面向对象是C++的重要特性.   
但是c++在c的基础上新增加的几点优化也是很耀眼的   
就const直接可以取代c中的#define   
以下几点很重要,学不好后果也也很严重   
  
const   
1. 限定符声明变量只能被读   
  const int i=5;   
  int j=0;   
  ...   
  i=j;  //非法，导致编译错误   
  j=i;  //合法   
2. 必须初始化   
  const int i=5;    //合法   
  const int j;      //非法，导致编译错误   
3. 在另一连接文件中引用const常量   
  extern const int i;    //合法   
  extern const int j=10;  //非法，常量不可以被再次赋值   
4. 便于进行类型检查   
  用const方法可以使编译器对处理内容有更多了解。   
  #define I=10   
  const long &i=10;  /\*dapingguo提醒：由于编译器的优化，使   
      得在const long i=10; 时i不被分配内存，而是已10直接代入   
      以后的引用中，以致在以后的代码中没有错误，为达到说教效   
      果，特别地用&i明确地给出了i的内存分配。不过一旦你关闭所   
      有优化措施，即使const long i=10;也会引起后面的编译错误。\*/   
  char h=I;      //没有错   
  char h=i;      //编译警告，可能由于数的截短带来错误赋值。   
5. 可以避免不必要的内存分配   
  #define STRING "abcdefghijklmn\n"   
  const char string[]="abcdefghijklm\n";   
  ...   
  printf(STRING);  //为STRING分配了第一次内存   
  printf(string);  //为string一次分配了内存，以后不再分配   
  ...   
  printf(STRING);  //为STRING分配了第二次内存   
  printf(string);   
  ...   
  由于const定义常量从汇编的角度来看，只是给出了对应的内存地址，   
  而不是象#define一样给出的是立即数，所以，const定义的常量在   
  程序运行过程中只有一份拷贝，而#define定义的常量在内存中有   
  若干个拷贝。   
6. 可以通过函数对常量进行初始化   
  int value();   
  const int i=value();   
  dapingguo说：假定对ROM编写程序时，由于目标代码的不可改写，   
  本语句将会无效，不过可以变通一下：   
  const int &i=value();   
  只要令i的地址处于ROM之外，即可实现：i通过函数初始化，而其   
  值有不会被修改。   
7. 是不是const的常量值一定不可以被修改呢？   
  观察以下一段代码：   
  const int i=0;   
  int \*p=(int\*)&i;   
  p=100;   
  通过强制类型转换，将地址赋给变量，再作修改即可以改变const常量值。   
8. 请分清数值常量和指针常量，以下声明颇为玩味：   
  int ii=0;   
  const int i=0;            //i是常量，i的值不会被修改   
  const int \*p1i=&i;        //指针p1i所指内容是常量，可以不初始化   
  int  \* const p2i=&ii;    //指针p2i是常量，所指内容可修改   
  const int \* const p3i=&i; //指针p3i是常量，所指内容也是常量   
  p1i=&ii;                  //合法   
  \*p2i=100;                //合法   
关于C++中的const关键字的用法非常灵活，而使用const将大大改善程序的健壮性，参考了康建东兄的const使用详解一文，对其中进行了一些补充，写下了本文。   
  
  
1.      const常量，如const int max = 100;    
优点：const常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查，而对后者只进行字符替换，没有类型安全检查，并且在字符替换时可能会产生意料不到的错误（边际效应）   
  
2.      const 修饰类的数据成员。如：   
class A   
  
{   
  
    const int size;   
  
    …   
  
}   
  
const 数据成员只在某个对象生存期内是常量，而对于整个类而言却是可变的。因为类可以创建多个对象，不同的对象其const数据成员的值可以不同。所以不能在类 声明中初始化const数据成员，因为类的对象未被创建时，编译器不知道const 数据成员的值是什么。如   
  
class A   
  
{   
  
const int size = 100;    //错误   
  
int array[size];        //错误，未知的size   
  
}   
  
const数据成员的初始化只能在类的构造函数的初始化表中进行。要想建立在整个类中都恒定的常量，应该用类中的枚举常量来实现。如   
  
class A   
  
{…   
  
enum {size1=100, size2 = 200 };   
  
int array1[size1];   
  
int array2[size2];   
  
}   
  
枚举常量不会占用对象的存储空间，他们在编译时被全部求值。但是枚举常量的隐含数据类型是整数，其最大值有限，且不能表示浮点数。   
  
3.      const修饰指针的情况，见下式：   
  
int b = 500;   
const int\* a = &          [1]   
int const \*a = &          [2]   
int\* const a = &          [3]   
const int\* const a = &    [4]   
如 果你能区分出上述四种情况，那么，恭喜你，你已经迈出了可喜的一步。不知道，也没关系，我们可以参考《Effective c++》Item21上的做法，如果const位于星号的左侧，则const就是用来修饰指针所指向的变量，即指针指向为常量；如果const位于星号的 右侧，const就是修饰指针本身，即指针本身是常量。因此，[1]和[2]的情况相同，都是指针所指向的内容为常量（const放在变量声明符的位置无 关），这种情况下不允许对内容进行更改操作，如不能\*a = 3 ；[3]为指针本身是常量，而指针所指向的内容不是常量，这种情况下不能对指针本身进行更改操作，如a++是错误的；[4]为指针本身和指向的内容均为常 量。   
  
4.    const的初始化   
先看一下const变量初始化的情况   
1) 非指针const常量初始化的情况：A b;   
const A a = b;   
2) 指针const常量初始化的情况：   
  
A\* d = new A();   
const A\* c = d;   
或者：const A\* c = new A();   
3）引用const常量初始化的情况：   
A f;   
const A& e = f;      // 这样作e只能访问声明为const的函数，而不能访问一              
  
般的成员函数；   
    [思考1]： 以下的这种赋值方法正确吗？   
    const A\* c=new A();   
    A\* e = c;   
    [思考2]： 以下的这种赋值方法正确吗？   
    A\* const c = new A();   
    A\* b = c;   
  
5.    另外const 的一些强大的功能在于它在函数声明中的应用。在一个函数声明中，const 可以修饰函数的返回值，或某个参数；对于成员函数，还可以修饰是整个函数。有如下几种情况，以下会逐渐的说明用法：A& operator=(const A& a);   
void fun0(const A\* a );   
void fun1( ) const; // fun1( ) 为类成员函数   
const A fun2( );   
  
1） 修饰参数的const，如 void fun0(const A\* a ); void fun1(const A& a);   
调 用函数的时候，用相应的变量初始化const常量，则在函数体中，按照const所修饰的部分进行常量化，如形参为const A\* a，则不能对传递进来的指针的内容进行改变，保护了原指针所指向的内容；如形参为const A& a，则不能对传递进来的引用对象进行改变，保护了原对象的属性。   
[注意]：参数const通常用于参数为指针或引用的情况，且只能修饰输入参数;若输入参数采用“值传递”方式，由于函数将自动产生临时变量用于复制该参数，该参数本就不需要保护，所以不用const修饰。   
  
[总结]对于非内部数据类型的输入参数，因该将“值传递”的方式改为“const引用传递”，目的是为了提高效率。例如，将void Func(A a)改为void Func(const A &a)   
  
      对于内部数据类型的输入参数，不要将“值传递”的方式改为“const引用传递”。否则既达不到提高效率的目的，又降低了函数的可理解性。例如void Func(int x)不应该改为void Func(const int &x)   
  
2）  修饰返回值的const，如const A fun2( ); const A\* fun3( );   
这样声明了返回值后，const按照"修饰原则"进行修饰，起到相应的保护作用。const Rational operator\*(const Rational& lhs, const Rational& rhs)   
{   
return Rational(lhs.numerator() \* rhs.numerator(),   
lhs.denominator() \* rhs.denominator());   
}   
返回值用const修饰可以防止允许这样的操作发生:Rational a,b;   
Radional c;   
(a\*b) = c;   
一般用const修饰返回值为对象本身（非引用和指针）的情况多用于二目操作符重载函数并产生新对象的时候。   
[总结]   
  
1.    一般情况下，函数的返回值为某个对象时，如果将其声明为const时，多用于操作符的重载。通常，不建议用const修饰函数的返回值类型为某个对象或对 某个对象引用的情况。原因如下：如果返回值为某个对象为const（const A test = A 实例）或某个对象的引用为const（const A& test = A实例） ，则返回值具有const属性，则返回实例只能访问类A中的公有（保护）数据成员和const成员函数，并且不允许对其进行赋值操作，这在一般情况下很少 用到。   
  
2.      如果给采用“指针传递”方式的函数返回值加const修饰，那么函数返回值（即指针）的内容不能被修改，该返回值只能被赋给加const 修饰的同类型指针。如：   
  
const char \* GetString(void);   
  
如下语句将出现编译错误：   
  
char \*str=GetString();   
  
正确的用法是：   
  
const char \*str=GetString();   
  
3.    函数返回值采用“引用传递”的场合不多，这种方式一般只出现在类的赙值函数中，目的是为了实现链式表达。如：   
  
class A   
  
{…   
  
A &operate = (const A &other);  //负值函数   
  
}   
A a,b,c;              //a,b,c为A的对象   
  
…   
  
a=b=c;            //正常   
  
(a=b)=c;          //不正常，但是合法   
  
若负值函数的返回值加const修饰，那么该返回值的内容不允许修改，上例中a=b=c依然正确。(a=b)=c就不正确了。   
[思考3]： 这样定义赋值操作符重载函数可以吗？   
const A& operator=(const A& a);   
  
6.    类成员函数中const的使用   
一般放在函数体后，形如：void fun() const;   
任何不会修改数据成员的函数都因该声明为const类型。如果在编写const成员函数时，不慎修改了数据成员，或者调用了其他非const成员函数，编译器将报错，这大大提高了程序的健壮性。如：   
  
class Stack   
  
{   
  
public:   
  
      void Push(int elem);   
  
      int Pop(void);   
  
      int GetCount(void) const;  //const 成员函数   
  
private:   
  
      int m\_num;   
  
      int m\_data[100];   
  
};   
  
int Stack::GetCount(void) const   
  
{   
  
  ++m\_num;              //编译错误，企图修改数据成员m\_num   
  
  Pop();                    //编译错误，企图调用非const函数   
  
  Return m\_num;   
  
}   
  
7.      使用const的一些建议   
1 要大胆的使用const，这将给你带来无尽的益处，但前提是你必须搞清楚原委；   
2 要避免最一般的赋值操作错误，如将const变量赋值，具体可见思考题；   
3 在参数中使用const应该使用引用或指针，而不是一般的对象实例，原因同上；   
4 const在成员函数中的三种用法（参数、返回值、函数）要很好的使用；   
5 不要轻易的将函数的返回值类型定为const;   
6除了重载操作符外一般不要将返回值类型定为对某个对象的const引用;   
[思考题答案]   
1 这种方法不正确，因为声明指针的目的是为了对其指向的内容进行改变，而声明的指针e指向的是一个常量，所以不正确；   
2 这种方法正确，因为声明指针所指向的内容可变；   
3 这种做法不正确；   
在const A::operator=(const A& a)中，参数列表中的const的用法正确，而当这样连续赋值的时侯，问题就出现了：   
A a,b,c:   
(a=b)=c;   
因为a.operator=(b)的返回值是对a的const引用，不能再将c赋值给const常量。

1.1.1. 定义普通常量  
使 用#define来定义常量也是常用方法，但const也可以用来定义常量，在[Effective C++]中建议使用const代替#define来定义常量，因为const定义的常量具有类型信息，而宏没有，所以使用const定义的常量在进行赋值 操作时编译器会进行更严格的类型检查，是类型安全的。

const double PI = 3.1414926;

const int POOL\_SIZE = 20;

定义常量有三种方法：宏、const、enum，其中宏应该尽量避免，而const 与enum也各有优缺点，最大的区别就是enum只能用于定义整数，而不能定义浮点数；而对于定义逻辑关系较近的一组整数时比较适合使用enum，也可以 考虑使用类代替enum(参见[??])。

常量必须在定义时进行初始化，之后便不能再赋值。说它不能被赋值并不是说常量的值是绝对不会改变的，只是说不能直接赋值，但可以通过指针及强制类型转换、const\_cast是可以改变常量的值的。

#include

using namespace std;

int main( void )

{

const int ci = 5;

const int\* cpci = &ci;

int \*pci = (int\*)&ci;

cout<<"cpci = "<

return 0;

}

输出结果：

cpci = 002DFAC8, pci = 002DFAC8

ci=5, \*cpci=1, \*pci=1

ci=5, \*cpci=2, \*pci=2

ci != \*cpci

之所以使用ci直接输出变量的值时显示其值始终没有改变，但通过指针间接显示出来的 值是改变了，而且输出结果的最后一行很奇怪，ci的值与\*cpci的值居然不相等，只因为编译器在编译时进行了优化，将代码中的ci直接替换成了5，与宏 替换是相同的效果，而指针的值则是实际内存中的值。

所以，千万不要试图使用指针强行改变const变量的值，否则程序可能表现出错误的 行为，而且查找起来这种错误非常困难。在gcc 4.3.4和visual C++ 2010中均默认打开了对常量的优化选项，目前还没找到关闭该优化的命令行选项，一定不要自作聪明去改const变量的值。

1.1.2. 修饰指针  
把 const与指针放到一起，很多人便会想到一个绕口令“指针常量与常量指针。“指针常量”即一个指针变量，该变量不能被赋值，而指针指向的内存单元的内容 是可以改变的；“常量指针”即一个指向常量的指针，指针变量本身可以赋值，而指针指向的内存单元的内容是不可以被重新赋值的。

char a = 'A', b = 'B';

const char\* ptoc = &a; // 常量指针

\*ptoc = 'C'; // 改变指针指向内存单元的内容，不可以

ptoc = &b; // 改变指针的值，可以

char\* const cp = &a; // 指针常量

cp = &b; // 改变指针的值，不可以

\*cp = 'D'; // 改变指针指向内存单元的内容，可以

const char\* const cptoc = &a; // 指向常量的指针常量

\*cptoc = 'E'; // 不可以

cptoc = &b; // 不可以

const是修饰类型还是修饰指针，要看const的位置，放在\*前就是修饰数据类型，放到\*后就是修饰指针，const char和char const是一样的。

建议：在不打算修改数据内容的时候都将指针定义成常量指针，不打算指针本身被修改的场合都定义成指针常量。尽可能地多用const，用错了没关系，编译器会提示你的，只要能够编译通过，就不会因为用错const而导致程序逻辑错误，应该说const负作用极小。

1.1.3. 修饰类成员常量  
当使用const修饰类成员变量时便定义了常数据成员，它的使用与使用类外定义的常量本质上并没有什么区别，在这里只想指出一点：有网友提到const数据成员只能被const修饰的函数使用这是没有根据的，是错误的。

1.1.4. 修饰类成员函数  
const修饰成员函数语法：

class Socket

{

public:

typedef unsigned short socket\_port\_t;

socket\_port\_t LocalPort( void ) const

{

++readCount;

return \_port;

}

private:

socket\_port\_t \_port;

mutable int \_readCount;

};

使用const修饰的成员函数不能修改类的成员变量，如成员\_port，而且只能调用成员类对象const函数，但有个例外，就是mutable修饰的成员变量可以在const修饰的成员函数中被修改，如\_readCount。

另外，const只能修饰非静态函数。

建议：将所有不改变对象状态的函数都使用const修饰符标识，以提高程序的可读 性。其实，头文件就是最好的类接口的说明文档，越多的提供信息就能使程序的可读性越好，越利于维护。看到成员函数的const修饰符，读者便立即明白该函 数不会改变程序的状态，这也有利于当程序状态出现异常时的问题定位。

1.1.5. 修饰类对象、对象引用或对象指针（常量指针）  
当 const修饰自定义的类对象时，与修饰C++内置类型的变量的思想是一致的，但稍有不同，除了不能被赋值外，还不能调用没有使用const修饰的非静态 成员函数。当const修饰类对象引用、指针时限制是一样的，因为引用本身与直接使用该变量实质上没有区别，而使用指针只是将.操作符改为了->本 质上还是一样的

const std::string hello = “Hello from Noock Tian;

std::cout<

hello = "Hi"; // 不可以赋值

hello.push\_back("!"); // 不可以

1.1.6. 修饰函数参数  
const修饰函数的例子是很常见了，表示函数的参数在函数体内不会被意外修改，一般用于修饰输入参数，例如标准库中的字符串连接函数。str1是输出参数，其内容会被修改，而str2为输入参数，其内容不会修改。

char\* strcat(char\* str1, const char\* str2);

实际上在说到const用法一开始就提到，const只是一种声明，但并不能保证， 例如strcat函数虽然声明了str2为const char\*型，但并不能保证内部绝对不会修改str2的内容。但const从语言本身提供了一种编写自描述性代码的方法，只要使用函数与实现函数的双方都 达成一致的约定，按照契约编程，我们就可以认为const修饰的类型在函数体内不会被修改，这与const修饰类成员函数一样，可以提高软件的可读性。

1.1.7. 修饰函数返回值  
const 可以用于修饰任何类型，只要返回值类型不是void，const就可以用来修饰返回值的类型。但实际上const用于修饰非引用的返回值类型是没有意义 的，因为返回值一般都会被赋值给另一个变量，此时用于传递返回值的对象已经被销毁，修饰返回值类型的const的作用也就终结了。

当返回值是引用类型时，如果该引用的值不希望被修改是可以声明为常引用的，例如：

class Socket

{

public:

const string& IP( void ) const{ return \_ip; }

private:

string \_ip;

};

Socket sock;

string& ip = sock.IP(); // 不可以

const string& ip2 = sock.IP(); // 可以

string ip3 = sock.IP(); // Socket::\_ip被复制，可以

此处，为了减少构造临时变量，将IP函数返回值定义为引用类型以提高程序运行效率， 但为了保护内部状态不会被客户端代码意外，返回值使用const修饰为常引用。但是，如果对于软件安全性较高的场合，最好不要定义为引用，因为恶意的客户 端代码是有可能修改Socket::\_ip的值的。

在C++中赋值运算符反默认返回值都是引用，但笔者认为定义为常引用更为合适，例如：

int main( void )

{

int a = 1,b = 2,c = 3,d = 4;

((a=b)=c)=d;

cout<<”a=”<

return 0;

}

输出结果：

a=4, b=2, c=3, d=4

显然，在实际工程中谁也不会写出这样的代码，这段代码却是合法的，无疑这给程序员多了一种出错的可能，如果把赋值运算符的返回值定义为常引用，则会减少程序员出错的机会，例如[??]:

class Object

{

public:

const Object& operator=(const Object& a) { return \*this; }

};

int main( void )

{

Object a, b, c, d;

a = b = c = d; // 可以

((a=b)=c)=d; // 不可以

return 0;

}

在gcc 编译时则会出现错误提示：

error: passing ‘const Object’ as ‘this’ argument of ‘const Object& Object::operator=(const Object&)’ discards qualifiers.   
当然，不同的编译器可能错误提示不同。