着分门别类的职掌。

2.5.3 避免返回handles (reference, 指针, 迭代器) 指向对象内部成分

对象内部的成分：对象成员中不被公开使用的成员函数，成员变量（都应该是 `private`）

1. 成员变量的封装性最多只等于“返回其reference”的函数的访问级别
2. 如果 `const` 成员函数传出一个 `reference`，后者所指数据与对象自身又关联，而它又被存储于对象之外，那么这个函数的调用者可以修改那笔数据
3. 返回一个“代表对象内部数据”的 `handle`，随之而来的是“降低对象封装性”的风险。并且，可能让原先声明为 `const` 的成员函数却造成对象状态的更改

通常我们认为，对象的“内部”就是指它的成员变量，但其实不被公开使用的成员函数（也就是被声明为 `protected` 或 `private` 者）也是对象“内部”的一部分。因此也应该留心不要返回它们的 handles。这意味着你绝对不该令成员函数返回一个指针指向“访问级别较低”的成员函数。如果你那么做，后者的实际访问级别就会提高如同前者（访问级别较高者），因为客户可以取得一个指针指向那个“访问级别较低”的函数，然后通过那个指针调用它。

```
class GUIObject { ... };
const Rectangle boundingBox(const GUIObject& obj); //条款 3 谈过为什么返回类型是 const
```

现在，客户有可能这么使用这个函数：

```
GUIObject* pgo; //让 pgo 指向某个 GUIObject
...
const Point* pUpperLeft = //取得一个指针指向外框左上点
    &(boundingBox(*pgo).upperLeft());
```

对 `boundingBox` 的调用获得一个新的、暂时的 `Rectangle` 对象，这个对象没有名称，所以我们权且称它为 `temp`。随后 `upperLeft` 作用于 `temp` 身上，返回一个 `reference` 指向 `temp` 的一个内部成分，更具体地说是指向一个用以标示 `temp` 的 `Point`s。于是 `pUpperLeft` 指向那个 `Point` 对象。目前为止一切还好，但故事尚未结束，因为在那个语句结束之后，`boundingBox` 的返回值，也就是我们所说的 `temp`，将被销毁，而那间接导致 `temp` 内的 `Point`s 析构。最终导致 `pUpperLeft` 指向一个不再存在的对象；也就是说一旦产出 `pUpperLeft` 的那个语句结束，`pUpperLeft` 也就变成空悬、虚吊 (*dangling*)！

这就是为什么函数如果“返回一个 handle 代表对象内部成分”总是危险的原因。

不论这所谓的 handle 是个指针或迭代器或 reference，也不论这个 handle 是否为 const，也不论那个返回 handle 的成员函数是否为 const。这里的唯一关键是，有一个 handle 被传出去了，一旦如此你就是暴露在“handle 比其所指对象更长寿”的风险下。

2.5.4 为“异常安全”而努力是值得的

从“异常安全性”的观点来看，这个函数很糟。“异常安全”有两个条件，而这个函数没有满足其中任何一个条件。

当异常被抛出时，带有异常安全性的函数会：

- ■ 不泄漏任何资源。上述代码没有做到这一点，因为一旦 "new Image(imgSrc)" 导致异常，对 unlock 的调用就绝不会执行，于是互斥器就永远被把持住了。
- ■ 不允许数据败坏。如果 "new Image(imgSrc)" 抛出异常，bgImage 就是指向一个已被删除的对象，imageChanges 也已被累加，而其实并没有新的图像被成功安装起来。（但从另一个角度说，旧图像已被消除，所以你可能会争辩说图像

```
1 //设计一个class来表现夹带背景图案的GUI菜单，这个class希望用于多
2 线程环境。所以他有个互斥器（mutex）作为并发控制之用
3 class PrettyMenu{
4 public:
5     void changeBackground(std::istream& imgSrc); //改变
6     背景图像
7 private:
8     Mutex mutex;
9     Image* bgImage;
10    int imageChanges;
11 };
12 //changeBackground()的一种实现
13 void PrettyMenu::changeBackground(std::istream& imgSrc)
14 {
15     lock(&mutex); //取得互斥器
16     delete bgImage; //删除旧的背景图像
17     ++imageChanges;
18     bgImage = new Image(imgSrc);
19     unlock(&mutex); //释放互斥器
20 }
```

上述代码存在的问题：

- 可能存在资源泄露：一旦new失败，那么mutex就不会被释放，造成互斥器占用
- 数据败坏问题：一旦new失败，会导致bgImage指向一个被删除的对象，imageChanges也被累加

```

1 //资源泄露解决方案
2 void PrettyMenu::changeBackground(std::istream& imgSrc)
3 {
4     Lock m1(&mutex); //条款14：利用资源管理类的构造和析构自动管
理资源
5     delete bgImage; //删除旧的背景图像
6     ++imageChanges;
7     bgImage = new Image(imgSrc);
8 }
```

2.5.4.1 异常安全的函数三个保证

- 基本承诺：**如果异常被抛出，程序内的任何事物仍然保持在有效状态下，没有任何对象或数据结构会因此而败坏
- 强烈保证：**如果异常被抛出，程序状态不改变。如果函数成功，就是完全成功，如果函数失败，程序会回复到“调用函数之前”的状态

当“强烈保证”不切实际时，你就必须提供“基本保证”。现实中你或许会发现，你可以为某些函数提供强烈保证，但效率和复杂度带来的成本会使它对许多人而言摇摇欲坠。只要你曾经付出适当的心力试图提供强烈保证，万一实际不可行，使你退而求其次地只提供基本保证，任何人都不该因此责难你。对许多函数而言，“异常安全性之基本保证”是一个绝对通情达理的选择。

- 不抛掷保证 (nothrow)**，承诺绝不抛出异常，因为他们总是能够完成他们原先承诺的功能

```
int doSomething() throw(); //注意“空白的异常明细”
                           // (empty exception spec)
```

这并不是说 doSomething 绝不会抛出异常，而是说如果 doSomething 抛出异常，将是严重错误，会有你意想不到的函数被调用¹。实际上 doSomething 也许完全没有提供任何异常保证。函数的声明式（包括其异常明细——如果有的话）并不能够告诉你是否它是正确的、可移植的或高效的，也不能够告诉你它是否提供任何异常安全性保证。所有那些性质都由函数的实现决定，无关乎声明。

```

struct PMImpl {
    std::tr1::shared_ptr<Image> bgImage; //PMImpl = "PrettyMenu Impl";
    int imageChanges;
};

class PrettyMenu {
    ...
private:
    Mutex mutex;
    std::tr1::shared_ptr<PMImpl> pImpl;
};

void PrettyMenu::changeBackground(std::istream& imgSrc)
{
    using std::swap; //见条款 25
    Lock ml(&mutex); //获得 mutex 的副本数据 → 避免了资源泄露
    std::tr1::shared_ptr<PMImpl> pNew(new PMImpl(*pImpl)); //创建一个当前pImpl的副本，用于交换。
    if (pNew->bgImage.reset(new Image(imgSrc)) == nullptr) //如果失败，则对原有状态没有什么影响
        ++pNew->imageChanges;

    swap(pImpl, pNew); //置换 (swap) 数据，释放 mutex
}

```

copy-and-swap的关键在于：修改对象数据的副本，然后在一个不抛出异常的函数中将修改后的数据和原件置换

- 异常安全函数（Exception-safe functions）即使发生异常也不会泄漏资源或允许任何数据结构败坏，这样的函数区分为三种可能的保证：基本型、强烈型、不抛异常型。
- “强烈保证”往往能够以 copy-and-swap 实现出来，但“强烈保证”并非对所有函数都可实现或具备现实意义。因为会在copy的过程中占用额外的时间和空间
- 函数提供的“异常安全保证”通常最高只等于其所调用之各个函数的“异常安全保证”中的最弱者。

2.5.5 透彻了解inlining的里里外外

inline函数：可以调用他们却不需要函数调用所招致的额外开销（堆栈区的内存）

inline函数的整体观念是：“对此函数的每一个调用”都以函数本体替换之

在class中的函数被隐喻声明为 **inline** （包括定义在class内部的friend函数）

inline函数通常一定被置于头文件内，因为大多数build environments在编译过程中进行inlining，而为了将一个函数调用替换为 **被调用函数的本体**，编译器必须知道那个函数长什么样子

现在让我们先结束“`inline` 是个申请，编译器可加以忽略”的观察。大部分编译器拒绝将太过复杂（例如带有循环或递归）的函数 `inlining`，而所有对 `virtual` 函数的调用（除非是最平淡无奇的）也都会使 `inlining` 落空。这不该令你惊讶，因为 `virtual` 意味“等待，直到运行期才确定调用哪个函数”，而 `inline` 意味“执行前，先将调用动作替换为被调用函数的本体”。如果编译器不知道该调用哪个函数，你就很难责备它们拒绝将函数本体 `inlining`。

这使我们在决定哪些函数该被声明为 `inline` 而哪些函数不该时，掌握一个合乎逻辑的策略。一开始先不要将任何函数声明为 `inline`，或至少将 `inlining` 施行范围局限在那些“一定成为 `inline`”（见条款 46）或“十分平淡无奇”（例如 p.135 `Person::age`）的函数身上。慎重使用 `inline` 便是对日后使用调试器带来帮助，不过这么一来也等于把自己推向手工最优化之路。不要忘记 80-20 经验法则：平均而言一个程序往往将 80% 的执行时间花费在 20% 的代码上头。这是一个重要的法则，因为它提醒你，作为一个软件开发者，你的目标是找出这可以有效增进程序整体效率的 20% 代码，然后将它 `inline` 或竭尽所能地将它瘦身。但除非你选对目标，否则一切都是虚功。

请记住

不应该对逻辑分支结构的函数 `inline`

- 将大多数 `inlining` 限制在小型、被频繁调用的函数身上。这可使日后的调试过程和二进制升级（binary upgradability）更容易，也可使潜在的代码膨胀问题最小化，使程序的速度提升机会最大化。
- ~~不要只因为 `function templates` 出现在头文件，就将它们声明为 `inline`。~~

2.5.6 将文件间的编译依存关系降至最低

编译依存关系指的是：当前头文件的一些定义，需要引用一些其他的自定义头文件

如果这些头文件中有一个发生改变，或者这些头脑文件所依赖的其他头文件有任何改变。那么任何一个含入当前 class 的文件也得重新编译

利用声明的依存性替换定义的依存性

也就是 cpp 中 .h 和 .cpp 分开编写 的缘由

```

class Person {
public:
    Person(const std::string& name, const Date& birthday,
           const Address& addr);
    std::string name() const;
    std::string birthDate() const;
    std::string address() const;
    ...
private:
    std::string theName;           //实现细目
    Date theBirthDate;           //实现细目
    Address theAddress;          //实现细目
};

```

这里的 class Person 无法通过编译——如果编译器没有取得其实现代码所用到的 classes string, Date 和 Address 的定义式。这样的定义式通常由 #include 指示符提供，所以 Person 定义文件的最上方很可能存在这样的东西：

```

#include <string>
#include "date.h"
#include "address.h"

```

2.5.6.1 为什么需要前置声明

前置声明中的问题：编译器必须在编译期间知道对象的大小

对于自定义数据类型 Person，编译器需要通过访问 Person class 的定义式来确定一个 Person 对象的大小

问题出在 C++ 并没有把“将接口从实现中分离”这事做得很好。Class 的定义式不只详细叙述了 class 接口，还包括十足的实现细目。例如：

```

class Person {
public:
    Person(const std::string& name, const Date& birthday,
           const Address& addr);
    std::string name() const;
    std::string birthDate() const;
    std::string address() const;
    ...
private:
    std::string theName;           //实现细目
    Date theBirthDate;           //实现细目
    Address theAddress;          //实现细目
};

namespace std {
    class string;           //前置声明（不正确，详下）
    class Date;             //前置声明
    class Address;          //前置声明
    class Person {
public:
    Person(const std::string& name, const Date& birthday,
           const Address& addr);
    std::string name() const;
    std::string birthDate() const;
    std::string address() const;
    ...
    };
}

```

前置声明将实现细目分开叙述

于一个指针背后”的游戏。针对 Person 我们可以这样做：把 Person 分割为两个 classes，一个只提供接口，另一个负责实现该接口。如果负责实现的那个所谓 implementation class 取名为 PersonImpl，Person 将定义如下：

```
#include <string>           //标准程序库组件不该被前置声明。
#include <memory>          //此乃为了 tr1::shared_ptr 而含入；详后。
class PersonImpl;          //Person 实现类的前置声明。
class Date;                //Person 接口用到的 classes 的前置声明。
class Address;
class Person {
public:
    Person(const std::string& name, const Date& birthday,
            const Address& addr);
    std::string name() const;
    std::string birthDate() const;
    std::string address() const;
    ...
private:
    std::tr1::shared_ptr<PersonImpl> pImpl; //指针，指向实现物;
};                                //std::tr1::shared_ptr 见条款 13.
```

2.5.6.2 结论

1. 如果使用 object references 或者 object pointers 可以完成任务，就不要使用 objects 本身。

一个类型声明式就可以定义出指向该类型的 references 或者 pointers，但是如果定义某个类型的 objects，就需要用到该类型的定义式

- 如果能够，尽量以 **class 声明式替换 class 定义式**。注意，当你声明一个函数而它用到某个 class 时，你并不需要该 class 的定义；纵使函数以 *by value* 方式传递该类型的参数（或返回值）亦然：

```
class Date; → //class 声明式。
Date today();           //没问题 — 这里并不需要
void clearAppointments(Date d); → Date 的定义式。 ???
```

当然，*pass-by-value* 一般而言是个糟糕的主意（见条款 20），但如果你发现因为某种因素被迫使用它，并不能够就此为“非必要之编译依存关系”导入正当性。

- 为声明式和定义式提供不同的头文件。为了促进严守上述准则，需要两个头文件，一个用于声明式，一个用于定义式。当然，这些文件必须保持一致性，如果有个声明式被改变了，两个文件都得改变。因此程序库客户应该总是#include一个声明文件而非前置声明若干函数，程序库作者也应该提供这两个头文件。举个例子，Date 的客户如果希望声明 today 和 clearAppointments，他们不该像先前那样以手工方式前置声明 Date，而是应该 #include适当的、内含声明式的头文件：

```
#include "datefwd.h"          //这个头文件内声明（但未定义）class Date.  
Date today();                //同前。  
void clearAppointments(Date d);
```

2.6 继承于面向对象设计

尽管如此，C++ 的 OOP 有可能和你原本习惯的 OOP 稍有不同：“继承”可以是单一继承或多重继承，每一个继承连接（link）可以是 public, protected 或 private，也可以是 virtual 或 non-virtual。然后是成员函数的各个选项：virtual? non-virtual? pure virtual? 以及成员函数和其他语言特性的交互影响：缺省参数值与 virtual 函数有什么交互影响？继承如何影响 C++ 的名称查找规则？设计选项有哪些？如果 class 的行为需要修改，virtual 函数是最佳选择吗？

本章对这些题目全面宣战。此外我也解释 C++ 各种不同特性的真正意义，也就是当你使用某个特定构件你真正想要表达的意思。例如“public 继承”意味 “is-a”，如果你尝试让它带着其他意义，你会惹祸上身。同样道理，virtual 函数意味 “接口必须被继承”，non-virtual 函数意味 “接口和实现都必须被继承”。如果不能区分这些意义，会造成 C++ 程序员大量的苦恼。

2.6.1 确定你的public继承塑模出is-a关系

在C++进行面向对象编程，最重要的一个规则是：public inheritance 意味着 is - a 的关系

public 继承 意味着 is-a，适用于base classes身上的每一件事情也一定适用于derived classes身上。因为每一个derived class对象也都是一个base class对象

```
class B();
class D:public B;    
```

如果你令 class D ("Derived") 以 public 形式继承 class B ("Base")，你便是告诉 C++ 编译器（以及你的代码读者）说，每一个类型为 D 的对象同时也是一个类型为 B 的对象，反之不成立。你的意思是 B 比 D 表现出更一般化的概念，而 D 比 B 表现出更特殊化的概念。你主张“B 对象可派上用场的任何地方，D 对象一样可以派上用场”（译注：此即所谓 Liskov Substitution Principle），因为每一个 D 对象都是一种（是一个）B 对象。反之如果你需要一个 D 对象，B 对象无法效劳，因为虽然每个 D 对象都是一个 B 对象，反之并不成立。

于是，承上所述，在 C++ 领域中，任何函数如果期望获得一个类型为 Person（或 pointer-to-Person 或 reference-to-Person）的实参，都愿意接受一个 Student 对象（或 pointer-to-Student 或 reference-to-Student）：

```
void eat(const Person& p);      //任何人都会吃
void study(const Student& s);   //只有学生才到校学习
Person p;                      //p 是人
Student s;                     //s 是学生
eat(p);                        //没问题，p 是人
eat(s);                        //没问题，s 是学生，而学生也是 (is-a) 人
study(s);                      //没问题，s 是个学生
study(p);                      //错误！p 不是个学生
```

这种情况只针对 **public** 继承才成立

很多时候，A 通过 public 继承并不合理。

为了表现“企鹅不会飞，就这样”的限制，你不可以为 Penguin 定义 fly 函数：

```
class Bird {
    ...
};

class Penguin: public Bird {
    ...
};
```

现在，如果你试图让企鹅飞，编译器会对你的背信加以谴责：

```
Penguin p;
p.fly();
```

//错误！

这和采取“令程序于运行期发生错误”的解法极为不同。若以那种做法，编译器不会对 p.fly 调用式发出任何抱怨。条款 18 说过：好的接口可以防止无效的代码通过编译，因此 你应该宁可采取“在编译期拒绝企鹅飞行”的设计，而不是“只在运行期才能侦测它们”的设计。

在调用Penguin::fly()的时候打印显示错误信息的定义方法

2.6.2 避免掩盖继承而来的名称

derived class 作用域被嵌套在 base class 作用域内

1. derive classes 内的名称会遮掩 base classes 内的名称，在 public 继承下从来没有希望如此
2. 为了让被遮掩的名称再见天日，可使用 **using** 声明式或转交函数

```

class Base {
private:
    int x;
public:
    virtual void mf1() = 0;
    virtual void mf2();
    void mf3();
    ...
};

class Derived: public Base {
public:
    virtual void mf1();
    void mf4();
    ...
};

```

Base的作用域

x(成员变量)
mf1(1个函数)
mf2(1个函数)
mf3(1个函数)

Derived的作用域
mf1(1个函数)
mf4(1个函数)

```

class Base {
private:
    int x;
public:
    virtual void mf1() = 0;
    virtual void mf1(int);
    virtual void mf2();
    void mf3();
    void mf3(double);
    ...
};

class Derived: public Base {
public:
    virtual void mf1();
    void mf3();
    void mf4();
    ...
};

class Base {
private:
    int x;
public:
    virtual void mf1() = 0;
    virtual void mf1(int);
    virtual void mf2();
    void mf3();
    void mf3(double);
    ...
};

class Derived: public Base {
public:
    virtual void mf1();
    void mf3();
    void mf4();
    ...
};

class Base {
private:
    int x;
public:
    virtual void mf1() = 0;
    virtual void mf1(int);
    virtual void mf2();
    void mf3();
    void mf3(double);
    ...
};

class Derived: public Base {
public:
    using Base::mf1; //让 Base class 内名为 mf1 和 mf3 的所有东西
    using Base::mf3; //在 Derived 作用域内都可见 (并且 public)
    virtual void mf1();
    void mf3();
    void mf4();
    ...
};

```

这段代码带来的行为会让每一位第一次面对它的 C++ 程序员大吃一惊。以作用域为基础的“名称遮掩规则”并没有改变，因此 base class 内所有名为 mf1 和 mf3 的函数都被 derived class 内的 mf1 和 mf3 函数遮掩掉了。从名称查找观点来看，Base::mf1 和 Base::mf3 不再被 Derived 继承！

```

Derived d;
int x;
...
d.mf1();           //没问题, 调用 Derived::mf1
d.mf1(x);         //错误! 因为 Derived::mf1 遮掩了 Base::mf1
d.mf2();           //没问题, 调用 Base::mf2
d.mf3();           //没问题, 调用 Derived::mf3
d.mf3(x);         //错误! 因为 Derived::mf3 遮掩了 Base::mf3

```

```

Derived d;
int x;
...
d.mf1();           //仍然没问题, 仍然调用 Derived::mf1
d.mf1(x);         //现在没问题了, 调用 Base::mf1
d.mf2();           //仍然没问题, 仍然调用 Base::mf2
d.mf3();           //没问题, 调用 Derived::mf3
d.mf3(x);         //现在没问题了, 调用 Base::mf3

```

这意味着如果你继承 base class 并加上重载函数，而你又希望重新定义或覆盖（推翻）其中一部分，那么你必须为那些原本会被遮掩的每个名称引入一个 using 声明式，否则某些你希望继承的名称会被遮掩。

有时候你并不想继承 base classes 的所有函数，这是可以理解的。在 public 继承下，这绝对不可能发生，因为它违反了 public 继承所暗示的“base 和 derived classes 之间的 is-a 关系”。（这也就是为什么上述 using 声明式被放在 derived class 的 public 区域的原因：base class 内的 public 名称在 publicly derived class 内也应该是 public。）然而在 private 继承之下（见条款 39）它却可能是有意义的。例如假设 Derived 以 private 形式继承 Base，而 Derived 唯一想继承的 mf1 是那个无参数版本。using 声明式在这里派不上用场，因为 using 声明式会令继承而来的某给定名称之所有同名函数在 derived class 中都可见。不，我们需要不同的技术，即一个简单的转交函数

```

int x;
public: Base class
    virtual void mf1() = 0;
    virtual void mf1(int);
    virtual void mf2();
    void mf3();
    void mf3(double);
    ...
};

class Derived: public Base {
public:
    virtual void mf1();
    void mf3();
    void mf4();
    ...
};

```

对于base class中的多个同名重载形式，如果派生类中重写或者重载了一个那么其他的同名函数会被隐藏

这段代码带来的行为会让每一位第一次面对它的 C++ 程序员大吃一惊。以作用域为基础的“名称遮掩规则”并没有改变，因此 base class 内所有名为 mf1 和 mf3 的函数都被 derived class 内的 mf1 和 mf3 函数遮掩掉了。从名称查找观点来看，Base::mf1 和 Base::mf3 不再被 Derived 继承！

```

Derived d;
int x;
...
d.mf1();      //仍然没问题，仍然调用 Derived::mf1
d.mf1(x);    //现在没问题了，调用 Base::mf1
d.mf2();      //仍然没问题，仍然调用 Base::mf2
d.mf3();      //没问题，调用 Derived::mf3
d.mf3(x);    //现在没问题了，调用 Base::mf3

```

这意味着如果你继承 base class 并加上重载函数，而你又希望重新定义或覆写（推翻）其中一部分，那么你必须为那些原本会被遮掩的每个名称引入一个 using 声明式，否则某些你希望继承的名称会被遮掩。也就是需要using 声明base class中的重载函数名

2.6.3 区分接口继承和实现继承

表面上直截了当的 public 继承概念，经过更严密的检查之后，发现它由两部分组成：函数接口（function interfaces）继承和函数实现（function implementations）继承。这两种继承的差异，很像本书导读所讨论的函数声明与函数定义之间的差异。

身为 class 设计者，有时候你会希望 derived classes 只继承成员函数的接口（也就是声明）；有时候你又会希望 derived classes 同时继承函数的接口和实现，但又希望能够覆写（override）它们所继承的实现；又有时候你希望 derived classes 同时继承函数的接口和实现，并且不允许覆写任何东西。

如果成员函数是一个 non-virtual 函数，意味着它并不打算在 derived classes 中有不同的行为。

声明一个非虚函数的目的是为了让派生类继承函数的接口及一份强制性实现，意味着在 derived class 中不应该被重新定义

pure virtual 函数、simple (impure) virtual 函数、non-virtual 函数之间的差异，使你得以精确指定你想要 derived classes 继承的东西：只继承接口，或是继承接口和一份缺省实现，或是继承接口和一份强制实现。由于这些不同类型的声明意味根本意义并不相同的事情，当你声明你的成员函数时，必须谨慎选择。如果你确实履行，应该能够避免经验不足的 class 设计者最常犯的两个错误。

第一个错误是将所有函数声明为 non-virtual。这使得 derived classes 没有余裕空间进行特化工作。non-virtual 析构函数尤其会带来问题（见条款 7）。当然啦，设计一个并不想成为 base class 的 class 是绝对合理的，既然这样，将其所有成员函数都声明为 non-virtual 也很适当。但这种声明如果不是忽略了 virtual 和 non-virtual 函数之间的差异，就是过度担心 virtual 函数的效率成本。实际上任何 class 如果打算被用来当做一个 base class，都会拥有若干 virtual 函数（再次见条款 7）。

另一个常见错误是将所有成员函数声明为 virtual。有时候这样做是正确的，例如条款 31 的 Interface classes。然而这也可能是 class 设计者缺乏坚定立场的前兆。某些函数就是不该在 derived class 中被重新定义，果真如此你应该将那些函数声明为 non-virtual。没有人有权利妄称你的 class 适用于任何人任何事任何物而他们只需花点时间重新定义你的函数就可以享受一切。如果你的不变性（*invariant*）凌驾特异性（*specialization*），别害怕说出来。

- 接口继承和实现继承不同。在 public 继承之下，derived classes 总是继承 base class 的接口。
- pure virtual 函数只具体指定接口继承。
- 简朴的（非纯）impure virtual 函数具体指定接口继承及缺省实现继承。
- non-virtual 函数具体指定接口继承以及强制性实现继承。

2.6.4 考虑virtual函数以外的其他选择

借由Non-Virtual Interface手法实现Template Method模式 (NVI手法)

通过public的non-virtual function调用一个private的virtual函数，实现相关功能

请记住

- virtual 函数的替代方案包括 NVI 手法及 **Strategy** 设计模式的多种形式。NVI 手法自身是一个特殊形式的 **Template Method** 设计模式。
- 将机能从成员函数移到 class 外部函数，带来的一个缺点是，非成员函数无法访问 class 的 non-public 成员。
- tr1::function 对象的行为就像一般函数指针。这样的对象可接纳“与给定之目标签名式（target signature）兼容”的所有可调用物（callable entities）。

2.6.5 绝不重新定义继承而来的non-virtual函数

non-virtual函数是静态绑定的，这意味着，即使父类指针指向的是一个子类对象，但是通过父类指针调用的永远是静态绑定的版本

virtual函数是动态绑定的，所以通过父类指针还是派生类指针都是调用的派生类的对象

2.6.6 绝不重新定义继承而来的缺省参数值

- 绝对不要重新定义一个继承而来的缺省参数值，因为缺省参数值都是静态绑定，而 virtual 函数——你唯一应该覆写的东西——却是动态绑定。

当你想令 virtual 函数表现出你所想要的行为但却遭遇麻烦，聪明的做法是考虑替代设计。条款 35 列了不少 virtual 函数的替代设计，其中之一是 NVI (*non-virtual interface*) 手法：令 base class 内的一个 public non-virtual 函数调用 private virtual 函数，后者可被 derived classes 重新定义。这里我们可以让 non-virtual 函数指定缺省参数，而 private virtual 函数负责真正的工作：

```
class Shape {  
public:  
    enum ShapeColor { Red, Green, Blue };  
    void draw(ShapeColor color = Red) const      //如今它是 non-virtual  
    {  
        doDraw(color);                          //调用一个 virtual  
    }  
    ...  
private:  
    virtual void doDraw(ShapeColor color) const = 0; //真正的工作  
};                                              //在此处完成  
  
class Rectangle: public Shape {  
public:  
    ...  
private:  
    virtual void doDraw(ShapeColor color) const; //注意，不须指定  
    ...  
};
```

由于 non-virtual 函数应该绝对不被 derived classes 覆写（见条款 36），这个设计很清楚地使得 draw 函数的 color 缺省参数值总是为 Red。

2.6.7 通过复合塑模出has-a或根据某物实现出

复合是类型之间的一种关系，当某种类型对象内含它种类型对象时，就是这样的关系

同义词：分层、内含、聚合、内嵌

2.6.8 明智而审慎地使用private继承

public继承意味着is-a的关系，对于base class有效的行为，对于derived class也应该有效

而private继承并不意味is-a关系。如果class之间的继承关系是private，编译器不会自动将一个derived class对象转换为base class对象

由private base class继承而来所有成员，在derived class中都会变成private属性，即使他们在base class中原本是protected或public属性

private继承意味着只有实现部分被继承，接口部分应略去（尽可能使用复合，必要时才使用private继承）

必要时候：当面对“并不存在is-a关系”的两个classes，其中一个需要访问另一个的protected成员，或需要重新定义其中一个或多个virtual函数时，private继承是个很好的选择方式

这是个好设计，但不值几文钱，因为 private 继承并非绝对必要。如果我们决定以复合（composition）取而代之，是可以的。只要在 Widget 内声明一个嵌套式 private class，后者以 public 形式继承 Timer 并重新定义 onTick，然后放一个这种类型的对象于 Widget 内。下面是这种解法的草样：

```
class Widget {  
private:  
    class WidgetTimer: public Timer {  
public:  
    virtual void onTick() const;  
    ...  
};  
    WidgetTimer timer;  
    ...  
};
```



- Private 继承意味 is-implemented-in-terms of（根据某物实现出）。它通常比复合（composition）的级别低。但是当 derived class 需要访问 protected base class 的成员，或需要重新定义继承而来的 virtual 函数时，这么设计是合理的。
- 和复合（composition）不同，private 继承可以造成 empty base 最优化。这对致力于“对象尺寸最小化”的程序库开发者而言，可能很重要。

2.6.9 明智而审慎地使用多重继承

多重继承的意思是继承一个以上的base classes，而当这些被继承的base classes又有更高级的相同的base class，就会产生 钻石型多重继承

```
1 //C++多重继承的语法  
2 class BorrowableItem{  
3 public:  
4     void checkout();  
5 };
```

```

6
7 class ElectronicGadget{
8 private:
9     bool checkout() const;
10 };
11
12 class MP3Player: public BorrowableItem, public
13 ElectronicGadget{
14 }; //多重继承
15
16 MP3Player mp;
17 mp.checkout(); //歧义，调用的是哪个checkout?
18 mp.BorrowableItem::checkout(); //正确

```

注意此例之中对 `checkout` 的调用是歧义（模棱两可）的，即使两个函数之中
只有一个可取用（`BorrowableItem` 内的 `checkout` 是 `public`，`ElectronicGadget`
 内的却是 `private`）。这与 C++ 用来解析（resolving）重载函数调用的规则相符：在
看到是否有个函数可取用之前，C++ 首先确认这个函数对此调用之言是最佳匹配。
找出最佳匹配函数后才检验其可取用性。本例的两个 `checkouts` 有相同的匹配程度
（译注：因此才造成歧义）没有所谓最佳匹配。因此 `ElectronicGadget::checkout`
的可取用性也就从未被编译器审查。

请记住

- 多重继承比单一继承复杂。它可能导致新的歧义性，以及对 `virtual` 继承的需要。
- `virtual` 继承会增加大小、速度、初始化（及赋值）复杂度等等成本。如果 `virtual`
`base classes` 不带任何数据，将是最具实用价值的情况。
- 多重继承的确有正当用途。其中一个情节涉及“public 继承某个 Interface class”
 和“private 继承某个协助实现的 class”的两相组合。
abstract class

2.7 模板与泛型编程 (Templates and Generic Programming)

2.7.1 了解隐式接口和编译期多态

面向对象编程世界总是以显式接口和运行期多态解决问题

请记住

class 面向对象编程

templates 泛型编程

显式接口

隐式接口

运行期多态

编译器多态

■ classes 和 templates 都支持接口 (interfaces) 和多态 (polymorphism)。

■ 对 classes 而言接口是显式的 (explicit)，以函数签名为中心。多态则是通过 virtual 函数发生于运行期。

■ 对 template 参数而言，接口是隐式的 (implicit)，奠基于有效表达式。多态则是通过 template 具现化和函数重载解析 (function overloading resolution) 发生于编译期。

2.7.2 了解 typename 的双重意义

在template声明式中，class和template意义完全相同

但是在有时候必须得使用typename，但不得在base class lists或member initialization list内以它作为base class修饰符

```
1 template<typename C>
2 void print2nd(const C& container)
3 {
4     if(container.size() >= 2)
5     {
6         C::const_iterator iter(container.begin());
7         ++iter;
8         cout << *iter;
9     }
10 }
```

1. template内出现的名称如果依赖于某个template参数，称为从属名称

2. 如果从属名称在class内呈嵌套状，称为嵌套从属名称

C::const_iterator (实际上还是一个嵌套从属类型名称)

3. 嵌套从属名称可能会导致解析困难，因为如果解析器在template中遭遇一个嵌套从属名称，它便假设这个名称不是个类型。即默认情况下，**嵌套从属名称不是类型**

一般性规则：任何时候当想要在template中指涉一个嵌套从属类型名称，就必须在紧邻它的前一个位置放上关键字 typename (有另一个例外)

```
template<typename C> //这是合法的 C++代码
void print2nd(const C& container)
{
    if (container.size() >= 2) {
        typename C::const_iterator iter(container.begin());
        ...
    }
}
```

typename 只被用来表明嵌套从属类型名称；其他名称不该有它存在。例如下面这个 function template，接受一个容器和一个“指向该容器”的迭代器：

```
template<typename C>
void f(const C & container,
       typename C::iterator iter); //允许使用 "typename" (或"class")
                                //不允许使用 "typename"
                                //一定要使用 "typename"
```

```
1 template<typename IterT>
2 void workwithIterator(IterT iter)
3 {
4     typename std::iterator_traits<IterT>::value_type
5     temp(*iter);
6     //对traits成员名称进行typedef
7     typedef typename
8         std::iterator_traits<IterT>::value_type value_type;
9     //typedef typename:指涉嵌套丛书类型名称
10 }
```

2.7.3 学习处理模板化基类内的名称

```
template<>
class MsgSender<CompanyZ> {           //一个全特化的
public:                                     //MsgSender; 它和一般 template 相同,
                                              //差别只在于它删掉了 sendClear。
...
void sendSecret(const MsgInfo& info)
{ ... }
};
```

注意 class 定义式最前头的 “template<>” 语法象征这既不是 template 也不是 标准 class，而是个特化版的 MsgSender template，在 template 实参是 CompanyZ 时被使用。这是所谓的模板全特化 (*total template specialization*)：template MsgSender 针对类型 CompanyZ 特化了，而且其特化是全面性的，也就是说一旦类型参数被定义为 CompanyZ，再没有其他 template 参数可供变化。

2.7.4 将与参数无关的代码抽离templates

- Templates 生成多个 classes 和多个函数，所以任何 template 代码都不该与某个造成膨胀的 template 参数产生相依关系。
- 因非类型模板参数 (non-type template parameters) 而造成的代码膨胀，往往可消除，做法是以函数参数或 class 成员变量替换 template 参数。
- 因类型参数 (type parameters) 而造成的代码膨胀，往往可降低，做法是让带有完全相同二进制表述 (binary representations) 的具现类型 (instantiation types) 共享实现码。

2.7.5 运用成员函数模板接受所有兼容类型

就原理而言，此例中我们需要的构造函数数量没有止尽，因为一个 template 可被无限量具现化，以致生成无限量函数。因此，似乎我们需要的不是为 SmartPtr 写一个构造函数，而是为它写一个构造模板。这样的模板（templates）是所谓 member function templates（常简称为 member templates），其作用是为 class 生成函数：

```
template<typename T>
class SmartPtr {
public:
    template<typename U>
    SmartPtr(const SmartPtr<U>& other); //member template,
    ... //为了生成 copy 构造函数
};
```

以上代码的意思是 对任何类型 T 和任何类型 U，这里可以根据 SmartPtr<U> 生成一个 SmartPtr<T> —— 因为 SmartPtr<T> 有个构造函数接受一个 SmartPtr<U> 参数。这一类构造函数根据对象 u 创建对象 t（例如根据 SmartPtr<U> 创建一个 SmartPtr<T>），而 u 和 v 的类型是同一个 template 的不同具现体，有时我们称之为泛化（generalized）copy 构造函数。

请记住

- 请使用 member function templates（成员函数模板）生成“可接受所有兼容类型”的函数。
- 如果你声明 member templates 用于“泛化 copy 构造”或“泛化 assignment 操作”，你还是需要声明正常的 copy 构造函数和 copy assignment 操作符。

不然编译器会生成一个默认的CTor

2.7.6 需要类型转换时请为模板定义非成员函数

template 在实参推导过程中，并不会将隐式转换函数纳入考虑

当我们编写一个 class template，而它所提供的与此 template 相关的函数需要支持 所有参数类型隐式转换时，将这些函数定义为 class template 内部的 friend 函数

因为定义于 class 内部的所有函数都是 inline，包括 friend 函数。因此，在必要的时候，可以将 inline 的 friend 函数作空实现，只是调用一个定义于 class 外部的辅助函数

条款 24 讨论过为什么惟有 non-member 函数才有能力“在所有实参身上实施隐式类型转换”，该条款并以 Rational class 的 operator* 函数为例。我强烈建议你继续看下去之前先让自己熟稔那个例子，因为本条款首先以一个看似无害的改动扩充条款 24 的讨论；本条款将 Rational 和 operator* 模板化了：

因为在隐式转换的时候，
this 占位问题

```

template<typename T>
class Rational {
public:
    Rational(const T& numerator = 0,
             const T& denominator = 1); // 条款 20 告诉你为什么参数以 passed by reference 方式传递。
    const T numerator() const; // 条款 28 告诉你为什么返回值 以 passed by value 方式传递。
    const T denominator() const; // 条款 3 告诉你为什么它们是 const。
    ...
};

template<typename T>
const Rational<T> operator* (const Rational<T>& lhs,
                             const Rational<T>& rhs)
{ ... }

```

只要利用一个事实，我们就可以缓和编译器在 template 实参推导方面受到的挑战：template class 内的 friend 声明式可以指涉某个特定函数。那意味 class Rational<T> 可以声明 operator* 是它的一个 friend 函数。 Class templates 并不倚赖 template 实参推导（后者只施行于 function templates 身上），所以编译器总是能够在 class Rational<T> 具现化时得知 T。因此，令 Rational<T> class 声明适当的 operator* 为其 friend 函数，可简化整个问题：

但是存在编译问题连接问题：在 template 中声明的函数，应该定义该函数。而不是在 class template 外部去寻找定义式

2.7.7 请使用 traits classes 表现类型信息

1. STL 的迭代器分类

```

struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag: public input_iterator_tag { };
struct bidirectional_iterator_tag: public forward_iterator_tag { };
struct random_access_iterator_tag: public bidirectional_iterator_tag { };

```

这些 structs 之间的继承关系是有效的 *is-a* 关系（见条款 32）：是的，所有 forward 迭代器都是 input 迭代器，依此类推。很快我们会看到这个继承关系的效力。

好，现在有了 `iterator_traits`（实际上是 `std::iterator_traits`，因为它 是 C++ 标准程序库的一部分），我们可以对 `advance` 实践先前的伪码（pseudocode）：

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
    if (typeid(typename std::iterator_traits<IterT>::iterator_category)
        == typeid(std::random_access_iterator_tag))
    ...
}
```

虽然这看起来前景光明，其实并非我们想要。首先它会导致编译问题，但我将在条款 48 才探讨这一点，此刻有更根本的问题要考虑。IterT 类型在编译期间获知，所以 `iterator_traits<IterT>::iterator_category` 也可在编译期间确定。但 `if` 语句却是在运行期才会核定。为什么将可在编译期完成的事延到运行期才做呢？这不仅浪费时间，也造成可执行文件膨胀。

我们真正想要的是一个条件式（也就是一个 `if...else` 语句）判断“编译期核定成功”之类型。恰巧 C++ 有一个取得这种行为的办法，那就是重载（overloading）。 

编译器便根据传来的实参选择最适当的重载件。编译器的态度是“如果这个重载件最匹配传递过来的实参，就调用这个 `f`；如果那个重载件最匹配，就调用那个 `f`；如果第三个 `f` 最匹配，就调用第三个 `f`！”依此类推。看到了吗，这正是一个针对类型而发生的“编译期条件句”，为了让 `advance` 的行为如我们所期望，我们需要做的是产生两版重载函数，内含 `advance` 的本质内容，但各自接受不同类型的 `iterator_category` 对象。我将这两个函数取名为 `doAdvance`：

```
template<typename IterT, typename DistT>          //这份实现用于
void doAdvance(IterT& iter, DistT d,                //random access
               std::random_access_iterator_tag) //迭代器
{
    iter += d;
}

template<typename IterT, typename DistT>          //这份实现用于
void doAdvance(IterT& iter, DistT d,                //bidirectional
               std::bidirectional_iterator_tag) //迭代器
{
    if (d >= 0) { while (d--) ++iter; }
    else { while (d++) --iter; }
}

template<typename IterT, typename DistT>          //这份实现用于
void doAdvance(IterT& iter, DistT d,                //input 迭代器
               std::input_iterator_tag)
{
    if (d < 0) {
        throw std::out_of_range("Negative distance"); //详下
    }
    while (d--) ++iter;
}
```

有了这些 doAdvance 重载版本，advance 需要做的只是调用它们并额外传递一个对象，**后者必须带有适当的迭代器分类**。于是编译器运用重载解析机制（overloading resolution）调用适当的实现代码：

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
    doAdvance(
        iter, d,
        typename
        std::iterator_traits<IterT>::iterator_category()
    );
}
```

//调用的 doAdvance 版本
//对 iter 之迭代器分类而言
//必须是适当的。

现在我们可以总结如何使用一个 traits class 了：

2.7.8 认识template元编程(template metaprogramming)

TMP的C++程序可能在每一方面都更高效：较小的可执行文件、较短的运行期、较少的内存需求。

但是将工作从运行期转移到编译器的另一个结果是，编译时间变长了

traits解法就是一种TMP

TMP 的起手程序是在**编译期计算阶乘 (factorial)**。这不是个令人特别兴奋的程序，但 "hello world" 程序也不是，而两者对于语言的导入都很有帮助。TMP 的阶乘运算示范如何通过“**递归模板具现化**”（recursive template instantiation）实现循环，以及如何在 TMP 中创建和使用变量：便捷系列的tuple实现就是如此

```
template<unsigned n>           //一般情况: Factorial<n> 的值是
struct Factorial {             // n 乘以 Factorial<n-1> 的值。
    enum { value = n * Factorial<n-1>::value };
};

template<>                     //特殊情况:
struct Factorial<0> {         //Factorial<0> 的值是 1
    enum { value = 1 };
};
```

全特化版本

enum hacks

2.8 定制new和delete

operator new 和 operator delete 只适合用来分配单一对象，arrays 所用的内存由 **operator new[]** 来分配并且由 **delete[]** 释放

2.8.1 了解new-handler的行为

当 `operator new` 抛出异常以反映一个未获满足的内存需求之前，它会先调用一个客户指定的错误处理函数，一个所谓的 *new-handler*。（这其实并非全部事实。`operator new` 真正做的事情稍微更复杂些。详见条款 51。）为了指定这个“用以处理内存不足”的函数，客户必须调用 `set_new_handler`，那是声明于 `<new>` 的一个标准程序库函数：

```
namespace std {  
    typedef void (*new_handler)();  
    new_handler set_new_handler(new_handler p) throw();  
}
```

如你所见，`new_handler` 是个 `typedef`，定义出一个指针指向函数，该函数没有参数也不返回任何东西。`set_new_handler` 则是“获得一个 `new_handler` 并返回一个 `new_handler`”的函数。`set_new_handler` 声明式尾端的 “`throw()`” 是一份异常明细，表示该函数不抛出任何异常——虽然事实更有趣些，详见条款 29。

```
1 //以下是operator new无法分配足够内存时，该被调用的函数  
2 void outOfMem()  
3 {  
4     std::cerr << "Unable to statisfy request for  
memory\n";  
5     std::abort();  
6 }  
7  
8 int main()  
9 {  
10    std::set_new_handler(outOfMem);  
11    int* pBigdataArray = new int[1000000000L];  
12 }
```

2.8.2 了解new和delete的合理替换时机

替换理由：

- 用来检测运用上的错误。如果将“new 所得内存”`delete` 掉却不幸失败，会导致内存泄漏（memory leaks）。如果在“new 所得内存”身上多次`delete` 则会导致不确定行为。如果 `operator new` 持有一串动态分配所得地址，而 `operator delete` 将地址从中移走，倒是很容易检测出上述错误用法。此外各式各样的编程错误可能导致数据 “overruns”（写入点在分配区块尾端之后）或 “underruns”（写入点在分配区块起点之前）。如果我们自行定义一个 `operator news`，便可超额分配内存，以额外空间（位于客户所得区块之前或后）放置特定的 byte patterns（即签名，signatures）。`operator deletes` 便得以检查上述签名是否原封不动，若否就表示在分配区的某个生命时间点发生了 overrun 或 underrun，这时候 `operator delete` 可以志记（log）那个事实以及那个惹是生非的指针。

为了强化效能，编译器所带的 operator new 和 operator delete 主要用于一般目的，它们不但可被长时间执行的程序（例如网页服务器，web servers）接受，也可被执行时间少于一秒的程序接受。它们必须处理一系列需求，包括**大块内存、小块内存、大小混合型内存**。它们必须接纳各种分配形态，范围从程序存活期间的少量区块动态分配，到大量短命对象的持续分配和归还。它们必须考虑破碎问题（fragmentation），这最终会导致程序无法满足大区块内存要求，即使彼时有总量足够但分散为许多小区块的自由内存。

2.8.2.1 内存对齐

为此协议而写，所以这儿我暂且忽略之。我现在只想专注于一个比较微妙的主题：齐位（alignment）。

许多计算机体系结构（computer architectures）要求特定的类型必须放在特定的内存地址上。例如它可能会要求指针的地址必须是 4 倍数（four-byte aligned）或 doubles 的地址必须是 8 倍数（eight-byte aligned）。如果没有奉行这个约束条件，可能导致运行期硬件异常。有些体系结构比较慈悲，没有那么霹雳，而是宣称如果齐位条件获得满足，便提供较佳效率。例如 Intel x86 体系结构上的 doubles 可被对齐于任何 byte 边界，但如果它是 8-byte 齐位，其访问速度会快许多。因为寄存器读取方便

```
//将 signature 写入内存的最前段落和最后段落.  
*(static_cast<int*>(pMem)) = signature;  
*(reinterpret_cast<int*>(static_cast<Byte*>(pMem)  
+realSize-sizeof(int))) = signature;  
  
//返回指针，指向恰位于第一个 signature 之后的内存位置.  
return static_cast<Byte*>(pMem) + sizeof(int);  
}
```

在我们目前这个主题中，齐位（alignment）意义重大，因为 C++ 要求所有 operator new 返回的指针都有适当的对齐（取决于数据类型）。malloc 就是在这样的要求下工作，所以令 operator new 返回一个得自 malloc 的指针是安全的。然而上述 operator new 中我并未返回一个得自 malloc 的指针，而是返回一个得自 malloc 且偏移一个 int 大小的指针。没人能够保证它的安全！如果客户端调用 operator new 企图获取足够给一个 double 所用的内存（或如果我们写个 operator new[]，元素类型是 doubles），而我们在一部“ints 为 4 bytes 且 doubles 必须 8-byte”

2.8.2.2 何时可在“全局性的”或“class 专属的”基础上合理替换缺省的 new 和 delete

- 为了检测运用错误（如前所述）。
- 为了收集动态分配内存之使用统计信息（如前所述）。

- 为了增加分配和归还的速度。泛用型分配器往往（虽然并不总是）比定制型分配器慢，特别是当定制型分配器专门针对某特定类型之对象而设计时。Class 专属分配器是“区块尺寸固定”之分配器实例，例如 Boost 提供的 Pool 程序库便是。如果你的程序是个单线程程序，但你的编译器所带的内存管理器具备线程安全，你或许可以写个不具线程安全的分配器而大幅改善速度。当然，在获得“operator new 和 operator delete 有加快程序速度的价值”这个结论之前，首先请分析你的程序，确认程序瓶颈的确发生在那些内存函数身上。
- 为了降低缺省内存管理器带来的空间额外开销。泛用型内存管理器往往（虽然并非总是）不只比定制型慢^①，它们往往还使用更多内存，那是因为它们常常在每一个分配区块身上招引某些额外开销。针对小型对象而开发的分配器（例如 Boost 的 Pool 程序库）本质上消除了这样的额外开销。
- 为了弥补缺省分配器中的非最佳齐位（suboptimal alignment）。一如先前所说，在 x86 体系结构上 doubles 的访问最是快速——如果它们都是 8-byte 齐位。但是编译器自带的 operator news 并不保证对动态分配而得的 doubles 采取 8-byte 齐位。这种情况下，将缺省的 operator new 替换为一个 8-byte 齐位保证版，可导致程序效率大幅提升。~~②~~
- 为了将相关对象成簇集中。如果你知道特定之某个数据结构往往被一起使用，而你又希望在处理这些数据时将“内存页错误”（page faults）的频率降至最低，那么为此数据结构创建另一个 heap 就有意义，这么一来它们就可以被成簇集中在尽可能少的内存页（pages）上。new 和 delete 的“placement 版本”（见条款 52）有可能完成这样的集簇行为。

2.8.3 编写new和delete时需固守常规

让我们从 operator new 开始。实现一致性 operator new 必得返回正确的值，内存不足时必得调用 new-handling 函数（见条款 49），必须有对付零内存需求的准备，还需避免不慎掩盖正常形式的 new —— 虽然这比较偏近 class 的接口要求而非实现要求。正常形式的 new 描述于条款 52。

operator new 的返回值十分单纯。如果它有能力供应客户申请的内存，就返回一个指针指向那块内存。如果没有那个能力，就遵循条款 49 描述的规则，并抛出一个 bad_alloc 异常。

然而其实也不是非常单纯，因为 operator new 实际上不只一次尝试分配内存，并在每次失败后调用 new-handling 函数。这里假设 new-handling 函数也许能够做某些动作将某些内存释放出来。只有当指向 new-handling 函数的指针是 null，operator new 才会抛出异常。

```
1 //一个non-member operator new伪码
2 void* operator new(std::size_t size)
3     throw(std::bad_alloc)
```

```

3  {
4      using namespace std;
5      if(size == 0) size = 1; //C++规定即使客户要求0bytes，也会返回一个合法指针
6      while(true)
7      {
8          尝试分配size bytes;
9          if(分配成功) return (一个指针：指向分配得来的内存);
10
11         //分配失败：找出目前的new-handling函数
12         new_handler globalHandler = set_new_handler(0);
13         set_new_handler(globalHandler);
14
15         if(globalHandler) (*globalHandler)();
16         else throw std::bad_alloc();
17     }
18 }
```

条款 49 谈到 operator new 内含一个无穷循环，而上述伪码明白表明出这个循环；"while (true)" 就是那个无穷循环。退出此循环的唯一办法是：内存被成功分配或 new-handling 函数做了一件描述于条款 49 的事情：让更多内存可用、安装另一个 new-handler、卸除 new-handler、抛出 bad_alloc 异常（或其派生物），或是承认失败而直接 return。现在，对于 new-handler 为什么必须做出其中某些事你应该很清楚了。如果不那么做，operator new 内的 while 循环永远不会结束。

请记住

- operator new 应该内含一个无穷循环，并在其中尝试分配内存，如果它无法满足内存需求，就该调用 new-handler。它也应该有能力处理 0 bytes 申请。Class 专属版本则还应该处理“比正确大小更大的（错误）申请”。
- operator delete 应该在收到 null 指针时不做任何事。Class 专属版本则还应该处理“比正确大小更大的（错误）申请”。

2.8.3.1 定制array new即operator new[]的注意事项

如果你打算控制 class 专属之“arrays 内存分配行为”，那么你需要实现 operator new 的 array 兄弟版：operator new[]。这个函数通常被称为 “array new”，因为很难想出如何发音 “operator new[]”。如果你决定写个 operator new[]，记住，唯一需要做的一件事就是分配一块未加工内存（raw memory），因为你无法对 array 之内迄今尚未存在的元素对象做任何事情。实际上你甚至无法计算这个 array 将含有多少个元素对象。首先你不知道每个对象多大，毕竟 base class 的 operator new[] 有可能经由继承被调用，将内存分配给 “元素为 derived class 对象”的 array 使用，而你当然知道，derived class 对象通常比其 base class 对象大。



因此，你不能在 Base::operator new[] 内假设 array 的每个元素对象的大小是 sizeof(Base)，这也就意味你不能假设 array 的元素对象个数是 (bytes 申请数) / sizeof(Base)。此外，传递给 operator new[] 的 size_t 参数，其值有可能比 “将被填以对象”的内存数量更多，因为条款 16 说过，动态分配的 arrays 可能包含额外空间用来存放元素个数。

2.8.3.2 重载operator new的注意事项

- operator new 应该内含一个无穷循环，并在其中尝试分配内存，如果它无法满足内存需求，就该调用 new-handler。它也应该有能力处理 0 bytes 申请。Class 专属版本则还应该处理 “比正确大小更大的（错误）申请”。
- operator delete 应该在收到 null 指针对不做什么事。Class 专属版本则还应该处理 “比正确大小更大的（错误）申请”。

2.8.4 写了placement new 也要写placement delete

placement new 和 placement delete 的定义：

如果 operator new 接受的参数除了一定会有的那个 size_t 之外还有其他，这就可以被称为一个 placement new

如果 operator delete 接受额外参数，那么就称为 placement delete

要求：在提供 placement new 的时候，提供一个正常的 operator delete 用于构造期间无任何异常被抛出，和一个 placement delete 用于构造期间又异常被抛出 后者的额外参数必须和 operator new 一样

规则很简单：如果一个带额外参数的 `operator new` 没有“带相同额外参数”的对应版 `operator delete`，那么当 `new` 的内存分配动作需要取消并恢复旧观时就没有任何 `operator delete` 会被调用。因此，为了消弭稍早代码中的内存泄漏，**Widget 有必要声明一个 `placement delete`**，对应于那个有记忆功能（logging）的 `placement new`：

```
class Widget {           placement new
public:
    ...
    static void* operator new(std::size_t size, std::ostream& logStream)
        throw(std::bad_alloc);

    static void operator delete(void* pMemory) throw();
    static void operator delete(void* pMemory, std::ostream& logStream)
        throw();
    ...
};                         placement delete
```

这样改变之后，如果以下语句引发 `Widget` 构造函数抛出异常：

```
Widget* pw = new (std::cerr) Widget;      //一如以往，但这次不再泄漏
```

对应的 `placement delete` 会被自动调用，让 `Widget` 有机会确保不泄漏任何内存。

然而如果没有抛出异常（通常如此），而客户代码中有个对应的 `delete`，会发生什么事：

```
delete pw;           //调用正常的 operator delete
```

就如上一行注释所言，调用的是正常形式的 `operator delete`，而非其 `placement` 版本。`placement delete` 只有在“伴随 `placement new` 调用而触发的构造函数”出现异常时才会被调用。对着一个指针（例如上述的 `pw`）施行 `delete` 绝不会导致调用 `placement delete`。不，绝对不会。

2.8.4.1 声明在class中的new隐藏问题

附带一提，由于成员函数的名称会掩盖其外围作用域中的相同名称（见条款 33），你必须小心避免让 class 专属的 `news` 掩盖客户期望的其他 `news`（包括正常版本）。假设你有一个 base class，其中声明唯一一个 `placement operator new`，客户端会发现他们无法使用正常形式的 `new`：

在缺省情况下，C++ 在 global 作用域内提供以下形式的 `operator new`

如果在 class 内声明 `operator news`，他会遮掩 global 中的标准形式

完成上述所言的一个简单做法是：建立一个 base class，内含所有正常形式的 `new` 和 `delete`

凡是想以自定形式扩充标准形式的客户，可以通过 `public` 继承并使用 `using` 取得标准形式。

请记住

- 当你写一个 *placement* operator new，请确定也写出了对应的 *placement* operator delete。如果没有这样做，你的程序可能会发生隐微而时断时续的内存泄漏。
- 当你声明 *placement* new 和 *placement* delete，请确定不要无意识（非故意）地遮掩了它们的正常版本。

2.9 杂项讨论

争取让自己的程序无警告，或者在忽略每个警告的时候，知道警告的意义，发生原因

智能指针（smart pointers）`tr1::shared_ptr` 和 `tr1::weak_ptr`。前者的作用有如内置指针，但会记录有多少个 `tr1::shared_ptrs` 共同指向同一个对象。这便是所谓的 *reference counting*（引用计数）。一旦最后一个这样的指针被销毁，也就是一旦某对象的引用次数变成 0，这个对象会被自动删除。这在非环形（acyclic）数据结构中防止资源泄漏很有帮助，但如果两个或多个对象内含 `tr1::shared_ptrs` 并形成环状（cycle），这个环形会造成每个对象的引用次数都超过 0——即使指向这个环形的所有指针都已被销毁（也就是这一群对象整体看来已无法触及）。这就是为什么又有 `tr1::weak_ptr` 的原因。`tr1::weak_ptr` 的设计使其表现像是“非环形 `tr1::shared_ptr`-based 数据结构”中的环形感生指针（cycle-inducing pointers）。`tr1::weak_ptr` 并不参与引用计数的计算；当最后一个指向某对象的 `tr1::shared_ptr` 被销毁，纵使还有个 `tr1::weak_ptr` 继续指向同一对象，该对象仍旧会被删除。这种情况下的 `tr1::weak_ptr` 会被自动标示无效。

3. 自己的理解

3.1 关于宏定义 `#define`

1. 无法利用宏定义创建一个 class 专属常量，因为 `#defines` 并不重视作用域
2. 一旦宏被定义，在其后的编译过程中都有效（除非被 `#undefine`）
3. `#defines` 不仅不能够用来定义 class 专属常量，也不能够提供任何封装性

3.2 关于 `const` 关键字

3.3 构造函数与析构函数

1. 尽量将构造函数写为参数列表初始化

- 尽量在初始化列表中列出所有的成员变量
- 声明次序为罗列次序

```
1 //拷贝构造
2 ABEntry::ABEntry (const std::string& name, const
3 std::string& address)
4 :theName(name), theAddress(address)
5 {} //这些也都是初始化操作，而非赋值操作
6
7 ABEntry::ABEntry (const std::string& name, const
8 std::string& address)
9 {
10     theName = name;
11     theAddress = address;
12 } //非赋值操作
```

2. default构造函数的写法

```
1 //default构造函数
2 ABEntry::ABEntry () :theName(), theAddress()
3 {} //这些也都是初始化操作，而非赋值操作
4
```

3.3.1 成员初始化次序

1. 基类
2. 派生类
3. 成员变量总是以其声明的次序被初始化

3.3.2 析构函数的运作方式

析构函数的运作方式是：

1. 最深层派生类的析构函数最先被调用，
2. 其次是每一个基类的析构函数。

编译器会在抽象父类的派生类的析构函数中创建一个对抽象父类的析构函数的调用，因此必须为这个函数提供一份定义

3. class析构函数（无论是编译器生成的还是用户自定的）会自动调用其non-static成员变量的析构函数

3.4 static关键字

3.4.1 static对象

static对象寿命从构造出来到程序结束为止

- 全局的static对象
- namespace作用域内的对象
- classes, func, file作用域内的static对象

函数体内的static对象成为 local static 对象。其他的成为 non-local static 对象 程序结束时会被自动销毁，即 在main()函数结束的时候自动调用析构函数

但是c++对于定义在不同的编译单元内的non-local static对象的初始化次序没有明确的定义

解决这一问题：将non-local转为local static，即将每个non-local static对象搬到自己的专属函数内，然后在该函数内声明该对象为 static,再返回一个 reference指向它所含的对象

```
1 //解决跨编译单元之non-local static的初始化次序问题
2 class FileSystem
3 {
4     std::size_t numDisks() const;
5 };
6 FileSystem& tfs()
7 {
8     static FileSystem fs;
9     return fs;
10}
11
12 class Directory {
13     std::size_t disks = tfs().numDisks(); //改动
14 };
15
16 Directory& tempDir()
17 {
18     static Directory td;
19     return td;
20 }
```

3.5 友元friend

3.5.1 friend class

A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class. For example, a `LinkedList` class may be allowed to access private members of `Node`.

```
1 class Node {  
2     private:  
3         int key;  
4         Node* next;  
5         /* Other members of Node Class */  
6     // Now class LinkedList can  
7     // access private members of Node  
8     friend class LinkedList;  
9 }
```

3.5.2 friend function

Friend Function Like friend class, a friend function can be given a special grant to access private and protected members. A friend function can be:

- a) A member of another class
- b) A global function

```
1 class Node {  
2     private:  
3         int key;  
4         Node* next;  
5         /* Other members of Node Class */  
6     friend int LinkedList::search();  
7         // only search() of linkedList  
8         // can access internal members  
9 }
```

Following are some important points about friend functions and classes:

- 1) Friends should be used only for limited purpose. **too many functions or external classes are declared as friends of a class with protected or private data**, it lessens the value of encapsulation of separate classes in object-oriented programming.
- 2) **Friendship is not mutual.** If class A is a friend of B, then B doesn't become a friend of A automatically.
- 3) Friendship is **not inherited** (See [this](#) for more details)

4) The concept of friends is not there in Java.

A simple and complete C++ program to demonstrate friend Class

3.6 虚函数virtual

任何class只要带有virtual都几乎确定应该有一个virtual析构函数。当一个class不包含virtual函数的时候，通常表明它太会被用来充当基类

心得：只有当class至少有一个virtual函数的时候，才为他声明virtual析构函数

3.6.1 为什么不要将析构函数声明为virtual析构函数

如果该class不准备被用来充当base class，那么就不要将其成员函数包括析构函数声明为virtual

这是因为，c++在实现虚函数机制的过程中，对象必须携带虚函数表指针，指向一个由函数指针构成的数组

当对象调用某一个虚函数的时候，实际上取决于vptr指向的虚函数指针

最重要的是，会增加对象的大小

3.6.2 纯虚函数

带有纯虚函数的class是一个抽象类，不能实例化，即不能为之创建对象。并且一个抽象类通常被用来充当父类，而父类又要求有一个virtual析构函数。因此

将希望成为抽象类的析构函数声明为纯虚析构函数

```
1 class MOV{  
2 public:  
3     virtual ~MOV() = 0; //声明为纯虚析构函数  
4 };  
5 //并为纯虚析构函数提供一份定义  
6 MOV::~MOV()  
7 {  
8  
9 }
```

Shape 是个抽象 class；它的 pure virtual 函数 draw 使它成为一个抽象 class。所以客户不能够创建 Shape class 的实体，只能创建其 derived classes 的实体。尽管如此，Shape 还是强烈影响了所有以 public 形式继承它的 derived classes，因为：

- 成员函数的接口总是会被继承。一如条款 32 所说，public 继承意味 is-a（是一种），所以对 base class 为真的任何事情一定也对其 derived classes 为真。因此如果某个函数可施行于某 class 身上，一定也可施行于其 derived classes 身上。

Shape class 声明了三个函数。第一个是 draw，于某个隐喻的视屏中画出当前对象。第二个是 error，准备让那些“需要报导某个错误”的成员函数调用。第三个是 objectID，返回当前对象的一个独一无二的整数识别码。每个函数的声明方式都不相同：draw 是个 pure virtual 函数；error 是个简朴的（非纯）impure virtual 函数；objectID 是个 non-virtual 函数。这些不同的声明带来什么样的暗示呢？

声明非纯虚函数的目的：让 derived classes 继承该函数的接口和缺省实现

但是在继承虚函数实现的时候，如果在 base class 中已经对该虚函数进行重载，那么在 derived class 中选择部分重写后，需要通过 using 声明来默认继承其他的实现

3.6.2.1 纯虚函数的两个特性

1. 纯虚函数要求任何 继承了它们 的具象 class 重新实现
2. 而且他们在抽象 class 中通常没有定义
3. 声明一个纯虚函数的目的是为了让派生类只继承函数接口

我们仍然可以为 pure virtual 函数提供定义，但是在调用的过程中，必须明确指出其 class 名称

```
Shape* ps = new Shape;           // 错误！Shape 是抽象的
Shape* ps1 = new Rectangle;       // 没问题
ps1->draw();                   // 调用 Rectangle::draw
Shape* ps2 = new Ellipse;         // 没问题
ps2->draw();                   // 调用 Ellipse::draw
ps1->Shape::draw();             // 调用 Shape::draw
ps2->Shape::draw();             // 调用 Shape::draw
```

除了能够帮助你在鸡尾酒派对上留给大师级程序员一个深刻的印象，一般而言这项性质 用途有限。但是一如稍后你将看到，它可以实现一种机制，为简朴的（非纯）impure virtual 函数提供更平常更安全的缺省实现。

3.7 多态的理解

有许多种做法可以记录时间，因此，设计一个 TimeKeeper base class 和一些 derived classes 作为不同的计时方法，相当合情合理：

```
class TimeKeeper {  
public:  
    TimeKeeper();  
    ~TimeKeeper();  
    ...  
};  
class AtomicClock: public TimeKeeper { ... }; //原子钟  
class WaterClock: public TimeKeeper { ... }; //水钟  
class WristWatch: public TimeKeeper { ... }; //腕表
```

许多客户只想在程序中使用时间，不想操心时间如何计算等细节，这时候我们可以设计 factory (工厂) 函数，返回指针指向一个计时对象。Factory 函数会“返回一个 base class 指针，指向新生成之 derived class 对象”：

```
TimeKeeper* getTimeKeeper(); //返回一个指针，指向一个  
                           //TimeKeeper 派生类的动态分配对象
```

多态：通过base class接口处理derived class对象

```
class Base{  
public:  
    virtual void printInfo()  
    {  
        cout << "I am your father" << endl;  
    }  
};  
  
class Son: public Base{  
public:  
    void printInfo(){  
        cout << "I am a son of Base" << endl;  
    }  
};  
  
void test01()  
{  
    Base* bptr = new Base(); // "I am your father"  
    Base* bptr = new Son(); // "I am a son of Base"  
    bptr->printInfo();  
}
```

3.7.1 多态的理解2

如上所述，当父类的成员函数被声明为virtual的时候，多态特性就已经建立。在此后，无论是继承还是继承的继承类也好。只要通过父类指针指向派生类的对象，都会调用覆盖的版本

但是，如果在对于没有声明为virtual的成员函数，多态特性仍然不成立

```

void test01()
{
    Base* bptr = new Base(); // "I am your father"
    bptr->printInfo();
    bptr = new Son(); // "I am a son of Base"
    bptr->testSon(); // I am a grandfather
    bptr->printInfo();
    bptr = new GrandSon();
    bptr->printInfo(); // "I am a grandSon of Base"

    Son* ptrs = new Son();
    ptrs->testSon(); // "I am a father of my son"
    ptrs = new GrandSon();
    ptrs->testSon(); // "I am a father of my son" (添加virtual前)
    // "I am a son of my father" (添加virtual后)
}

```

在Base中，printInfo()声明为virtual，而testSon () 没有声明为virtual
在Son中，无论printInfo是否声明为virtual，父类指针都能成功调用Son重写的printInfo
在GrandSon中，因为Son的testSon为virtual，所以具有多态，但是bptr无法调用Son的testSon

3.7.2 多态的理解3

在父类中声明为virtual的成员函数，如果在派生类中有对其进行重写，那么父类指针就会调用该重写版本的函数。而如果没有重写该版本，则默认调用父类的virtual版本成员函数

3.8 tr1::shared_ptr

核算其使用成本。最常见的 `tr1::shared_ptr` 实现品来自 Boost（见条款 55）。Boost 的 `shared_ptr` 是原始指针（raw pointer）的两倍大，以动态分配内存作为簿记用途和“删除器之专属数据”，以 `virtual` 形式调用删除器，并在多线程程序修改引用次数时蒙受线程同步化（thread synchronization）的额外开销。（只要定义一个预处理器符号就可以关闭多线程支持）。总之，它比原始指针大且慢，而且使用辅助动态内存。在许多应用程序中这些额外的执行成本并不显著，然而其“降低客户错误”的成效却是每个人都看得到。

3.9 定义式和声明式

所谓声明式（*declaration*）是告诉编译器某个东西的名称和类型（*type*），但略去细节。下面都是声明式：

<code>extern int x;</code>	//对象（object）声明式
<code>std::size_t numDigits(int number);</code>	//函数（function）声明式
<code>class Widget;</code>	//类（class）声明式
<code>template<typename T></code>	//模板（template）声明式
<code>class GraphNode;</code>	//"typename" 的使用见条款 42

定义式 (*definition*) 的任务是提供编译器一些声明式所遗漏的细节。对对象而言，定义式是编译器为此对象拨发内存的地点。对 function 或 function template 而言，定义式提供了代码本体。对 class 或 class template 而言，定义式列出它们的成员。

```
int x;                                //对象的定义式
std::size_t numDigits(int number)        //函数的定义式
{
    std::size_t digitsSoFar = 1;          //此函数返回其参数的数字个数,
                                         //例如十位数返回 2, 百位数返回 3.

    while ((number /= 10) != 0) ++digitsSoFar;
    return digitsSoFar;
}

class Widget {                          //class 的定义式
public:
    Widget();
    ~Widget();
    ...
}
```

3.10 empty class

这个激进情况真是有够激进，只适用于你所处理的 class 不带任何数据时。这样的 classes 没有 non-static 成员变量，没有 virtual 函数（因为这种函数的存在会为每个对象带来一个 vptr，见条款 7），也没有 virtual base classes（因为这样的 base classes 也会招致 体积上的额外开销，见条款 40）。于是这种所谓的 empty classes 对象不使用任何空间，因为没有任何隶属对象的数据需要存储。然而由于技术上的理由，C++ 裁定凡是独立（非附属）对象都必须有非零大小。所以如果你这样做：

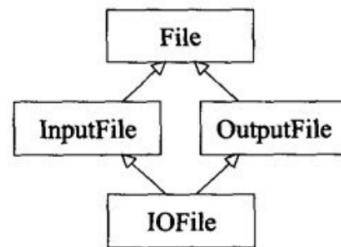
在大多数编译器中，`sizeof(empty)` 大小为 1，因为面对“大小为零的独立（非附属）对象”，通常 C++ 官方会勒令安插一个 char 到空对象内，

```
1 class Empty{
2
3 };
4
5 void TestEmpty()
6 {
7     Empty e;
8     std::cout << "size of empty class:" << sizeof(e) <<
9     std::endl;
10    //size of empty class:1
11 }
```

3.11 钻石型多重继承（菱形继承）

多重继承的意思是继承一个以上的 base classes，但这些 base classes 并不常在继承体系中又有更高级的 base classes，因为那会导致要命的“钻石型多重继承”：

```
class File { ... };
class InputFile: public File { ... };
class OutputFile: public File { ... };
class IOFile: public InputFile,
    public OutputFile
{ ... };
```



思考问题：

- 是否打算让base class内的成员变量经由每一条路径被复制？那么每个对象应该有两份fileName成员变量（**缺省做法**）
- IOFile对象只该有一个文件名称，所以它继承自两个base classes而来的fileName不该重复。如果要实现这一操作，应该要令那个带有此数据的 class (File) 成为一个virtual base class，并且令所有直接继承自它的 classes采用virtual继承

```
1 class File{...};
2 class InputFile: virtual public File {...};
3 class OutputFile: virtual public File {...};
4 class IOFile: public InputFile, public OutputFile {};
```

从正确行为的观点看，public 继承应该总是 virtual。如果这是唯一一个观点，规则很简单：任何时候当你使用 public 继承，请改用 virtual public 继承。但是，啊呀，**正确性并不是唯一观点**。为避免继承得来的成员变量重复，编译器必须提供若干幕后戏法，而其后果是：使用 virtual 继承的那些 classes 所产生的对象往往比使用 non-virtual 继承的兄弟们体积大，访问 virtual base classes 的成员变量时，也比访问 non-virtual base classes 的成员变量速度慢。种种细节因编译器不同而异，但基本重点很清楚：你得为 virtual 继承付出代价。

请记住

- 多重继承比单一继承复杂。它可能导致新的歧义性，以及对 virtual 继承的需要。
- virtual 继承会增加大小、速度、初始化（及赋值）复杂度等等成本。如果 virtual base classes 不带任何数据，将是最具实用价值的情况。**
- 多重继承的确有正当用途。其中一个情节涉及“public 继承某个 Interface class”和“private 继承某个协助实现的 class”的两相组合。
abstract class

3.12 面向对象编程vs泛型编程

面向对象编程世界总是以**显示接口和运行期多态解决问题**

而Templates及泛型编程的世界，与面向对象有根本的不同，在此世界中显示接口和运行期多态仍然存在，但是重要性降低。反倒是**隐式接口和编译器多态**占据主要地位

```
1 //面向对象编程
2 class wwidget{
3 public:
4     widget();
5     virtual ~widget();
6     virtual std::size_t size() const;
7     virtual void normalize();
8     virtual swap(widget& other);
9 };
10
11 void doProcessing(widget& w)
12 {
13     if(w.size() > 10 && w != someNastywidget){
14         widget temp(w);
15         temp.normalize();
16         temp.swap(w);
17     }
18 }
19 /*
20 由于w的类型被声明为widget，所以w必须支持widget接口，这些接口称为
21 显示接口
22 在源码中明确可见
23 其次：由于widget的部分成员函数是virtual，w对于那些函数的调用将表
24 现出运行期多态
25 */
26
27 //Templates及泛型编程
28 template<typename T>
29 void doProcessing(T& w)
30 {
31     if(w.size() > 10 && w != someNastywidget)
32     {
33         T temp(w);
34         temp.normalize();
35         temp.swap(w);
36     }
37 }
```

```
36 /*  
37 w必须支持哪一种接口，是有template中执行于w身上的操作来决定的。  
38 本例中，类型T好像必须支持size, normalize和swap成员函数，copy  
ctor, 以及operator!=  
39 这一组表达式就是T必须支持的隐式接口  
40 凡涉及w的任何函数调用，比如operator>和operator!=有可能造成  
template具现化行为发生在编译器  
41 “以不同的template参数具现化function templates”会调用不同的函  
数，这便是所谓的编译器多态  
42 */
```

3.12.1 运行期多态和编译期多态

1. 运行期多态：由于base class的某些成员函数是virtual，因此handle对于那些函数的调用将表现出运行期多态（runtime polymorphism）也就是在运行期间根据w的动态类型决定究竟调用哪个函数
2. 编译器多态：“以不同的template参数具现化function templates”会调用不同的函数，这便是所谓的编译器多态

哪一个重载函数被调用（发生在编译器）和哪一个virtual函数该被绑定（发生在编译器）

3.12.2 隐式接口和显示接口

通常显式接口由函数的签名式（也就是函数名称、参数类型、返回类型）构成。

例如 Widget class:

```
class Widget {  
public:  
    Widget();  
    virtual ~Widget();  
    virtual std::size_t size() const;  
    virtual void normalize();  
    void swap(Widget& other);  
};
```

其 public 接口由一个构造函数、一个析构函数、函数 size, normalize, swap 及其参数类型、返回类型、常量性（constnesses）构成。当然也包括编译器产生的 copy 构造函数和 copy assignment 操作符（见条款 5）。另外也可以包括 typedefs，以及如果你大胆违反条款 22（令成员变量为 private）而出现的 public 成员变量。

隐式接口就完全不同了。它并不基于函数签名式，而是由有效表达式（valid expressions）组成。再次看看 doProcessing template 一开始的条件：

```
template<typename T>
void doProcessing( T& w)
{
    if (w.size() > 10 && w != someNastyWidget) {
        ...
    }
}
```

T (w 的类型) 的隐式接口看来好像有这些约束：

- 它必须提供一个名为 size 的成员函数，该函数返回一个整数值。
- 它必须支持一个 operator!= 函数，用来比较两个 T 对象。这里我们假设 someNastyWidget 的类型为 T。

加诸于 template 参数身上的 隐式接口，就像加诸于 class 对象身上的 显式接口一样真实，而且两者都在编译期完成检查。就像你无法以一种“与 class 提供之显式接口矛盾”的方式来使用对象（代码将通不过编译），你也无法在 template 中使用“不支持 template 所要求之隐式接口”的对象（代码一样通不过编译）。

3.13 c++的traits是什么？

Traits 并不是 C++ 关键字或一个预先定义好的构件；它们是一种技术，也是一个 C++ 程序员共同遵守的协议。这个技术的要求之一是，它对内置 (built-in) 类型和用户自定义(user-defined)类型的表现必须一样好。举个例子，如果上述 advance 收到的实参是一个指针（例如 const char*）和一个 int，上述 advance 仍然必须有效运作，那意味 traits 技术必须也能够施行于内置类型如指针身上。
在定义class的时候，添加一个iterator type的做法

“traits 必须能够施行于内置类型”意味“类型内的嵌套信息 (nesting information)”这种东西出局了，因为我们无法将信息嵌套于原始指针内。因此类型的 traits 信息必须位于类型自身之外。标准技术是把它放进一个 template 及其一或多个特化版本中。这样的 templates 在标准程序库中有若干个，其中针对迭代器者被命名为 iterator traits:

至于 iterator_traits，只是鹦鹉学舌般地响应 iterator class 的嵌套式 typedef:

```
//类型 IterT 的 iterator_category 其实就是用来表现“IterT 说它自己是什么”。
//关于 "typedef typename" 的运用，见条款 42。
template<typename IterT>
struct iterator_traits {
    typedef typename IterT::iterator_category iterator_category;
    ...
};
```

```
template<typename IterT>           //template, 用来处理
struct iterator_traits;             //迭代器分类的相关信息
```

如你所见，`iterator_traits` 是个 `struct`。是的，习惯上 `traits` 总是被实现为 `structs`，但它们却又往往被称为 `traits classes`。

`iterator_traits` 的运作方式是，针对每一个类型 `IterT`，在 `struct iterator_traits<IterT>` 内一定声明某个 `typedef` 名为 `iterator_category`。这个 `typedef` 用来确认 `IterT` 的迭代器分类。

`iterator_traits` 以两个部分实现上述所言。首先它要求每一个“用户自定义的迭代器类型”必须嵌套一个 `typedef`，名为 `iterator_category`，用来确认适当的卷标结构（tag struct）。例如 `deque` 的迭代器可随机访问，所以一个针对 `deque` 迭

对于指针（也是一种迭代器）的实现：偏特化版本

为了支持指针迭代器，`iterator_traits` 特别针对指针类型提供一个偏特化版本（*partial template specialization*）。由于指针的行径与 `random access` 迭代器类似，所以 `iterator_traits` 为指针指定的迭代器类型是：

```
template<typename IterT>           //template 偏特化
struct iterator_traits<IterT*>     //针对内置指针
{
    typedef random_access_iterator_tag iterator_category;
    ...
};
```

3.13.1 函数重载的特性

重载机制的应用：advance，根据迭代器类型做不同的工作

当重载某个函数f时，必须详细叙述各个重载件的参数类型，但调用f时，编译器便根据传来的实参选择最适当的重载件。

编译器态度是：如果这个重载件最匹配传递过来的实参，就调用这个f，如果那个重载件最匹配，就调用那个f。

这正是编译期间的条件语句