C++学习笔记

目录

0．问题

- 已经有 C，为什么要发明 C++?
- C++的设计遵循哪些原则？
- C++的特征？
- C++与面对对象编程有何关系？
- C++设计时在操作系统方面有什么考虑？
- C++＋与 C 有什么重要差别？试比较 C 与 C++。
- C++对 C 的改进主要体现在哪些方面？
- C++的 class 与 C 的 struct 的区别是什么？
- 没有定义析构函数的类会如何？会自己清除吗？如果会，什么时候清除？
- 如 X:X (int I):i(I) {},构造函数初始化表达式有什么作用?在什么时候必须使用?
- 什么是引用？它与指针有什么区别？
- 使用关键字 const 的好处是什么? 何时需要使用 const ？
- 引进 const 的原因?
- volatile 关键字与 const 关键字有何区别?
- C 中的常量与 C++中的常量有什么区别?
- const int f(const int& a) const ,如此定义的函数有什么特别之处? 三个 const 各起什么作用？
- const 指针和指向 const 的指针有什么区别？如何使用？
- f(const X& x), f(X& x), f(X x)的三种引用方式有什么区别？分别在什么时候使用？
- 在 const 成员函数中改变数据成员有什么方法？
- const 与 volatile 的区别？
- 在类中使用 const 与使用 enum 有什么区别?
- 为什么要引进输入输出流? 原来 C 的标准 I/O 的不足?
- endl 操纵算子起什么作用? 它与 flush 算子有什么不同?
- 输入输出流的类层次结构? 有哪些头文件?分别处理哪些事务?
- 如何在输入输出流中寻找?
- C 和 C++中的库函数 printf 和 cout 等是如何与计算机硬件打交道的？为什么可以从屏幕产生输出？
- 什么是内联函数？内联函数与非内联函数有什么区别？
- 变量用 static 修饰与不用 static 修饰有什么区别。static，extern，auto，register 指定符号的意义?
- 命名空间(namespace)有什么用处？如何使用？
- 哪些运算符不能重载？哪些运算符必须作为成员函数重载，哪些运算符必须作为友元函数重载？
- 对于既能作为成员函数重载又能作为友元函数重载的运算符一般应用哪种方式重载？
- ＝运算符重载有什么特别之处？
- C++中如何实现动态内存分配？较 C 有何优点？
- 在 C++中如何实现 buf[N]的创建？其中 N 为变量。

- C++中是否有主动释放内存的机制？如何实现？
- 值于何时内存会发生泄漏?怎样人工造成内存泄漏?
- 如何用一个程序测试系统可用内存的大小？
- 在堆里创建对象与在栈里创建对象有什么区别？
- 为什么需要多态？
- 虚函数有什么意义？怎样使用？
- 构造函数可以是虚的吗？析构函数可以是虚的吗？
- 什么是抽象类？它与虚函数有什么联系？
- 什么是纯虚函数？纯虚函数有什么用？
- 可以对纯虚函数提供定义吗？为什么？
- 多重继承为什么在 C++中是必须的？
- 多重继承在什么情况下可能有用？
- 怎样避免使用多重继承？
- 何时该使用组合，何时该使用继承？
- 继承后的访问控制将又怎样的改变？
- 什么是包容器？它与模板有何联系？
- 模板使用于什么场合？如何使用？
- C++中 complex 类可以简单地使用吗？
- 何时应该使用异常捕获？捕获所有的还是在适当地方捕获？
- C 与 C++在出错处理方面的不同？
- 编译器是如何实现异常处理的？
- 异常处理类的层次？
- 什么是 RTTI？ RTTI 在什么时候需要使用？怎样使用？

1．C++设计原则[1]

1.1 C++设计目标

Aims

C++ makes programming more enjoyable for serious programmers.

C++ is a general-purpose programming language that

- －is a better C
- －supports data abstraction
- －supports object-oriented programming

1.2 C++设计一般原则

General rules

C++’s evolution must be driven by real problems.

Don’t get involved in a sterile quest for perfection.

C++ must be useful now.

Every feature must have a reasonably obvious implementation.

Always provide a transition path

C++ is a language, not a complete system

Provide comprehensive support for each supported style

Don’t try to force people.

1.3 C++设计支持原则

Design Support rules

Support sound design notions.

Provide facilities fro program organization.

Say what you mean.

All features must be affordable.

It is more important to allow a useful feature than to prevent every misuse.

Support composition of software from separately developed parts.

[1] 直接引自文献【3】，原文引用。

1.4 语言技术规则

Language-technical rules
No implicit violations of the static type system.
Provide as good support for user-defined types as for built-in types.
Locality is good.
Avoid order dependencies.
If in doubt, pick the variant of a feature that is easiest to teach.
Syntax matters(often in perverse ways).
Preprocessor usage should be eliminated.

1.5 低层编程支持规则

Low-level programming support rules
Use traditional (dumb) linkers.
No gratuitous incompatibilities with C.
Leave no room for a lower-level language below C++(except assembler).
What you don't use, you don't pay for(zero-overhead rule).
If in doubt, provide means for manual control.

1.6 关于 C++设计的补充说明

- C++设计保证：
  (1) 一个 C 编译过的程序用 C++也能编译.C++编译器仅有的编译错误和警告源于 C 语言的"漏洞",修正这些需要局部编译.
  (2) 用 C++重新编译,C++编译器不会暗地里改变 C 程序的行为.
- C++不是纯的面对对象的语言.friend 关键字的使用正好说明了这一点。因为它是 C 的超集，所以若不利用其面向对象的特性，很多时候看起来更像一种面向过程的语言。当将 C++当成 C 或 Fortran 使用时，它就是一种面向过程的语言。但 C++之所以是 C++,在于你可以利用它的面向对象特性，进行面向对象设计。C++也并非完全兼容 C 的，比如，class 和 new 在 C 中是合法的标识符，而在 C++中是关键字。又如 x=a//* divide */b 在 C++意为 x=a,在 C 中意为 x=a/b。

2．C++特性

2.1 数据抽象与类

- 数据封装与隐藏实现是面向对象的特征。C++的类(class)整个语言的基础。C++的 class 与 C 中的 struct 的区别（1）是 class 成员默认是 private 的，而 struct 成员默认是 public 的；(2)class 中不仅可包括数据，一般还包括对数据的操作；
- C++的存取控制由三个关键字实现：private, public 和 protected。private 关键字意味着，除了该类型的创建者与类的内部成员函数之外，任何人都不能存取这些成员。private 的权限限制可以由 friend 关键子突破。friend 的存在也说明 C++不是纯的面向对象的语言。public 声明意味着其后声明的所有成员任何人都可以存取。protected 与 private 基本类似，只有一点不同：继承的结构可以访问 protected 成员，但不能访问 private 成员。

2.2 构造函数、析构函数、拷贝构造函数

- 构造函数和析构函数保证正确地初始化和清除对象（编译器将不允许没有调用构造函数与析构函数就直接创建与销毁一个对象），这样可以得到控制与安全。
- 构造函数与析构函数都不必显然定义。如 class X{int i; };没有显然定义构造函数与析构函数，编译器会自动创建一个构造函数与析构函数。
- 缺省构造函数是不带任何参数的构造函数。当编译器需要创建一个对象又不知任何细节时，缺省构造函数就显得非常重要。当没有显然定义构造函数时，编译器将自动创建一个缺省构造函数，若定义了构造函数却没有缺省构造函数，试图不带参数构造对象时会产生编译错误。
- Copy constructor 是形如 X(X&)或 X(const X&)的构造函数，这个构造函数是控制传值方式传递和返回用户定义类型的根本所在。 它可以被用来从现有的类创建新类。当用传值方式传递或返回一个对象时，编译器调用这个拷贝构造函数。因为 C＋编译器默认是进行位拷贝(bitcopy)的，位拷贝方式在类中存在指针时通常是危险的，因为这将使多个对象指向同一片内存区域。因为缺省构造函数是进行位拷贝的，所以要防止位拷贝应自己定义 copy constructor。
- 与 copy constructor 紧密相关的是运算符＝的重载。应能明白何时会调用拷贝构造函数，何时会调用＝运算符重载。看下面的例子：

  ```
  AAA b;
  AAA a=b;      //调用 copy constructor
  a=b;          //调用＝运算符重载
  ```
- 若没有定义＝运算符重载，编译器也会自动创建一个。应记住=默认也是进行位拷贝的。若要防止进行赋值运算，可以将＝重载为 private 函数。若同时声明一个私有拷贝构造函数，则彻底防止按值传递。
- 在定义有指针的数据类型中（如链表、链栈等），一般都要自己定义拷贝构造函数和＝运算符，以防止位拷贝。
- 应掌握构造函数初始化函数表的使用。

```
class fred{
    const int size;
public:
    fred();
};
fred::fred():size(100){}
```

此时用初始化表是唯一初始化 size 的方法.

如果在类中有继承与组合，如果子对象没有缺省构造函数或如果我们想改变某个构造函数的缺省行为，只能使用初始化表达式。因为新类的构造函数不能访问它的子对象的 private 数据成员，所以不能直接地对它们初始化。但应明白对于继承和组合使用初始化表达式时的差别：对继承用类名初始化，对组合用对象名初始化。对内置类型使用初始化表达式不是真正调用构造函数，而是一种伪构造函数调用。

下例说明初始化表达式的使用：

```
class A{
   int i;
public:
    A(int I){i=I;}
    ~A(){}
    void f() const {}
};
class B{
    int i;
public:
    B(int I){i=I;}
    ~B(){}
     void f() const {}
};

class C:public B{
  A a;
public:
     C(int I):B(I),a(I) {}
     ~C(){}
     void f() const{
         a.f();
         B::f();
     }
 };
 main(){
    C c(47);
}
```

2.3 引用与指针

- 引进引用是 C++的特色之一。“Reference is like a constant pointer that is automatically dereferenced by the compiler. A reference holds an address, but you treat it like an object. References are essential for clean syntax with operator overloading, but they also add syntactic convenience for passing and returning objects for ordinary functions. ”
- 应能区别引用和指针。C++中同时存在引用与指针。引用与指针的区别如下：
  （1） 引用在建立时必须被初始化；指针可以在定义后任何时间被初始化。
  （2） 一旦引用被初始化，它不能被改变指向另一个对象；指针可以在任何时间指向其它对象。
  （3） 不能有 NULL 引用。引用总是与一片合法存储区相联系。而指针可以是 NULL 的，而且经常是 NULL 的。
- 用引用作为函数参数是一般应遵循的编程原则，如果不需要改变对象，则加 const, 如是返回参数，则不加 const。这可以大大提高程序的效率。对对象进行拷贝是一件费时费力的事情。使用引用不仅方便，而且清晰。
- 引用不是总可以用指针代替的。因为运算符重载使用的场合，指针与引用的意义不同。如函数 f(X& b, X& c)定义中 a=b-c 可以方便使用。而若定义函数 f(X* b, X* c)，a=&b-&c 与上面的意义完全不同，因为&b-&c 在 C 已经有意义，C++不好去改变它。
- 不管何时在一行里最好只放一个指针定义，且在定义的地方初始化每个指针。可以把*附于数据类型上。但是事实上， *是与标识符结合，而不是类型结合！ int* u=&w, v=0; 等价于 int *u=&w; int v=0; v 不是指针！
- 由于引用的语法的原因,传递一个临时对象给一个带有引用的函数是可能的,但不能传递一个临时对象给带有一个指针的函数—因为它必须清楚地带有地址.所以,通过引用传递会产生一个在 C 中不会出现的新情形:一个总是常量的临时变量,它的地址可以传递给一个函数,但这个函数的参数一定是常量引用.

```
class X();
X f() {return X();}
void g1(X&){}
void g2(const X&) {}
main()
{
  g1(f()); //!error: invalid initialization of non-const refererence of
         //type 'X&' from a temporary  of type 'X'
  g2(f()); //OK!
}
```

2.4 函数重载

- 重载可以使得不同类型参数的函数使用同一函数名，可以使程序更简单明了，也更具有可读性。重

载也是构造函数存在的需要，因为构造函数只能有固定的函数名。

- C++禁止通过返回值重载，只能通过范围和参数重载。
- 函数重载时可以使用缺省参数。缺省参数的使用为程序员提供了很多方便。但应注意：只有参数列表后面的参数才可以是缺省的，一旦开始使用缺省参数，那么这个参数后面的所有参数都必须是缺省的。
- 缺省参数只能放在函数声明中，而不能在定义中。缺省参数可以让声明的参数没有标识符，如声明 void f(int x,int=0,float=1.1)；
- 缺省参数的一个重要应用是在开始定义函数时用了一组参数，而使用一段时间后发现要增加一些参数。这时可以将新增参数作为缺省的参数，而以前编写的代码不需要任何改变。

2.5 常量

- 引进 const 的最初动机是取代预处理器#define 进行值替代。它曾被用于指针，函数变量，返回类型，类对象及成员函数。用＃define 可以避免出现不可思异的数，这对于支持维护代码是很重要的。但是预处理器只做文本替代，它没有类型检查的思想，也没有类型检查工具，所以预处理器的值替代会产生一些小的问题。这些问题可以通过 C＋＋的 const 避免。如果一个值不变，就可使之成为常量(const)，这不仅为意外的更改提供安全措施，也消除了存储和读内存操作，使编译器产生的代码更有效。
- 应掌握 C 与 C++中 const 定义的区别。C 中的常量 const 的意思是“一个不能改变的普通常量”，在 C 中，它总是占用存储而且它的名字是全局符。C 编译器不能把 const 看成一个编译期间的常量。在 C 中，如果写：

  ```
  const bufsize=100;
  char buf[bufsize];
  ```

  将得到错误结果，因为 bufsize 占用存储的某个堤防，所以 C 编译器不知道它在编译时的值。在 C 中，可用 const bufsize; 声明一个常量，这个声明指明在别的地方有存储分配。在 C++中这是不对的！因为 C 默认 const 是外部连接的，C++默认 const 是内部连接的。在 C++中做同样的事情必须使用 extern const bufsize。C++中 const 只在被定义过的文件里才是可见的。定义一个常量时，必须附值给它，除非用 extern 作了说明。
- 应掌握如下定义的函数 const int f(const int& a) const 中三个 const 所起的作用。
  最后一个 const 说明此函数不改变对象，只用在成员函数中。第一个 const 表示返回值是常量。
  第二个 const 表示不改变引用的值。注意：构造函数和析构函数都不能定义为 const 成员函数！
- 应掌握 const 指针和指向 const 指针的区别和使用。

  const int* x; int const* y ;都是指指向 const int 的指针。
  int* const z=&d; 是指指向 int 的 const 指针。
  const int* const w=&d; int const* const w=&d; 都指指向 const int 的 const 指针。
  下面的程序及错误提示说明 const 指针和指向 const 指针的区别：

  ```
  int i=23;
  const int* px=&i;
  cout<<”px=&i=”<<px<<” px=”<<*px<<endl;
  ```

```
//output: px=&i=0xbfe31478 *px=23
i++;   //legal
cout<< " px=&i= " <<px<< "   *px= " <<*px<<endl;
//output: px=&i=0xbfe31478 *px=24
px++;     //legal since px is not a constant point
cout<< " px=&i= " <<px<< "   *px= " <<*px<<endl;
 //output: px=&i=0xbfe3147c *px=10026996
 (*px)++;  //!illegal.  since px point to const  int.
 //error:increment of read-only location
 int* const  py; //!illegal. uninitialized const 'py'
 int* const  py=&i;
 cout<< " py=&i= " <<px<< "   *py= " <<*px<<endl;
 //output: py=&i=0xbfe3147c *py=24
 py++;// !illegal. increment of const 'py'
 (*py)++;  //legal
 cout<< " py=&i= " <<px<< "   *py= " <<*px<<endl;
 //output: py=&i=0xbfe3147c *py=25
 const int* const pz=&i;
 pz++; //!illegal
(*pz)++;  //!illegal
 const int j=345;
 px=&j;    //ok!
 py=&j;   //!illegal. py is constant
 int* pw=&j;  //!illegal!
       //error: invalid conversion from 'const int*' to 'int*'
int* pw=(int*)&j;  //legal but bad practice.  强制类型转换！
```

- 应掌握 f(const X& x)，f(X& x)，f(X x)的三种引用方式的区别和使用。
  (1) f(const X& x)与使用值传递是一样的，但是此方式比值传递更加有效，尤其在使用一个占用较大空间的类对象来说。这种方式不会建立类对象的拷贝，因此省内存。由于使用 const,类对象在 f 运行期间不会被改变。
  (2) f(X& x)同样使用引用方式，同样省内存，但是原类对象可能被改变，除非要刻意改变原类对象，不要这样使用。但需要返回多个参数值时要如此使用。
  (3) f(X x) 将建立一个 X 的拷贝，只适用于需要使用值传递的场合。
- 应掌握 在 const 成员函数中改变数据成员有什么方法。有两种方法：一种是使用类指针强制转换 const 类型，如((Y*)this)->j++; 第二种方法是要改变的数据定义为 mutable，这是更好的方法。
- 应掌握 const 与 mutable、volatile 的区别。mutable 数据成员在 const 对象中可以改变。volatile 对象可以在编译器之外被改变。
- C++中的 const 默认是内部连接，const 可用于集合，但编译器不能把一个集合存放在它的符号表里，所以必须分配内存，此时，const 意味着不能改变的一块存储。然而，其值在编译时不能使用，

因为编译器在编译时不需要知道存储的内容。

- 不能把一个 const 对象的地址赋给一个非 const 指针，因为可能通过被赋值指针改变这个 const 对象。当然总能通过强制类型转换强制进行这样的赋值。C++有助于防止错误发生，但程序员自己打破这种安全机制，它也无能为力。
- 应掌握在类中使用 const 与 enum 的区别。

```
class  bob{
      const int size=100; //illegal
      static const int size=100; //OK!
      int array[size];
      enum{ size1=1000;}
     int  array1[size1];
}
```

在类中使用 const 必须用 static;使用 enum 不会占用对象中的存储空间,因为枚举对象在编译时全部求值。另外，C＋＋中的枚举有比 C 更严格的类型检查。

- 关键字 const 能将对象、函数参数、返回值及成员函数定义为常量，并能消除预处理器值替代而不对预处理器有任何影响。所有这些都为程序设计提供了非常好的类型检查形式以及安全性。使用所谓的 const correctness(在可能的任何地方使用 const)已成为项目的救星。

2.6 输入输出流

- 应掌握原来 C 的标准 I/O 的不足和引进输入输出流的原因。

  标准 C I/O 的不足：

  (1) 即使用解释程序的一部分功能,所有的东西都将被装载；

  (2) 解释发生在运行期间,编译时不能对其求值；

  (3) 没有编译时的错误检查；

  (4) printf()族是不能扩展的,不容易处理 C++中新增的数据类型。

  C++引进输入输出流正可弥补 C 输入输出函数的不足。
- 掌握各种文件打开方式：

  1) ios::in 打开一个输入文件,用这个标志作为 ifstream 的打开方式,以防止截断一个现成件；

  2) ios::out 打开一个输出文件,当用于一个没有 ios::app, ios::ate, ios::in 的 ofstream 时,ios::trunc 被隐含.

  3) ios::app 追加方式.

  4) ios::ate 打开一个现成文件(不论是输入还是输出)并寻找末尾.

  5) ios::nocreate 仅打开一个存在的文件,否则失败。

  6) ios::noreplace 仅打开一个不存在的文件.

  7) ios::trunc 如果一个文件存在,打开并删除旧的文件.

  8) ios::binary 打开一个二进制文件,缺省的是文本文件.
- 掌握在输入输出流中查找的方法：

  1) 流定位 用 streampos tellg()(输入流), tellp()(输出流)函数。

  2) 相对查找: seekp(n,direction), seekg(n, direction)

n 为移动的字节数;
direction 为查找方向， ios::beg 从流的开始位置
ios::cur 从流的当前位置
ios::end 从流的末尾位置.

- 掌握 endl 与 flush 操纵算子的区别。
  endl 算子的作用: 插入一新行并清空流(消除所有存储在内部缓冲区里还没有输出的字符)，
  flush 只清空流。
- 应熟练掌握<iostream> <fstream> <iomanip>主要的对象和操纵算子：cin, cout, cerr, clog, ofstream, ifstream 等。
- 掌握最初<strstrea.h>头文件中 istrstream 和 ostrstream 在标准 C++中的替代。GNU C++是用<sstream>中定义的 istringstream 和 ostringstream 代替。不同的编译器可能有不同。

2.7 内联函数

- 应明白为什么要在 C++中引进内联函数。C 中保护效率的一个方法是使用预处理器宏。但是 C++中使用存在两个问题：一是宏看起来像函数调用，但并不总是如此，这隐藏了难以发现的错误（C 中也有此问题）。二是预处理器不容许存取私有数据，这意味着预处理器在用作成员函数时变得无用。为克服这两个问题，C++引进了 inline function。
- 内联函数相当于宏，在编译阶段转化，不会有函数调用，因此也不会有参数压栈、生成汇编语言的 CALL、返回参数、执行汇编语言的 RETURN 的时间花费。
- 应明白在类中定义的函数自动成为内联函数（**因此决不要将不想内联的函数放在函数体中定义**），但也可以使用 inline 关键字放在类外定义的函数前面使之成为内联函数。但为使之有效，必须使函数体和声明一起，否则，编译器将它作为普通函数对待。
- 应明白内联的目的是为了减少函数调用的开销，只有极小的函数才应该使用内联，将大函数内联将造成代码膨胀。
- 存取函数是使用内联函数的好地方，这种函数的函数体只有一个语句。使用内联函数避免将某些数据定义为公有成员。
- 并非所有定义为 inline 的函数都会真正成为内联。假如函数太复杂，编译器将可能不执行内联，这取决于特定编译器。
- 预处理在某些地方仍然是有用的：字符串定义／字符串串联／标志粘贴。

```
#define DEBUG(X) cout<<#X ”= ” <<X<<endl
#define TRACE(S) cout<<#S<<endl;S
#define FIELD(A) char* A##_string; int A##_size
FIELD(one);
```

2.8 命名控制

- static 有两种含义:
  1)在固定的地址上分配,即静态存储.

2)对一个特定的编译单位来说是本地的.这里 static 控制名子的可见性.此时它与 extern 是相对的.所有的全局变量都是隐含的静态存储类．函数则默认为 extern.

extern:表示这个名子对所有的编译单位都是可见的。

auto：告诉编译器这是一个局部变量。(几乎多余)

register: 告诉编译器这是经常用到的局部变量。

- 掌握函数内部的静态对象和类内的静态变量的使用。

1）函数内部 static: 这个对象将存储在程序的静态数据区中，而不是在堆栈中，以后在多次函数调用时保存它的值。

2）类的静态成员：属于类（动物），而非属于某一对象(一只老虎)。所有对象的静态数据成员都共享这一静态存储空间，这为对象间提供了互相通信的方法！任何预定义类型都可以成为静态数据成员，而用 enum 只能用整数值。

- 使用 static 的另一重要场合是控制连接。一般情况下，在文件范围内的所有名字（即不嵌套在类和函数中的名字）对程序中的所有单元都是可见的，即所谓的外部连接。外部的编译单位、全局变量、和普通函数都是外部连接。在文件范围内，一个被明确声明为 static 的对象或函数的名字对编译单元来说是局部变量；这些名字有内部连接。内部连接的一个好处是这个名字可以放在一个头文件中而不用担心连接时发生冲突。
- 可以创建静态成员函数，它为类的全体服务而不是为一个类的对象服务。这样就不需要定义一个全局函数，减少了全局和局部名字空间的占用。静态成员函数不能访问一般的数据成员，它只能访问静态数据成员，也只能调用其它的静态成员函数。
- 命名空间(namespace)的作用是控制 C++名字。

1)namespace 只能在全局范围内定义，但它们之间可以相互嵌套。

2)namespace 定义结尾，不必加分号。

3）一个 namespace 可以在多个文件中用一个标识符定义。

4）可以使用别名。

- 掌握命名空间的使用: using namespace X;
- 使用 using namespace X 语句,并不是从现在开始所有定义的变量都加到 X namespace 中,它只是使 X namespace 中的名子可以直接使用。所以可以使用多个 using namespace 语句。若变量使用没有歧义，可以直接使用，若有歧义，必须指明是哪个命名空间。C++中变量定义默认是局域的。
- 全局变量构成一个特殊的命名空间：无名命名空间。其变量可以通过::globalVariableName 指定。
- 标准 C++定义了一个特殊命名空间 std,从而使头文件的定义方式有了改变。原来引进头文件，一般用#include <iostream.h>，标准 C++使用#include <iostream>,但必须加上语句 using namespace std;才能使用 cout,cin 等对象。

2.9 运算符重载

- 运算符重载可以使编程变得容易，比如直观地实现矩阵相加、相乘等。
- 不能重载的运算符有：成员选择运算符“.”，成员逆向引用运算符“.*”,没有求幂运算符，也不存在用户定义的运算符。
- 仅允许作为成员函数重载的运算符：=、（）、[]、->。[]运算符重载时需要单个参数。
- 仅能作为友元函数重载的运算符：<<、>>，因为左侧运算符是别的对象，不能用成员函数进行重载。

```
friend ostream& operator<<(ostream& os, const MyClass& myclass);
friend istream& oprator>>(istream& is, MyClass& myclass);
```

注意都使用引用引入自定义类作为参数，但第一个加 const,第二个不加。

- 尽管大多数运算符既可以用成员函数也可以用友元函数方式重载。但是建议所有的一元运算符和二元运算符“ +=、-=、/=、*=、^=、&=、|=、%=、>>=、<<= ”用成员函数方式重载，而其余的二元运算符用友元函数方式重载。因为这样的程序可读性更强（其余二元运算符的两个参数的地位是相等的）。使用友元函数重载还可以让运算符转换任一个参数。
- 注意前缀++和后缀++(++a，a++)运算符重载的区别，后缀++中加一个哑 int 参数加以区别。下面是通常的定义方式：

```
class X{
    …
    X& operator++() ;    //prefix: no argument
    X operator++(int);    //postfix, because of the argument
};
```

- 运算符重载时何时用引用，何时不用引用，何时加 const，何时不加 const，一定要心中有数。
- =、+=、-=、*=、/=运算符重载时一般应检查是否是自赋值，一般用

```
X& operator=(const X& right){
      if(this!=&right){
         …
      }
      return *this;
}
```

来定义。

- 在用成员函数重载运算符时，会经常用到 this 指针，this 是对象自身的指针，*this 则指对象本身。

2.10 内存管理

- C++通过 new 和 delete 在堆上安全创建对象。当用 new 创建一个对象时，它在堆里为对象分配内存并为这块内存调用构造函数。它带有内置的长度计算、类型转换和安全检查。delete 表达式首先调用析构函数，然后释放内存（经常是调用 free()）。delete 只用于删除由 new 创建的对象。如果用 malloc()(或 calloc()或 realloc())创建一个对象，然后用 delete 删除它，这个行为是未定义的。因为大多数缺省的 new 和 delete 实现机制都使用了 malloc()和 free()，所以我们很可能没有调用析构函数就释放了内存。如果正在删除的对象指针是 0，将不会发生任何事情。为此，建议在删除指针后立即把指针赋值为 0 以免多它删除两次。
- 定义数组的方式：foo *fp=new foo[N]; 删除数组：delete[] fp;
- 当运算符 new 找不到足够大的连续内存来安排对象时，一个称为 new_handler 的函数被调用。或者，检查指向函数的指针，如果指针非 0，那么它指向的函数被调用。
- 运算符 new 和 delete 可以被重载，以满足我们的需要。

2.11 继承与组合

- 继承与组合是代码重用的重要方式。应明白何时使用继承，何时使用组合。当是 is-a 关系时用继承，has-a 关系时用组合。
- 私有继承的存在主要了为了语言的完整性，如果没有其它理由，则应当减少混淆，所以建议用 private 成员而不是 private 继承。
- 构造函数和析构函数的与众不同之处是每一层函数都被调用。构造在类层次的最根处开始，而在每一层，首先调用基类构造函数，然后调用成员函数构造函数。调用构造函数则严格按照构造函数相反的次序。
- 对于一般的成员函数，只是这个函数被调用，而不是任意基类的版本被调用。如果想调用一般成员函数的基类版本，必须显式地调用。
- 多重继承在 C++中存在的原因，一是 C++不存在 Smalltalk 和 Java 的 Object 基类（所有对象都由此继承），C++可以创建任意个继承树，因此从逻辑完备性来说，必须要求有多重继承性。多重继承存在的另一原因是许多类库（如 ios,istream,ostream,iostream）都是以这种模式构建的。
- 菱形继承方式通常通过定义虚基类的方式实现。一般定义方式如下：
  class AW: public virtual W{}
  class BW: public virtual W{}
  class CW:public AW,public BW {}
  使用虚基类时，most-derived class 有责任对虚基类进行初始化。为了促进 most-derived class 初始化一个虚基类，最好通过创建一个虚基类的缺省构造函数，这样，派生类就不用管虚基类的初始化。
- 一般情况下，应当避免使用多重继承，唯一的例外是使用控制之外的代码。有了模板后，多重继承的重要性大大降低了。MI 是 C++中次要的但更为高级的特性。

2.12 多态与虚函数

- 面向对象编程的三块基石是：数据抽象、继承、多态。封装是通过特性和行为的组合来创建新数据类型的，通过让细节 private 来使得接口与具体实现相隔离。而虚函数则根据类型的不同来实现不同的隔离。多态性在 C++中用虚函数来实现。如果不用虚函数，就等于不懂得 OOP。虚函数帮助实现晚绑定和向下映射。
- 多态的实现在内部是通过 virtual pointer(VPTR)和 virtual table(VTABLE)来实现的。
- 纯虚函数是类似如下定义的类：
  virtual void x()=0;
  纯虚函数使得所在的类成为一个接口，而不是真正的类。
  可以对纯虚函数提供定义。这对于在继承类中重复使用某段代码经常是有用的。允许对纯虚函数提供定义还可以使得从普通函数改为纯虚函数而不改变现有代码。
- 如果使用对象而不是使用地址或引用进行向上映射，则这个对象被”切片”，直到所剩下的是适合目的的子对象。通常要提防这种操作。我们可以通过在基类中放置纯虚函数来防止对象切片。这时如果进行对象切片就将引起编译时的出错信息。
- 虚机制在构造函数中不工作，对于在构造函数中调用一个虚函数的情况，被调用的只是这个函数的本地版本。这样做的理由是显然的：如果构造函数可以调用派生类的构造函数，将会可能操作还没有初始化

的对象，从而导致灾难的发生。

- 构造函数不能是虚的。但是析构函数能够而且常常必须是虚的。析构函数必须拆卸所有可能层次的对象：从最晚派生的类开始，并上升到基类。不定义基类的析构函数为虚的，可能引起内存泄漏。**作为准则，任何时候在类中有虚函数，我们就应该直接增加虚析构函数。这样保证以后不发生意外。**但值得注意的是，在析构函数中，只有成员函数的本地版本被调用，虚机制被忽略。
- 抽象类是至少有一个纯虚函数的类。抽象类实际是一个接口，不能用构造函数直接创建抽象类对象。从抽象类继承的对象或者定义它的基类的纯虚函数，或者再次定义它为纯虚函数。
- 纯虚析构函数必须提供函数体，继承函数不需为虚析构函数提供定义。

2.13 模板

- 模板的起源在于表达参数化的包容器类的愿望。异常起源于提供处理运行时异常的标准方式的愿望。应C++是多重继承的，不能像 Smalltalk 和 Java 那样用动态类型确定和继承来实现包容器类。
- 应明白模板与异常的关系。用 Bjarne Stroustrup 的语言，“ Templates and exceptions are two sides of the same coin：templates allow a reduction in the number of run-time errors by extending the range of problems handled by static type checking; exceptions provide a mechanism for dealing with the remaining run-time errors. Templates make exception handling manageable by reducing the need for run-time error handling to the essential cases. Exceptions make general template-based libraries manageable by providing a way for such libraries to report errors. ”
- 模板参数不受限制，所有的类型检查都推迟到实例化时完成。模板参数也允许非类型参数，如：

```
template <class T, int i> class Buffer{
    T v[i];
    int sz;
public:
    Buffer():sz(i) {}
    //…
};
```

- 应记住模板是一种抽象，模板类的编写者不可能预见到所有今后模板可能使用的情况。因此不要期望使用模板永远不会产生问题。
- 模板包括类模板和函数模板，应掌握其使用方法。

2.14 异常处理

- 错误修复技术的改进是提高代码健壮性的最有效方法之一。C 语言中实现出错处理的方法将用户函数与出错处理程序紧密地结合起来，但这将造成出错处理使用的不方便和难以接受。C++的异常处理使得出错处理程序与通常代码不必紧密结合，而且错误发生不会被忽略。
- 设计异常处理时的假设有：
    1） Exceptions are used primarily for error handling.
    2） Exception handlers are rare compared to function definitions.
    3） Exceptions occur infrequently compared to function calls.
    4） Exceptions are language-level concept－not just implementation, and not an error-handling policy.

- 异常处理的意义：

1) Isn’t intended as simply an alternative return mechanism, but specifically as a mechanism for supporting the construction of fault-tolerant systems.
2) Isn’t intended to turn every function into a fault-tolerant entity, but rather as a mechanism by which a subsystem can be given a large measure of fault tolerance even if its individual functions are written without regard for overall error-handling strategies.
3) Isn’t meant to constrain designers to a single ‘correct’notion of error handling, but to make the language more expressive.

- C++异常处理的理想：

1） Type-safe transmission of arbitrary amounts of information from a throw-point to a handler.
2） No added cost(in time and space) to code that does not throw an exception.
3） A guarantee that every exception raised is caught by an appropriate handler.
4） A way of grouping exceptions so that handlers can be written to catch groups of exceptions as well as individual ones.
5） A mechanism that by default will work correctly in a multi-threaded program.
6） Easy to use.
7） Easy to implement.

- 应明白异常处理不能解决所有的问题。“Trying to provide facilities that allow a single program to recover from all errors is misguided and leads to error-handling strategies so complex that they themselves become a source of errors.”异常不能被保证的情况有：
  （1） 异步事件；
  （2） 普通错误情况；
  （3） 流控制；
  （4） 新异常，老代码。

  如果不必要使用异常，就不要使用。

- 异常是一些类，可以继承。比如：

```
class Matherr{};
class Overflow:public Matherr{};
class Underflow: public Matherr{};
class Zerodivide: public Matherr{};
//…
void g(){
   try{
      f();
    }catch(Overflow){
      //handle
    }
    catch(Matherr){
      //handle
    }
}
```

- 掌握异常抛出，异常捕获和异常处理的一般方法。
- 应掌握捕获所有异常和异常重新抛出的方法：

```
catch(…){
    cout<< “ an exception was thrown  ” <<endl;
    throw;
}
```

- 如果函数实际抛出的异常类型与我们的异常规格说明不一致，会调用特殊函数 unexpected()，可以通过 set_unexpected()函数设置自己的不一致异常处理函数。
- 如果异常未被捕获，特殊函数 terminate()将被自动调用，可以使用标准函数 set_terminate()来安装自己的终止函数 terminate()。
- 掌握常见的标准异常：
  logic_error(domain_error,invalid_argument,length_error,out_of_range,bad_cast,bad_typeid)
  runtime_error(range_error,overflow_error,bad_alloc)
- 异常的典型使用情况包括：
  （1） 把问题固定下来和重新调用这个函数；
  （2） 把事情修补好而继续运行，不去重试函数；
  （3） 计算一些选择结果用于代替函数假定产生的结果。
  （4） 在当前上下文环境尽其所能并且再把同样的异常弹向更高的上下文中。
  （5） 在当前上下文环境尽其所能并且再把不同的异常弹向更高的上下文中。
  （6） 终止程序。
  （7） 包装使用错误方案的函数（尤其是 C 的库函数），以便产生异常替代。
  （8） 简化，假若我们的异常方案建造得过于复杂，使用时会令人懊恼。
  （9） 使我们的库和程序更安全。
- 使用异常处理的建议包括：
  （1） 随时使用异常规格说明；
  （2） 起始于标准异常；
  （3） 套装我们自己的异常；
  （4） 使用异常层次；
  （5） 多重继承；
  （6） 用引用而非值去捕获；
  （7） 在构造函数中抛出异常；
  （8） 不要在析构函数中导致异常；
  （9） 避免无保护的指针。

2.15 运行时类型识别

- Run-time type identification, RTTI,是在我们只有一个指向基类的指针或引用时确定一个对象的准确类型。
- RTTI 机制包括三个部分：
  （1）一个操作符 dynamic_cast，以从基类指针中得到继承类的指针；
  （2）一个操作符 type_id, 以通过基类指针判定对象的确切类型；
  （3）一个结构 type_info,以作进一步运行类型识别使用。type_info 类由 type_id 操作符产生。
- 谨慎使用 RTTI，尽可能使用虚函数，必要时才使用 RTTI。

- 掌握 static_cast, const_cast 和 reinterpret_cast 的使用：
  (1) static_cast：为了行为良好和行为较好而使用的映射；
  (2) const_cast：用于映射常量和变量，用于将 const 转换为非 const，把 volatile 转换为非 volatile.
  (3) Reinterpret_cast：为了映射到一个完全不同的意思，假设对象是一个比特模式，危险，慎用。

3．对参考文献的评价

文献【3】是掌握 C++设计思想的不可多得的好书，里面也有许多 C++形成与演化过程的许多笑话。这本书对设计任何一个复杂系统也非常有借鉴意义。文献【1】【2】可以作为深入学习 C++重要特性的重点阅读书目。【2】与【1】的区别是【2】以标准 C++为基础改写了代码，对于 STL 也有较详细介绍，另外还有设计模板的简要介绍。【1】【2】【3】似乎都不可作为学习 C++的入门教科书，可以选择一种中文书籍作为 C++的入门教材（如钱能的《C++程序设计语言》）。中文书籍的通常优点是简单，便于查阅；缺点通常是不够深入，思想性不强。

对于优秀的 C++程序员而言，应该好好阅读文献【4】。这本书的特色是其中充满了如何写好的 C++程序的建议（每一章后都有 Advice 的总结），如何用 C++进行软件设计的思想贯穿整本书（事实上，第四部分的三章全部在讲如何用 C++进行好的设计）。第三部分较为详细地讲述了 STL 的设计。STL 可以说是 C++的第一个优美的设计，其设计原理可以给一般 C++程序员提供一个好的设计的范例。它对 C++语言的各方面作了最权威、详尽的描述，但它对于初学者而言似乎太难了，阅读书中充满模板的代码对初学者是一个挑战。

4．总结

C++作为更好的 C,其威力在于能与传统系统合作、运行时间与空间效率、类概念的适应性，其弱点在于某些继承 C 的特性，它的特别新的特性（如内在数据库支持）以及由于并非纯的面向对象语言而造成的整个语言的复杂性和不纯性。但是毫无疑问，它将在很长一段时间内对计算机科学起着不可替代的作用，正如 C 语言的经典和不可替代一样。

5．参考文献

【1】 Bruce Eckel 著.刘宗田等译.C++编程思想.北京:机械工业出版社.2000.1
【2】 Bruce Eckel. Thinking in C++ (2nd edition).网络共享版.
【3】 Bjarne Stoustrup. The Design and Evolution of C++.北京:机械工业出版社.2002.1
【4】 Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley,Reading,MA. third edition,1997.

廖海仁(liaohairen@gmail.com)
2006-10-8 于南极中山站，2007-11-2 Revised