

- 一个空类编译器会自动生成哪些函数？ 哪些需要禁止？
- 如何限制一个类对象只在栈（堆）上分配空间？
- 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？
- this指针
- 友元函数
  - 为什么要有？
  - 使用友元函数的优缺点
  - 友元函数的使用
- C++中哪些函数不可以是虚函数？
- C 指针指向的是物理地址吗？
- 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
  - 实例化
  - 模板的特化
- 运算符重载
  - 运算符重载规则
  - 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
  - 重载[]（下标运算符）
  - 赋值运算符重载
  - 取地址及const取地址操作符重载
  - Operator char()什么意思
- extern c有什么用
- printf和sprintf的区别？ strcpy和strncpy的区别？
- 实现strcat, strcpy, strncpy, memset, memcpy
- 内存泄漏和内存溢出的区别？
- 栈溢出和堆溢出的区别？
- 定位内存泄露
- 哈希是什么？哈希冲突是什么？如何解决？
- 说说引用，什么时候用引用好，什么时候用指针好？
- 说下const, static, typeof, volatile, typedef, #define
  - const作用：
  - #define
  - typedef
  - typedef和#define区别
  - const与#define区别
  - static
  - typeof
  - volatile

- malloc的内存分配方式
- 全局变量和静态变量
- C语言是怎么进行函数调用的？
- C语言参数压栈顺序？
- C语言如何处理返回值？
- C++4个cast的区别
- new和malloc的区别
- 堆和栈的区别(从数据结构和内存方面)
  - 数据结构中的堆和栈
  - 内存中的堆和栈
- C++的内存分配有哪些？
- 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

## 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针调用静态成员函数。

### 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**virtual**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和普通函数，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址到物理地址的映射。

## 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）

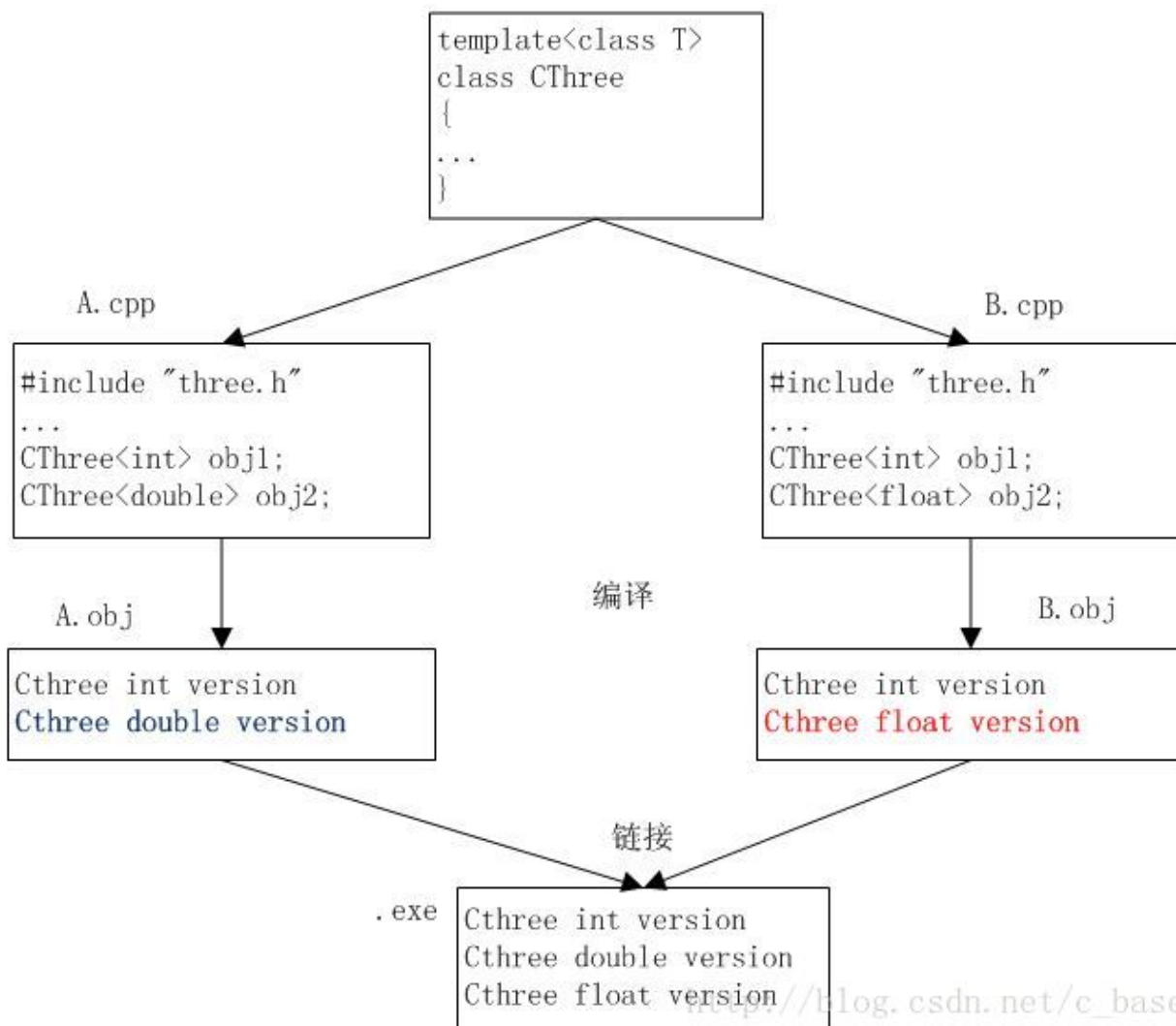
### 编译和链接：

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 << "destructor" << 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。



**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 ,则返回负数
    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. mtrace

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  $f(\text{key}) = \text{key}$  或  $f(\text{key}) = a * \text{key} + b$ ，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  $f(\text{key}) = \text{key} \% p$ ,  $p < m$ 。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：

0	1	2	3	4	5	6	7
		26	35	36			

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：

$$f_i = (f(\text{key}) + i) \% m, \quad 0 \leq i \leq m-1$$

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。

0	1	2	3	4	5	6	7
		26	35	36	42		

## 2) 二次探查法

$$f_i = (f(\text{key}) + d_i) \% m, 0 \leq i \leq m-1$$

探查时从地址  $d$  开始，首先探查  $T[d]$ ，然后依次探查  $T[d+d_i]$ ， $d_i$  为增量序列  $1^2, -1^2, 2^2, -2^2, \dots, q^2, -q^2$  且  $q \leq 1/2(m-1)$ ，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，被占据后接着探查地址7，然后插入。

0	1	2	3	4	5	6	7
		26	35	36			42

## 3) 双哈希函数探测法

$$f_i = (f(\text{key}) + i * g(\text{key})) \% m \quad (i=1, 2, \dots, m-1)$$

其中， $f(\text{key})$  和  $g(\text{key})$  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  $f(\text{key})$  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  $g(\text{key})$  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， $f(\text{key})=a$  时产生地址冲突，就计算  $g(\text{key})=b$ ，则探测的地址序列为  $f_1=(a+b) \% m$ ， $f_2=(a+2b) \% m$ ， $\dots$ ， $f_{m-1}=(a+(m-1)b) \% m$ ，假设  $b$  为 3，那么关键字42应放在“5”的位置。

0	1	2	3	4	5	6	7
		26	35	36	42		

## 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 $\alpha$ 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 $\alpha \geq 1$ ，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，typedef，#define

## const作用:

- 1: 可以用来定义常量, 修饰函数参数, 修饰函数返回值, 且被const修饰的东西, 都受到强制保护, 可以预防其它代码无意识的进行修改, 从而提高了程序的健壮性 (是指系统对于规范要求以外的输入能够判断这个输入不符合规范要求, 并能有合理的处理方式。ps: 即所谓高手写的程序不容易死)
- 2: 使编译器保护那些不希望被修改的参数, 防止无意代码的修改, 减少bug;
- 3: 给读代码的人传递有用的信息, 声明一个参数, 是为了告诉用户这个参数的应用目的;

### 1. 修饰局部变量

```
const int n = 5;  
int const n = 5;
```

这两种写法是一样的, 都是表示变量n的值不能被改变了, 需要注意的是, 用const修饰变量时, 一定要给变量初始化, 否则之后就不能进行赋值了。

接下来看看const用于修饰常量静态字符串:

```
const char* str = "fddsfsff";
```

如果没有const修饰, 我们可能会在后面有意无意的写str[4]="x"这样的语句, 这样会导致对只读内存区域的赋值, 然后程序会立刻异常终止。有了const, 这个错误就能在程序被编译的时候就立即检查出来, 这就是const的好处, 让逻辑错误在编译期被发现。

### 2. 常量指针和指针常量

常量指针是指针指向的内容的常量, 如下两种定义:

```
const int* n;  
int const* n;
```

常量指针说的是不能通过这个指针改变变量的值, 但是还是可以通过其他的引用来改变变量的值。

```
int a = 5;  
const int *n = &a;  
a = 6;
```

常量指针指向的值不能被改变, 但是这并不意味着指针本身不能改变, 常量指针可以指向其他的地址。

```
int a = 5;  
int b = 6;  
const int *n = &a;  
n = &b;
```

指针常量是指指针本身是个常量，不能再指向其他的地址。

```
int *const n;
```

指针常量指向的地址不能改变，但是地址中保存的数值是可以改变的，可以通过其他指向改地址的指针来修改。

```
int a = 5;
int *p = &a;
int *const n = &a;
*p = 8;
```

区分常量指针和指针常量的关键在于\*的位置，我们以\*为分界线，如果const在\*的左边，则为常量指针，在右边则为指针常量。如果我们将\*读作‘指针’，将const读作常量，内容正好符合。

### 指向常量的指针：

是以上两种的结合，指针指向的位置不能改变并且也不能通过这个指针改变变量的值，但是依然可以通过其他普通的指针来改变变量的值。

```
const int* const p;
```

## 3. 修饰函数的参数

根据常量指针和指针常量，const修饰函数的参数也分为三种情况。

### 1) 防止修改指针指向的内容

```
void stringCopy(char* strDestination, const char* strSource);
```

其中strSource是输入参数，strDestination是输出参数。给strSource加上const后，如果函数体内的语句试图改动strSource的内容，编译器将指出错误。

### 2) 防止修改指针指向的地址

```
void swap(int* const p1, int* cconst p2);
```

指针p1和指针p2指向的地址都不能修改。

### 3) 以上两种的结合

## 4. 修饰函数的返回值

如果给以“指针传递”方式的函数返回值加const修饰，那么函数的返回值（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。

```
const char* getString();
char* str = getString();//错误
const char* str = getString();
```

## 5. 修饰全局变量

全局变量的作用域是整个文件，我们应该尽量避免使用全局变量，因为一旦有一个函数改变了全局变量的值，它也会影响到引用这个变量的函数，导致出了bug很难发现，如果一定要用全局变量，应该尽量使用const修饰符。

## #define

宏定义，在预处理阶段的替换。

不管是在某个函数内，还是在所有函数之外，define的作用域都是从定义开始到整个文件结尾。

## typedef

来为类型取一个新的名字

## typedef和#define区别

1. typedef 仅限于为类型定义符号名称，#define 不仅可以为类型定义别名，也能为数值定义别名，比如您可以定义 1 为 ONE。
2. typedef 是由编译器执行解释的，#define 语句是由预编译器进行处理的。

## const与#define区别

const是用于定义一个有类型的只读常变量。而#define是宏定义，是在编译期间对定义的变量进行直接文本替换，不做类型检查。变量定义时，大多数情况下const相对于#define来说有着更优的表现效果。基于以下几点原因：

- 1.const对数据进行类型检查，#define不进行类型检查；
- 2.某些编译器支持对const对象进行调试，所有编译器都不支持对#define进行调试；
- 3.const变量存放在内存的静态数据区域，所以在程序运行期间const变量只有一个拷贝，而#define修饰的变量则会在每处都进行展开，拥有多个拷贝；

其余区别：

- 1.#define可以用来定义宏函数，而const不行。
- 2.const除了可以修饰常变量外，还可以用于修饰指针、函数形参等等，修饰功能更强大。

## static

- (1) 设置变量的存储方式或链接属性

1. 用**static**声明局部变量，使其变为静态存储方式(静态数据区)，作用域不变；用**static**声明外部变量，其本身就是静态变量，这只会改变其连接方式，使其只在本文件内部有效，而其他文件不可连接或引用该变量。
2. 编译时，在全局数据区分配内存，默认值为0。
3. 不同于**auto**变量，该变量的内存只被分配一次，因此其值在下次调用时仍维持上次的值。

## (2) 限制函数的作用域

1. 使用**static**用于函数定义时，对函数的连接方式产生影响，使得函数只在本文件内部有效，对其他文件是不可见的。这样的函数又叫作静态函数。使用静态函数的好处是不用担心与其他文件的同名函数产生干扰，另外也是对函数本身的一种保护机制。
2. 如果想要其他文件可以引用本地函数，则要在函数定义时使用关键字**extern**，表示该函数是外部函数，可供其他文件调用。另外在要引用别的文件中定义的外部函数的文件中，使用**extern**声明要用的外部函数即可。

(3) 在类中的**static**成员变量意味着它为该类的所有实例所共享，也就是说当某个类的实例（对象）修改了该静态成员变量，其修改值为该类的其它所有实例所见；

(4) 在类中的**static**成员函数属于整个类所拥有，只能访问类的**static**成员变量，因为这个函数不隐式的接收类的实例（对象）的**this**指针，实例也无法直接调用静态成员函数。

## typeof

指定变量的数据类型

## volatile

来修饰被不同线程访问和修改的变量；**volatile**的作用是作为指令关键字，确保本条指令不会因编译器的优化而省略，且要求每次直接从内存读值。

## malloc的内存分配方式

**malloc**基本的实现原理就是维护一个内存空闲链表，当申请内存空间时，搜索内存空闲链表，找到适配的空闲内存空间，然后将空间分割成两个内存块，一个变成分配块，一个变成新的空闲块。如果没有搜索到，那么就会用**sbrk()**来推进**brk**指针来申请内存空间。

搜索空闲块最常见的算法有：首次适配，下一次适配，最佳适配。

首次适配：第一次找到足够大的内存块就分配，这种方法会产生很多的内存碎片。

下一次适配：也就是说等第二次找到足够大的内存块就分配，这样会产生比较少的内存碎片。

最佳适配：对堆进行彻底的搜索，从头开始，遍历所有块，使用数据区大小大于**size**且差值最小的块作为此次分配的块。



## 合并空闲块

在释放内存块后，如果不进行合并，那么相邻的空闲内存块还是相当于两个内存块，会形成一种假碎片。所以当释放内存后，我们需要将两个相邻的内存块进行合并。

## 全局变量和静态变量

全局变量与全局静态变量的区别：

- 1.若程序由一个源文件构成时，全局变量与全局静态变量没有区别。
- 2.若程序由多个源文件构成时，全局变量与全局静态变量不同：全局静态变量使得该变量成为定义该变量的源文件所独享，即：全局静态变量对组成该程序的其它源文件是无效的。
- 3.具有外部链接的静态，可以在所有源文件里调用，除了本文件，其他文件可以通过**extern**的方式引用。

## C语言是怎么进行函数调用的？

### 什么是栈帧？

每一次函数调用都为这次函数调用开辟一块空间，这个空间就叫做栈帧。每个函数的每次调用，都有它自己独立的一个栈帧。

首先必须明确一点也是非常重要的一点，栈是向下生长的，所谓向下生长是指从内存高地址->低地址的路径延伸，那么就很明显了，栈有栈底和栈顶，那么栈顶的地址要比栈底低。对x86体系的CPU而言，其中

---> 寄存器**ebp**（**base pointer**）可称为“帧指针”或“基址指针”，其实语意是相同的。

---> 寄存器**esp**（**stack pointer**）可称为“栈指针”。

要知道的是：

---> **ebp** 在未受改变之前始终指向栈帧的开始，也就是栈底，所以**ebp**的用途是在堆栈中寻址用的。

---> **esp**是会随着数据的入栈和出栈移动的，也就是说，**esp**始终指向栈顶。

假设函数A调用函数B，我们称A函数为“调用者”，B函数为“被调用者”则函数调用过程可以这么描述：

- （1）先将调用者（A）的堆栈的基址（**ebp**）入栈，以保存之前任务的信息。
- （2）然后将调用者（A）的栈顶指针（**esp**）的值赋给**ebp**，作为新的基址（即被调用者B的栈底）。
- （3）然后在这个基址（被调用者B的栈底）上开辟（一般用**sub**指令）相应的空间用作被调用者B的栈空间。
- （4）函数B返回后，从当前栈帧的**ebp**即恢复为调用者A的栈顶（**esp**），使栈顶恢复函数B被调用前的位置；然后调用者A再从恢复后的栈顶可弹出之前的**ebp**值（可以这么做是因为这个值在函数调用前一步被压入堆栈）。这样，**ebp**和**esp**就都恢复了调用函数B前的位置，也就是栈恢复函数B调用前的状态。

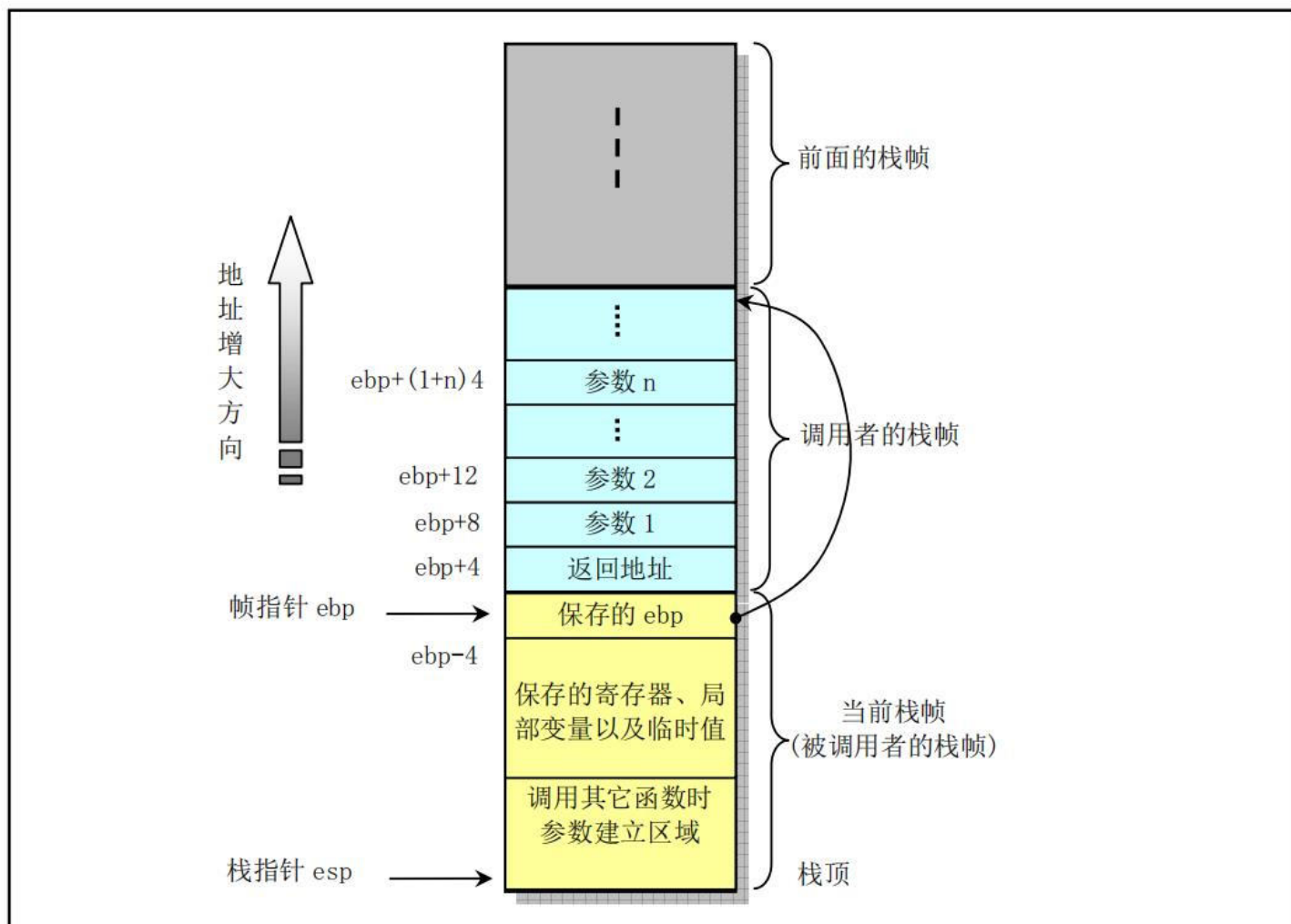


图 3-4 栈中帧结构示意图

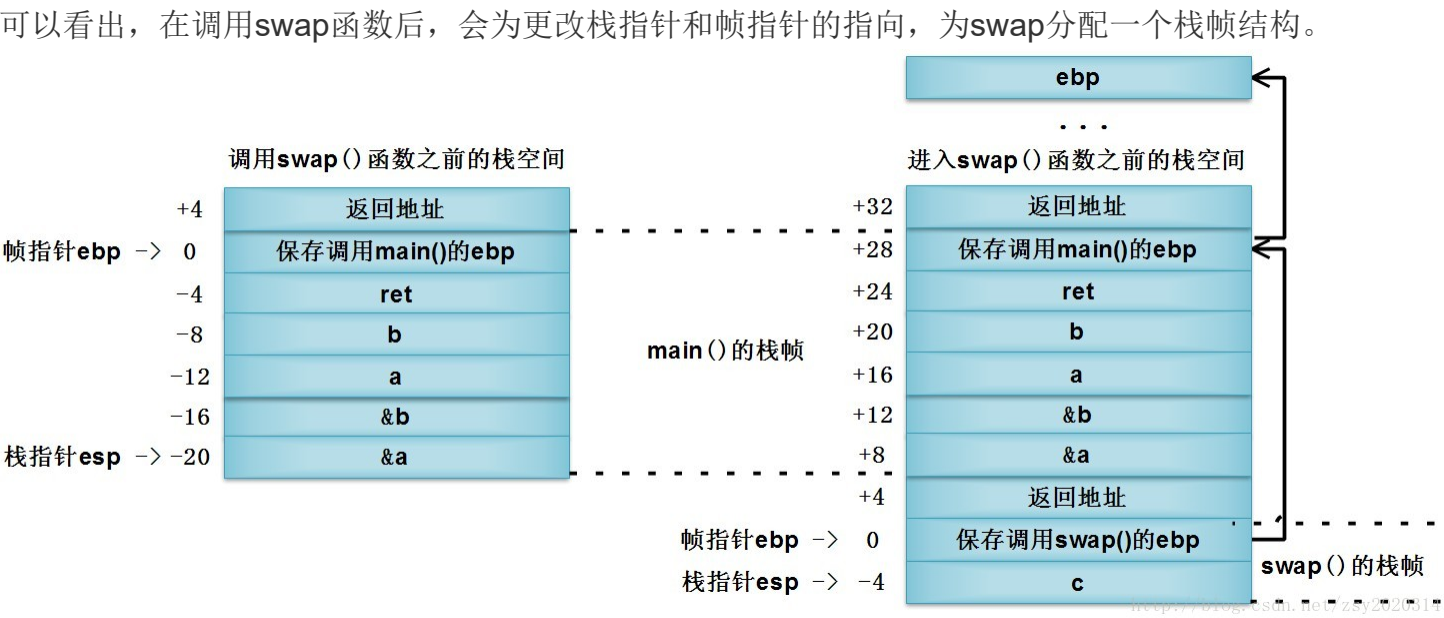
[https://blog.csdn.net/weixin\\_42462202](https://blog.csdn.net/weixin_42462202)

来分析下面的程序，下面的程序的功能就是调用swap函数交换两个数的值，然后返回两个数的差。

```
void swap(int *a,int *b)
{
    int c;
    c = *a;
    *a = *b;
    *b = c;
}

int main(void)
{
    int a ;
    int b ;
    int ret;
    a =16;
    b = 64;
    ret = 0;
    swap(&a,&b);
    ret = a - b;
    return ret;
}
```

下面是这个过程对应的栈结构



## C语言参数压栈顺序？

C函数的参数压栈顺序是从右到左。

printf函数的原型是：int printf(const char \*format,...);

没错，它是一个不定参函数，那么我们在实际使用中是怎么样知道它的参数个数呢？这就要靠format了，编译器通过format中的%占位符的个数来确定参数的个数。

现在我们假设参数的压栈顺序是从左到右的，这时，函数调用的时候，**format**最先进栈，之后是各个参数进栈，最后**pc**进栈，此时，由于**format**先进栈了，上面压着未知个数的参数，想要知道参数的个数，必须找到**format**，而要找到**format**，必须要知道参数的个数，这样就陷入了一个无法求解的死循环了！！

而如果把参数从右到左压栈，情况又是怎么样的？函数调用时，先把若干个参数都压入栈中，再压**format**，最后压**pc**，这样一来，栈顶指针加2便找到了**format**，通过**format**中的%占位符，取得后面参数的个数，从而正确取得所有参数。

## C语言如何处理返回值？

函数在返回返回值的时候汇编代码一般都会将待返回的值放入**eax**寄存器暂存，接着再调用**mov**指令将**eax**中返回值写入对应的变量中。

函数在调用结束后不会会自动清理栈上的内存，如果我们再次访问原来栈空间上的数据，那么我们读取到的数据还是可以读取到的。

**return** 语句可以有多个，可以出现在函数体的任意位置，但是每次调用函数只能有一个 **return** 语句被执行，所以只有一个返回值

函数一旦遇到 **return** 语句就立即返回，后面的所有语句都不会被执行到了。从这个角度看，**return** 语句还有强制结束函数执行的作用。

## C++4个cast的区别

### 1、static\_cast

在编译期处理

```
static_cast < type-id > ( expression )
```

该运算符把**expression**转换为**type-id**类型，但没有运行时类型检查来保证转换的安全性。它主要有如下几种用法：

①用于类层次结构中基类（父类）和派生类（子类）之间指针或引用的转换。

进行上行转换（把派生类的指针或引用转换成基类表示）是安全的；

进行下行转换（把基类指针或引用转换成派生类表示）时，由于没有动态类型检查，所以是不安全的。

②用于基本数据类型之间的转换，如把**int**转换成**char**，把**int**转换成**enum**。这种转换的安全性也要开发人员来保证。

③把空指针转换成目标类型的空指针。

④把任何类型的表达式转换成**void**类型。

注意：**static\_cast**不能转换掉**expression**的**const**、**volatile**、或者**\_\_unaligned**属性。

## 2、const\_cast

在编译期处理

```
const_cast<type_id> (expression)
```

const\_cast将转换掉表达式的const性质。

该运算符用来修改类型的const或volatile属性。除了const 或volatile修饰之外， type\_id和expression的类型是一样的。

- 一、常量指针被转化成非常量的指针，并且仍然指向原来的对象；
- 二、常量引用被转换成非常量的引用，并且仍然指向原来的对象。

## 3、dynamic\_cast

在运行期，会检查这个转换是否可能。

```
dynamic_cast<type_id> (expression)
```

dynamic\_cast 支持运行时识别指针或引用所指向的对象。

该运算符把expression转换成type-id类型的对象。type-id必须是类的指针、类的引用或者void\*；如果type-id是类指针类型，那么expression也必须是一个指针，如果type-id是一个引用，那么expression也必须是一个引用。

dynamic\_cast运算符可以在执行期决定真正的类型。如果downcast是安全的（也就是说，如果基类指针或者引用确实指向一个派生类对象）这个运算符会传回适当转型过的指针。如果downcast不安全，这个运算符会传回空指针（也就是说，基类指针或者引用没有指向一个派生类对象）。

dynamic\_cast主要用于类层次间的上行转换和下行转换，还可以用于类之间的交叉转换。

在类层次间进行上行转换时，dynamic\_cast和static\_cast的效果是一样的；

在进行下行转换时，dynamic\_cast具有类型检查的功能，比static\_cast更安全。

```
class B{
public:
    int m_iNum;
    virtual void foo();
};
class D:public B{
public:
    char *m_szName[100];
};
void func(B *pb){
    D *pd1 = static_cast<D *>(pb);
    D *pd2 = dynamic_cast<D *>(pb);
}
```

在上面的代码段中，如果**pb**指向一个**D**类型的对象，**pd1**和**pd2**是一样的，并且对这两个指针执行**D**类型的任何操作都是安全的；但是，如果**pb**指向的是一个**B**类型的对象，那么**pd1**将是一个指向该对象的指针，对它进行**D**类型的操作将是不安全的（如访问**m\_szName**），而**pd2**将是一个空指针。

#### 4、reinterpret\_cast

在编译期处理

```
reinterpret_cast<type_id> (expression)
```

该运算符把**expression**重新解释成**type-id**类型的对象。对象在这里的范围包括变量以及实现类的对象。此标识符的意思即为数据的二进制形式重新解释，但是不改变其值。

## new和malloc的区别

### 1.申请的内存所在位置

**new**操作符从自由存储区（**free store**）上为对象动态分配内存空间，而**malloc**函数从堆上动态分配内存。

### 2.返回类型安全性

**new**操作符内存分配成功时，返回的是对象类型的指针，类型严格与对象匹配，无须进行类型转换，故**new**是符合类型安全性的操作符。而**malloc**内存分配成功则是返回**void \***，需要通过强制类型转换将**void**指针转换成我们需要的类型。

### 3.内存分配失败时的返回值

**new**内存分配失败时，会抛出**bad\_alloc**异常，它不会返回**NULL\***；**malloc**分配内存失败时返回**NULL**。

### 4.是否需要指定内存大小

使用**new**操作符申请内存分配时无须指定内存块的大小，编译器会根据类型信息自行计算，而**malloc**则需要显式地指出所需内存的尺寸。

5.**malloc**与**free**是C++/C语言的标准库函数，**new/delete**是C++的运算符。

6.**new** 建立的是一个对象,**malloc**分配的是一块内存。

## 堆和栈的区别(从数据结构和内存方面)

### 数据结构中的堆和栈

栈就像装数据的桶或箱子，是一种具有后进先出性质的数据结构。

堆像是一颗倒立的大树，堆是一种经过排序的树形数据结构，每个节点都有一个值。通常我们所说的堆的数据结构是指二叉树。堆的特点是根节点的值最小（或最大），且根节点的两个子树也是一个堆。由于堆的这个特性，常用来实现优先队列，堆的存取是随意的，这就如同我们在图书馆的书架上取书，虽然书的摆放是有顺序的，但是我们想取任意一本时不必像栈一样，先取出前面所有的书，书架这种机制不同于箱子，我们可以直接取出我们想要的书。

# 内存中的堆和栈

## 堆栈空间分配

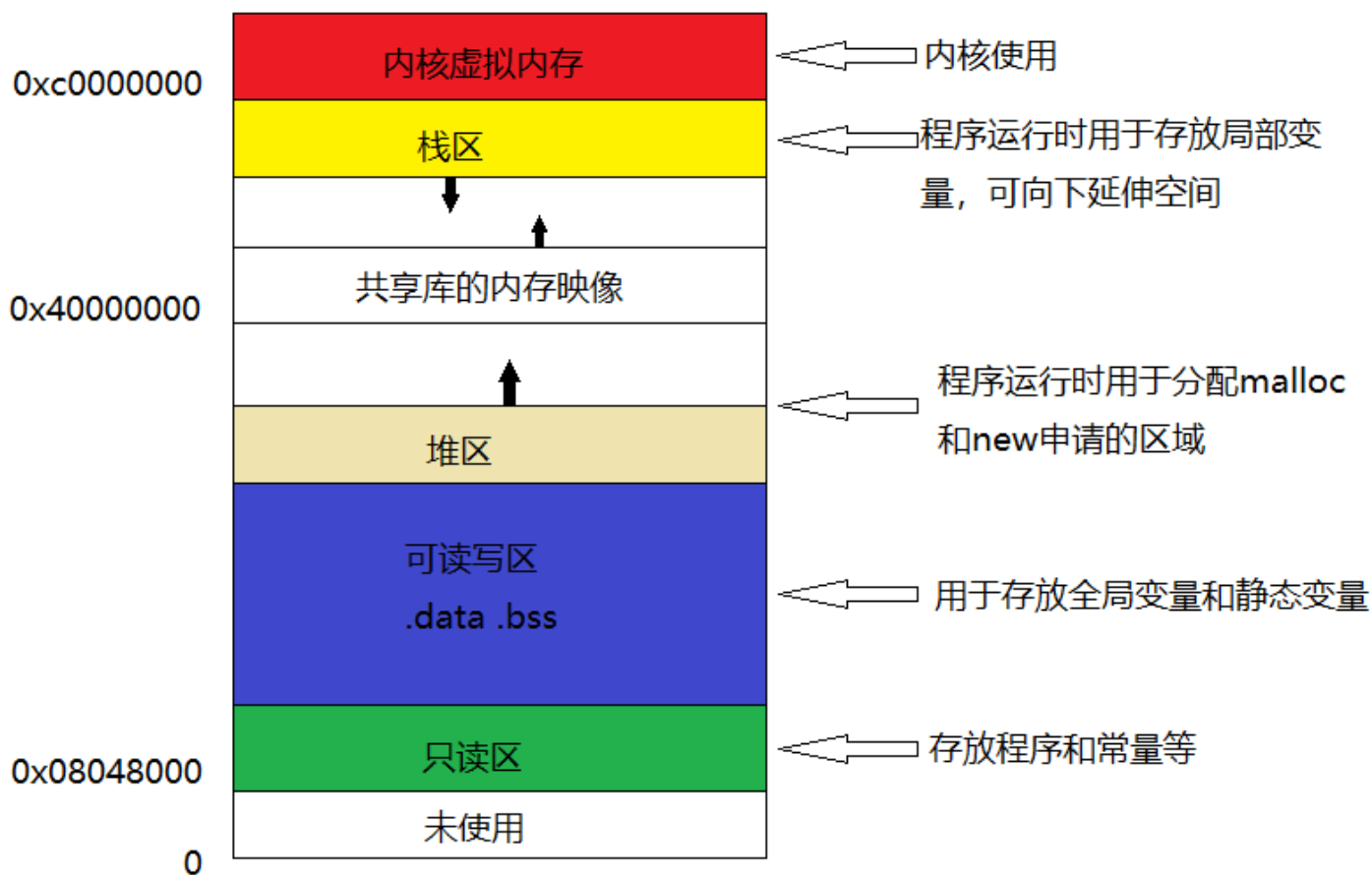
栈（操作系统）：由操作系统自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

堆（操作系统）：一般由程序员分配释放，若程序员不释放，程序结束时可能由OS回收，分配方式倒是类似于链表。

## 堆栈缓存方式

栈使用的是一级缓存，他们通常都是被调用时处于存储空间中，调用完毕立即释放。

堆则是存放在二级缓存中，生命周期由虚拟机的垃圾回收算法来决定（并不是一旦成为孤儿对象就能被回收）。所以调用这些对象的速度要相对来得低一些。





```

int a = 0; 全局初始化区
char *p1; 全局未初始化区
main()
{
    int b; 栈
    char s[] = "abc"; 栈
    char *p2; 栈
    char *p3 = "123456"; //123456\0在常量区，p3在栈上。
    static int c =0; 全局（静态）初始化区
    p1 = (char *)malloc(10); 堆
    p2 = (char *)malloc(20); 堆
}

```

### 1. 申请方式和回收方式不同

栈（stack）:由系统自动分配。例如，声明在函数中一个局部变量int b;系统自动在栈中为b开辟空间。

堆（heap）:需程序员自己申请（调用malloc,realloc,calloc）,并指明大小，并由程序员进行释放。容易产生memory leak。

```

char *p;
p = (char *)malloc(sizeof(char));

```

但是，p本身是在栈中。

由于栈上的空间是自动分配自动回收的，所以栈上的数据的生存周期只是在函数的运行过程中，运行后就释放掉，不可以再访问。而堆上的数据只要程序员不释放空间，就一直可以访问到，不过缺点是一旦忘记释放会造成内存泄露。

### 2. 申请后系统的响应

1）栈：只要栈的空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

2）堆：首先应该知道操作系统有一个记录空闲内存地址的链表，但系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲链表中删除，并将该结点的空间分配给程序，另外，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样，代码中的free语句才能正确的释放本内存空间。另外，找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。

说明：对于堆来讲，频繁的new/delete势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低。对于栈来讲，则不会存在这个问题

堆会在申请后还要做一些后续的工作这就会引出申请效率的问题。

### 3. 申请效率

栈由系统自动分配，速度快。但程序员是无法控制。

堆是由malloc分配的内存，一般速度比较慢，而且容易产生碎片，不过用起来最方便。



#### 4. 申请大小的限制

1) 栈：在Windows下,栈是向低地址扩展的数据结构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在 Windows下，栈的大小是2M（也有的说是1M，总之是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示overflow。因此，能从栈获得的空间较小。

2) 堆：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

#### 5. 堆和栈中的存储内容

1) 栈：在函数调用时，第一个进栈的是主函数中函数调用后的下一条指令（函数调用语句的下一条可执行语句）的地址，然后是函数的各个参数，在大多数的C编译器中，参数是由右往左入栈的，然后是函数中的局部变量。注意静态变量是不入栈的。

当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行。

2) 堆：一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容有程序员安排。

#### 6. 存取效率

堆：char \*s1="hellow tigerjibo";是在编译时就确定的；

栈：char s1[]="hellow tigerjibo";是在运行时赋值的；用数组比用指针速度更快一些，指针在底层汇编中需要用edx寄存器中转一下，而数组在栈上读取。

#### 7. 分配方式

堆都是动态分配的，没有静态分配的堆。

栈有两种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配由alloca函数进行分配，但是栈的动态分配和堆是不同的。它的动态分配是由编译器进行释放，无需手工实现。

## C++的内存分配有哪些？

**1.从静态存储区分配：**此时的内存在程序编译的时候已经分配好，并且在程序的整个运行期间都存在。全局变量，static变量等在此存储。

**2.在栈区分配：**相关代码执行时创建，执行结束时被自动释放。局部变量在此存储。栈内存分配运算内置于处理器的指令集中，效率高，但容量有限。

**3.在堆区分配：**动态分配内存。用new/malloc时开辟，delete/free时释放。生存期由用户指定，灵活。但有内存泄露等问题。

## 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