C++编程思想（第一卷）笔记

By Ken 2008-10-26

（一） 对象导言

- ✧ **Alan Kay 总结了 Smalltalk 五个特性，这些特性代表了纯面向对象程序设计的方法**
  1) 万物皆对象
  2) 程序就是一组对象，对象之间通过发送消息相互通知做什么
  3) 每个对象都有它自己的由其它对象构成的存储区
  4) 每个对象都有一个类型
  5) 一个特定类型的所有对象都能接受相同的消息

（二） 对象的创建和使用

- ✧ 解释器：将源代码转化成一系列动作（可能由多组机器指令组成）并立即执行这些动作
- ✧ 编译器：直接把源代码转化为汇编语言或机器指令。最终的结果是一个或多个机器的代码文件
- ✧ C++实行编译期间的静态类型的检查
- ✧ 对于带空参数的函数，C 和 C++中有很大的不同。在 C 语言中，申明：int func(); 表示“一个可带任意参数的函数”。这就妨碍了类型检查。而在 C++中它意味着“不带参数的函数”
- ✧ extern 关键字在 C/C++中表示：这只是个申明，它的定义在别的地方，它表示变量是在文件以外定义的，或在文件后面部分才定义
- ✧ C 的库函数在 C++中包含格式有两种：#include<stdio.h>和#include<cstdio>。新的包含格式和老的效果是不一样的：使用.h 的文件是老的、非模板化的版本；而没有.h 的文件是新的模板化版本。如果在同一个程序中混用这两种形式，可能会遇到某些问题
- ✧ 因为连接器按指定的顺序查找文件，所以用户使用与库函数同名的函数时，把带有这种函数的文件插到库文件名列表之前，就能使用自己定义的函数取代库函数。由于在找到库文件之前，连接器已用用户定义的函数来解释引用，因此被使用的是用户的函数而不是库函数。注意这可能是个 bug。
- ✧ cin 代表：console input；cout 代表 console output

（三） C++和 C

- ✧ 在 C 语言中**定义**函数，必须给参数命名。如：int func(int a);声明函数时，可以不给出参数的名字；在 C++中声明和定义都可以不给出参数的名字
- ✧ register 变量的限制：不能得到或计算 register 变量的地址。register 变量只能在一个块中申明（不可能有全局的或静态的 register 变量）。函数的形式参数也可以是 register 类型
- ✧ 函数之外定义的所有变量（在 C++中除了 const，C++中的 const 全局变量自动成为全局的 static 变量，所以此 const 变量只在本模块中可见。这也说明在 C++中，在模块中定义两个同名的全局 const 变量不会引起连接错误，而在 C 中连接错误必会发生）和函数定义默认为外部连接。可以使用关键字 static 特地强制它们具有内部

连接，也可以在定义时使用关键字 extern 显式指定标志符具有外部连接。在 C++中定义全局的 const 变量必须使用 extern 关键字，比如：extern const int cst = 100; 这样别的模块才能引用这个全局的 const 变量

- ✧ 函数调用时，自动（局部）变量只是临时存在于堆栈中。连接器不知道自动变量，所以这些变量没有连接
- ✧ 位运算的几个小例子：

```
void printBinary(const unsigned char val){
    for(int i=(int)sizeof(val)-1; i>=0; i--){
        if(val & (i<<i))    cout<<"1";
        else    cout<<"0";
    }
}
unsigned char rol(unsigned char val){
    int highbit;
    if(val & 0x80)   highbit = 1;
    else    highbit = 0;
    //left shift (bottom bit becomes 0)
    val    <<= 1;
    val |= highbit; //Rotate the high bit onto the bottom
    return val;
}
unsigned char ror(unsigned char val){
    int lowbit;
    if(val & 1) lowbit = 1;
    else  lowbit = 0;
    val >>= 1;
    val |= (lowbit<<7);
    return val;
}
```

- ✧ sizeof 是一个运算符，不是函数。如果把它应用于一个类型，必须要加括号，如果对一个变量使用它，可以不加括号；sizeof(int); sizeof a;
- ✧ enum 枚举数据类型是把名字和数字相联系的一种方式。在 C 和 C++中对 enum 类型变量的类型检查是不相同的：C 的枚举很简单，只是把整数值和名字联系起来，并不提供类型检查。对于一个 color 枚举类型的实例 a，在 C 中，可以写 a++，但在 C++中不能这样写。这是因为枚举的增量运算执行两种类型转换，其中一个在 C++中是合法的，另一个是不合法的。首先枚举的值隐式地从 color 强制转换成 int，然后递增该值，再把 int 强制转换回 color 类型。在 C++中，这是不允许的，因为 color 是一个独特的类型，并不等价于一个 int。在 C++中，union 有很相似的附加类型检查。
- ✧ '#'具有把变量和表达式转换成字符串的功能。如：#define PR(x) cout<<#x<<endl ;
- ✧ assert 调试宏；当完成调试时，通过在程序的#include<cassert>之前插入语句行：#define NDEBUG 或者在命令行中定义 ndebug，可以消除宏产生的代码

（四） 数据抽象

- ✧ 在 C 中，可以赋 void*给任何指针和赋任何指针给 void*；在 C++中，可以赋任何

指针给 void*，但不能赋 void*给任何其它类型指针

- ✧ 无数据成员的结构的大小非 0 的原因是：每个对象必须有唯一的内存地址
- ✧ 头文件的重要性：如果将所有的函数声明都放在一个头文件中，并且将这个头文件包含在使用这些函数和定义这些函数的任何文件中，就能确保在整个系统中申明的一致性。通过将这个头文件包含在定义文件中，还可以确保申明和定义匹配。这样可以减少编程错误。
- ✧ 我们应该把什么放在头文件中：1）基本原则是，“只限于申明”，即只限于对编译器的信息，不涉及通过生成代码或创建变量而分配存储的任何信息。这是因为头文件一般会包含在项目的多个翻译单元中，如果一个标识符在多于一处被分配内存，那么连接器就报告多次定义的错误（这就是 C++中的一次定义，多次申明的规则）。2）如果在头文件中定义了一个“文件静态”变量，那么在整个项目中会有该数据的多个实例，但连接器不会冲突。
- ✧ C 和 C++都允许重声明函数，只要两个声明匹配即可，但是两者都不允许重声明结构。在 C++中，这条规则特别重要，因为如果编译器允许重声明一个结构而且这两个声明不同，那么应该使用哪个声明呢？
- ✧ 不要在头文件中使用 using 指令

（五） 隐藏实现

- ✧ friend 必须在一个结构内声明，这一点很重要，因为编译器必须能读取这个结构的声明以理解这个数据类型的大小、行为等方面的所有规则。可以把一个全局函数声明为 friend，也可以把另一个结构中的成员函数甚至整个结构都声明为 friend。例如：

```
struct X;
struct Y{
    void f(X*);
};
struct X{
private:
    int i;
public:
    void init();
    friend void g(X*, int); //global friend
    friend void Y::f(X*);     //struct member friend
    friend struct Z;       //entire struct is a friend
};

struct Z{
private:
    int j;
public:
    void init();
};
```

- ✧ 嵌套友元：嵌套结构并不能自动获得访问 private 成员的权限。要获得访问私有成员的权限，必须遵守特定的规则：首先声明（而不定义）一个嵌套结构，然后声明它是全局范围使用的一个 friend，最后定义这个结构。结构的定义必须与 friend 声

明分开，否则编译器将不把它看做成员。例如：

```
struct Holder{
private:
    int a[100];
public:
    void init();
    struct Pointer; //this
    friend Pointer; //three
    struct Pointer{ //step
    private:
        Holder* h;
        int*  p;
    public:
        void init(Holder* h);
        void next();
        void previous();
    };
};
```

- ✧ 在一个特定的“访问块（被访问说明符限定的一组声明）内”，这些变量在内存中肯定是连续存放的，这和 C 语言中一样，然后这些“访问块”本身可以不按声明的顺序在对象中出现。**虽然编译器通常都是按访问块出现的顺序给它们分配内存，但并不是一定是这样的，因为特定机器的体系结构和操作系统环境可对 private 成员和 protected 成员提供明确的支持，将其放在特定的内存位置上。**C++的访问说明符并不限制这种长度。
- ✧ 访问说明符是结构的一部分，它们并不影响从这个结构创建对象。程序开始运行之前，所有的访问说明信息都消失了。访问说明符信息通常是在编译期间消失的。在程序运行期间，对象变成了一个存储区域，别无它物。事实上，只有编译器知道类成员的保护级别，与成员关联的这些访问控制信息并没有被传递给连接器，所有的访问保护检查都是由编译器来完成的，在运行期间并不再检查
- ✧ 如何隐藏实现：C++的访问控制符允许将实现部分和结构部分分开，但实现部分的隐藏是不完全的。因为即使客户程序员不能轻易地访问私有实现部分，但可以看到它，这提供了机会让程序员不顾一切地使用指针和类型转换来访问私有成员。还有一个问题就是造成一些不必要的重复编译。解决这两个问题，就有必要把一个编译好的实际结构放在实现文件中，而不是暴露在头文件中。
- ✧ 减少重复编译：在我们的编程环境中，当一个文件被修改，或它所依赖的头文件被修改时，需要重复编译该文件。这意味着程序员无论何时修改了一个类，无论修改的是公共的接口部分，还是私有成员的声明部分，他都必须再次编译包含该头文件的所有文件。这对于一个大型项目来说，有可能是不能接受的。解决这个问题的技术是使用**句柄类（handle class）或称“Cheshire cat”（其实就是 bridge 设计模式）。**有关实现的任何东西都消失了，只剩下一个单指针“smile”。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同出现在实现文件中。这样只要接口部分不改变，头文件就不需要变动了。而实现部分可以按需要任意更改，完成后只需要对实现文件进行重新编译，然后重新连接到项目中。

```
//Handle.h
```

```
#ifndef HANDLE_H
#define HANDLE_H
class Handle{
    struct Cheshire; //class declaration only
    Cheshire* smile;
public:
    Handle();
    ~Handle();
    int read();
    void change(int);
};
#endif //HANDLE_H

//Handle.cpp
#include"Handle.h"
struct Handle::Cheshire{
    int i;
};

Handle::Handle(){
    smile = new Cheshire;
    smile->i = 0;
}

Handle::~Handle(){
    delete smile;
}

int Handle::read(){
    return smile->i;
}

void Handle::change(int x){
    smile->i = x;
}
```

（六） 初始化与清除

- ✧ 构造函数和析构函数是两个非常特殊的函数：它们没有返回值。这与返回值为 void 的函数显然不同。后者虽然也不返回任何值，但可以让它做点别的事情，而构造函数和析构函数则不允许。在程序中创建和消除一个对象的行为非常特殊，就像出生和死亡，而且总是由编译器来调用这些函数以确保它们被执行。如果它们有返回值，那么编译器必须知道如何处理返回值，要么就只能由程序员自己来显示地调用构造函数和析构函数，这样一来，安全性就破坏了
- ✧ 析构函数的调用的惟一证据是包含该对象的右括号。即使使用 goto 语句跳出这一程序块，析构函数仍然被调用。应该注意非局部的 goto 语句(nonlocal goto)，它们

使用标准 C 语言库中的 setjmp 和 longjmp 函数实现的，这些非局部的 goto 语句将不会引发析构函数的调用。

- ✧ C99 和 C++一样，可以在某一块的任意地方定义变量。一般说来，应该在尽量可能靠近变量使用点处定义变量，并在定义时就初始化。这是出于安全性的考虑，通过减少变量在块中的生命周期，就可以减少该变量在块的其它地方被无用的机会，另外程序的可读性也增强了，因为读者不需要跳到开头去确定变量的类型。小作用域是良好设计的指标。
- ✧ 编译器会检查有没有把一个对象的定义（构造函数的调用）放在一个条件块中，比如 switch 块中声明，或可能被 goto 跳过的地方。下面有个典型的例子：

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

void fun(int i)
{
  if(i < 10){
      //goto jmp1;
  }
  X x1;
jmp1:
  switch(i){
      case 1:
          X x2;
          break;
      //case 2:
          X x3;
          break;
  }
}
```

注释掉的语句都有问题，它们分别跳过了 x1 的初始化和 x2 的初始化；在 switch 中可以给每个 case 加上{}来解决

- ✧ 在定义数组时，如果给的初始化值少于元素的个数，这时编译器把第一个初始化值赋给数组的第一个元素，然后用 0 赋给其余的元素。需要注意的是，如果定义了一个数组而没有给出一列初始值时，编译器并不会去做初始化工作。所以 int b[6]={0}；表达式是将一个数组初始化为 0 的简洁方法，可能效率很高。例如：

```
struct X{
    int i;
    float f;
    char c;
};
X x2[3] = {{1, 1.1, 'a'},{2, 2.2, 'b'}};
```

这里第三个对象 x2[2]被初始化为 0，即 x2[2].i = 0; x2[2].f = 0.0; x2[2].c = 0;

（七） 函数重载与默认参数

✧ 在 C++中，所有的函数使用之前都必须事先声明；在 C 中，如果用户错误地声明了一个函数，或者一个函数没有声明就调用了，而编译器按函数被调用的方式去推断函数的声明。这有可能造成难以发现的 bug。看如下一个例子：

```
//Def.cpp
void f(int){}

//Use.cpp
//function misdeclaration
void f(char);

int main(){
    f(1); //causes a link error
}
```

在 C 中，编译和链接都成功了。但在 C++中却不行。因为编译器会修饰这些名字，把它变成 f_int()之类的名字，而使用的函数则是 f_char。当连接器试图引用 f_char 时，它只能找到 f_int，所以它就会报告一条连接出错信息。这就是类型安全连接。这是也是利用 C++编译器查找 C 语言程序中很隐蔽的错误的例子

✧ union 类型也可以带有构造函数、析构函数、成员函数甚至访问控制。但 union 不能作为基类被继承。下面有个关于 union 的典型例子：

```
class SuperVar{
    enum{
        character,
        integer,
        floating;
    }vartype; //define one
    union{//anonymous union
        char c;
        int i;
        float f;
    };
public:
    SuperVar(char ch){vartype = character; c = ch;}
    SuperVar(int ii){vartype = integer; i = ii;}
    SuperVar(float ff){vartype = floating; f = ff;}
    void print();
};

void SuperVar::print(){
    switch(vartype){
        case character:
            break;
        case integer:
            break;
        case floating:
```

```
                    break;
            }
        }
```

在这个例子中，union 没有类型名和标志符。这就做匿名联合，为这个 union 创建空间，但并不需要用标志符的方式和以点操作符（‘.’）方式访问这个 union 的元素。再看一个匿名 union 的例子：

```
int main(){
    union{
        int i;
        float f;
    };
    i = 12; //acess members without using qualifiers
    f = 1.22;
}
```

注意：我们访问一个匿名联合变量就像访问普通的变量一样。惟一的区别在于：该联合的两个变量占用同一空间。如果匿名联合变量在文件作用域（在所有函数和类之外），则它必须被声明为 static，以使它有内部连接。

- ✧ 在使用默认参数时必须记住两条规则：1）只有参数列表的后部参数才是可默认的，也就是说，不可以在一个默认参数后面又跟一个非默认参数；2）一旦在一个函数调用中开始使用默认参数，那么这个参数后面的所有参数都必须是默认的。另外，默认参数只能放在函数声明中，通常在一个头文件中。编译器必须在使用该函数之前知道默认值。在函数定义处放上一些默认的注释值可以增加程序的阅读方便：void fun(int x /* = 0*/){// ...
- ✧ 占位符参数：在函数声明时，参数可以没有标志符，当这些不带标志符的参数用作默认参数时，可以这样声明：void f(int x, int = 0, float = 1.1) ;在 C++中，在函数定义时，并不一定需要标志符，如：void f(int x, int, float f) ;{/*...*/}
- ✧ 选择重载还是默认参数：不要把默认值当成一个标志，来使单个函数执行不同的分支，在这种情况下，我们应该选择重载，把函数分解成两个或多个重载函数。一个默认参数应该是一个在一般情况下放在这个位置的值。这个值出现的可能比其他值要大，所以程序员可以忽略它或只在需要改变默认值时才去用它。默认参数的一个重要应用情况是在开始定义函数时用了一组参数，而使用了一段时间后发现要增加一些参数。通过把这些新增的参数都作为默认的参数，就可以保证所有使用这一函数的客户代码不会受到影响
- ✧ 设计类时，最重要的问题是类的接口。如果类容易使用和重用，那说明成功了。要是有必要，总是可以为了效率而作适当的调整。但是，如果程序员过分强调效率的话，设计类的效果将是可怕的。总是应该关心接口清晰，使使用和阅读代码的人易于理解

（八） 常量

- ✧ #define BUFSIZE 100，这样定义的常量时，BUFSIZE 是一个名字，它只是在预处理期间存在，因此它不占用存储空间且能放在一个头文件里，目的是为使用它的所有编译单元提供一个值。它没有类型信息。C++中可以如此定义常量：const int bufsize = 100 ;这是一个编译器常量（内部数据类型），编译器还可以执行“常量迭代（constant folding）”，也就是说，编译器在编译时可以通过必要的计算把一个复杂的常量表达式通过缩减简单化。C++中应该总是使用 const 来定义常量，而不是

使用 define

- 把 const 变量放在头文件中，那么每个包含该头文件的实现文件都有该常量的一个私有拷贝。C++中的 const 默认为内部连接（internal linkage），也就是说，const 仅在 const 被定义过的文件里才是可见的，而在连接时不能被其它编译单元看到。当定义一个 const 时，必须赋一个值给它，除非用 extern 作出了清楚的说明：extern const int size ;
- 通常 C++编译器并不为 const 创建存储空间，相反它把这个定义保存在它的符号表里。但是 extern 关键字强制进行了存储空间分配（另外还有一些情况，如取一个 const 的地址或 const 对象比较复杂时，也要进行存储空间的分配），由于 extern 意味着使用外部连接，因此必须分配存储空间，这也就是说有几个不同的编译单元应当能够引用它，所以它必须有存储空间
- 来看一下 const 的几个例子：

```
//const int i = 100;
int main(int argc, char** argv){
    const char c = cin.get();
    const int i = 100;
    int* p = (int*)(&i);
    *p = 200 ;
    cout<<i<<" "<<*p<<endl;
    return 0;
}
```

在这个例子中 c 是一个运行期常量，所以在编译期间它的值是不可知的，这也意味着必须为 c 分配存储空间。i 是一个编译器常量。当 i 是在一个函数内部时（如上代码，i 在 main 函数内部，对 i 取了地址，说明了 i 在栈上分配了存储空间）输出的是 100 200。出现此现象的原因是编译器使用了常量迭代（虽然对 i 取了地址，分配了存储空间，但编译器还是选择了常量迭代）。当 i 在函数外部时（如上代码注释部分），也即 i 是个全局的常量时，出现运行期错误。这说明全局的常量 i 被存放在了一个只读数据存储区。这是在 VS2005 下做的测试

- const 可以用于集合，因为编译器不会复杂到把一个集合保存到它的符号表中，所以必须为之分配内存。在这种情况下，const 意味着“不能改变的一块存储空间”。然而，不能在编译期间使用它的值，因为编译器在编译期间不知道存储空间的内容。下面这个例子是个很好的说明：

```
const int i[] = {1,2, 3, 4};
struct S{int i, j;};
const S s[] = {{1,2}, {3,4}};
// double d[s[1].j]; //illegal
```

所以只有常量保存在符号表中，才能用这个常量去定义一个数组的大小

- const 在 C 和 C++中的区别：1）在 C 中如果这样写：const int size = 100; char buf[size]; 是错误的。因为 const 常量在 C 中总是分配存储空间的，所以 size 占用某块内存，所以 C 编译器不知道它的编译时的值。2）在 C 中可以这样写：const int size;这在 C++中是不对的，而 C 编译器则把它作为一个声明，指明在别的地方有存储分配，因为 C 默认 const 是外部连接的，所以这样做是合理的。C++默认 const 是内部连接的，如果在 C++中想完成与 C 中同样的事情，必须用 extern 明确地把连接改为外部连接：extern const int size; //declaration only

- 只要可能，一行只定义一个指针，并尽可能在定义时初始化
- 字符数组的字面值：对于 char* cp = “howdy”;字符数组的字面值（”howdy”）是被编译器作为一个常量字符数组建立的，存放在字符常量存储区，所引用该字符数组得到的结果是它在内存里的首地址。修改该字符数组中的任何一个字符都会导致运行期错误。(C 中并不会出现运行期错误)
- 临时变量自动成为常量。看一个典型例子：

```
class X{
    int i;
public:
    X(int ii=0){i = ii;}
    void modify(){i++;}
};

X f5(){
    return X();
}

const X f7(X& x){
    x.modify();
}
f7(f5());       //(1)
f5() = X(1);    //(2)
f5().modify(); //(3)
```

上述代码的(1)在遵循 C++标准的编译器中是不能编译通过的（vs2005 能够编译通过，gcc 不能编译通过），因为 f5()返回的临时变量自动成为一个常量对象，这和 f7 的参数比匹配。(2)、(3)语句是合法的，尽管它们可以编译通过，但实际上存在问题。f5()返回一个 X 对象，而且对编译器来说，要满足(2)(3)表达式，它必须创建临时对象来保存返回值。于是，在这两个表达式中，临时对象也被修改，表达式被编译过之后，临时对象也将被清除。结果丢失了所有的修改，从而代码可能存在问题。

- enum hack。在旧版 C++中，不支持类中使用 static const，解决这个问题的办法是使用不带实例的无标记 enum。一个枚举在编译期间必须有值，它在类中局部出现，而且它的值对于常量表达式是可以使用的。例如：

```
class Bunch{
    enum{size = 10};
    int i[size];
};
```

```
class T{
    enum{size = 10, size1};
};
```

```
enum E{
    size = 100,
    size1
};
```

sizeof(Bunch) = 10*sizeof(int) = 40;在 Bunch 类中 enum 是保证不占对象的存储空间的，编译期间得到枚举值。sizeof(T) = 1; sizeof(E) = 4;

- const 对象只能调用 const 成员函数，非 const 对象两个版本的成员函数都可以调用。
- 类中的 const 成员函数，在申明和定义时都要写上 const 说明，const 已成为函数标识符的一部分，所以编译器和连接器都要检查 const。
- 如果想建立一个 const 成员函数，但仍然想在对象里改变某些数据，这时该怎么办呢？这涉及到按位（bitwise）const 和按逻辑（logical）const 的区别。按位 const

的意思是对象中的每个字节都是固定的，所以对象的每个位映射从不改变。按逻辑 const 的意思是，虽然整个对象从概念上是不变的，但是可以以成员为单位改变。**当编译器被告知一个对象是 const 对象时，它将绝对保护这个对象按位的常量性**。要实现按逻辑 const 的属性，有两种由内部 const 成员函数改变数据成员的办法。1）“强制转除常量性（casting away constness）”。在 const 成员函数内部，this 指针实际上是一个 const 指针，可以把它强制转换成普通指针，然后改变成员变量。例如：

```
class Y{
    int i;                  // mutable int j;
public:
    Y(){i = 0;}
    void f()const;
};

void Y::f()const{
    //i++; //error              //j++;
    ((Y*)this)->i++; //ok:cast away constness
    (const_cast<Y*>(this))->i++;
}
```

上述这种方法已成为过去。2）声明变量为 mutable，以指定一个特定的数据成员可以在一个 const 对象里被改变

- ✧ volatile 的语法和 const 是一样的。volatile 的意思是“在编译器认识的范围外，这个数据可以改变”。和 const 一样，我们可以对数据成员、成员函数和对象本身使用 volatile，可以对 volatile 对象（只能）调用 volatile 成员函数。volatile 和 const 一起通常称为 c-v 限定词

（九） 内联函数

- ✧ 应该（几乎）永远不使用宏，只使用内联函数
- ✧ 任何在类中定义的函数自动成为内联函数，但也可以在非类的函数前面加上 inline 关键字使之成为内联函数。但为了使之有效，必须是函数体和声明结合在一起，否则编译器将它作为普通函数对待。因此：inline int plusOne(int x);没有任何效果成功的方法是：inline int plusOne(int x){return ++x;}。编译器要检查内联函数参数列表是否正确，并返回值（进行必要的转换）。这些事是预处理器无法完成的（类型检查和做必要的转换工作）。
- ✧ 一般应该把内联定义放在头文件中。当编译器看到这个定义时，它把函数类型（函数名+返回值）和函数体放到符号表里。当使用函数时，编译器检查以确保调用时正确的且返回值被正确使用，然后将函数调用替换为函数体，因而消除了函数调用的开销
- ✧ 在头文件中，内联函数处于一种特殊状态。在每个使用到内联函数的文件中，我们都必须包含该内联函数声明和定义的头文件，但是不会出现多个定义错误的情况（不过，在包含该内联函数的所有地方，该内联函数的定义都必须是相同的）
- ✧ 有两种情况，编译器不能执行内联。1）函数太复杂，比如包含了循环语句或许多条件语句；2）要显式地或隐式地取函数地址时，编译器不能执行内联，因为这时编译器必须为函数代码分配内存从而产生一个函数地址。但当地址不需要时，编译器仍然将可能内联代码。在这两种情况下，编译器就像对非内联函数一样，根据内联函数的定义为函数建立存储空间，简单地将其转换成函数的普通形式。假如它必

须在多个编译单元内做这些转换工作（通常这将会产生一个多重定义的错误），连接器就会被告知忽略此多重定义。

✧ 向前引用：看如下一个例子，

```
class Forward{
    int i;
public:
    Forwar():i(0){}
    //call to undeclared function
    int f()const{ return g()+1;}
    int g()const{return i;}
};
```

上面的代码中，函数 f()调用 g()，但此时还没有声明 g()。这也能正常工作，因为C++语言规定：只有在类声明结束后，其中的内联函数才会被计算

✧ 预处理的三个特殊的特征：字符串定义、字符串拼接和标志粘帖。字符串定义使用#指示。字符串拼接实在两个相邻的字符串没有分隔符时发生，在这种情况下字符串组合在一起。在写调试代码的时候，这两个特征很有用。

```
#define DEBUG(x) cout<<#x“ = ”<<x<<endl
```

标志粘帖直接用“##”实现，在写代码时非常有用的。它允许把两个标识符粘帖在一起自动产生一个新的标识符。例如：

```
#define FIELD(a) char* a##string; int a##_size
class Record{
    FIELD(one);
    FIELD(two);
    FIELD(three);
    // …
};
```

（十） 名字控制

✧ 在 C 和 C++中，static 都有两种基本含义

1) 在固定的地址上进行存储分配，也就是说对象是在一个特殊的静态数据区上创建的，而不是每次函数调用时在堆栈上产生的
2) 对一个特定的编译单元来说是局部的。这样，static 控制名字的可见性，所以这个名字在这个单元或类之外是不可见的

✧ 如果没有为一个内部类型的静态变量提供一个初始值的话，编译器也会确保在程序开始时把它初始化为零

✧ 静态对象的析构函数（包括静态存储的所有对象，不仅仅是局部静态对象）在程序从 main()中退出时，或标准的 C 库函数 exit()被调用时才被调用。这意味着在析构函数中使用 exit()是很危险的，因为这样导致了无穷的递归调用。如果用标准的 C 库函数 abort()退出程序，静态对象的析构函数并不会调用

✧ 同普通对象的销毁一样，静态对象的销毁也是按与初始化时相反的顺序进行的。编译器会记录对象初始化的顺序和那些已被创建的对象。全局对象总是在 main()执行之前被创建，在退出 main()时销毁。如果一个包含局部静态对象的函数从未调用过，那么这个对象的构造函数也就不会执行，这样自然也不会执行析构函数

✧ namespace 与 class、struct、union 和 enum 有明显的区别

1) namespace 只能在全局范围内定义，但它们之间可以互相嵌套

2) 在 namespace 定义的结尾，右花括号的后面不必跟一个分号

3) 一个 namespace 可以在多个头文件中用一个标志符来定义，就好象重复定义一个类一样

```
//Head1.h
#ifndef HEAD1_H
#define HEAD1_H
namespace MyLib{
    extern int x;
    void f();
    //...
}
#endif //HEAD1_H

//Head2.h
#ifndef HEAD2_H
#define HEAD2_H
//add more names to MyLib
namespace MyLib{    //not a redefinition
    extern int y;
    void g();
    //...
}
#endif //HEAD2_H
```

4) 一个 namespace 的名字可以用另外一个名字来作为它的别名，这样就不必敲打那些开发商提供的冗长的名字了

```
//BobSuperDuperLibrary.cpp
namespace BobSuperDuperLibrary{
    class Widget{/*...*/};
    class Poppit{/*...*/};
    //...
}
namespace Bob = BobSuperDuperLibrary;
int main(){}
```

5) 不能像类那样去创建一个名字空间的实例

✧ 未命名的名字空间：每个翻译单元都可以包含一个未命名的名字空间——可以不用标识符而只用“namespace”增加一个名字空间。在这个空间中的名字自动地在翻译单元内无限制有效。但要确保每个翻译单元只有一个未命名的名字空间。如果把一个局部名字放在一个未命名的名字空间中，不需要加 static 说明就可以让它们作内部连接

✧ 可以在一个名字空间的类定义中插入一个友元声明：

```
//FriendInjection.cpp
namespace Me{
    class Us{
        //...
```

```
        friend void you();
    };
}
```

这样函数 you()就成了名字空间 Me 的一个成员

✧ 可以采取三种方法来引用一个名字空间的名字：1）用作用域运算符；2）用 using 指令把所有名字引入到名字空间中；3）用 using 声明一次性引用名字。using 声明给出了标识符的完整的名字，但没有类型方面的信息。也就是说，如果名字空间中包含了一组用相同名字重载的函数，using 声明就声明了这个重载的集合内的所有函数

✧ 定义静态数据成员：例如，在一个类中定义一个静态数据成员如下：

```
class A{
    static int i;
public:
    //...
};
```

之后，必须在定义文件中为静态数据成员定义存储区：

```
int A::i = 1;
```

这里有一点需要注意，有些人对 A::i 是私有的这点有点疑惑不解，在这里似乎在公开地直接对它处理。这不是破坏了类结构的保护性吗？有两个原因可以保证它绝对安全。1）这些变量的初始化惟一合法的地方是在定义时；2）一旦这些数据被定义了，最终的用户就不能再定义它——否则连接器会报告错误。

✧ 静态成员的初始化表达式是在一个类的作用域内。下面是个典型的例子：

```
int x = 100;
class WithStatic{
    static int x;
    static int y;
public:
    void print()const{
        cout<<"WithStatic::x = " << x <<endl;
        cout<<"WithStatic::y = "<< y <<endl;
    }
};
int WithStatic::x = 1;
int WithStatic::y = x + 1;
//WithStatic::x NOT ::x

int main(){
    WithStatic ws;
    ws.print();
}
```

这段代码的输出是 WithStatic::x = 1；WithStatic::y = 2；这里，WithStatic::限定符把 WithStatic 的作用域扩展到全部定义中

✧ 静态数组的初始化。看下面一个实例：

```
//StaticArray.cpp
```

```
// Initializing static arrays in classes
class Values {
// static consts are initialized in-place:
static const int scSize = 100;
static const long scLong = 100;
// Automatic counting works with static arrays.
// Arrays, Non-integral and non-const statics
// must be initialized externally:
static const int scInts[];
static const long scLongs[];
static const float scTable[];
static const char scLetters[];
static int size;
static const float scFloat;
static float table[];
static char letters[];
};

int Values::size = 100;
const float Values::scFloat = 1.1;
const int Values::scInts[] = {
    99, 47, 33, 11, 7
};
const long Values::scLongs[] = {
    99, 47, 33, 11, 7
};
const float Values::scTable[] = {
    1.1, 2.2, 3.3, 4.4
};
const char Values::scLetters[] = {
    'a', 'b', 'c', 'd', 'e',
    'f', 'g', 'h', 'i', 'j'
};
float Values::table[4] = {
    1.1, 2.2, 3.3, 4.4
};
char Values::letters[10] = {
    'a', 'b', 'c', 'd', 'e',
    'f', 'g', 'h', 'i', 'j'
};
int main() { Values v; }
```

对于 static const 类型的整型，可以在类内提供这些定义，但是对于其它的对象（包括整数类型的数组，即使它们为静态常量），必须为这些成员提供专用的外部定义。这些定义是内部连接的，所以可以把它们放在头文件中。

也可以创建类的静态常量对象和这样的对象的数组。不过，不能使用“内联语法”初始化它们。

```
//StaticObjectArrays.cpp
// Static arrays of class objects
class X {
    int i;
public:
    X(int ii) : i(ii) {}
};

class Stat {
// This doesn't work, although you might want it to:
//! static const X x(100);
// Both const and non-const static class objects must be initialized externally:
    static X x2;
    static X xTable2[];
    static const X x3;
    static const X xTable3[];
};

X Stat::x2(100);
X Stat::xTable2[] = {
    X(1), X(2), X(3), X(4)
};
const X Stat::x3(100);
const X Stat::xTable3[] = {
    X(1), X(2), X(3), X(4)
};

int main() { Stat v; }
```

- ✧ 局部类（在函数内部定义的类）中不能有静态数据成员
- ✧ 静态成员函数为类的全体对象服务，而不是为一个类的特殊对象服务。把函数移到类内部，这样就不需要定义一个全局函数，减少了全局或局部名字空间的占用。当产生一个静态成员函数时，也就表达了与一个特定类的联系。静态成员函数不能访问一般的数据成员，而只能访问静态数据成员，也只能调用其它的静态成员函数。通常，当前对象的地址（this）是被隐式地传递到被调用的函数的。但一个静态成员函数没有 this，所以它无法访问一般的成员。
- ✧ 静态初始化的相依性：看一个静态变量初始化相互依赖的例子：

```
在一个文件中定义：
extern int y;
int x = y + 1;
在另外一个文件中
extern int x;
int y = x + 1;
```

上面这段代码如果 x 首先初始化，那么结果为：x=1，y=2；反之，若 y 首先初始化，那么结果为：x=2，y=1；
我们并能保证 x 首先初始化，或 y 首先初始化；
有三种方法解决这个问题

1) 不用它，避免初始化时的相互依赖。这是最好的解决办法
2) 如果实在要用，就把那些关键的静态对象的定义放在一个文件中，这样只要它们在文件中顺序正确就可以保证它们正确初始化
3) 如果确性把静态对象放在几个不同的编译单元是不可避免的话，可以通过两种技术加以解决

技术一：
这一技术要求在库的头文件中加上一个额外的类。这个类负责库中的静态变量的动态初始化。

```
//Initializer.h
// Static initialization technique
#ifndef INITIALIZER_H
#define INITIALIZER_H
#include <iostream>
extern int x; // Declarations, not definitions
extern int y;
class Initializer {
    static int initCount;
public:
    Initializer() {
        std::cout << "Initializer()" << std::endl;
        // Initialize first time only
        if(initCount++ == 0) {
            std::cout << "performing initialization"<< std::endl;
            x = 100;
            y = 200;
        }
    }
    ~Initializer() {
        std::cout << "~Initializer()" << std::endl;
        // Clean up last time only
        if(--initCount == 0) {
            std::cout << "performing cleanup"<< std::endl;
            // Any necessary cleanup here
        }
    }
};
// The following creates one object in each file where Initializer.h is included, but
//that object is only visible within that file:
static Initializer init;
#endif // INITIALIZER_H
```

技术二：

这一技术基于这样的事实：函数内部的静态对象在函数第一次被调用的时候初始化，且只被初始化一次。

```
//Dependency1.h
#ifndef DEPENDENCY1_H
#define DEPENDENCY1_H
#include <iostream>
class Dependency1 {
    bool init;
public:
    Dependency1() : init(true) {
        std::cout << "Dependency1 construction"<< std::endl;
    }
    void print() const {
        std::cout << "Dependency1 init: "<< init << std::endl;
    }
};
#endif // DEPENDENCY1_H

//Dependency2.h
#ifndef DEPENDENCY2_H
#define DEPENDENCY2_H
#include "Dependency1.h"
class Dependency2 {
    Dependency1 d1;
public:
    Dependency2(const Dependency1& dep1): d1(dep1){
        std::cout << "Dependency2 construction ";
        print();
    }
    void print() const { d1.print(); }
};
#endif // DEPENDENCY2_H
```

当如下顺序定义时就会出问题：

```
// Simulate the dependency problem:
extern Dependency1 dep1;
Dependency2 dep2(dep1); //此时dep1还没初始化
Dependency1 dep1;
```

解决方法如下：

```
//Dependency1StatFun.h
#ifndef DEPENDENCY1STATFUN_H
#define DEPENDENCY1STATFUN_H
#include "Dependency1.h"
extern Dependency1& d1();
```

```
#endif // DEPENDENCY1STATFUN_H

//Dependency2StatFun.h
#ifndef DEPENDENCY2STATFUN_H
#define DEPENDENCY2STATFUN_H
#include "Dependency2.h"
extern Dependency2& d2();
#endif // DEPENDENCY2STATFUN_H

//Dependency1StatFun.cpp {O}
#include "Dependency1StatFun.h"
Dependency1& d1() {
    static Dependency1 dep1;
    return dep1;
}

//Dependency2StatFun.cpp {O}
#include "Dependency1StatFun.h"
#include "Dependency2StatFun.h"
Dependency2& d2() {
    static Dependency2 dep2(d1());
    return dep2;
}

//Technique2b.cpp
//{L} Dependency1StatFun Dependency2StatFun
#include "Dependency2StatFun.h"
int main() { d2(); }
```

我们也许在函数d1()和d2()的头文件中把它们写为内联函数，但是这样做是不行的。内联函数在它出现的每个文件中都会有一份副本——这种副本包括静对象的定义。因为内联函数自动地默认为内部连接，所以这将导致在不同的编译单元有多个重复的静态对象，这当然会出现问题。所以必须保证每一个定义了静态对象的函数只有一份定义，这就意味着不能把定义了静态对象的函数作为内联函数

（十一） 引用和拷贝构造函数

- ✧ 引用的规则
  1) 当引用被创建时，它必须初始化（指针可以在任何时候初始化）
  2) 一旦一个引用被初始化为指向一个对象，它就不能改变为指向另一个对象的引用（指针则可以在任何时候指向另外一个对象）
  3) 不可能有 NULL 的引用。必须确保引用是和一块合法的内存单元关联
- ✧ 常量引用。看个例子：

```
void f(int&) {}
void g(const int&) {}
int main() {
    //! f(1); // Error
```

```
    g(1);
}
```

- 在 C 和 C++中，参数是从右向左进栈的，然后调用函数，调用代码负责清理栈中的参数
- 传递和返回大对象：

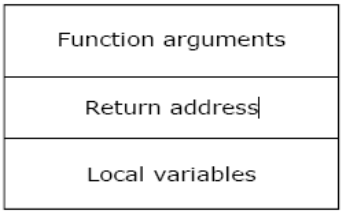
```
//PassingBigStructures.cpp
struct Big {
    char buf[100];
    int i;
    long d;
} B, B2;

Big bigfun(Big b) {
    b.i = 100; // Do something to the argument
    return b;
}

int main() {
    B2 = bigfun(B);
}
```

这段代码在 main()函数中，首先调用 bigfun()，整个 B 的内容被压栈（我们可能发现有些编译器把 B 的地址和大小装入寄存器，然后调用辅助(helper)函数把它压栈）。在调用 bigfun 之前，这里有个附加的操作就是 B2 的地址压栈，虽然它明显不是一个参数，B2 的地址压栈的理由是：bigfun 函数返回时，向这个地址中进行赋值操作

- 函数调用栈框架：当编译器为函数调用产生代码时，它首先把所有的参数压栈，然后调用函数。在函数内部，产生代码，向下移动栈指针为函数局部变量提供存储单元。但是在汇编语言 CALL 中，CPU 把程序代码中的函数调用指令的地址压栈，所以汇编语言 RETURN 可以使用这个地址返回到调用点。这个地址是非常重要的，因为没有它程序将迷失方向。下面是一个 CALL 后栈框架的样子，此时在函数中已为局部变量分配了存储单元。



试图从栈中得到返回值是合理的。因为编译器简单地把返回值压栈，函数可以返回一个偏移值，它告诉返回值的开始在栈中所处的位置。但这将会出现一些问题。因为不能触及堆栈返回地址以上任何部分，所以函数必须在返回地址以下将值压栈。但当汇编语言 RETURN 执行时，堆栈指针必须指向返回地址，所以在恰要执行 RETURN 之前，堆栈指针必须指向返回地址，这便清除了所有局部变量。如果此时试图从堆栈中的返回地址以下返回数值，因为中断可能在此时发生，这也是最易被攻击的时候。这时 ISR 将向下移动堆栈指针，保存返回地址和局部变量，这样

就覆盖掉返回值。

为了解决这个问题，在调用函数之前，调用者应负责在堆栈中为返回值分配额外的存储单元。然而 C 不是按这种方法设计的，C++也不是。

下一个想法可能是在全局数据区域返回数值，但这不可行。重入意味着任何函数可以中断任何其它的函数，包括当前所处的相同函数。因此，如果把返回值放在全局区域，可能又返回到相同的函数中，这将重写返回值。对于递归也是同样的道理

惟一安全的返回场所是寄存器，问题是当寄存器没有用于存放返回值的足够大小时该怎么做？答案是把返回值的地址像一个函数参数一样压栈，让函数直接把返回值信息拷贝到目的地。这也是 PassingBigStructures.cpp 的 main()中 bigfun()调用之前将 B2 的地址压栈的原因。如果看了 bigfun()的汇编输出，可以看到它接收这个隐藏的参数并在函数内完成向目的地的拷贝。

✧ 拷贝构造函数的一个经典例子：

```
//HowMany2.cpp
// The copy-constructor
#include <fstream>
#include <string>
using namespace std;
ofstream out("HowMany2.out");

class HowMany2 {
    string name; // Object identifier
    static int objectCount;
public:
    HowMany2(const string& id = "") : name(id) {
        ++objectCount;
        print("HowMany2()");
    }
    ~HowMany2() {
        --objectCount;
        print("~HowMany2()");
    }
// The copy-constructor:
    HowMany2(const HowMany2& h) : name(h.name) {
        name += " copy";
        ++objectCount;
        print("HowMany2(const HowMany2&)");
    }
    void print(const string& msg = "") const {
        if(msg.size() != 0)
            out << msg << endl;
        out << '\t' << name << ": "<< "objectCount = "<< objectCount << endl;
    }
};
```

```
int HowMany2::objectCount = 0;
// Pass and return BY VALUE:
HowMany2 f(HowMany2 x) {
	x.print("x argument inside f()");
	out << "Returning from f()" << endl;
	return x;
}

int main() {
	HowMany2 h("h");
	out << "Entering f()" << endl;
	HowMany2 h2 = f(h);
	h2.print("h2 after call to f()");
	out << "Call f(), no return value" << endl;
	f(h);
	out << "After call to f()" << endl;
}
```

上述代码的输出是：

```
1) HowMany2()
2) h: objectCount = 1
3) Entering f()
4) HowMany2(const HowMany2&)
5) h copy: objectCount = 2
6) x argument inside f()
7) h copy: objectCount = 2
8) Returning from f()
9) HowMany2(const HowMany2&)
10) h copy copy: objectCount = 3
11) ~HowMany2() //复制完后再析构
12) h copy: objectCount = 2
13) h2 after call to f()
14) h copy copy: objectCount = 2
15) Call f(), no return value
16) HowMany2(const HowMany2&)
17) h copy: objectCount = 3
18) x argument inside f()
19) h copy: objectCount = 3
20) Returning from f()
21) HowMany2(const HowMany2&) //即使忽略返回值，但还是要进行对象构造
22) h copy copy: objectCount = 4
23) ~HowMany2()   // f中的x临时对象先析构
24) h copy: objectCount = 3
25) ~HowMany2()   // x的拷贝，也就是被忽略的返回值，后析构
```

26) h copy copy: objectCount = 2
27) After call to f()
28) ~HowMany2()
29) h copy copy: objectCount = 1
30) ~HowMany2()
31) h: objectCount = 0

✧ 指向成员的指针：所有的指针需要地址，但在类内部是没有地址的；选择一个类的成员意味着在类中偏移。只有把这个偏移和具体对象的开始地址结合，才能得到实际地址。成员指针的语法要求**选择一个对象的同时间接引用成员指针**。定义一个名字为 pointerToMember 的成员指针，该指针可以指向在 ObjectClass 类中的任一 int 类型的成员。

int ObjectClass::*pointerToMember = &ObjectClass::a;

因为仅仅提到了一个类而非那个类的对象，所以没有 ObjectClass::a 的确切“地址”，它仅代表类中的一个偏移。因而，&ObjectClass::a 仅是作为成员指针的语法被使用。

✧ 求成员函数的地址时，符号&是必须的，求非成员函数的地址时符号&可有可无。但是，可以给出不含参数列表的函数标识符，因为重载方案可以由成员指针类型所决定。

```
class Simple2 {
public:
  int f(float) const { return 1;}
  int f(int)const{return 1;}
};

int (Simple2::*fp)(float) const;
int (Simple2::*fp2)(float) const = &Simple2::f;

int main() {
  fp = &Simple2::f;
}
```

✧ 一个成员函数指针的典型例子：

```
class Widget {
    void f(int) const { cout << "Widget::f()\n"; }
    void g(int) const { cout << "Widget::g()\n"; }
    void h(int) const { cout << "Widget::h()\n"; }
    void i(int) const { cout << "Widget::i()\n"; }
    enum { cnt = 4 };
    void (Widget::*fptr[cnt])(int) const;
public:
    Widget() {
        fptr[0] = &Widget::f; // Full spec required    //pay attention
        fptr[1] = &Widget::g;
        fptr[2] = &Widget::h;
        fptr[3] = &Widget::i;
```

```
    }
    void select(int i, int j) {
        if(i < 0 || i >= cnt) return;
        (this->*fptr[i])(j);     //pay attention
    }
    int count() { return cnt; }
};

int main() {
    Widget w;
    for(int i = 0; i < w.count(); i++)
        w.select(i, 47);
}
```

上述代码，在 Widget 构造函数中，成员函数的初始化似乎过分指定了。是否可以这样写：fptr[1] = &g;因为名字 g 在成员函数中出现，这是否可以自动地认为在这个类范围内呢？问题是这不符合成员函数的语法，它的语法要求编译器能够判断将要做什么。相似地，当成员指针被间接引用时，它看起来像这样：(this->*fptr[i])(j);它仍然是过分指定的，this 似乎多余。正如前面所讲的，当成员指针被间接引用时，语法也需要成员指针总是和一个对象绑定在一起

（十二） 运算符重载

- ✧ 前缀自增和后缀自增运算符：后缀自增（减）运算符比前缀形式多一个 int 参数，比如前缀自增运算符为：Integer& operator++(Integer& a);那么相应的后缀形式为：Integer operator++(Integer&a, int);
- ✧ 重载赋值运算符时注意“自赋值（self-assignment）”的检查，这非常重要
- ✧ 运算符重载时的几个模式：
  1) 对于任何函数参数，如果仅需要从参数中读而不改变它，默认地应当作为 const 引用来传递它。
  2) 返回值的类型取决于运算符的具体含义。如果使用该运算符的结果是产生一个新值，就需要产生一个作为返回值的新对象。
  3) 所有赋值运算符均改变左值。为了使赋值结果能用于链式表达式（如 a=b=c），应该能够返回一个刚刚改变了的左值的引用。所有赋值运算符的返回值对于左值应该是非常量引用
  4) 对于逻辑运算符，我们希望至少得到一个 int 返回值，最好是 bool 返回值
  5) 前缀和后缀版本返回值：对于前缀版本，在对象被改变后，我们希望返回改变后的对象。这样用前缀版本只需要作为一个引用返回*this。因为后缀版本返回改变之前的值，所以必须创建一个代表这个值得独立对象并返回它。
- ✧ 返回值优化（return value optimization）：通过传值方式返回要创建的新对象时，应注意使用的形式。例如在 operator+中：return Integer(left.i + right.i);咋看起来就像是一个“对一个构造函数的调用”，其实并非如此。这是临时对象语法，它是在说：“创建一个临时 Integer 对象并返回它”。这和下面这种写法是不同的：Integer tmp(left.i + right.i); return tmp;这种创建一个局部对象然后返回，会发生三件事：1)创建 tmp 对象，其中包括构造函数的调用；2)拷贝函数把 tmp 拷贝到外部返回值的存储单元里；3)当 tmp 在作用域的结尾时调用析构函数。与此相反，“返回临时对象”的方式是完全不同的，当编译器看到我们这样做时，它明白对创建的对象没有

其它需求，只是返回它，所以编译器直接把这个对象创建在外部返回值的内存单元。因为不是真正创建一个局部对象，所以仅需要一个普通构造函数的调用，且不会调用析构函数。这种方法不需要什么花费，因此效率非常高。

- ✧ 下标运算符 operator[]，必须是成员函数并且它只接受一个参数。因为它作用的对象应该像数组一样操作，可以经常从这个运算符返回一个引用
- ✧ operator->指针间接引用运算符一定是一个成员函数。它有着额外的限制：它必须返回一个对象（或对象的引用），该对象也有一个指针间接引用运算符；或者必须返回一个指针，被用于选择指针间接引用运算符箭头所指向的内容
- ✧ operator->*是个二元运算符。要想创建一个 operator->*，必须首先创建带有 operator()类，这是 operator->*将返回对象的类。The trick when defining operator->* is that it must return an object for which the operator( ) can be called with the arguments for the member function you're calling.看下面的一个关于 operator->*的例子：

```
class Dog {
public:
    int run(int i) const {
        cout << "run\n";
        return i;
    }
    int eat(int i) const {
        cout << "eat\n";
        return i;
    }
    int sleep(int i) const {
        cout << "ZZZ\n";
        return i;
    }

    typedef int (Dog::*PMF)(int) const;

    // operator->* must return an object
    // that has an operator():
    class FunctionObject {
        Dog* ptr;
        PMF pmem;
    public:
    // Save the object pointer and member pointer
        FunctionObject(Dog* wp, PMF pmf): ptr(wp), pmem(pmf) {
            cout << "FunctionObject constructor\n";
        }
    // Make the call using the object pointer
    // and member pointer
        int operator()(int i) const {
            cout << "FunctionObject::operator()\n";
            return (ptr->*pmem)(i); // Make the call
```

```
        }
    };

    FunctionObject operator->*(PMF pmf) {
        cout << "operator->*" << endl;
        return FunctionObject(this, pmf);
    }
};

int main() {
    Dog w;
    Dog::PMF pmf = &Dog::run;
    cout << (w->*pmf)(1) << endl;
    pmf = &Dog::sleep;
    cout << (w->*pmf)(2) << endl;
    pmf = &Dog::eat;
    cout << (w->*pmf)(3) << endl;
}
```

✧ 不能重载的运算符

1) 成员选择 operator.
2) 成员指针间接引用 oprator.*
3) 不存在的用户定义的运算符
4) 不能改变运算符的优先级

✧ 运算符重载为成员或非成员的基本方针

Operator	Recommended use
All unary operators	member
= ( ) [ ] -> ->*	*must* be member
+= -= /= *= ^= &= |= %= >>= <<=	member
All other binary operators	non-member

✧ 设计一个类时，我们应该尤其首先要考虑这么几个函数（如果我们不考虑编译器可能在合适的时候自动产生）：contructor、copy-constructor、operator=、de-constructor. 我们应该总是明确地给出这些函数的定义。如果不想定义这些函数可以把它们声明为 private，且不必定义它们

✧ 自动类型转换有两种类型：特殊的构造函数和重载运算符。看下面两个例子：

```
//AutomaticTypeConversion.cpp
// Type conversion constructor
class One {
public:
    One() {}
};
class Two {
```

```
public:
    Two(const One&) {}
};
void f(Two) {}
int main() {
    One one;
    f(one); // Wants a Two, has a One
}

//OperatorOverloadingConversion.cpp
class Three {
    int i;
public:
    Three(int ii = 0, int = 0) : i(ii) {}
};

class Four {
    int x;
public:
    Four(int xx) : x(xx) {}
    operator Three() const { return Three(x); }
};
void g(Three) {}
int main() {
    Four four(1);
    g(four);
    g(1); // Calls Three(1,0)
}
```

- ✧ 用构造函数技术，是目的类执行转换。使用运算符技术，是源类执行转换。构造函数的价值是在创建一个新类时为现有系统增加了新的转换途径。然而，创建一个单一参数的构造函数总是定义一个自动类型转换（即使它有不止一个参数也是一样，其余参数都有默认值），这可能并不是我们所要的（这可以使用 explicit 来避免）。另外，使用构造函数技术没有办法实现从用户定义类型向内置类型转换，这只有运算符重载可能做到

（十三） 动态对象创建

- ✧ 默认的 new 进行检查以确信在传递地址给构造函数之前内存分配是成功的，所以不必显式地确定 new 的调用是否成功
- ✧ new 一个对象发生两件事：1）分配足够的内存；2）调用对象的构造函数初始化这片内存；delete 一个对象也发生两件事：1）调用对象的析构函数清理对象；2）释放对象所占内存；new 和 delete 成对使用，new []和 delete []成对使用，malloc 和 free 成对使用
- ✧ 使用 set_new_handler()函数设置当内存不足时，编译器应该调用的函数
- ✧ 重载全角 new 和 delete。operator new()的返回值是一个 void*，而不是指向任何特定类型的指针。所做的是分配内存，而不是完成一个对象的建立——直到构造函数

调用了才完成了对象的创建，它是编译器确保的动作，不在我们的控制范围之内。operator delete()的参数是一个指向由 operator new()分配的内存的 void*。参数是一个 void*是因为是一个在调用析构函数后得到的指针。析构函数从内存单元里移去对象。operator delete()返回 void。一个典型例子：

```
//GlobalOperatorNew.cpp
// Overload global new/delete
#include <cstdio>
#include <cstdlib>
using namespace std;

void* operator new(size_t sz) {
    void* m = malloc(sz);
    if(!m) puts("out of memory");
        return m;
}
void operator delete(void* m) {
    puts("operator delete");
    free(m);
}
```

✧ 对一个类重载 new 和 delete：为一个类重载 new 和 delete 时，虽然不必显式地使用 static，但实际上仍是在创建 static 成员函数。它的语法和重载任何其它运算符一样。当编译器看到使用 new 创建自己定义的类的时，它选择成员版本的 operator new()而不是全局版本的 new()。但全局版本的 new 和 delete 仍为所有其它类型对象使用。一个典型例子：

```
//Framis.cpp
// Local overloaded new & delete
#include <cstddef> // Size_t
#include <fstream>
#include <iostream>
#include <new>
using namespace std;
ofstream out("Framis.out");

class Framis {
    enum { sz = 10 };
    char c[sz]; // To take up space, not used
    static unsigned char pool[];
    static bool alloc_map[];
public:
    enum { psize = 100 }; // frami allowed
    Framis() { out << "Framis()\n"; }
    ~Framis() { out << "~Framis() ... "; }
    void* operator new(size_t) throw(bad_alloc);
    void operator delete(void*);
```

```
};

unsigned char Framis::pool[psize * sizeof(Framis)];
bool Framis::alloc_map[psize] = {false};

// Size is ignored -- assume a Framis object
void* Framis::operator new(size_t) throw(bad_alloc) {
    for(int i = 0; i < psize; i++)
        if(!alloc_map[i]) {
            out << "using block " << i << " ... ";
            alloc_map[i] = true; // Mark it used
            return pool + (i * sizeof(Framis));
        }
    out << "out of memory" << endl;
    throw bad_alloc();
}

void Framis::operator delete(void* m) {
    if(!m) return; // Check for null pointer
    // Assume it was created in the pool
    // Calculate which block number it is:
    unsigned long block = (unsigned long)m - (unsigned long)pool;
    block /= sizeof(Framis);
    out << "freeing block " << block << endl;
    // Mark it free:
    alloc_map[block] = false;
}
```

✧ 定位 new 和 delete：重载的 operator new()可以带一个或多个参数。但第一个参数总是对象的长度，它在内部计算出来并由编译器传递给 new。其它参数可由我们自己定义：一个对象放置的地址、一个是对内存分配函数或对象的引用，或其它任何使我们方便的设置。下面是个定位 new 的例子：

```
//PlacementOperatorNew.cpp
// Placement with operator new
#include <cstddef> // Size_t
#include <iostream>
using namespace std;

class X {
    int i;
public:
    X(int ii = 0) : i(ii) {
        cout << "this = " << this << endl;
    }
    ~X() {
```

```
        cout << "X::~X(): " << this << endl;
    }
    void* operator new(size_t, void* loc) {
        return loc;
    }
};

int main() {
    int l[10];
    cout << "l = " << l << endl;
    X* xp = new(l) X(47); // X at location l
    xp->X::~X(); // Explicit destructor call ONLY use with placement!
                 // or xp->~X()
}
```

同样存在定位 operator delete，它仅在一个定位 operator new 表达式的构造函数产生一个异常信息时才被调用（因此该内存在异常处理操作中被自动清除了）。定位 operator delete 有一个和定位 operator new 相对应的参数表，该定位 operator new 是指构造函数产生异常信息之前被调用的那一个

（十四） 继承和组合

- ✧ 构造是从类层次的最根处开始的，而在每一层，首先会调用基类构造函数，然后调用成员对象构造函数。调用析构函数则严格按照构造函数相反的次序。对于成员对象，构造函数的调用次序完全不受构造函数的初始化列表中的次序影响。该次序是由成员对象在类中声明的次序所决定的。
- ✧ 在类继承时，注意名字的隐藏问题
- ✧ 非自动继承的函数：构造函数和析构函数不能被继承，必须为每个特定的派生类分别创建。operator=也不能被继承，因为它完成类似于构造函数的活动。这就是说，尽管我们知道如何由等号右边的对象初始化左边的对象的所有成员，但这并不意味着这个初始化在继承后仍然具有相同的意义。一个典型例子：

```
//SynthesizedFunctions.cpp
// Functions that are synthesized by the compiler
#include <iostream>
using namespace std;

class GameBoard {
public:
    GameBoard() { cout << "GameBoard()\n"; }
    GameBoard(const GameBoard&) {
        cout << "GameBoard(const GameBoard&)\n";
    }
    GameBoard& operator=(const GameBoard&) {
        cout << "GameBoard::operator=()\n";
        return *this;
    }
    ~GameBoard() { cout << "~GameBoard()\n"; }
```

```
};
class Game {
    GameBoard gb; // Composition
public:
    // Default GameBoard constructor called:
    Game() { cout << "Game()\n"; }
    // You must explicitly call the GameBoard
    // copy-constructor or the default constructor
    // is automatically called instead:
    Game(const Game& g) : gb(g.gb) {
        cout << "Game(const Game&)\n";
    }
    Game(int) { cout << "Game(int)\n"; }
    Game& operator=(const Game& g) {
        // You must explicitly call the GameBoard
        // assignment operator or no assignment at
        // all happens for gb!
        gb = g.gb;
        cout << "Game::operator=()\n";
        return *this;
    }
    class Other {}; // Nested class
    // Automatic type conversion:
    operator Other() const {
        cout << "Game::operator Other()\n";
        return Other();
    }
    ~Game() { cout << "~Game()\n"; }
};

class Chess : public Game {};
void f(Game::Other) {}

class Checkers : public Game {
public:
    // Default base-class constructor called:
    Checkers() { cout << "Checkers()\n"; }
    // You must explicitly call the base-class
    // copy constructor or the default constructor
    // will be automatically called instead:
    Checkers(const Checkers& c) : Game(c) {
        cout << "Checkers(const Checkers& c)\n";
    }
    Checkers& operator=(const Checkers& c) {
```

```
            // You must explicitly call the base-class
            // version of operator=() or no base-class
            // assignment will happen:
            Game::operator=(c);
            cout << "Checkers::operator=()\n";
            return *this;
        }
};

int main() {
        Chess d1; // Default constructor
        Chess d2(d1); // Copy-constructor
        //! Chess d3(1); // Error: no int constructor
        d1 = d2; // Operator= synthesized
        f(d1); // Type-conversion IS inherited
        Game::Other go;
        //! d1 = go; // Operator= not synthesized
        // for differing types
        Checkers c1, c2(c1);
        c1 = c2;
}
```

仔细阅读上述代码的注释部分

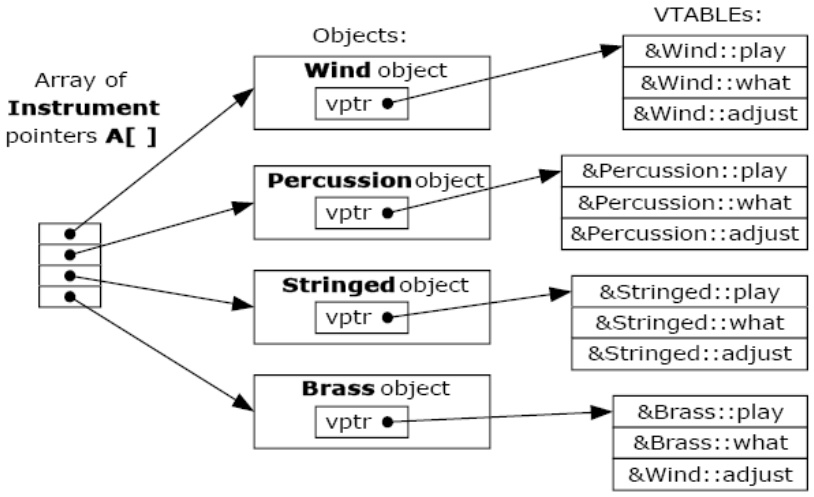
- ✧ 静态成员函数和非静态成员函数的共同点
  1) 它们均可被继承到派生类中
  2) 如果我们重新定义了一个静态成员函数，所有基类中的其它重载函数会被隐藏
  3) 如果我们改变了基类中一个函数的特征，所有使用该函数名字的基类版本都将会隐被藏
- ✧ 组合通常是在希望新类内部具有已存在类的功能时使用，而不是希望已存在类作为它的接口。这就是说，嵌入一个对象用以实现新类的功能，而新类的用户看到的是新定义的接口而不是老类的接口。为此，在新类的内部嵌入已存在类的 private 对象。private 继承的意义和组合差不多
- ✧ 如果希望子类把基类中的每件东西都加入进来，这就是子类化（subtyping），因为我们正由已存在的类创建一个新类，并且希望这个新类与已存在的类有着严格相同的接口（可以增加任何我们想加入的其他成员函数），所以能在已经使用过这个已存在类的任何地方使用这个新类，这就是必须使用继承的地方
- ✧ 私有继承：当私有继承时，我们是“照此实现（implement in terms of）”，也就是说，创建的新类具有基类的所有数据和功能，但这些功能是隐藏的，所以它只是部分的内部实现。该类的用户访问不到这些内部功能，并且一个对象不能被看做是这个基类的实例。通常是选择使用组合而不是 private 继承。然而，这里可能偶然有这种情况，即可能想产生像基类接口一样的接口部分，而不允许该对象的处理像一个基类对象，private 继承提供了这个功能
- ✧ 除了赋值运算符，其它的运算符可以自动地继承到派生类中
- ✧ 继承的最重要的方面不是它为新类提供了成员函数，而是它是基类与新类之间的关系，这种关系可背描述为：“新类属于原有类的类型”。也即，一个派生类对象是一

个基类对象

- ✧ **确定应当使用组合还是继承，最清楚的方法之一是询问是否需要从新类型向上类型转换**

（十五） 多态性和虚函数

- ✧ 如果不用虚函数，就等于不懂 OOP
- ✧ 把函数体与函数调用相联系称为捆绑（binding）。当捆绑在程序运行之前（由编译器和连接器）完成时，称为早捆绑（early binding）。C 编译器只有一种函数调用方式，就是早捆绑。晚捆绑（late binding）意味着捆绑根据对象的类型，发生在运行时。晚捆绑又称为动态捆绑（dynamic binding）或运行时捆绑（runtime binding）。当一种语言实现晚捆绑时，必须有某种机制来确定运行时对象的类型并调用合适的成员函数
- ✧ 对于特定的函数，为了引起晚捆绑，C++要求在基类中声明这个函数时使用 virtual 关键字。晚捆绑只对 virtual 函数起作用，而且只在使用含有 virtual 函数的基类的地址时发生。
- ✧ 为了创建一个 virtual 成员函数，可以简单地在这个函数声明额前面加上关键字 virtual。仅仅在声明的时候需要使用关键字 virtual，定义时并不需要。如果一个函数在基类中被声明为 virtual，那么在所有的派生类中它都是 virtual 的。在派生类中 virtual 函数的重定义通常称为重写（overriding）
- ✧ C++实现晚捆绑的机制：典型的编译器对每个包含虚函数的类创建一个表，成为 VTABLE，在 VTABLE 中，编译器放置特定类的虚函数的地址。在每个带有虚函数的类中，编译器秘密地放置一个指针，成为 vpointer（VPTR），指向这个对象的 VTABLE。当通过基类指针做虚函数调用时（也就是多态调用时），编译器静态地插入能取得这个 VPTR 并在 VTABLE 表中查找函数地址的代码，这样就能调用正确的函数并引起晚捆绑的发生
- ✧ 一个典型的类继承层次的 VTABLE 和 VPTR 示意图：



每当创建一个包含有虚函数的类或从包含有虚函数的类派生一个类时，编译器就为这个类创建一个惟一的 VTABLE。在这个表中，编译器放置了在这个类中或在它的基类中所有已声明为 virtual 的函数的地址。如果在这个派生类中没有对在基类中声明为 virtual 的函数进行重定义，编译器就是用基类的这个虚函数地址（在 Brass 的 VTABLE 中，adjust 的入口就是这种情况）。然后编译器在这个类中放置 VPTR。当使用简单继承时，对于每个对象只有一个 VPTR。VPTR 必须被初始化

为指向相应的 VTABLE 的起始地址，这是在构造函数中发生的。一旦 VPTR 被初始化为指向相应的 VTABLE，对象就“知道”它自己是什么类型。但只有当虚函数被调用时这种自我认知才有用

- ✧ 撩开面纱：如果能看到由虚函数调用而产生的汇编语言代码，这将是很有帮助的，这样可以看到后捆绑实际上是如何发生的。下面是在函数 f(instrument&i)中调用

  i . adjust(1);

  某个编译器所产生的输出：

  ```
  push 1
  push si
  mov bx, word ptr[si]
  call  word ptr[bx+4]
  add  sp, 4
  ```

  C++函数调用的参数与C函数调用一样，是从右向左进栈的（这个顺序是为了支持C的变量参数表），所以参数1首先压栈。在这个函数的这个地方，寄存器si（intel x86处理器的一部分）存放i的首地址。因为它是被选中的对象的首地址，它也被压进栈。记住，这个首地址对应于this的值，正因为调用每个成员函数时this都必须作为参数压进栈，所以成员函数知道它工作在哪个特殊对象上。这样，我们总能看到，在成员函数调用之前压栈的次数等于参数个数加一（除了static成员函数，它没有this）。

  现在，必须实现实际的虚函数调用。首先，必须产生VPTR，使得能找到VTABLE。对于这个编译器，VPTR在对象的开头，所以this的内容对应于VPTR。下面这一行

  ```
  mov bx , word ptr[si]
  ```

  取出si（即this）所指的字，它就是VPTR。将这个VPTR放入寄存器bx中。放在bx中的这个VPTR指向这个VTABLE的首地址，但被调用的函数在VTABLE中不是第0个位置，而是第二个位置（因为它是这个表中的第三个函数）。对于这种内存模式，每个函数指针是两个字节长，所以编译器对VPTR加四，计算相应的函数地址所在的地方，注意，这是编译时建立的常值。所以我们只要保证在第二个位置上的指针恰好指向adjust()。幸好编译器仔细处理，并保证在VTABLE中的所有函数指针都以相同的次序出现。一旦在VTABLE中相应函数指针的地址被计算出来，就调用这个函数。所以取出这个地址并马上在这个句子中调用：

  ```
  call word ptr [bx+4]
  ```

  最后，栈指针移回去，以清除在调用之前压入栈的参数。在C和C++汇编代码中，我们将常常看到调用者清除这些参数，但这依处理器和编译器的实现而有所变化。

- ✧ 要知道向上类型转换仅处理地址（或引用），这是重要的。如果编译器有一个它知道确切类型的对象，那么，在 C++中，对任何函数的调用将不再使用晚捆绑，或至少编译器不必使用晚捆绑

- ✧ 抽象基类和纯虚函数：在设计时，常常希望基类仅仅作为其派生类的一个接口。这就是说，仅想对基类进行向上类型转换，使用它的接口，而不希望用户实际地创建一个基类的对象。要做到这一点，可以在基类中加入至少一个纯虚函数（pure virtual function），来使基类成为抽象（abstract）类。纯虚函数使用关键字 virtual，并且在其后面加上=0。如果某人试着生成一个抽象类的对象，编译器会阻止他。建立公共接口的惟一原因是它能对于每个不同的子类有不同的表示。它建立一个基本的格

式，用来确定什么是对于所有派生类是公共的——除此之外，别无用途。

- ✧ 声明一个纯虚函数等于告诉编译器在 VTABLE 中为函数保留一个位置，但在这个特定位置中不放置地址。只要有一个函数在类中被声明为纯虚函数，那么 VTABLE 就是不完全的
- ✧ 如果一个类中的成员函数全部是纯虚函数，那么这个类为纯抽象类（pure abstract class）。纯虚函数禁止对抽象类以传值的方式调用。这也是防止对象切片的一种方法（这也许是纯虚函数最重要的作用）。通过抽象类，可以保证在向上类型转换期间总是使用指针或引用
- ✧ 纯虚函数定义：在基类中，对纯虚函数提供定义是可能的。我们仍然告诉编译器不允许产生抽象基类的对象，而且如果要创建对象，则纯虚函数必须在派生类中定义。然而，我们可能希望一段公共代码，使一些或所有派生类定义都能调用，而不必在每个函数中重复这段代码，那么我们可能会对纯虚函数进行定义。看下面一个例子：

```
class Pet{
public:
    virtual void speak()const = 0;
    virtual void eat()const = 0;
    //inline pure virtual definitions illegal:  但是vs2005能够通过
    //virtual void sleep()const = 0{}
};

void Pet::speak()const
{
    cout<<"Pet::speak"<<endl;
}

void Pet::eat()const
{
    cout<<"Pet::eat"<<endl;
}

class Dog:public Pet{
public:
    void speak()const
    {
        cout<<"Dog speak:";
        Pet::speak();
    }
    void eat()const
    {
        cout<<"Dog eat:";
        Pet::eat();
    }
    //void sleep()const{}
};
```

上述代码中 Pet 的 VTABLE 表仍然空着，但在派生类中刚好有一个函数，可以通过名字调用它。这个特点的另一个好处是，它允许我们实现从常规虚函数到纯虚函数的改变，而无需打乱已有的代码

- ✧ 如果对一个对象进行向上类型转换，而不使用地址或引用，这个对象将被“切片”，直到剩下来的是适合目的的子对象
- ✧ 重载和重写（overloading&overriding）：我们知道重新定义一个基类中的重载函数将会隐藏所有该函数的其它基类版本。而当对虚函数进行这些操作时，情况会有点不同。看下面一个例子：

```
//NameHiding2.cpp
// Virtual functions restrict overloading
#include <iostream>
#include <string>
using namespace std;
class Base {
public:
    virtual int f() const {
        cout << "Base::f()\n";
        return 1;
    }
    virtual void f(string) const {}
    virtual void g() const {}
};

class Derived1 : public Base {
public:
    void g() const {}
};

class Derived2 : public Base {
public:
    // Overriding a virtual function:
    int f() const {
        cout << "Derived2::f()\n";
        return 2;
    }
};

class Derived3 : public Base {
public:
    // Cannot change return type:
    //! void f() const{ cout << "Derived3::f()\n";}
};
class Derived4 : public Base {
public:
```

```
        // Change argument list:
        int f(int) const {
            cout << "Derived4::f()\n";
            return 4;
        }
};

int main() {
    string s("hello");
    Derived1 d1;
    int x = d1.f();
    d1.f(s);
    Derived2 d2;
    x = d2.f();
    //! d2.f(s); // string version hidden
    Derived4 d4;
    x = d4.f(1);
    //! x = d4.f(); // f() version hidden
    //! d4.f(s); // string version hidden
    Base& br = d4; // Upcast
    //! br.f(1); // Derived version unavailable
    br.f(); // Base version available
    br.f(s); // Base version abailable
}
```

首先注意到，在 Derived3 中，编译器不允许我们改变重写（overridden）过的函数的返回值（如果函数 f()不是虚函数，则是允许的）。这是一个非常重要的限制，因为编译器必须保证我们能够多态地通过基类调用函数，并且如果基类希望 f()返回一个 int 值，则 f()的派生类版本必须保持这种约定，否则会出问题。前面的规则仍然有效：如果重新定义了基类中的一个重载成员函数，则在派生类中的其它的重载函数将会被隐藏。这可由 main()中测试 Derived4 的代码显示出来，即使 f()的新版本实际上没有改写（override）已存在的的虚函数的接口，f()的两个基类版本会被 f(int)隐藏。

- ✧ 上面的一个例子中类 Derived3 显示了我们不能在重写（override）过程中修改虚函数的返回值类型。通常是这样的，但也有特例，我们可以稍稍修改返回类型。如果返回一个指向基类 B 的指针或引用，则该函数的重写版本可以返回一个指向 B 的派生类的指针或引用（这是 C++新版的一个语言特性）
- ✧ 构造函数不能是虚函数，但析构函数往往是虚函数
- ✧ 纯虚析构函数：纯虚析构函数在标准 C++中是合法的，但有一个限制：必须为纯虚析构函数提供一个函数体。纯虚析构函数和非纯虚析构函数之间惟一的不同之处在于纯虚析构函数使得基类是抽象类，所以不能创建一个基类的对象（虽然如果基类的任何其它函数是纯虚函数，也是具有相同的效果）。当从某个含有纯虚析构函数的类继承出一个类时，情况变得有点复杂。不像其它的纯虚函数，我们不要求在派生类中提供基类的纯虚析构函数的定义。一般来说，如果在派生类中基类的纯虚函数没有重新定义，则派生类会自动成为抽象类。但对于纯虚析构函数来说，情况不

是这样。因为如果不进行析构函数的定义，编译器将会自动地为每个类生成一个析构函数定义。那就是这里所发生的——基类的析构函数被重写，因此编译器会提供定义并且派生类实际上不会成为抽象类

- ✧ 任何时候在我们的类中如果有一个虚函数，我们应当立即增加一个虚析构函数（即使它什么也不做）
- ✧ 在析构函数中调用的成员函数，不管它是虚的还是非虚的，只有成员函数的“本地”版本被调用；虚机制被忽略。在构造函数中虚机制同样会被忽略。
- ✧ 当使用 dynamic_cast 时，必须对一个真正多态的层次进行操作——它含有虚函数——这因为 dynamic_cast 使用了存储在 VTABLE 中的信息来判断实际的类型
- ✧ 无论何时进行向下类型转换，我们都有责任进行检查以确保类型转换的返回值非 0。但我们不用确保指针完全一样，这是因为通常在向上类型转换和向下类型转换时指针会进行调整（特别是在多重继承的情况下）

（十六） 模板介绍

- ✧ 继承和组合提供了重用对象代码的方法，而 C++的模板特征提供了重用源代码的方法
- ✧ 即使是在创建非内联函数定义时，我们还是通常想把模板的所有声明和定义都放入一个头文件中。这似乎违背了通常的头文件规则：“不要放置分配存储空间的任何东西”（这条规则是为了防止在连接期间的多重定义错误），但模板定义很特殊。在 template<…>之后的任何东西都意味着编译器在当时不为它分配存储空间，而是一直处于等待状态直到被一个模板示例告知。在编译器和连接器中有机制能去掉同一模板的多重定义。所以几乎为了使用方便，几乎总是在头文件中放置全部的模板声明和定义。
- ✧ 列出一个带有迭代器的 TStack 代码，以供参考模板和迭代器的相关技术：

```
#ifndef TSTACK2_H
#define TSTACK2_H

template<class T>
class Stack {
    struct Link {
        T* data;
        Link* next;
        Link(T* dat, Link* nxt): data(dat), next(nxt) {}
    }* head;
public:
    Stack() : head(0) {}
    ~Stack();
    void push(T* dat) {
        head = new Link(dat, head);
    }
    T* peek() const {
        return head ? head->data : 0;
    }
    T* pop();
    // Nested iterator class:
```

```
    class iterator; // Declaration required
    friend class iterator; // Make it a friend
    class iterator { // Now define it
        Stack::Link* p;
    public:
        iterator(const Stack<T>& tl) : p(tl.head) {}
        // Copy-constructor:
        iterator(const iterator& tl) : p(tl.p) {}
        // The end sentinel iterator:
        iterator() : p(0) {}
        // operator++ returns boolean indicating end:
        bool operator++() {
            if(p->next)
            p = p->next;
            else p = 0; // Indicates end of list
            return bool(p);
        }
        bool operator++(int) { return operator++(); }
        T* current() const {
            if(!p) return 0;
            return p->data;
        }
        // Pointer dereference operator:
        T* operator->() const {
            require(p != 0,"PStack::iterator::operator->returns 0");
            return current();
        }
        T* operator*() const { return current(); }
        // bool conversion for conditional test:
        operator bool() const { return bool(p); }
        // Comparison to test for end:
        bool operator==(const iterator&) const {
            return p == 0;
        }
        bool operator!=(const iterator&) const {
            return p != 0;
        }
    };
    iterator begin() const {
        return iterator(*this);
    }
    iterator end() const { return iterator(); }
};
```

```
template<class T>
Stack<T>::~Stack() {
    while(head)
        delete pop();
}

template<class T>
T* Stack<T>::pop() {
    if(head == 0) return 0;
    T* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
}
#endif // TSTACK2_H
```

（十七） 编程准则

- ✧ 头文件包含顺序是从“最特殊到最一般”。也就是，在本地目录中的任何头文件首先被包含，然后是我们自己的所有“工具”头文件，随后是第三方库头文件，接着是标准 C++库头文件和 C 库头文件
- ✧ 编程准则：

1) 不要用 C + +主动重写我们已有的 C 代码，除非我们需要对它的功能做较大的调整，（也就是说，不破不立）。用 C++重新编译是很有价值的，因为这可以发现隐藏的错误。把一段运行得很好的 C 代码用 C++重写可能是在浪费时间，除非 C++的版本以类的形式提供许多重用的机会。
2) 要区别类的创建者和类的使用者（客户程序员）。类的使用者才是“顾客”，他们并不需要或许也不想知道类的内部是怎样运作的。类的创建者必须是设计类和编写类的专家，以使得被创建的类可以被最没有经验的程序员使用，而且在应用程序中工作良好。库只是在透明的情况下才会容易使用。
3) 当我们创建一个类时，要尽可能用有意义的名字来命名类。我们的目标应该是使用户接口要领简单。可以用函数重载和缺省参数来创建一个清楚、易用的接口。
4) 数据隐藏允许我们（类的创建者）将来在不破坏用户代码（代码使用了该类）的情况下随心所欲地修改代码。为实现这一点，应把对象的成员尽可能定义为 private，而只让接口部分为 public，而且总是使用函数而不是数据。只有在迫不得已时才让数据为 public。如果类的使用者不需要调用某个函数，就让这个函数成为 private。如果类的一部分要让派生类可见，就定义成 protected，并提供一个函数接口而不是直接暴露数据，这样，实现部分的改变将对派生类产生最小的影响。
5) 不要陷入分析瘫痪之中。有些东西只有在编程时才能学到并使各种系统正常。C++有内建的防火墙，让它们为我们服务。在类或一组类中的错误不会破坏整个系统的完整性。
6) 我们的分析和设计至少要在系统中创建类、它们的公共接口、它们与其他类的关系、特殊的基类。如果我们的方法产生的东西比这些更多，就应当问问自己，

是不是所有的成分在程序的整个生命期中都是有价值的，如果不是，将会增加我们对它们的维护开销。开发小组的人都认为不应该维护对他们的产品没有用的东西。许多设计方法并不大奏效，这是事实。

7) 记住软件工程的基本原则：所有的问题都可以通过引进一个额外的间接层来简化(Andrew Koenig 向我解释了这一点）。这是抽象方法的基础，而抽象是面向对象编程的首要特征。
8) 使类尽可能地原子化。也就是每个类有一个单一、清楚的目的。如果我们的类或我们设计的系统过于复杂，就应当将所有复杂的类分解成多个简单的类。
9) 从设计的角度，寻找并区分那些变化和不变的成分。也就是在系统中寻找那些修改时不需要重新设计的成分，把它们封装到一个类中。
10) 注意不同点。两个语义上不同的对象可能有同样的操作或反应，自然就会试着把一个作为另一个的子类以便利用继承性的好处。这就叫差异，但并没有充分的理由来强制这种并不存在的父子关系。一个好的解决办法是产生一个共同的父类：它包含两个子类——这可能要多占一点空间，但我们可以从继承中获益，并且可能对这种自然语言的解有一个重要发现。
11) 注意在继承过程中的限制。最清晰的设计是向被继承者加入新的功能，而如果在继承过程删除了原有功能，而不是加入新功能，那这个设计就值得怀疑了。但这也不是绝对的，如果我们正在与一个老的类库打交道，对已有的类在子类中进行限制可能更有效，而不必重建一套类层次来使我们的新类适应新的应用。
12) 不要用子类去扩展基类的功能。如果一个类接口部分很关键的话，应当把它放在基类中，而不是在继承时加入。如果我们正在用继承来添加成员函数，我们可能应该重新考虑我们的设计。
13) 一个类一开始时接口部分应尽可能小而精。在类使用过程中，我们会发现需要扩展类的接口。然而一个类一旦投入使用，我们要想减少接口部分，就会影响那些使用了该类的代码，
14) 但如果我们需要增加函数则不会有影响，一切正常，只需重新编译一下即可。但即使用新的成员函数取代了原来的功能，也不要去改正原有接口（如果我们愿意的话，可以在低层将两个函数合并）。如果我们需要对一个已有的函数增加参数，我们可以让原来的参数保持不变，把所有新参数作为缺省参数，这样不会妨碍对该函数已有的调用。
15) 大声朗读我们的类，确保它们是合理的。读基类时用“is-a”，读成员对象时用“has-a”。
16) 在决定是用继承还是用组合时，问问自己是不是需要向上映射到基类。如果不需要，就用组合（成员对象）而不用继承。这样可以减少多重继承的可能。如果我们选择继承，用户会认为他们被假设向上映射。
17) 有时我们为了访问基类中的 p r o t e c t e d 成员而采用继承。这可能导致一个可察觉的对多重继承的需求。如果我们不需要向上映射，首先导出一个新类来完成保护成员的访问，然后把这个新类作为一个成员对象，放在需要用到它的所有对象中去。
18) 一个典型的基类仅仅是它的派生类的一个接口。当我们创建一个基类时，缺省情况下让成员函数都成为纯虚函数。析构函数也可以是纯虚函数（强制派生类对它重新定义），但记住要给析构函数一个函数体，因为继承关系中所有的析构函数总是被调用。当我们在类中放一个虚函数时，让这个类的所有函数都成

为虚函数，并在类中定义一个虚析构函数。当我们要求高效时再把 virtual 关键词去掉，这种方法防止了接口的行为出格。

19) 用数据成员表示值的变化，用虚函数表示行为的变化。如果我们发现一个类中有几个状态变量和几个成员函数，而成员函数在这些变量的作用下改变行为，我们可能要重新设计它，用子类和虚函数来区分这种不同的作用。
20) 如果我们必须做一些不可移植的事，对这种服务做一个抽象并将它定位在一个类的内部，这个额外的间接层可防止这种不可移植性影响我们的整个程序。
21) 尽量不用多重继承。这可帮我们摆脱困境，尤其是修复我们无法控制的类的接口。除非我们是一个经验相当丰富的程序员，否则不要在系统中设计多重继承。
22) 不要用私有继承。虽然 C + +中可以有私有继承，而且似乎在某些场合下很有用，但它和运行时类型识别一起使用时，常常引起语义的模棱两可。我们可以用一个私有成员对象来代替私有继承。
23) 运算符重载仅仅是“语法糖”：另一种函数调用方法。如果重载一个运算符不会使类的接口更清楚、更易于使用，就不要重载它。一个类只创建一个自动类型转换运算符，一般情况下，重载运算符应遵循第 11 章介绍的原则和格式。
24) 首先保证程序能运行，然后再考虑优化。特别是，不要急于写内联函数、使一些函数为非虚函数或者紧缩代码以提高效率。这些在我们开始构建系统时都不用考虑。我们开始的目标应该是证明设计的正确性，除非设计要求一定的效率。
25) 不要让编译器来为我们产生构造函数、析构函数或“ =”运算符。这些是训练我们的机会。类的设计者应该明确地说出类应该做什么，并完全控制这个类。如果我们不想要拷贝构造函数或“=”运算符，就把它们声明为私有的。记住，只要我们产生了任何构造函数，就防止了缺省构造函数被生成。
26) 如果我们的类中包含指针，我们必须产生拷贝构造函数、“=”运算符和析构函数，以使类运行正常。
27) 为了减少大项目开发过程中的重复编译，应使用第 3 章介绍的类句柄/Cheshire cat 技术，只有需要提高运行效率时才把它去掉。
28) 避免用预处理器。可以用常量来代替值，用内联函数代替宏。
29) 保持范围尽可能的小，这样我们的对象的可见性和生命期也就尽可能的小。这就减少了错用对象和隐藏难以发现的错误的可能性。比方说，假设我们有一个包容器和一段扫描这个包容器的代码，如果我们拷贝这些代码来用一个新的包容器，我们可能无意间用原有的包容器的大小作为新包容器的边界。如果原来的这个包容器超过了这个范围，就会引起一个编译错误。
30) 避免使用全局变量。尽可能把数据放在类中。全局函数存在的可能性要比全局变量大，虽然我们后来发现一个全局函数作为一个类的静态成员更合适。
31) 如果我们需要声明一个来自库中的类或函数，应该用包含一个头文件的方法。比如，如果我们想创建一个函数来写到 ostream 中，不要用一个不完全类型指定的方法自己来声明 ostream，如 class ostream；这样做会使我们的代码变得很脆弱（比如说 ostream 实际上可能是一个 typedef）。我们可以用头文件的形式，例如：#include <iostream.h>当创建我们自己的类时，如果一个库很大，应提供给用户一个头文件的简写形式，文件中包含有不完全的类型说明（这就是类型名声明），这是对于只需要用到指针的情况（它可以提高编译速度）。
32) 当选择重载运算符的返回值类型时候，要一起考虑串连表达式：当定义运算符“ =”时应记住 x = x。对左值返回一个拷贝或一个引用（返回* this），所以它可以用在串连表达式 A = B = C）中。

33) 当写一个函数时，我们的第一选择是用 const 引用来传递参数。只要我们不需要修改正在被传递进入的对象，这种方式是最好的。因为它有着传值方式的简单，但不需要费时的构造和析构来产生局部对象，而这在传值方式时是不可避免的。通常我们在设计和构建我们的系统时不用注意效率问题，但养成这种习惯仍是件好事。
34) 当心临时变量。当调整完成时，要注意临时创建的对象，尤其是用运算符重载时。如果我们的构造函数和析构函数很复杂，创建和销毁临时对象就很费时。当从一个函数返回一个值时，总是应在 return 语句 return foo(i,j);中调用构造函数来“就地”产生一个对象。这优于 foo x(i,j);return x;前一个返回语句避免了拷贝构造函数和析构函数的调用。
35) 当产生构造函数时，要考虑到异常，在最好的情况下，构造函数不需要引起一个异常，另一种较好的情况：类将只从健壮的类被组合和继承，所以当一个异常产生时它会自动清除它自己。如果我们必须使用一个裸指针，我们应该负责捕获自己的异常，然后在我们的构造函数中释放所有异常出现之前指针指向的资源。如果一个构造函数无法避免失败，最好的方法是抛出一个异常。
36) 在我们的构造函数中只做一些最必要的事情，这不仅使构造函数的调用有较低的时间花费（这中间有许多可能不受我们控制），而且我们的构造函数更少地抛出异常和引起的问题。
37) 析构函数的作用是释放在对象的整个生命期内分配的所有资源，而不仅仅是在创建期间。
38) 使用异常层次，从标准 C + +异常层次中继承并嵌套，作为能抛出这个异常的类中的一个公共类。捕获异常的人然后可以确定异常的类型。如果我们加上一个新的派生异常，客户代码还是通过基类来捕获这个异常。
39) 用值来抛出异常，用引用来捕获异常。让异常处理机制处理内存管理。如果我们抛出一个指向在堆上产生的异常的指针，则异常处理器必须知道怎样破坏这个异常，这不是一种好的搭配。如果我们用值来捕获异常，我们需要额外的构造和析构，更糟的是，我们的异常对象的派生部分可能在以值向上映射时被切片。
40) 除非确有必要，否则不要写自己的类模板。先查看一个标准模板库，然后查问创建特殊工具的开发商。当我们熟悉了这些产品后，我们就可大大提高我们的生产效率。
41) 当创建模板时，留心那些不带类型的代码并把它们放在非模板的基类中，以防不必要的代码膨胀。用继承或组合，我们可以产生自己的模板，模板中包含的大量代码都是类型有关的，因此也是必要的。
42) 不要用 STDIO.H 中的函数，例如 printf( )。学会用输入输出流来代替，它们是安全类型和可扩展类型，而且功能也更强。我们在这上面花费的时间肯定不会白费（参看第 6 章）。一般情况下都要尽可能用 C++中的库而不要用 C 库。
43) 不要用 C 的内部数据类型，虽然 C++为了向后兼容仍然支持它们，但它们不像 C++的类那样强壮，所以这会增加我们查找错误的时间。
44) 无论何时，如果我们用一个内部数据类型作为一个全局或自动变量，在我们可以初始化它们之前不要定义它们。每一行定义一个变量，并同时对它初始化。当定义指针时，把‘ *’紧靠在类型的名字一边。如果我们每个变量占一行，我们就可以很安全地定义它们。这种风格也使读者更易于理解。
45) 保证初始化出现在我们的代码的所有方面。在构造函数初始化表达式表中完成

所有成员的初始化，甚至包括内部数据类型（用伪构造函数调用）。用任何簿记技术来保证没有未初始化的对象在我们的系统中运行。在初始化子对象时用构造函数初始化表达式表常常更有效，否则调用缺省构造函数，并且我们最终调用其他成员函数，也可能是运算符“ =”，为了得到我们想要的初始化。

46) 不要用“foo a=b;”的形式来定义一个对象。这是常常引起混乱的原因。因为它调用构造函数来代替运算符“ =”。为了清楚起见，可以用“ foo a(b);”来代替。这个语句结果是一样的，但不会引起混乱。

47) 使用 C + +中新的类型映射。一个映射践踏了正常的类型系统，它往往是潜在的错误点。通过把 C 中一个映射负责一切的情况改成多种表达清楚的映射，任何人来调试和维护这些代码时，都可以很容易地发现这些最容易发生逻辑错误的地方。

48) 为了使一个程序更强壮，每个组件都必须是很强壮的。在我们创建的类中运用 C + +中提供的所有工具：隐藏实现、异常、常量更正、类型检查等等。用这些方法我们可以在构造系统时安全地转移到下一个抽象层次。

49) 建立常量更正。这允许编译器指出一些非常细微且难以发现的错误。这项工作需要经过一定的训练，而且必须在类中协调使用，但这是值得的。

50) 充分利用编译器的错误检查功能，用完全警告方式编译我们的全部代码，修改我们的代码，直到消除所有的警告为止。在我们的代码中宁可犯编译错误也不要犯运行错误（比如不要用变参数列表，这会使所有类型检查无效）。用 assert() 来调试，但要用异常来处理运行时错误。

51) 宁可犯编译错误也不要犯运行错误。处理错误的代码离出错点越近越好。尽量就地处理错误而不要抛出异常。用最近的异常处理器处理所有的异常，这里它有足够的信息处理它们。在当前层次上处理我们能解决的异常，如果解决不了，重新抛出这个异常。

52) 如果我们用异常说明，用 set_unexpected() 函数安装我们自己的 unexpected() 函数。我们的 unexpected ()应该记录这个错误并重新抛出当前的异常。这样的话，如果一个已存在的函数被重复定义并且开始引起异常时，它不会不引起整个程序中止。

53) 建立一个用户定义的 terminate()函数（指出一个程序员的错误）来记录引起异常的错误，然后释放系统资源，并退出程序。

54) 如果一个析构函数调用了任何函数，这些函数都可能抛出异常。一个析构函数不能抛出异常（这会导致 terminate()调用，它指出一个程序设计错误）。所以任何调用了其他函数的析构函数都应该捕获和管理它自己的异常。

55) 不要自己创建私有数据成员名字“分解”，除非我们有了许多已有的全局值，否则让类和命名空间来为我们做这些事。

56) 如果我们打算在 for 循环结束之后使用一个循环变量，要在 for 控制表达式之前定义这个变量，这样，当 for 控制表达式中定义的变量的生命期被限制在 for 循环之内时，我们的程序依然正确。

57) 注意重载，一个函数不应该用某一参数的值来决定执行哪段代码，如果遇到这种情况，应该产生两个或多个重载函数来代替。

58) 把我们的指针隐藏在包容器类中。只有当我们要对它们执行一个立即可以完成的操作时才把它们带出来。指针已经成为出错的一大来源，当用 new 运算符时，应试着把结果指针放到一个包容器中。让包容器拥有它的指针，这样它就会负责清除它们。如果我们必须有一个游离状态的指针，记住初始化它，最好

是指向一个对象的地址，必要时让它等于 0。当我们删除它时把它置 0，以防意外的多次删除。

59) 不要重载全局 new 和 delete，我们可以在一个类跟随类的基础上去重载它们。重载全局 new 和 delete 会影响整个客户程序员的项目，有些事只能由项目的创建者来控制。当为类重载 new 和 delete 时，不要假定我们知道对象的大小，有些人可能是从我们的类中继承的。用提供的参数，如果我们做任何特殊的事，要考虑到它可能对继承者产生的影响。

60) 不要自我重复。如果一段代码在派生类的许多函数中重复出现，就把这段代码放在基类的一个单一的函数中然后在派生类中调用它。这样我们不仅节省了代码空间，也使将来的修改容易传播。这甚至适用于纯虚函数（见第 14 章）。我们可以用内联函数来提高效率。有时这种相同代码的发现会为我们的接口添加有用的功能。

61) 防止对象切片。实际上用值向上映射到一个对象毫无意义。为了防止这一点，在我们的基类中放入一些纯虚函数。

62) 有时简单的集中会很管用。一个航空公司的“旅客舒适系统”由一系列相互无关的因素组成：座位、空调、电视等等，而我们需要在一架飞机上创建许多这样的东西。我们要创建私有成员并建立一个全部的接口吗？不，在这种情况下组件本身也是公开接口的一部分，所以我们应该创建公共成员对象。这些对象有它们自己的私有实现，所以也是很安全的。