一个空类编译器会自动生成哪些函数？ 哪些需要禁止？

当空类Empty_one定义一个对象时Empty_one pt;sizeof(pt)仍是为1，但编译器会生成6个成员函数：一个缺省的构造函数、一个拷贝构造函数、一个析构函数、一个赋值运算符、两个取址运算符。

```
class Empty
{
  public:
    Empty();                                //缺省构造函数
    Empty(const Empty &rhs);                //拷贝构造函数
    ~Empty();                               //析构函数
    Empty& operator=(const Empty &rhs); //赋值运算符
    Empty* operator&();                     //取址运算符
    const Empty* operator&() const;     //取址运算符(const版本)
};
```

对于某些类而言，对象的拷贝或赋值时不合法的，例如定义了一个学生类，但对于学生对象而言，只能有一个，世界上不存在两个一样的学生对象，我们应该明确阻止学生对象之间的拷贝或赋值，也就是说学生类是不支持拷贝或赋值的。
阻止拷贝构造函数及拷贝赋值运算符的生成，下面主要介绍三种：
1、在C++11标准下，将这些函数声明为删除的函数，在函数参数的后面加上=delete来指示出我们定义的删除的函数
2、将这些函数声明为private,并且不提供函数定义
3、将待定义的类成为一个不支持copy的类的子类

如何限制一个类对象只在栈（堆）上分配空间？

在C++中，类的对象建立分为两种，一种是静态建立，如A a；另一种是动态建立，如A* ptr=new A；这两种方式是有区别的。
静态建立类对象：
是由编译器为对象在栈空间中分配内存，是通过直接移动栈顶指针，挪出适当的空间，然后在这片内存空间上调用构造函数形成一个栈对象。使用这种方法，直接调用类的构造函数。
动态建立类对象：
是使用new运算符将对象建立在堆空间中。这个过程分为两步，第一步是执行operator new()函数，在堆空间中搜索合适的内存并进行分配；第二步是调用构造函数构造对象，初始化这片内存空间。这种方法，间接调用类的构造函数。
只能在堆上分配类对象
当对象建立在栈上面时，是由编译器分配内存空间的，调用构造函数来构造栈对象。当对象使用完后，编译器会调用析构函数来释放栈对象所占的空间。编译器管理了对象的整个生命周期。如果编译器无法调用类的析构函数，情况会是怎样的呢？比如，类的析构函数是私有的，编译器无法调用析构函数来释放内存。所以，编译器在为类对象分配栈空间时，会先检查类的析构函数的访问性，其实不光是析构函数，只要是非静态的函数，编译器都会进行检查。如果类的析构函数是私有的，则编译器不会在栈空间上为类对象分配内存。
因此，将**析构函数设为私有**，类对象就无法建立在栈上了。

```
class A
{
public:
    A(){}
    void destory(){delete this;}
private:
    ~A(){}
};
```

试着使用A a;来建立对象，编译报错，提示析构函数无法访问。这样就只能使用new操作符来建立对象，构造函数是公有的，可以直接调用。类中必须提供一个destory函数，来进行内存空间的释放。类对象使用完成后，必须调用destory函数。

上述方法的缺点：

一、无法解决继承问题。

如果A作为其它类的基类，则析构函数通常要设为virtual，然后在子类重写，以实现多态。

因此析构函数不能设为private。

还好C++提供了第三种访问控制，protected。

将析构函数设为protected可以有效解决这个问题，类外无法访问protected成员，子类则可以访问。

二、类的使用很不方便，

使用new建立对象，却使用destory函数释放对象，而不是使用delete。

(使用delete会报错，因为delete对象的指针，会调用对象的析构函数，而析构函数类外不可访问。这种使用方式比较怪异。)

为了统一，可以将构造函数设为protected，然后提供一个public的static函数来完成构造，这样不使用new，而是使用一个函数来构造，使用一个函数来析构。

```
class A
{
protected:
    A(){}
    ~A(){}
public:
    static A* create()
    {
        return new A();
    }
    void destory()
    {
        delete this;
    }
};
```

这样，调用create()函数在堆上创建类A对象，调用destory()函数释放内存。

只能在栈上分配类对象

只有使用new运算符，对象才会建立在堆上，因此，只要禁用new运算符就可以实现类对象只能建立在栈上。

虽然你不能影响new operator的能力（因为那是C++语言内建的），但是你可以利用一个事实：new operator 总是先调用 operator new，而后者我们是可以自行声明重写的。

因此，将operator new()设为私有即可禁止对象被new在堆上。

```
class A
{
private:
    void* operator new(size_t t){}      // 注意函数的第一个参数和返回值都是固定的
    void operator delete(void* ptr){} // 重载了new就需要重载delete
public:
    A(){}
    ~A(){}
};
```

类静态成员函数的特点、静态成员函数可以是虚函数吗、静态成员函数可以是const函数吗？

它为类的全部服务，而不是为某一个类的具体对象服务。静态成员函数与静态数据成员一样，都是在类的内部实现，属于类定义的一部分。普通的成员函数一般都隐藏了一个this指针，this指针指向类的对象本身，因为普通成员函数总是具体的属于某个类的具体对象的。通常情况下，this指针是缺省的、但是与普通函数相比，静态成员函数由于不是与任何的对象相联系，因此它不具有this指针，从这个意义上讲，它无法访问属于类对象的非静态数据成员，也无法访问非静态成员函数，它只能调用其余的静态成员函数。

特点：

1.出现在类体外的函数不能指定关键字static；

2.静态成员之间可以互相访问，包括静态成员函数访问静态数据成员和访问静态成员函数；

3.非静态成员函数可以任意地访问静态成员函数和静态数据成员；

4.静态成员函数不能访问非静态成员函数和非静态数据成员

5.由于没有this指针的额外开销，因此静态成员函数与类的全局函数相比，速度上会有少许的增长

6.调用静态成员函数，可以用成员访问操作符(.)和(->)为一个类的对象或指向类对象的指调用静态成员函数。

不能为虚函数。

1. static成员不属于任何类对象或类实例，所以即使给此函数加上virutal也是没有任何意义的。
2. 静态与非静态成员函数之间有一个主要的区别。那就是静态成员函数没有this指针。
   虚函数依靠vptr和vtable来处理。vptr是一个指针，在类的构造函数中创建生成，并且只能用this指针来访问它，因为它是类的一个成员，并且vptr指向保存虚函数地址的vtable.

对于静态成员函数，它没有this指针，所以无法访问vptr. 这就是为何static函数不能为virtual.
虚函数的调用关系：this -> vptr -> vtable ->virtual function

不能为**const**。
静态成员函数是属于类的，而不是某个具体对象，在没有具体对象的时候静态成员就已经存在，静态成员函数不会访问到非静态成员，也不存在this指针。而成员函数的const就是修饰this指针的，既然静态成员函数不会被传递this指针，那const自然就没有必要了

不能为**volatile**
volatile关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。声明时语法：int volatile vInt; 当要求使用 volatile 声明的变量的值的时候，系统总是重新从它所在的内存读取数据，即使它前面的指令刚刚从该处读取过数据。而且读取的数据立刻被保存。
与const同理

this指针

this指针指向类的某个实例（对象），叫它当前对象。在成员函数执行的过程中，正是通过“This指针”才能找到对象所在的地址，因而也就能找到对象的所有非静态成员变量的地址。

友元函数

为什么要有？

友元函数是一个不属于类成员的函数，但它可以访问该类的私有成员。换句话说，友元函数被视为好像是该类的一个成员。友元函数可以是常规的独立函数，也可以是其他类的成员。实际上，整个类都可以声明为另一个类的友元。

为了使一个函数或类成为另一个类的友元，必须由授予它访问权限的类来声明。类保留了它们的朋友的"名单"，只有名字出现在列表中的外部函数或类才被授予访问权限。通过将关键字 friend 放置在函数的原型之前，即可将函数声明为友元。

使用友元函数的优缺点

优点：能够提高效率，表达简单、清晰。
缺点：友元函数破坏了封装机制，尽量不使用成员函数，除非不得已的情况下才使用友元函数。

友元函数的使用

可以直接调用友元函数，不需要通过对象或指针;此外，友元函数没有this指针，则参数要有三种情况：
1、要访问非static成员时，需要对象做参数；
2、要访问static成员或全局变量时，则不需要对象做参数；
3、如果做参数的对象是全局对象，则不需要对象做参数.

```
class Box{
        double width;  // 默认是private
        public:
            double length;
            friend void printWidth(Box box);  // 友元函数声明
            friend class BigBox;  // 友元类的声明
            void setWidth(double wid);
    };

    // 成员函数的定义
    void Box::setWidth(double wid){
        width = wid;
    }
    // 友元函数的定义
    // 请注意：printWidth() 不是任何类的成员函数!!!
    void printWidth(Box box){
        /* 因为printWidth()是Box的友元函数，它可以直接访问该类的任何成员 */
        cout << "Width of Box: " << box.width << endl;
    }

    // 友元类的使用
    class BigBox{
        public:
            void Print(int width, Box &box){
                // BigBox是Box类的友元类，它可以直接访问Box类的任何成员
                box.setWidth(width);
                cout << "Width of Box: " << box.width << endl;
            }
    };

    int main(){
        Box box;
        BigBox big;
        // 使用成员函数设置宽度
        box.setWidth(10.0);
        // 使用友元函数输出宽度
        printWidth(box); // 调用友元函数!
        cout << "-------------------------------------\n";
        // 使用友元类中的方法设置宽度
        big.Print(20, box);
        return 0;
    }
```

C++中哪些函数不可以是虚函数?

1、**普通函数（非成员函数）**：我在前面多态这篇博客里讲到，定义虚函数的主要目的是为了重写达到多态，所以普通函数声明为虚函数没有意义，因此编译器在编译时就绑定了它。

2、**静态成员函数**：静态成员函数对于每个类都只有一份代码，所有对象都可以共享这份代码，他不归某一个对象所有，所以它也没有动态绑定的必要。

3、**内联成员函数**：内联函数本就是为了减少函数调用的代价，所以在代码中直接展开。但虚函数一定要创建虚函数表，这两者不可能统一。另外，内联函数在编译时被展开，而虚函数在运行时才动态绑定。

4、**构造函数**：
1）因为创建一个对象时需要确定对象的类型，而虚函数是在运行时确定其类型的。而在构造一个对象时，由于对象还未创建成功，编译器无法知道对象的实际类型，是类本身还是类的派生类等等

2）虚函数的调用需要虚函数表指针，而该指针存放在对象的内存空间中；若构造函数声明为虚函数，那么由于对象还未创建，还没有内存空间，更没有虚函数表地址用来调用虚函数即构造函数了

5、**友元函数**：当我们把一个函数声明为一个类的友元函数时，它只是一个可以访问类内成员的普通函数，并不是这个类的成员函数，自然也不能在自己的类内将它声明为虚函数。

C 指针指向的是物理地址吗？

c/c++的指针是指向逻辑地址。
以windows平台为例，任何一个C++程序肯定是运行在某一个进程中，windows的32位系统对每一个用户进程都管理着一个寻址范围为4GB的地址空间, 各个进程的地址空间是相互独立的，很显然这是一个逻辑的地址空间，C++指针指向进程内的一个逻辑内存地址，然后由操作系统管理着从逻辑地十到物理地址的映射。

模板的编译过程，模板是什么时候实例化的？模板特化·

编译过程

通常我们将函数或类的声明放在头(.h)文件中，定义放在(.cpp)文件中，在其他文件中使用该函数或类时引用头文件即可，编译器是怎么工作的呢？编译器首先编译所有cpp文件，如果在程序中用到某个函数或类，只是判断这个函数或类是否已经声明，并不会立即找到这个函数或类定义的地址，只有在链接的过程中才会去寻找具体的地址，所以我们如果只是对某个函数或类声明了，而不定义它的具体内容，如果我不再其他地方使用它，这事没有任何问题的。
而编译器在编译模板所在的文件时，模板的内容是不会立即生成二进制代码的，直到有地方使用到该模板时，才会对该模板生成二进制代码（即模板实例化）。但是，如果我们将模板的声明部分和实现部分分别放在.h和.cpp两个文件中时，问题就出现了：由于模板的cpp文件中使用的不是具体类型，所以编译器不能为其生成二进制代码，在其他文件使用模板时只是引用了头文件，编译器在编译时可以识别该模板，编译可以通过；但是在链接时就不行了，二进制代码根本就没有生成，链接器当然找不到模板的二进制代码的地址了，就会出现找不到函数地址类似的错误信息了。
所以，在通常情况下，我们在定义模板时将声明和定义都放在了头文件中，STL就是这样。当然了，也

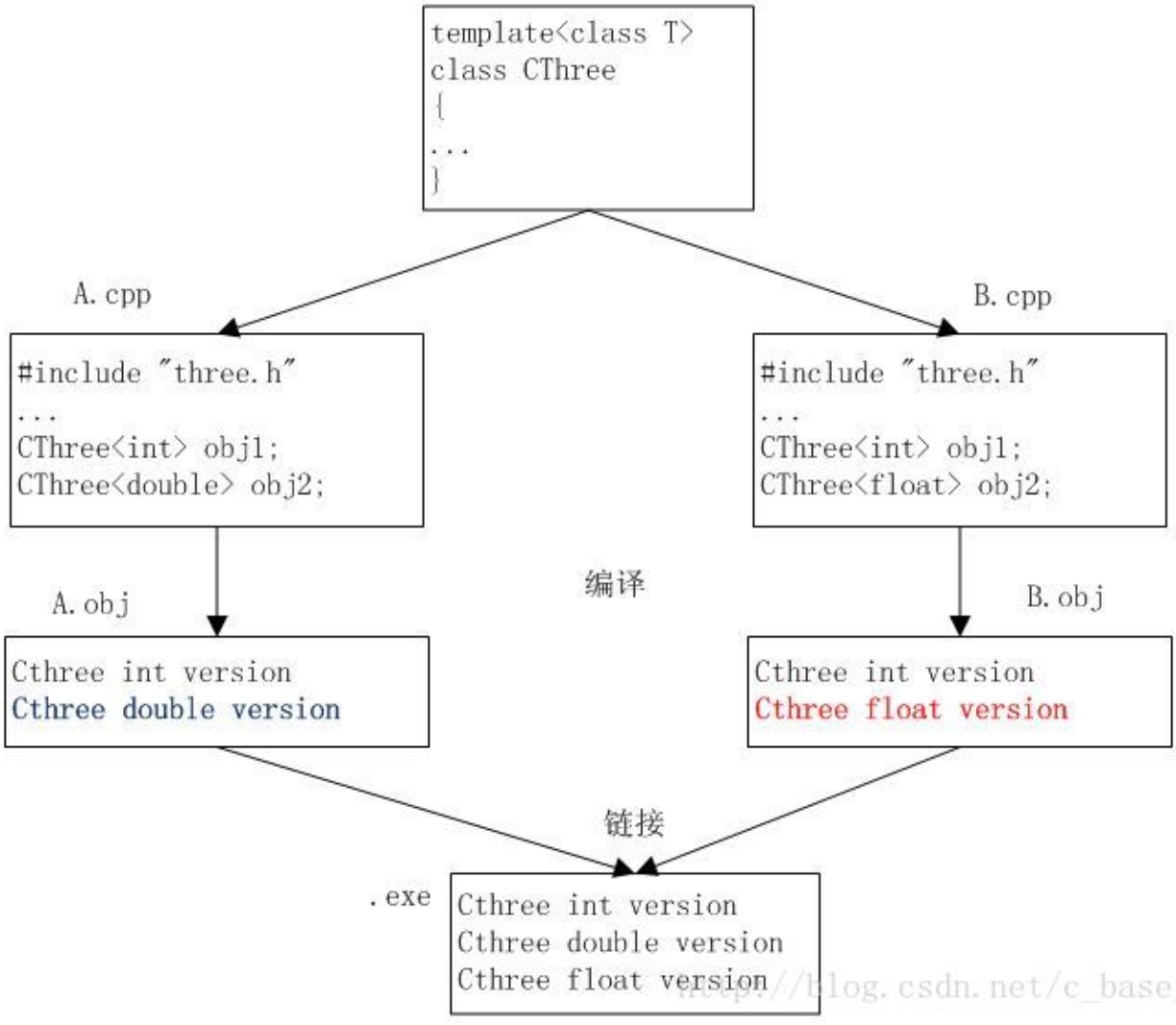
可以像C++ Primer中所述的，使用export关键字。（你可以在.h文件中，声明模板类和模板函数；在.cpp文件中，使用关键字export来定义具体的模板类对象和模板函数；然后在其他用户代码文件中，包含声明头文件后，就可以使用该这些对象和函数了）

编译和链接：

当编译器遇到一个template时，不能够立马为他产生机器代码，它必须等到template被指定某种类型。也就是说，函数模板和类模板的完整定义将出现在template被使用的每一个角落。

对于不同的编译器，其对模板的编译和链接技术也会有所不同，其中一个常用的技术称之为Smart，其基本原理如下：

1. 模板编译时，以每个cpp文件为编译单位，实例化该文件中的函数模板和类模板
2. 链接器在链接每个目标文件时，会检测是否存在相同的实例；有存在相同的实例版本，则删除一个重复的实例，保证模板实例化没有重复存在。
   比如我们有一个程序，包含A.cpp和B.cpp，它们都调用了CThree模板类，在A文件中定义了int和double型的模板类，在B文件中定义了int和float型的模板类；在编译器编译时.cpp文件为编译基础，生成A.obj和B.obj目标文件，即使A.obj和B.obj存在重复的实例版本，但是在链接时，链接器会把所有冗余的模板实例代码删除，保证exe中的实例都是唯一的。编译原理和链接原理，如下所示：



实例化

在我们使用类模板时，只有当代码中使用了类模板的一个实例的名字，而且上下文环境要求必须存在类的定义时，这个类模板才被实例化。

1. 声明一个类模板的指针和引用，不会引起类模板的实例化，因为没有必要知道该类的定义。
2. 定义一个类类型的对象时需要该类的定义，因此类模板会被实例化。
3. 在使用sizeof()时，它是计算对象的大小，编译器必须根据类型将其实例化出来，所以类模板被实例化.
4. new表达式要求类模板被实例化。
5. 引用类模板的成员会导致类模板被编译器实例化。
6. 需要注意的是，类模板的成员函数本身也是一个模板。标准C++要求这样的成员函数只有在被调用或者取地址的时候，才被实例化。用来实例化成员函数的类型，就是其成员函数要调用的那个类对象的类型

模板的特化

一是特化为绝对类型（全特化）； 二是特化为引用，指针类型(半特化、偏特化)

模板函数只能全特化
模板类都可以

全特化，就是模板中参数全被指定为确定的类型。
全特化就是定义了一个全新的类型，全特化的类中的函数可以与模板类不一样。
偏特化：就是模板中的模板参数没有被全部确定，需要编译器在编译时进行确定。
在类型中加上const，&，（*const int int& int* 等等），并没有产生新的类型，只是类型被修饰了。模板在编译时，可以得到这些修饰信息。

全特化的标志就是产生出完全确定的东西，而不是还需要在编译期间去搜寻合适的特化实现，全特化的东西无论是类还是函数都有该特点。
一个特化的模板类的标志：在定义类实现时，加上了<>
比如 class A <int T>; 但是在定义一个模板类的时候，class A后面是没有<>的。
全特化的标志：template<> 然后是完全和模板类型没有一点关系的类实现或者函数定义。
偏特化的标志：
template<typename T,...>还剩点东西，不像全特化那么彻底。

运算符重载

重载的运算符是带有特殊名称的函数，函数名是由关键字 operator 和其后要重载的运算符符号构成的。与其他函数一样，重载运算符有一个返回类型和一个参数列表。

```
Box operator+(const Box&);
```

声明加法运算符用于把两个 Box 对象相加，返回最终的 Box 对象。大多数的重载运算符可被定义为普通的非成员函数或者被定义为类成员函数。如果我们定义上面的函数为类的非成员函数，那么我们需要为每次操作传递两个参数，如下所示：

```
Box operator+(const Box&, const Box&);
```

例子：

```
#include <iostream>
using namespace std;
class complex{
public:
    complex();
    complex(double real, double imag);
public:
    //声明运算符重载
    complex operator+(const complex &A) const;
    void display() const;
private:
    double m_real;  //实部
    double m_imag;  //虚部
};
complex::complex(): m_real(0.0), m_imag(0.0){ }
complex::complex(double real, double imag): m_real(real), m_imag(imag){ }
//实现运算符重载
complex complex::operator+(const complex &A) const{
    complex B;
    B.m_real = this->m_real + A.m_real;
    B.m_imag = this->m_imag + A.m_imag;
    return B;
}
void complex::display() const{
    cout<<m_real<<" + "<<m_imag<<"i"<<endl;
}
int main(){
    complex c1(4.3, 5.8);
    complex c2(2.4, 3.7);
    complex c3;
    c3 = c1 + c2;
    c3.display();
    return 0;
}
```

我们在 complex 类中重载了运算符+，该重载只对 complex 对象有效。当执行c3 = c1 + c2;语句时，编译器检测到+号左边（+号具有左结合性，所以先检测左边）是一个 complex 对象，就会调用成员函数

operator+()，也就是转换为下面的形式：

```
c3 = c1.operator+(c2);
```

c1 是要调用函数的对象，c2 是函数的实参。
运算符重载函数不仅可以作为类的成员函数，还可以作为全局函数。更改上面的代码，在全局范围内重载+，实现复数的加法运算：

```
#include <iostream>
using namespace std;
class complex{
public:
    complex();
    complex(double real, double imag);
public:
    void display() const;
    //声明为友元函数
    friend complex operator+(const complex &A, const complex &B);
private:
    double m_real;
    double m_imag;
};
complex operator+(const complex &A, const complex &B);
complex::complex(): m_real(0.0), m_imag(0.0){ }
complex::complex(double real, double imag): m_real(real), m_imag(imag){ }
void complex::display() const{
    cout<<m_real<<" + "<<m_imag<<"i"<<endl;
}
//在全局范围内重载+
complex operator+(const complex &A, const complex &B){
    complex C;
    C.m_real = A.m_real + B.m_real;
    C.m_imag = A.m_imag + B.m_imag;
    return C;
}
int main(){
    complex c1(4.3, 5.8);
    complex c2(2.4, 3.7);
    complex c3;
    c3 = c1 + c2;
    c3.display();
    return 0;
}
```

运算符重载函数不是 complex 类的成员函数，但是却用到了 complex 类的 private 成员变量，所以必须在 complex 类中将该函数声明为友元函数。

当执行c3 = c1 + c2;语句时，编译器检测到+号两边都是 complex 对象，就会转换为类似下面的函数调用：

```
c3 = operator+(c1, c2);
```

运算符重载规则

1. 不是所有的运算符都可以重载。长度运算符sizeof、条件运算符: ?、成员选择符.和域解析运算符::不能被重载。
2. 重载不能改变运算符的优先级和结合性。
3. 重载不会改变运算符的用法。
4. 运算符重载函数不能有默认的参数，否则就改变了运算符操作数的个数。
5. 运算符函数既可以作为类的成员函数，也可以作为全局函数。
6. 箭头运算符->、下标运算符[ ]、函数调用运算符( )、赋值运算符=只能以成员函数的形式重载。

到底以成员函数还是全局函数（友元函数）的形式重载运算符

（1）一般而言，对于双目运算符，最好将其重载为友元函数；而对于单目运算符，则最好重载为成员函数。
但是也存在例外情况。有些双目运算符是不能重载为友元函数的，比如赋值运算符=、函数调用运算符()、下标运算符[]、指针运算符->等，因为这些运算符在语义上与this都有太多的关联。比如=表示“将自身赋值为…”，[]表示“自己的第几个元素”，如果将其重载为友元函数，则会出现语义上的不一致。

2）还有一个需要特别说明的就是输出运算符<<。因为<<的第一个操作数一定是ostream类型，所以<<只能重载为友元函数，如下：

```
friend ostream& operator <<(ostream& os, const Complex& c);
ostream& operator <<(ostream& os, const Complex& c)
{
    os << c.m_Real << “+” << c.m_Imag << “i” << endl;
    return os;
}
```

（3）所以，对于 =、[]、()、->以及所有的类型转换运算符只能作为非静态成员函数重载。如果允许第一操作数不是同类对象，而是其他数据类型，则只能作为非成员函数重载（如输入输出流运算符>>和<<就是这样的情况）。

重载[]（下标运算符）

下标运算符[ ]必须以成员函数的形式进行重载。该重载函数在类中的声明格式如下：

```
返回值类型 & operator[ ] (参数);
```

或者：

```
const 返回值类型 & operator[ ] (参数) const;
```

使用第一种声明方式，[ ]不仅可以访问元素，还可以修改元素。使用第二种声明方式，[ ]只能访问而不能修改元素。在实际开发中，我们应该同时提供以上两种形式，这样做是为了适应 const 对象，因为通过 const 对象只能调用 const 成员函数，如果不提供第二种形式，那么将无法访问 const 对象的任何元素。

```
#include <iostream>
using namespace std;
class Array{
public:
    Array(int length = 0);
    ~Array();
public:
    int & operator[](int i);
    const int & operator[](int i) const;
public:
    int length() const { return m_length; }
    void display() const;
private:
    int m_length;  //数组长度
    int *m_p;  //指向数组内存的指针
};
Array::Array(int length): m_length(length){
    if(length == 0){
        m_p = NULL;
    }else{
        m_p = new int[length];
    }
}
Array::~Array(){
    delete[] m_p;
}
int& Array::operator[](int i){
    return m_p[i];
}
const int & Array::operator[](int i) const{
    return m_p[i];
}
void Array::display() const{
    for(int i = 0; i < m_length; i++){
        if(i == m_length - 1){
            cout<<m_p[i]<<endl;
        }else{
            cout<<m_p[i]<<", ";
        }
    }
}
int main(){
    int n;
    cin>>n;
    Array A(n);
    for(int i = 0, len = A.length(); i < len; i++){
        A[i] = i * 5;
    }
    A.display();

    const Array B(n);
```

```
    cout<<B[n-1]<<endl;  //访问最后一个元素

    return 0;
}
```

重载[ ]运算符以后，表达式arr[i]会被转换为：

```
arr.operator[ ](i);
```

需要说明的是，B 是 const 对象，如果 Array 类没有提供 const 版本的operator[ ]，那么倒数第二行代码将报错。虽然这行代码只是读取对象的数据，并没有试图修改对象，但是它调用了非 const 版本的 operator[ ]，编译器不管实际上有没有修改对象，只要是调用了非 const 的成员函数，编译器就认为会修改对象（至少有这种风险）。

赋值运算符重载

```
#include <iostream>
#include <string>
using namespace std;

class MyStr {
public:
    MyStr() {}
    MyStr(int _id, char *_name)
    {
        cout << "constructor" << endl;
        id = _id;
        name = new char[strlen(_name) + 1];
        strcpy_s(name,strlen(_name) + 1,_name);
    }
    MyStr(const MyStr &str)
    {
        cout << "copy constructor" << endl;
        id = str.id;
        if (name != NULL)
            delete name;
        name = new char[strlen(str.name) + 1];
        strcpy_s(name,strlen(str.name) + 1,str.name);
    }

    MyStr& operator=(const MyStr& str)
    {
        cout << "operator=" << endl;
        if (this != &str)
        {
            if (name != NULL)
                delete name;
            this->id = str.id;
            name = new char[strlen(str.name) + 1];
            strcpy_s(name,strlen(str.name) + 1,str.name);

            return *this;
        }
    }
    ~MyStr()
    {
        cout << "deconstructor" << endl;
        delete name;
    }
private:
    char *name;
    int id;
};

void main()
{
    MyStr str1(1,"Jack");
```

```
    MyStr str2;
    str2 = str1;
    MyStr str3 = str2;
    return;

}
```

如果将上述例子显示提供的拷贝函数注释掉，然后同样执行MyStr str3 = str2;语句，此时调用默认的拷贝构造函数，它们指向内存中的同一区域。
这样会有两个致命错误：
1）str2修改name时，str3的name也会被修改；
2）当执行str2和str3的析构函数时，会导致同一内存区域释放两次，程序崩溃。
所以，必须通过显示提供拷贝构造函数以避免这样的问题，如上述例子，先判断被拷贝者的name是否为空，若否，delete name，然后为name重新申请空间，再将拷贝者name中的数据拷贝到被拷贝者的name中，这样，str2.name和str3.name各自独立，避免了上面两个错误。赋值运算符重载函数也是同样的道理。

赋值运算符重载函数只能是类的非静态的成员函数
1、因为静态成员函数只能操作类的静态成员，无法操作类的非静态成员，可以参考静态成员变量和静态成员函数在C++类中的作用来进行理解；
2、避免二义性
当程序没有显示提供一个以本类或者本类的引用为参数的赋值运算符重载函数时，编译器会自动提供一个。现在假设C++允许友元函数定义的赋值运算符重载函数，而且以引用为参数，与此同时，编译器也提供一个默认的赋值运算符重载函数（由于友元函数不属于这个类，所以此时编译器会自动提供一个）。但是当再执行类似str2 = str1;这样的代码时，编译器就困惑了。
为了避免这样的二义性，C++强制规定，赋值运算符重载函数只能定义为类的成员函数，这样编译器就能判断是否需要提供默认版本了。

取地址及const取地址操作符重载

取地址是什么意思呢？就是返回当前对象的地址，对于成员函数来讲，this指针就是它的地址，需要返回指针。
"&" 运算符是一个单目运算符，其只有一个参数，而这个参数就是一个对象，所以说这个对象是不用传的，定义为成员函数时函数参数就应该少一个，第一个函数参数就被this指针所代替。所以，在此不需要进行传参。

const成员函数及const对象去调用，普通的成员函数普通的对象来进行调用，若没有普通成员函数，那么普通对象也能够调用const成员函数。

```
class Date {
public:
	Date(int year, int month, int day) {
		_year = year;
		_month = month;
		_day = day;
	}
	Date(const Date& d) {
		_year = d._year;
	}
	Date* operator&() {
		cout << "Date* operator&()" << endl;
		return this;
	}

	const Date* operator&() const {
		cout << "const Date* operator&() const" << endl;
		return this;
	}

private:
	int _year;
	int _month;
	int _day;
};

int main() {
	Date d1(2019, 4, 1);
	const Date d2(2019, 3, 31);

	Date* pa1 = &d1;
	const Date* pd2 = &d2;
	system("pause");
	return 0;
}
```

如果不写这两个函数的时候，编译器会帮助默认生成，若无其它操作完全够用了，因为这两个函数只返回this指针，也没有其他的操作。除非，你想返回别的地址，可以做到"返回你想返回的地址"，比如，返回一个病毒的地址，返回一个很深的调用链等等，可以自己按照需求进行重载实现，否则不必实现也无影响。

Operator char()什么意思

operator用于类型转换函数：
类型转换函数的特征：

1. 型转换函数定义在源类中；
2. 须由 operator 修饰，函数名称是目标类型名或目标类名；

3. 函数没有参数，没有返回值，但是有return 语句，在return语句中返回目标类型数据或调用目标类的构造函数。

   类型转换函数主要有两类：

1） 对象向基本数据类型转换：

```
class D {
  public:
     D(double d) : d_(d) {}

     /* “(int)D”类型转换:将类型D转换成int */
     operator int() const {
         std::cout << "(int)d called!" << std::endl;
         return static_cast<int>(d_);
     }

   private:
     double d_;
 };

 int add(int a, int b) {
     return a + b;
 }

  int main() {
     D d1 = 1.1;
     D d2 = 2.2;
     std::cout << add(d1, d2) << std::endl;
     return 0;
 }
```

执行add(d1,d2)函数时“(int)D”类型转换函数将被自动调用，程序运行的输出为：

(int)d called!

(int)d called!

3

2）对象向不同类的对象的转换：

```
class A
{
  public:
     A(int num = 0) : dat(num) {}

     /* "(int)a"类型转换 */
     operator int() { return dat; }

  private:
     int dat;
};


class X
{
  public:
     X(int num = 0) : dat(num) {}

     /* "(int)a"类型转换 */
     operator int() { return dat; }

     /* "(A)a"类型转换 */
     operator A() {
         A temp = dat;
         return temp;
     }

  private:
     int dat;
};


int main()
{
     X stuff = 37;
     A more = 0;
     int hold;

     hold = stuff;    // convert X::stuff to int
     std::cout << hold << std::endl;

     more = stuff;    // convert X::stuff to A::more
     std::cout << more << std::endl;        // convert A::more to int

     return 0;
}
```

上面这个程序中X类通过“operator A()”类型转换来实现将X类型对象转换成A类型，这种方式需要先创建一个临时A对象再用它去赋值目标对象；更好的方式是为A类增加一个构造函数：

```
A(const X& rhs) : dat(rhs) {}
```

extern c有什么用

指示编译器这部分代码按C语言的进行编译，而不是C++的。由于C++支持函数重载，因此编译器编译函数的过程中会将函数的参数类型也加到编译后的代码中，而不仅仅是函数名；而C语言并不支持函数重载，因此编译C语言代码的函数时不会带上函数的参数类型，一般之包括函数名。

printf和sprintf的区别？strcpy和strncpy的区别？

函数printf(...)根据指定的格式（format）将参数（argument）输出到屏幕上；
函数sprintf(...)根据指定的格式（format）将参数（argument）输出到由指针buffer指定的字符数组（字符缓冲区）中；

strcpy函数：顾名思义字符串复制函数，功能：把从src地址开始且含有NULL结束符的字符串赋值到以dest开始的地址空间，返回dest（地址中存储的为复制后的新值）。**要求：src和dest所指内存区域不可以重叠且dest必须有足够的空间来容纳src的字符串。**
strncpy函数：复制字符串的前n个字符，功能：strncpy()会将字符串src前n个字符拷贝到字符串dest。不像strcpy()，strncpy()不会向dest追加结束标记'\0'。

1. 如果src的前n个字节不含NULL字符，则结果不会以NULL字符结束。
2. 如果src的长度小于n个字节，则以NULL填充dest直到复制完n个字节。
3. src和dest所指内存区域不可以重叠且dest必须有足够的空间来容纳src的字符串。

strncpy 的标准用法为：（手工写上 \0）

```
strncpy(path, src, sizeof(path) - 1);
path[sizeof(path) - 1] = '\0';
len = strlen(path);
```

memcpy：从源src所指的内存地址的起始位置开始拷贝n个字节到目标dest所指的内存地址的起始位置中，src和dest有可能出现空间重叠，它可以复制任何内容；

实现strcat，strcpy，strncpy，memset，memcpy

strcat 函数要求 dst 参数原先已经包含了一个字符串（可以是空字符串）。它找到这个字符串的末尾，并把 src 字符串的一份拷贝添加到这个位置。如果 src 和 dst 的位置发生重叠，其结果是未定义的。

```
char *strcat (char * dst, const char * src)
{
    assert(NULL != dst && NULL != src);   // 源码里没有断言检测
    char * cp = dst;
    while(*cp )
         cp++;                       /* find end of dst */
    while(*cp++ = *src++) ;          /* Copy src to end of dst */
    return( dst );                   /* return dst */
}
```

strcpy:

```
char *strcpy(char *dst, const char *src)    // 实现src到dst的复制
{
    if(dst == src) return dst;               //源码中没有此项
    assert((dst != NULL) && (src != NULL)); //源码没有此项检查，判断参数src和dst的有效性
    char *cp = dst;                          //保存目标字符串的首地址
    while (*cp++ = *src++);                  //把src字符串的内容复制到dst下
    return dst;
}
```

memcpy:

```
void *memcpy(void *dst, const void *src, size_t length)
{
    assert((dst != NULL) && (src != NULL));
    char *tempSrc= (char *)src;            //保存src首地址
    char *tempDst = (char *)dst;           //保存dst首地址
    while(length-- > 0)                    //循环length次，复制src的值到dst中
          *tempDst++ = *tempSrc++ ;
    return dst;
}
```

strncpy:

```
char *strncpy(char *dst, const char *src, size_t len)
{
    assert(dst != NULL && src != NULL);     //源码没有此项
    char *cp = dst;
    while (len-- > 0 && *src != '\0')
        *cp++ = *src++;
    *cp = '\0';                             //源码没有此项
    return dst;
}
```

memset:将参数a所指的内存区域前length个字节以参数ch填入，然后返回指向a的指针。在编写程序的时候，若需要将某一数组作初始化，memset()会很方便。

```
void *memset(void *a, int ch, size_t length)
{
    assert(a != NULL);
    void *s = a;
    while (length--)
    {
        *(char *)s = (char) ch;
        s = (char *)s + 1;
    }
    return a;
}
```

strcmp:字符串比较

```
int mystrcmp(const char *dest,const char *src)
{
      int i=0;
      //判断str1与str2指针是否为NULL,函数assert的头文件为#include<assert.h>
      assert(dest!=NULL && src !=NULL);  //[1]

      //如果dest > source,则返回值大于0，如果dest = source,则返回值等于0，如果dest  < source ,则返回
      while (*dest && *src && (*dest == *src))
      {
          dest ++;
          src ++;
      }
      return *dest - *source; [2]
}
```

内存泄漏和内存溢出的区别？

系统已经不能再分配出你所需要的空间，比如你需要100M的空间，系统只剩90M了，这就叫内存溢出。
比方说栈，栈满时再做进栈必定产生空间溢出，叫上溢，栈空时再做退栈也产生空间溢出，称为下溢。就是分配的内存不足以放下数据项序列,称为内存溢出。

用资源的时候为他开辟了一段空间，当你用完时忘记释放资源了，这时内存还被占用着，一次没关系，但是内存泄漏次数多了就会导致内存溢出。
向系统申请分配内存进行使用(new)，可是使用完了以后却不归还(delete)，结果你申请到的那块内存你自己也不能再访问（也许你把它的地址给弄丢了），而系统也不能再次将它分配给需要的程序。

栈溢出和堆溢出的区别？

堆溢出:不断的new 一个对象，一直创建新的对象，但是没有delete
栈溢出：死循环或者是递归太深，递归的原因，可能太大，也可能没有终止。

定位内存泄露

1. 查看进程maps表
   在实际调试过程中，怀疑某处发生了内存泄漏，可以查看该进程的maps表，看进程的堆或mmap段的虚拟地址空间是否持续增加。如果是，说明可能发生了内存泄漏。如果mmap段虚拟地址空间持续增加，还可以看到各个段的虚拟地址空间的大小，从而可以确定是申请了多大的内存。
2. 重载new/delete操作符
   重载new/delete操作符，用list或者map记录对内存的使用情况。new一次，保存一个节点，delete一次，就删除节点。
   最后检测容器里是否还有节点，如果有节点就是有泄漏。也可以记录下哪一行代码分配的内存被泄漏。
   类似的方法：在每次调用new时加个打印，每次调用delete时也加个打印。
3. top 指令。 在Linux上面可以快速定位泄漏的程序和程度。
4. matrace
   mtrace的原理是记录每一对malloc-free的执行，若每一个malloc都有相应的free，则代表没有内存泄露；
   对于任何非malloc/free情况下所发生的内存泄露问题，mtrace并不能找出来。

哈希是什么？哈希冲突是什么？如何解决？

哈希函数可以将任意长度的输入，变换成一个固定长度的输出（哈希值）。哈希函数具有如下性质：

1. 典型的哈希函数有无限的输入域
2. 当哈希函数传入相同的输入值时，返回值一样
3. 当给哈希函数传入不同的输入值时，返回值可能一样，也可能不一样。
4. 最重要的性质是很多不同的输入值所得到的的返回值会均匀分布在输出域上。

常见的哈希算法：
1） 直接定址法
取关键字或关键字的某个线性函数值为散列地址。
即 f(key) = key 或 f(key) = a*key + b，其中a和b为常数。

2） 除留余数法
取关键字被某个不大于散列表长度 m 的数 p 求余，得到的作为散列地址。
即 f(key) = key % p, p < m。这是最为常见的一种哈希算法。

3） 数字分析法
当关键字的位数大于地址的位数，对关键字的各位分布进行分析，选出分布均匀的任意几位作为散列地

址。
仅适用于所有关键字都已知的情况下，根据实际应用确定要选取的部分，尽量避免发生冲突。

4） 平方取中法
先计算出关键字值的平方，然后取平方值中间几位作为散列地址。
随机分布的关键字，得到的散列地址也是随机分布的。

5） 随机数法
选择一个随机函数，把关键字的随机函数值作为它的哈希值。
通常当关键字的长度不等时用这种方法。

哈希冲突：
不同是输入得到了相同的返回值就是哈希冲突。

如何解决
1.开放地址法
当冲突发生时，使用某种探测算法在散列表中寻找下一个空的散列地址，只要散列表足够大，空的散列地址总能找到。按照探测序列的方法，一般将开放地址法区分为线性探查法、二次探查法、双重散列法等。
用以一个模为8的哈希表为例，采用除留余数法，往表中插入三个关键字分别为26，35，36的记录，分别除8取模后，在表中的位置如下：

这个时候插入42，那么正常应该在地址为2的位置里，但因为关键字30已经占据了位置，所以就需要解决这个地址冲突的情况，接下来就介绍三种探测方法的原理，并展示效果图。

1） 线性探查法：
fi=(f(key)+i) ％ m ，0 ≤ i ≤ m-1
探查时从地址 d 开始，首先探查 T[d]，然后依次探查 T[d+1]，…，直到 T[m-1]，此后又循环到 T[0]，T[1]，…，直到探查到有空余的地址或者到 T[d-1]为止。

插入42时，探查到地址2的位置已经被占据，接着下一个地址3，地址4，直到空位置的地址5，所以39应放入地址为5的位置。
缺点：需要不断处理冲突，无论是存入还是查找效率都会大大降低。

2） 二次探查法
fi=(f(key)+di) ％ m，0 ≤ i ≤ m-1
探查时从地址 d 开始，首先探查 T[d]，然后依次探查 T[d+di]，di 为增量序列1^2， -1^2， 2^2， -2^2，……， q^2， -q^2 且q≤1/2 (m-1) ,直到探查到 有空余地址或者到 T[d-1]为止。

缺点：无法探查到整个散列空间。

所以插入42时，探查到地址2被占据，就会探查T[2+1^2]也就是地址3的位置，被占据后接着探查到地址7，然后插入。

3） 双哈希函数探测法
fi=(f(key)+i*g(key)) % m (i=1，2，……，m-1)
其中，f(key) 和 g(key) 是两个不同的哈希函数，m为哈希表的长度

步骤：
双哈希函数探测法，先用第一个函数 f(key) 对关键码计算哈希地址，一旦产生地址冲突，再用第二个函数 g(key) 确定移动的步长因子，最后通过步长因子序列由探测函数寻找空的哈希地址。

比如，f(key)=a 时产生地址冲突，就计算g(key)=b，则探测的地址序列为 f1=(a+b) mod m，f2=(a+2b) mod m，……，fm-1=(a+(m-1)b) % m，假设 b 为 3，那么关键字42应放在 “5” 的位置。

2.再哈希法
当发生冲突时，使用第二个、第三个、哈希函数计算地址，直到无冲突时。缺点：计算时间增加。
比如上面第一次按照姓首字母进行哈希，如果产生冲突可以按照姓字母首字母第二位进行哈希，再冲突，第三位，直到不冲突为止

3.链地址法
将所有哈希地址为i的元素构成一个称为同义词链的单链表，并将单链表的头指针存在哈希表的第i个单元中，因而查找、插入和删除主要在同义词链中进行。
拉链法的优点:
①拉链法处理冲突简单，且无堆积现象，即非同义词决不会发生冲突，因此平均查找长度较短；
②由于拉链法中各链表上的结点空间是动态申请的，故它更适合于造表前无法确定表长的情况；
③开放定址法为减少冲突，要求装填因子α较小，故当结点规模较大时会浪费很多空间。而拉链法中可取α≥1，且结点较大时，拉链法中增加的指针域可忽略不计，因此节省空间；
④在用拉链法构造的散列表中，删除结点的操作易于实现。只要简单地删去链表上相应的结点即可。

拉链法的缺点：
指针需要额外的空间，故当结点规模较小时，开放定址法较为节省空间，而若将节省的指针空间用来扩大散列表的规模，可使装填因子变小，这又减少了开放定址法中的冲突，从而提高平均查找速度。

说说引用，什么时候用引用好，什么时候用指针好？

使用引用参数的主要原因有两个：

1. 程序员能修改调用函数中的数据对象
2. 通过传递引用而不是整个数据–对象，可以提高程序的运行速度

一般的原则：
对于使用引用的值而不做修改的函数：

1. 如果数据对象很小，如内置数据类型或者小型结构，则按照值传递
2. 如果数据对象是数组，则使用指针（唯一的选择），并且指针声明为指向const的指针
3. 如果数据对象是较大的结构，则使用const指针或者引用，已提高程序的效率。这样可以节省结构所需的时间和空间
4. 如果数据对象是类对象，则使用const引用（传递类对象参数的标准方式是按照引用传递）

对于修改函数中数据的函数：

1. 如果数据是内置数据类型，则使用指针
2. 如果数据对象是数组，则只能使用指针
3. 如果数据对象是结构，则使用引用或者指针
4. 如果数据是类对象，则使用引用

说下const，static，typeof，violiate

const作用：
1.用于修饰右边变量(基本变量,指针变量)
2.被const修饰变量只读(普通的变量是可读可写的)
const与宏区别：
1.编译时刻: 宏:预编译 const:编译
2.编译检查: 宏不会做编译检查 const会
3.宏好处: 宏可以定义函数和方法 const不行
4.宏坏处: 大量使用宏,会导致预编译时间过长

static

不用加减乘除求一个数的7倍

两个鸡蛋，100层楼，判断出鸡蛋会碎的临界层