- 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，violiate，typedef，#define
  - const作用:
  - #define
  - typedef
  - typedef和#define区别
  - const与#define区别
  - #define和inline的区别
  - static
  - typeof

    - violiate
- 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？
- C++编译过程
    - 头文件是否参与编译
    - 如何防止头文件重复编译
- 静态库和动态库的区别
- 成员函数可以调用delete this吗？
- explicit的作用
- 什么是隐式类型转换？
- #include 的顺序以及尖叫括号和双引号的区别
- 对内存对齐的理解，为什么要内存对齐
- new具体是怎么开辟内存的
- stable_sort和sort区别
- 机器为什么使用补码？
- main函数在执行前和执行后有哪些操作
- 模板类了解吗（类模板）
- 编译器如何识别函数重载
- 对线程池有什么了解？
- emplace / empalce_front / emplace_back
- int atoi(char *)
- 库函数和系统调用区别
- inline函数怎么理解 为什么可以加快运行
- 负数二进制表示
- 指针和引用的区别
- new/delete、malloc/free的区别和联系
- 声明和定义的区别？
- 解释一下.so文件；
- int a[10]，求sizeof（a）和sizeof（a*）；
- int (*a)[10] 解释；（指针数组）

- 堆内存和栈内存区别与联系？
- 字节序的概念？（LINUX大端小端的概念，字节存储的顺序高低位）
- 枚举类型的大小？
- union和结构体的区别与联系？
- Qt当中信号与槽机制是怎么实现的
- 进行的相关操作，槽函数没有响应会因为什么
- 内存4G，malloc申请4.1G会发生什么
- 内存4G，已经有其他进程申请了2.5G了，此时malloc申请4G，会发生什么
- 如果物理内存是2G 如果mallco 4G可以么？会有什么问题？
- const char* p=“hello world” 这个“hello world”在内存的哪个位置
- static变量放在头文件会产生什么问题？
- inline关键字在什么情况下会展开失败？
- debug和release的区别
- 讲一下指针传递和引用传递
- 矩阵乘法代码
- 局部性原理
- 如何在一个函数通过调用地址修改另一个函数的临时变量
- 在main函数中定义了a和b两个int变量，调用sum函数求其和，说一下其压栈过程
- 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

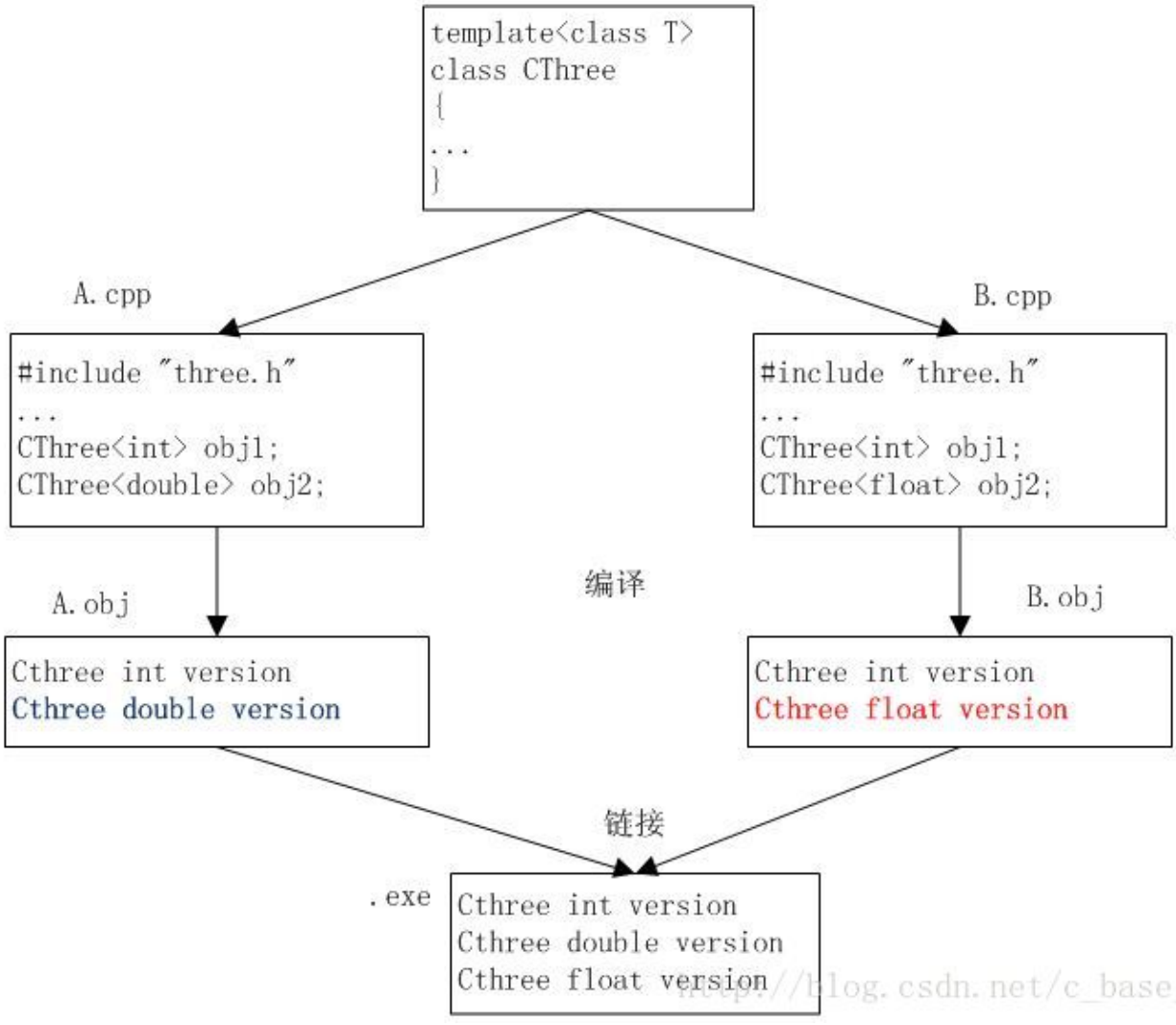
可以像C++ Primer中所述的，使用export关键字。（你可以在.h文件中，声明模板类和模板函数；在.cpp文件中，使用关键字export来定义具体的模板类对象和模板函数；然后在其他用户代码文件中，包含声明头文件后，就可以使用该这些对象和函数了）

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。

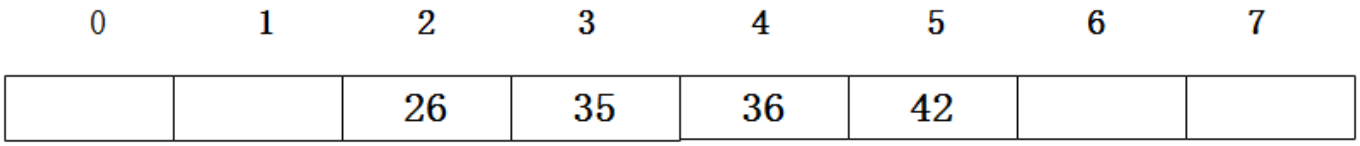
如何解决
1.开放地址法
当冲突发生时，使用某种探测算法在散列表中寻找下一个空的散列地址，只要散列表足够大，空的散列地址总能找到。按照探测序列的方法，一般将开放地址法区分为线性探查法、二次探查法、双重散列法等。
用以一个模为8的哈希表为例，采用除留余数法，往表中插入三个关键字分别为26，35，36的记录，分别除8取模后，在表中的位置如下：

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：
$fi=(f(key)+i) \% m$ ，$0 \le i \le m-1$
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。



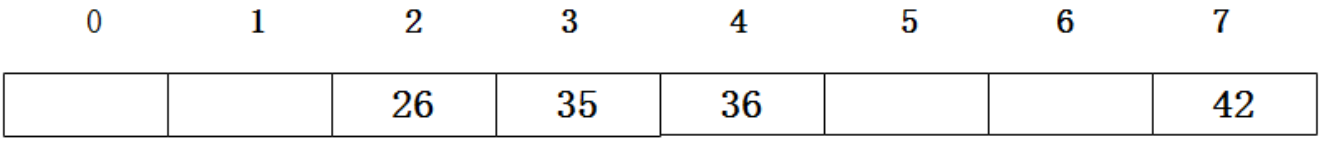
2） 二次探查法

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

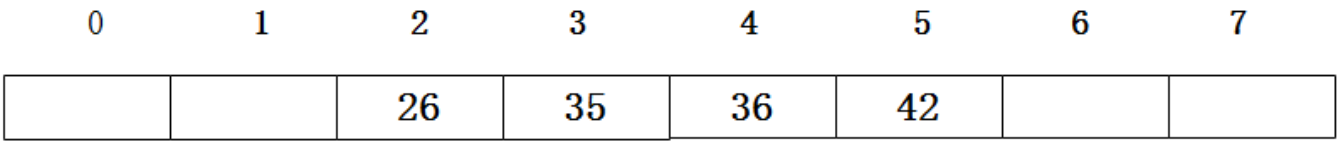
fi=(f(key)+i*g(key)) % m (i=1，2，……，m-1)

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，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，但是这并不是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cosnt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* cosnt 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。

#define和inline的区别

（1）内联函数在编译时展开，宏在预编译时展开；
（2）内联函数直接嵌入到目标代码中，宏是简单的做文本替换；

（3）内联函数有类型检测、语法判断等功能，而宏没有；
（4）inline函数是函数，宏不是；
（5）宏定义时要注意书写（参数要括起来）否则容易出现歧义，内联函数不会产生歧义；

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

violiate

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()才推进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。

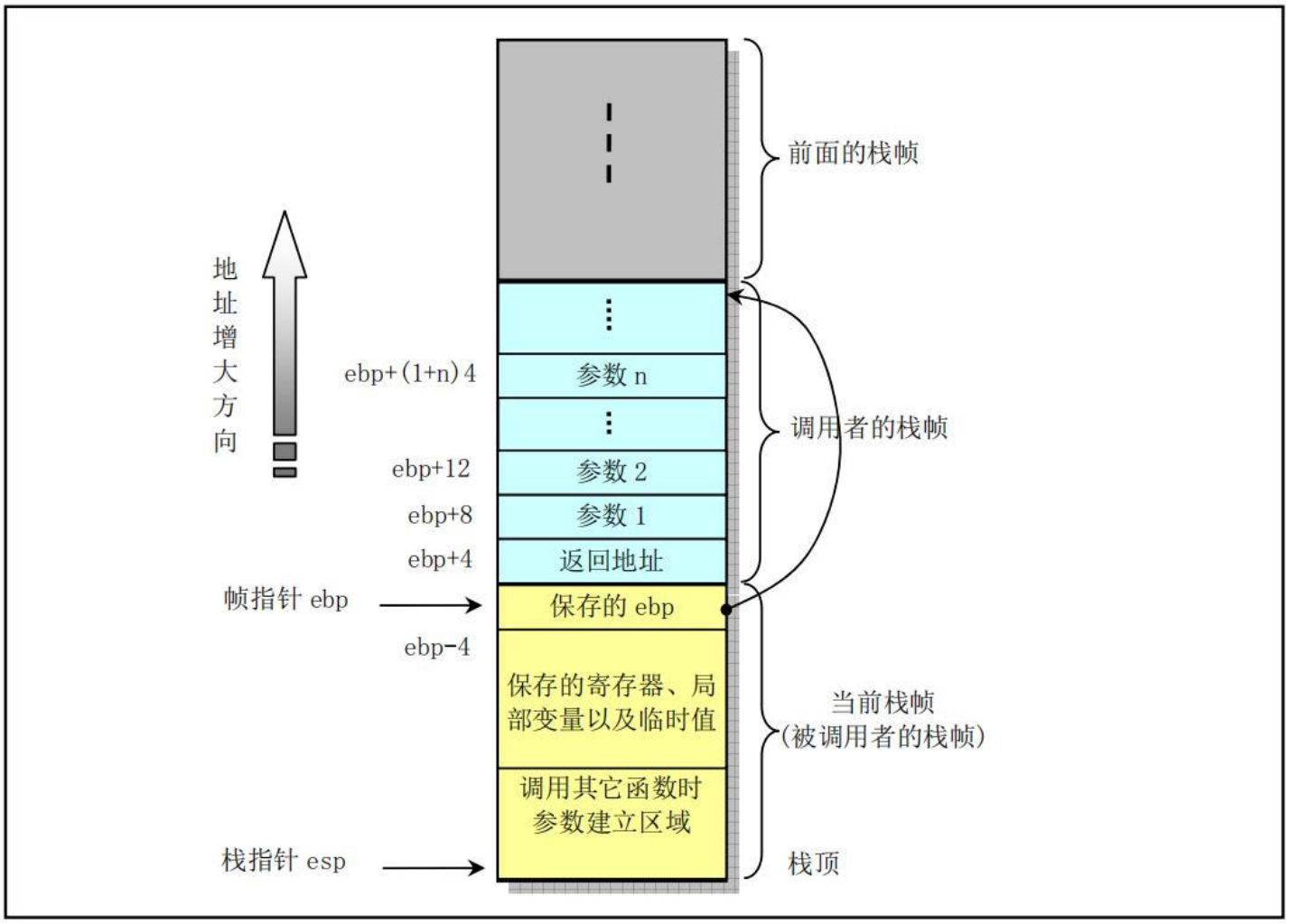


图 3-4 栈中帧结构示意图

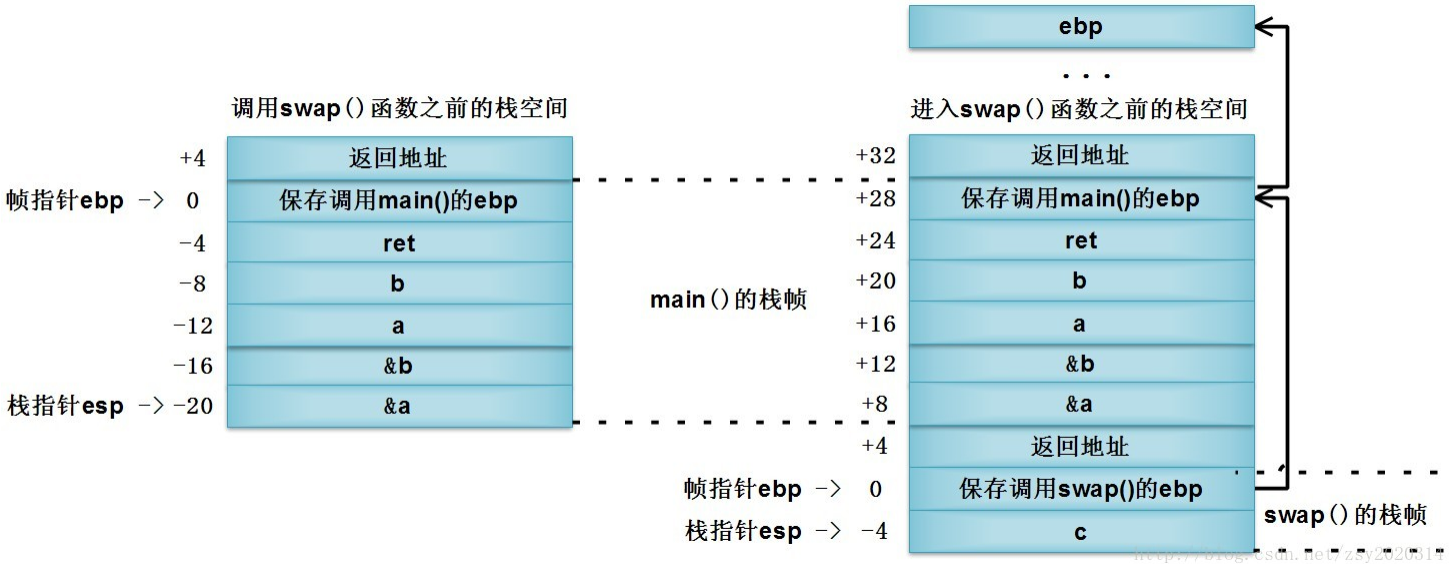
来分析下面的程序，下面的程序的功能就是调用swap函数交换两个数的值，然后返回两个数的差。

```
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

可以看出，在调用swap函数后，会为更改栈指针和帧指针的指向，为swap分配一个栈帧结构。



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bac_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

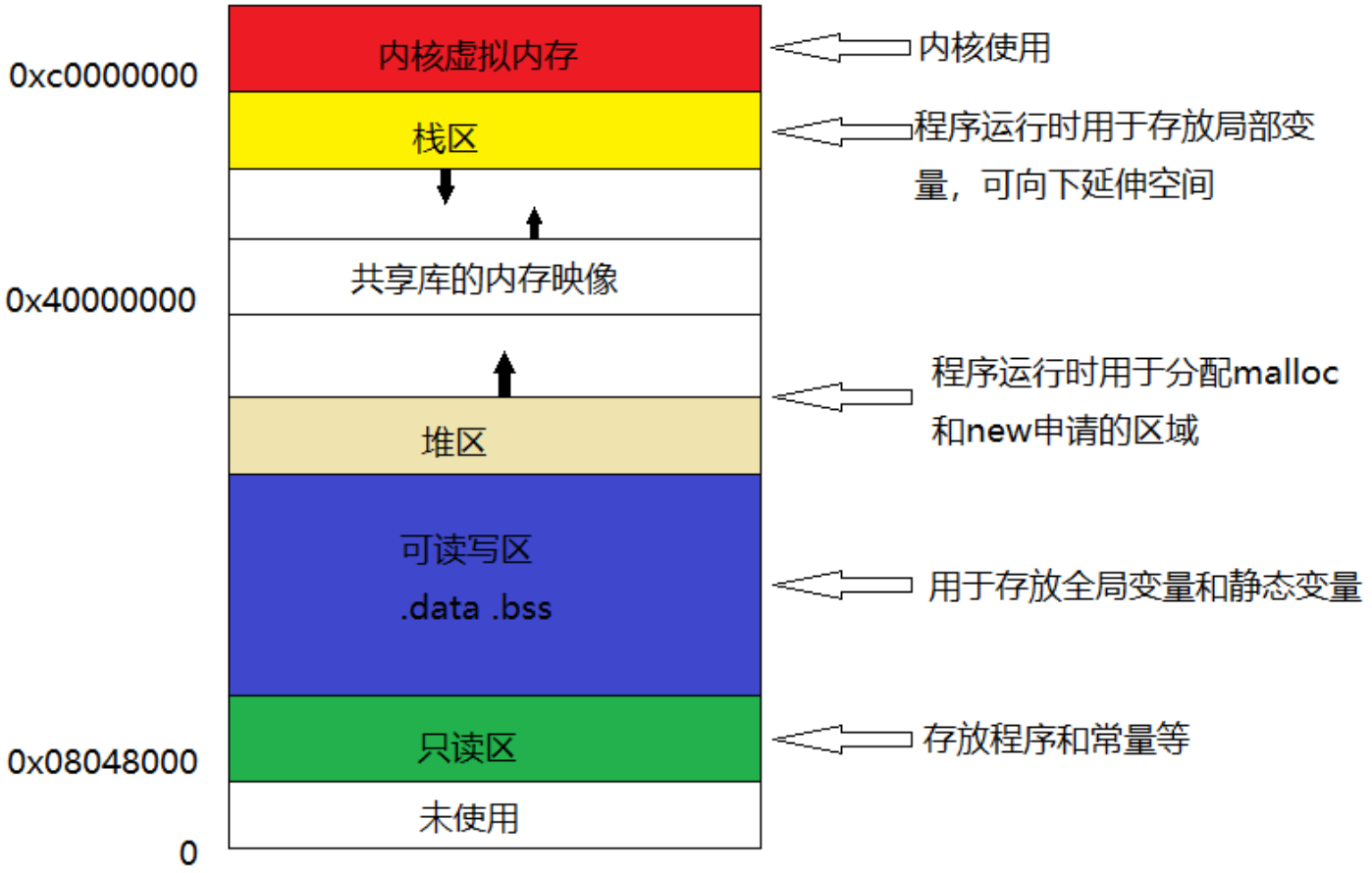
栈（操作系统）：由操作系统自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

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（satck）: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”;是在编译是就确定的；

   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。

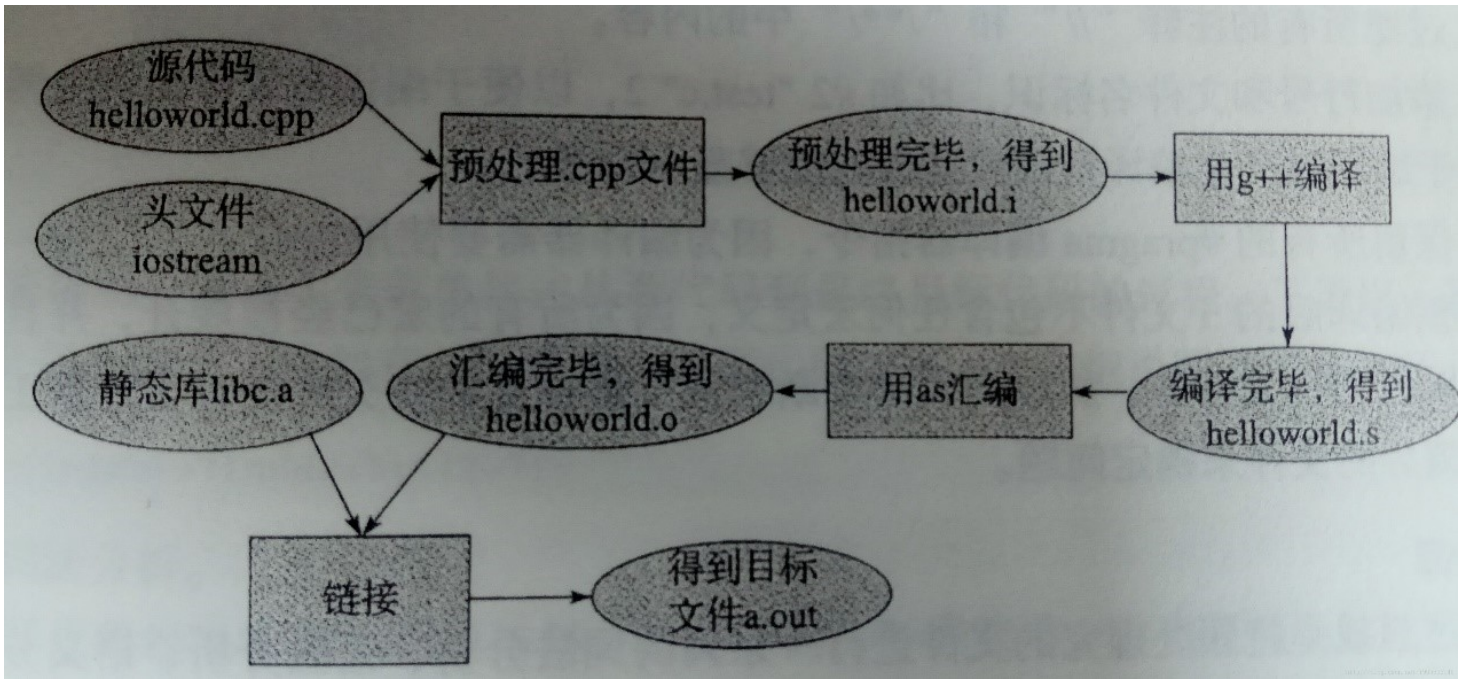
C++的内存分配有哪些？

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。

C++编译过程



a）预处理（产生.i文件）

```
g++ -E helloworld.cpp -o helloworld.i
```

1. 将#define删除，并将宏定义展开
2. 处理一些条件预编译指令，如#ifndef,#ifdef,#elif,#else,#endif（作用：防止重复包含头文件）等。将不必要的代码过滤掉。
3. 处理#include预编译指令，将被包含的文件插入到该预编译指令的位置。这个过程是递归进行的，因为被包含的文件也包含其他文件
4. 过滤掉所有注释里面的内容
5. 添加行号和文件名标识
6. 保留#program编译器指令，因为编译器需要使用他们

b）编译（产生.s文件）

```
g++ -S helloworld.i -o helloworld.s
```

编译就是将预处理的文件进行一系列的词法分析、语法分析、语义分析，以及优化后产生相应的汇编代码文件。

c）汇编（产生.o或.obj文件）

汇编过程实际上是把汇编语言代码翻译成目标机器指令的过程，即生成目标文件。目标文件中所存放的也就是与源程序等效的目标机器语言代码。目标文件由段组成，通常一个目标文件至少有两个段：

1. 代码段：该段中所包含的主要是程序的指令，该段一般是可读和可执行的，但一般不可写。

2. 数据段：主要存放程序中要用到的各种全局变量或静态的数据。一般数据段都是可读、可写、可执行的。

UNIX环境下主要有三类目标文件：

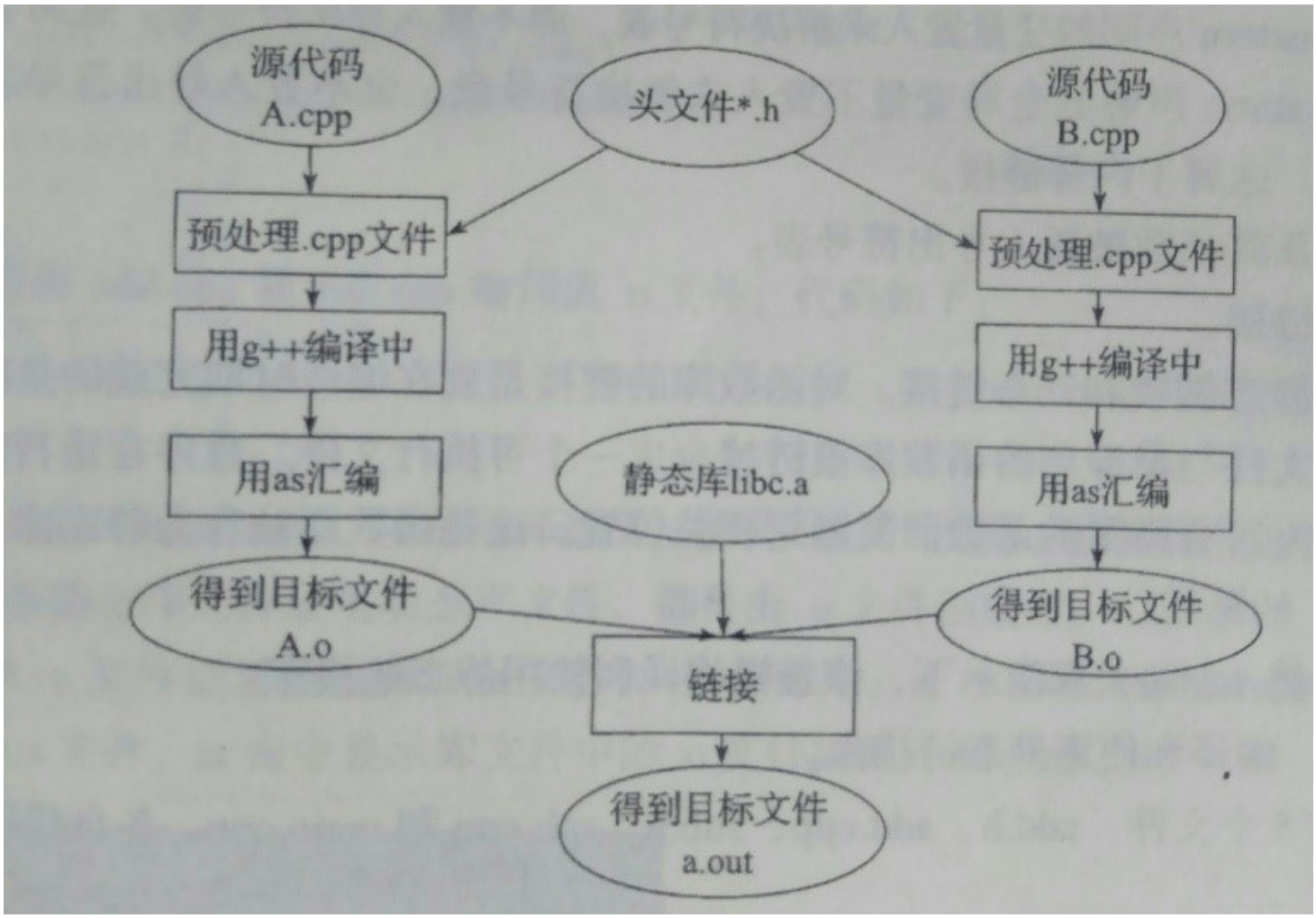
1. 可重定位文件：其中包含有适合于其它目标文件链接来创建一个可执行的或者共享的目标文件的代码和数据。
2. 共享的目标文件：这种文件存放了适合于在两种上下文里链接的代码和数据。第一种是链接程序可把它与其它可重定位文件及共享的目标文件一起处理来创建另一个目标文件；第二种是动态链接程序将它与另一个可执行文件及其它的共享目标文件结合到一起，创建一个进程映象。
3. 可执行文件：它包含了一个可以被操作系统创建一个进程来执行之的文件。

汇编程序生成的实际上是第一种类型的目标文件，对于后两种还需要其他的处理才能得到，这个就是连接程序的工作了。

d）链接（产生.out或.exe文件）

链接就是把每个源代码独立的编译，然后按照它们的要求将它们组装起来，链接主要解决的是源代码之间的相互依赖问题，链接的过程包括地址和空间的分配，符号决议，和重定位等这些步骤。

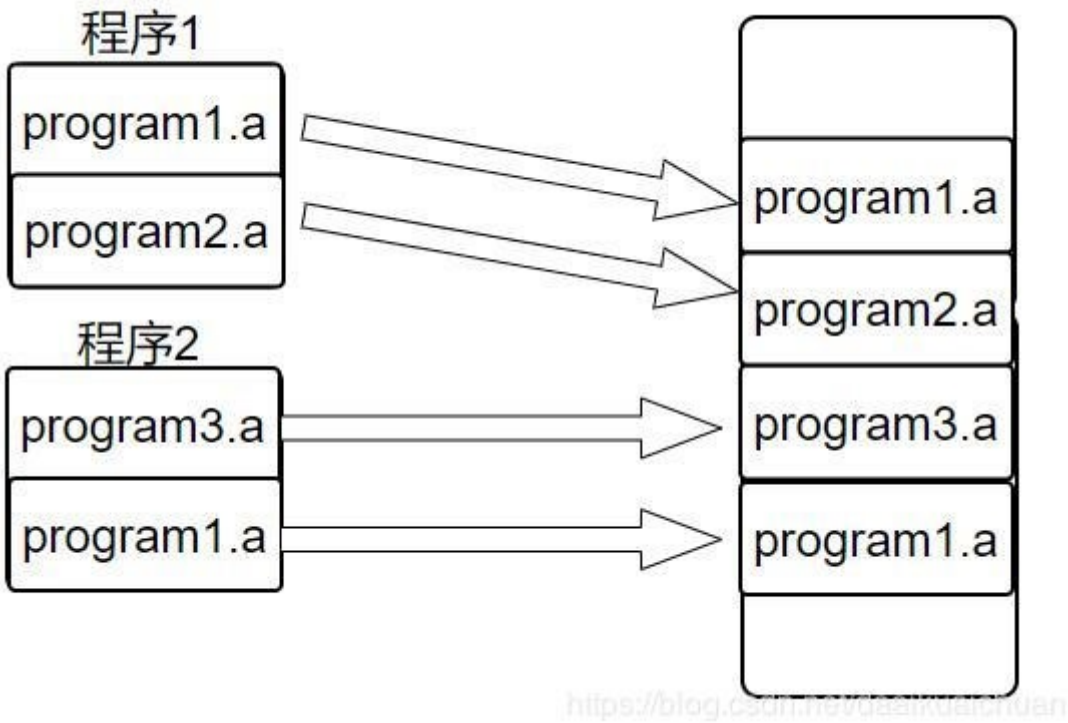
动态链接如图所示：



静态链接/库

在链接阶段，会将汇编生成的目标文件.o与引用到的库一起链接打包到可执行文件中，因此对应的链接方式称为静态链接。

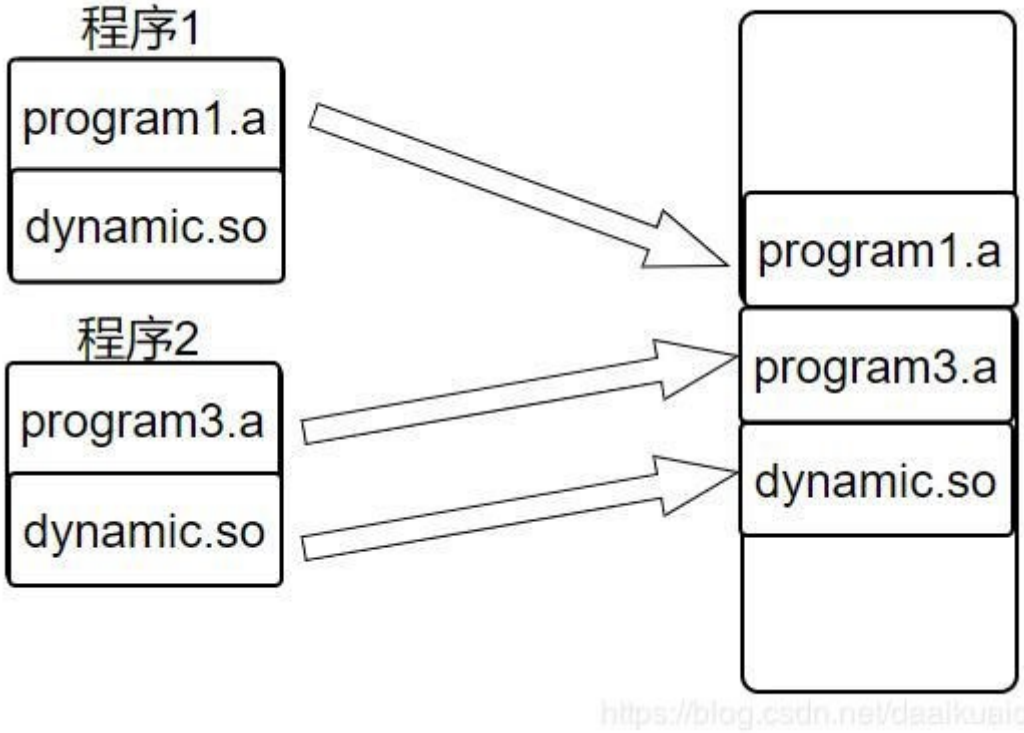
静态库可以简单看成是一组目标文件（.o/.obj文件）的集合，即很多目标文件经过压缩打包后形成的一个文件。



静态库的缺点在于：浪费空间和资源，因为所有相关的目标文件与牵涉到的函数库被链接合成一个可执行文件。

动态链接/库

动态库在程序编译时并不会被连接到目标代码中，而是在程序运行是才被载入。不同的应用程序如果调用相同的库，那么在内存里只需要有一份该共享库的实例，规避了空间浪费问题。动态库在程序运行是才被载入，也解决了静态库对程序的更新、部署和发布页会带来麻烦。用户只需要更新动态库即可，增量更新。



头文件是否参与编译

会参与预编译。
.h的内容被插入到.c中，作为.c的内容被编译。.h文件本身不直接参加编译。
头文件不用被编译。我们把所有的函数声明全部放进一个头文件中，当某一个.cpp源文件需要它们时，它们就可以通过一个宏命令 “#include”包含进这个.cpp文件中，从而把它们的内容合并到.cpp文件中去。当.cpp文件被编译时，这些被包含进去的.h文件的作用便发挥了。

如何防止头文件重复编译

1. #ifndef,#ifdef,#elif,#else,#endif
2. #pragma once

除了#pragma once是微软编译器所特有的之外，用宏和#pragma once的办法来避免重复包含头文件，主要区别在于宏处理的方法会多次打开同一头文件，而#pragma once则不会重复打开，从而#pragma once能够更快速。
#pragma once指定当前文件在构建时只被包含(或打开)一次,这样就可以减少构建的时间,因为加入#pragma once后,编译器在打开或读取第一个#include 模块后,就不会再打开或读取随后出现的同#include 模块。

静态库和动态库的区别

1、 静态链接库的后缀名为lib，动态链接库的导入库的后缀名也为lib。不同的是，静态库中包含了函数的实际执行代码，而对于导入库而言，其实际的执行代码位于动态库中，导入库只包含了地址符号表

等，确保程序找到对应函数的一些基本地址信息；
2、由于静态库是在编译期间直接将代码合到可执行程序中，而动态库是在执行期时调用DLL中的函数体，所以执行速度比动态库要快一点；
3、 静态库链接生成的可执行文件体积较大，且包含相同的公共代码，造成内存浪费；
4、 使用动态链接库的应用程序不是自完备的，它依赖的DLL模块也要存在，如果使用载入时动态链接，程序启动时发现DLL不存在，系统将终止程序并给出错误信息。而使用运行时动态链接，系统不会终止，但由于DLL中的导出函数不可用，程序会加载失败；
5、 DLL文件与EXE文件独立，只要输出接口不变（即名称、参数、返回值类型和调用约定不变），更换DLL文件不会对EXE文件造成任何影响，因而极大地提高了可维护性和可扩展性，适用于大规模的软件开发，使开发过程独立、耦合度小，便于不同开发者和开发组织之间进行开发和测试。

成员函数可以调用delete this吗？

在delete this之后，会释放掉类的对象的内存空间，因此如果在delete this之后进行的其他任何函数调用，只要不涉及到this指针的内容，都能够正常运行。一旦涉及到this指针，如操作数据成员，调用虚函数等，就会出现不可预期的问题。

delete this之后不是释放了类对象的内存空间了么，那么这段内存应该已经还给系统，不再属于这个进程。照这个逻辑来看，应该发生指针错误，无访问权限之类的令系统崩溃的问题才对啊？这个问题牵涉到操作系统的内存管理策略。delete this释放了类对象的内存空间，但是内存空间却并不是马上被回收到系统中，可能是缓冲或者其他什么原因，导致这段内存空间暂时并没有被系统收回。此时这段内存是可以访问的，你可以加上100，加上200，但是其中的值却是不确定的。当你获取数据成员，可能得到的是一串很长的未初始化的随机数；访问虚函数表，指针无效的可能性非常高，造成系统崩溃。

如果在类的析构函数中调用delete this，会发生什么？实验告诉我们，会导致堆栈溢出。原因很简单，delete的本质是“为将被释放的内存调用一个或多个析构函数，然后，释放内存” (来自effective c++)。显然，delete this会去调用本对象的析构函数，而析构函数中又调用delete this，形成无限递归，造成堆栈溢出，系统崩溃。

explicit的作用

explicit 关键字只能用于类内部的构造函数声明上。被修饰的构造函数的类，不能发生相应的隐式类型转换。

什么是隐式类型转换？

c++自动将一种类型转换成另一种类型，是编译器的一种自主行为。

#include 的顺序以及尖叫括号和双引号的区别

预处理器发现 #include 指令后，就会寻找后面跟的文件名并把这个文件的内容包含到当前文件中。被包含文件中的文本将替换源代码文件中的#include指令，就像你把被包含文件中的全部内容键入到源文件中的这个位置一样。但是包含头文件有两种方式，尖括号和双引号。
尖括号：表示编译器只在**系统默认目录**或尖括号内的工作目录下搜索头文件，并不去用户的工作目录下寻找，所以一般尖括号用于包含标准库文件，例如：stdio.h,stdlib.h。
双引号：表示编译器先在**用户的工作目录**下搜索头文件，如果搜索不到则到系统默认目录下去寻找，所以双引号一般用于包含用户自己编写的头文件。

对内存对齐的理解，为什么要内存对齐

访问未对齐的内存，处理器要访问两次（数据先读高位，再度地位），访问对齐的内存，处理器只要访问一次，为了提高处理器读取数据的效率，我们使用内存对齐。Windows 默认对齐数为8字节，Linux 默认对齐数为4字节。

new具体是怎么开辟内存的

（1）简单数据类型（包括基本数据类型和不需要构造函数的类型）

- 简单类型直接调用operator new分配内存；
- 可以通过new_handler来处理new失败的情况；
- new分配失败的时候不像malloc那样返回NULL，它直接抛出异常。要判断是否分配成功应该用异常捕获的机制；

（2）复杂数据类型（需要由构造函数初始化对象）

- new 复杂数据类型的时候先调用operator new，然后在分配的内存上调用构造函数。

stable_sort和sort区别

带有stable的函数可保证相等元素的原本相对次序在排序后保持不变。

机器为什么使用补码？

（1）根据运算法则减去一个正数等于加上一个负数， 即: 1-1 = 1+(-1)， 所以计算机被设计成只有加法而没有减法， 而让计算机辨别”符号位”会让计算机的基础电路设计变得十分复杂，于是就让符号位也参与运算，从而产生了反码
（2）用反码计算， 出现了”0”这个特殊的数值， 0带符号是没有任何意义的。 而且会有[0000 0000]和[1000 0000]两个编码表示0。于是设计了补码， 负数的补码就是反码+1，正数的补码就是正数本身，从而解决了0的符号以及两个编码的问题: 用[0000 0000]表示0，用[1000 0000]表示-128。
（3）-128实际上是使用以前的-0的补码来表示的，所以-128并没有原码和反码。使用补码， 不仅仅修

复了0的符号以及存在两个编码的问题， 而且还能够多表示一个最低数。 这就是为什么8位二进制， 使用补码表示的范围为[-128， 127]。

补码就是最方便的方式。它的便利体现在，所有的加法运算可以使用同一种电路完成。
数的原码表示形式简单，适用于乘除运算，但用原码表示的数进行加减法运算比较复杂，引入补码之后，减法运算可以用加法来实现，且数的符号位也可以当作数值一样参与运算，因此在计算机中大都采用补码来进行加减法运算

main函数在执行前和执行后有哪些操作

main函数执行之前，主要就是初始化系统相关资源：
1. 设置栈指针
2. 初始化static静态和global全局变量，即data段的内容
3. 将未初始化部分的全局变量赋初值：数值型short，int，long等为0，bool为FALSE，指针为NULL，等等，即.bss段的内容
4. 全局对象初始化，在main之前调用构造函数
5. 将main函数的参数，argc，argv等传递给main函数，然后才真正运行main函数

main函数执行之后：
1. 全局对象的析构函数会在main函数之后执行；
2. 可以用_onexit 注册一个函数，它会在main 之后执行;

模板类了解吗（类模板）

由类模板实例化得到的类叫模板类。

类模板使用template来声明。可以定义相同的操作，拥有不同数据类型的成员属性。

编译器如何识别函数重载

C++将会对重载的函数进行名称修饰或者叫名称矫正

```
int fun(int a)
int fun(float b)
```

这样的重载函数 在编译器下就可能是?fun@@YXX 和?fun@@XXY这样的进行了貌似无意义的修饰 用于编译器的识别。

对线程池有什么了解？

线程池采用预创建的技术，在应用程序启动之后，将立即创建一定数量的线程(N1)，放入空闲队列中。这些线程都是处于阻塞（Suspended）状态，不消耗CPU，但占用较小的内存空间。当任务到来后，缓冲池选择一个空闲线程，把任务传入此线程中运行。当N1个线程都在处理任务后，缓冲池自动创建一定数量的新线程，用于处理更多的任务。在任务执行完毕后线程也不退出，而是继续保持在池中等待下一次的任务。当系统比较空闲时，大部分线程都一直处于暂停状态，线程池自动销毁一部分线程，回收系统资源。

线程池通常适合下面的几个场合：
(1)单位时间内处理任务频繁而且任务处理时间短
(2)对实时性要求较高。如果接受到任务后在创建线程，可能满足不了实时要求，因此必须采用线程池进行预创建。

emplace / empalce_front / emplace_back

emplace操作是C++11新特性，新引入的的三个成员emlace_front、empace 和 emplace_back,这些操作构造而不是拷贝元素到容器中，这些操作分别对应push_front、insert 和push_back，允许我们将元素放在容器头部、一个指定的位置和容器尾部。

push_back()函数向容器中加入一个临时对象（右值元素）时， 首先会调用构造函数生成这个对象，然后条用拷贝构造函数将这个对象放入容器中， 最后释放临时对象。但是emplace_back()函数向容器中中加入临时对象， 临时对象原地构造，没有赋值或移动的操作。

int atoi(char *)

atoi()原型: int atoi(const char *str);
函数功能：把字符串转换成整型数。
参数str：要进行转换的字符串
返回值：每个函数返回 int 值，此值由将输入字符作为数字解析而生成。 如果该输入无法转换为该类型的值，则atoi的返回值为 0。

```c
int my_atoi(char* pstr)
{
        int Ret_Integer = 0;
        int Integer_sign = 1;

        /*
        * 判断指针是否为空
        */
        if(pstr == NULL)
        {
                printf("Pointer is NULL\n");
                return 0;
        }

        /*
        * 跳过前面的空格字符
        */
        while(isspace(*pstr) == 0)
        {
                pstr++;
        }

        /*
        * 判断正负号
        * 如果是正号，指针指向下一个字符
        * 如果是符号，把符号标记为Integer_sign置-1，然后再把指针指向下一个字符
        */
        if(*pstr == '-')
        {
                Integer_sign = -1;
        }
        if(*pstr == '-' || *pstr == '+')
        {
                pstr++;
        }

        /*
        * 把数字字符串逐个转换成整数，并把最后转换好的整数赋给Ret_Integer
        */
        while(*pstr >= '0' && *pstr <= '9')
        {
                Ret_Integer = Ret_Integer * 10 + *pstr - '0';
                pstr++;
        }
        Ret_Integer = Integer_sign * Ret_Integer;

        return Ret_Integer;
}
```

库函数和系统调用区别

系统调用

系统调用指运行在用户空间的程序向操作系统内核请求需要更高权限运行的服务。它通过软中断向内核态发出一个明确的请求。系统调用实现了用户态进程和硬件设备之间的大部分接口。

库函数

库函数用于提供用户态服务。它可能调用封装了一个或几个不同的系统调用（printf调用write），也可能直接提供用户态服务（atoi不调用任何系统调用）。

inline函数怎么理解 为什么可以加快运行

一个函数的程序进行代码拓展而不被调用，用相应的函数代码替换函数调用。
inline主要是为了减少函数调用的开销，而用相应的代码代替函数调用

负数二进制表示

在计算机中，
正数是直接用原码表示的，如单字节5，在计算机中就表示为：0000 0101。
负数以其正值的补码形式表示，如单字节-5，在计算机中表示为1111 1011。

原码：
一个正数的原码，是按照绝对值大小转换成的二进制数；
一个负数的原码，是按照绝对值大小转换成的二进制数，然后最高位补1。

比如 ：
00000000 00000000 00000000 00000101是 5的 原码。
10000000 00000000 00000000 00000101是 -5的 原码。

反码：
正数的反码与原码相同，负数的反码为对该数的原码除符号位外各位取反。

取反操作指：原为1，得0；原为0，得1。（1变0; 0变1）

比如：
正数00000000 00000000 00000000 00000101的反码还是00000000 00000000 00000000 00000101
负数10000000 00000000 00000000 00000101每一位取反（除符号位），得11111111 11111111 11111111 11111010。

称：11111111 11111111 11111111 11111010是 10000000 00000000 00000000 00000101的反码。
反码是相互的，所以也可称：
10000000 00000000 00000000 00000101和 11111111 11111111 11111111 11111010互为反码。

补码：
正数的补码与原码相同；
负数的补码为对该数的原码除符号位外各位取反，然后在最后一位加1.
比如：
10000000 00000000 00000000 00000101的反码是：11111111 11111111 11111111 11111010。
那么，补码为：
11111111 11111111 11111111 11111010 + 1 = 11111111 11111111 11111111 11111011

所以，-5 在计算机中表达为：11111111 11111111 11111111 11111011。转换为十六进制：0xFFFFFFFB。

指针和引用的区别

1、 指针有自己的一块空间，而引用只是一个别名
2、 使用sizeof，一个指针大小是4，而引用则是被引用对象的大小
3、 指针可以初始化为NULL，而引用必须被初始化且必须是一个已有对象的引用
4、 使用参数传递时，指针需要被解引用才可以对对象进行操作，而直接对引用的修改都会改变引用所指向的对象
5、 指针在使用中可以指向其他对象，而引用只能是一个对象的引用
6、 指针可以有多级指针，而引用只有一级
7、 指针和引用使用++运算符的意义不一样。对引用的操作直接反应到所指向的对象，而不是改变指向；而对指针的操作，会使指针指向下一个对象，而不是改变所指对象的内容。
指针自加，比如 int a[2] = {0,10} ； int *pa =a；pa++表示指针往后移动一个int的长度。指向下一个内存地址。及pa从指向a[0]变成指向a[1]引用是值++；比如b是引用a[0]的，++表示a[0]的值++从0变为1；
int a=0;
int b=&a;
int *p=&a;
b++;相当于a++;b只是a的一个别名，和a一样使用。
p++;后p指向a后面的内存
(*p)++;相当于a++
8、 如果返回动态内存分配的对象或者内存，必须使用指针，引用可能会引起内存泄漏
9、引用比指针更安全。由于不存在空引用，并且引用一旦被初始化为指向一个对象，它就不能被改变为另一个对象的引用，因此引用很安全。对于指针来说，它可以随时指向别的对象，并且可以不被初始化，或为NULL，所以不安全。const 指针虽然不能改变指向，但仍然存在空指针，并且有可能产生野指针（即多个指针指向一块内存，free掉一个指针之后，别的指针就成了野指针）。

new/delete、malloc/free的区别和联系

(1) 属性
new/delete是C++关键字，需要编译器支持。malloc/free是库函数，需要c头文件支持。

(2) 参数
使用new操作符申请分配内存时无需指定内存块的大小，编译器会根据类型信息自行计算。而malloc则需要显式得指出所需内存的尺寸。
(3) 返回类型
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。
(4) 分配失败
new内存分配失败时，会抛出bac_alloc异常。malloc分配内存失败时返回NULL。（5）自定义类型
new会先调用operator new函数，申请足够的内存（通常底层使用malloc实现）。然后调用类型的构造函数，初始化成员变量，最后返回自定义类型指针。delete先调用析构函数，然后调用operator delete函数释放内存（通常底层使用free实现）。
malloc/free是库函数，只能动态的申请和释放内存，无法强制要求其做自定义类型对象构造和析构工作。
（6）重载
C++允许重载new/delete操作符，特别的，布局new的就不需要为对象分配内存，而是指定了一个地址作为内存起始区域，new在这段内存上为对象调用构造函数完成初始化工作，并返回此地址。而malloc不允许重载。
（7）内存区域
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。自由存储区是C++基于new操作符的一个抽象概念，凡是通过new操作符进行内存申请，该内存即为自由存储区。而堆是操作系统中的术语，是操作系统所维护的一块特殊内存，用于程序的内存动态分配，C语言使用malloc从堆上分配内存，使用free释放已分配的对应内存。自由存储区不等于堆，如上所述，布局new就可以不位于堆中。
C++中，内存区分为5区，分别是堆、栈、自由存储区、全局/静态存储区、常量存储区；

声明和定义的区别？

①变量定义：用于为变量分配存储空间，还可为变量指定初始值。程序中，变量有且仅有一个定义。
②变量声明：用于向程序表明变量的类型和名字。
③定义也是声明：当定义变量时我们声明了它的类型和名字。
④extern关键字：通过使用extern关键字声明变量名而不定义它。

函数也有声明和定义，但由于函数的声明和定义是有区别的，函数的定义是有函数体的，所以函数的声明和定义都可以将extern省略掉，反正其他文件也是知道这个函数是在其他地方定义的。

解释一下.so文件；

- .o文件是源码编译出的二进制文件

- .a文件实质上就是.o文件打了个包。一般把它叫做静态库文件。它在使用的时候，效果和使用.o文件是一样的。
- .so文件就不一样了,它不是简单的.o文件打了一个包，它是一个ELF格式的文件，也就是linux的可执行文件。
  .so文件可以用于多个进程的共享使用（位置无关的才行），所以又叫共享库文件。程序在使用它的时候，会在运行时把它映射到自己进程空间的某一处，其不在使用它的程序中。

int a[10]，求sizeof（a）和sizeof（a*）；

sizeof(a) = sizeof(int)*10
sizeof(*a)就是指针的大小

int (*a)[10] 解释；（指针数组）

int *a[10] ：数组指针。数组a里存放的是10个int型指针
int (*a)[10] ：a是指针，指向一个数组。此数组有10个int型元素

int *a[10]
先找到声明符a,然后向右看，有[]说明a是个数组，再向左看，是int *，说明数组中的每个元素是int *。所以这是一个存放int指针的数组。

int(*a)[10]*
*先找到声明符a,被括号括着，先看括号内的(优先级高)，然后向右看，没有，向左看,是，*说明a是个指针，什么指针？在看括号外面的，先向右看，有[] 是个数组，说明a是个指向数组的指针，再向左看，是int，说明数组的每个元素是int。所以，这是一个指向存放int的数组的指针。

堆内存和栈内存区别与联系？

1 栈：为编译器自动分配和释放，如函数参数、局部变量、临时变量等等
2 堆：为成员分配和释放，由程序员自己申请、自己释放。否则发生内存泄露。典型为使用new申请的堆内容。

除了这两部分，还有一部分是：
3 静态存储区：内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。它主要存放静态数据、全局数据和常量。

字节序的概念？（LINUX大端小端的概念，字节存储的顺序高低位）

首先，要明确以下两点：

- 一 双字节数据以上有高字节和低字节之分
- 二 字节在内存中从低地址到高地址依次存放
  这样，以字WORD(双字节)数据0x1234为例：

大端字节序：字数据高字节存储在内存的低地址，而低字节存储在内存中的高地址 。如0x12存储在地址a处，则0x34存储在a+1处[即：高对低，低对高].

小端字节序：字数据高字节存储在内存的低地址，而低字节存储在内存中的高地址 。如0x34存储在地址a处，则0x12存储在a+1处[即：高对高，低对低].

```
//int--char的强制转换，是将低地址的数值截断赋给char，利用这个准则可以判断系统是大端序还是小端序
#include <iostream>
using namespace std;
int main()
{
    int a = 0x1234;
    char c = static_cast<char>(a);
    if (c == 0x12)
        cout << "big endian" << endl;
    else if(c == 0x34)
        cout << "little endian" << endl;
}
```

枚举类型的大小？

枚举类型的尺寸是以能够容纳最大枚举子的值的整数的尺寸

union和结构体的区别与联系？

union （ 共用体）：构造数据类型,也叫联合体
用途：使几个不同类型的变量共占一段内存(相互覆盖)

struct (结构体)：是一种构造类型
用途： 把不同的数据组合成一个整体——自定义数据类型

主要区别：

1. struct和union都是由多个不同的数据类型成员组成, 但在任何同一时刻, union中只存放了一个被选中的成员; 而struct的所有成员都存在。在struct中，各成员都占有自己的内存空间，它们是同时存在的,一个struct变量的总长度等于所有成员长度之和，遵从字节对其原则; 在Union中，所有成员不能同时占用它的内存空间，它们不能同时存在 , Union变量的长度等于最长的成员的长度。
2. 对于union的不同成员赋值, 将会对其它成员重写, 原来成员的值就不存在了,所以，共同体变量中起作用的成员是最后一次存放的成员; 而对于struct的不同成员赋值是互不影响的。

Qt当中信号与槽机制是怎么实现的

信号槽，实际就是观察者模式。当某个事件发生之后，比如，按钮检测到自己被点击了一下，它就会发出一个信号（signal）。这种发出是没有目的的，类似广播。如果有对象对这个信号感兴趣，它就会使用连接（connect）函数，意思是，将想要处理的信号和自己的一个函数（称为槽（slot））绑定来处理这个信号。也就是说，当信号发出时，被连接的槽函数会自动被回调。这就类似观察者模式：当发生了感兴趣的事件，某一个操作就会被自动触发。

槽的本质是类的成员函数，其参数可以是任意类型的。和普通C++成员函数几乎没有区别，它可以是虚函数；也可以被重载；可以是公有的、保护的、私有的、也可以被其他C++成员函数调用。唯一区别的是：槽可以与信号连接在一起，每当和槽连接的信号被发射的时候，就会调用这个槽。

进行的相关操作，槽函数没有响应会因为什么

1、类没有声明Q_OBJECT;

2、信号槽没有定义为pubic/private slots;

3、事件被子控件过滤掉了。比如QListWidget,当QListWidgetItem已经处理keypress事件后，QListWidget就不能响应itemDoubleClicked事件了。

4、信号槽的参数是自定义的，这时需要用qRegisterMetaType注册一下这种类型。具体操作可搜索 qRegisterMetaType的使用。

内存4G，malloc申请4.1G会发生什么

内存4G，已经有其他进程申请了2.5G了，此时malloc申请4G，会发生什么

如果物理内存是2G 如果mallco 4G可以么？会有什么问题？

malloc的实现与物理内存自然是无关的，分配到的内存只是虚拟内存，而且只是虚拟内存的页号，代表这块空间进程可以用，实际上还没有分配到实际的物理页面。

const char* p=“hello world” 这个“hello world”在内存的哪个位置

"hello wrold"是一个字符串字面常量，因此存储于程序的只读存储区中

static变量放在头文件会产生什么问题？

如果在头文件中定义了static变量，那么，所有包含这个头文件的源文件都会定义自己的static变量，而不是使用该头文件中的static变量。所以也就造成了，在头文件定义static变量，其他包含头文件的源文件也能使用该变量的假象。造成内存空间的浪费。

inline关键字在什么情况下会展开失败？

虚函数和递归函数就不会被正常内联. 通常, 递归函数不应该声明成内联函数.(递归调用堆栈的展开并不像循环那么简单, 比如递归层数在编译时可能是未知的, 大多数编译器都不支持内联递归函数).
inline是在编译器将函数类容替换到函数调用处，是静态编译的。而虚函数是动态调用的，在编译器并不知道需要调用的是父类还是子类的虚函数，所以不能够inline声明展开，所以编译器会忽略

debug和release的区别

一个为调试版本，其中包括了出错时能够定位源代码的在行，如果源文件已经改变，定位出来会有偏移，而且，在这个版本中编译器不会进行代码优化，并且在程序中能用宏定义_DEBUG来确定当前的版本。另一个为正式版本，程序出错只是进行简单的错误处理，编译器会优化代码，以提高性能。
Release代码更小,执行更快,编译更严格,更慢
当然就没有了调试信息

DEBUG和RELEASE 版本差异及调试相关问题：
内存分配问题

1. 变量未初始化。下面的程序在debug中运行的很好。而在release中却不行，因为debug中会自动给变量初始化found=FALSE,而在release版中则不会。所以尽可能的给变量、类或结构初始化。
2. 数据溢出的问题
   在debug版中buffer的NULL覆盖了counter的高位，但是除非counter>16M,什么问题也没有。但是在release版中，counter可能被放在寄存器中，这样NULL就覆盖了buffer下面的空间，可能就是函数的返回地址，这将导致ACCESS ERROR。
3. DEBUG版和RELEASE版的内存分配方式是不同的 。如果你在DEBUG版中申请 ele 为6*sizeof(DWORD)=24bytes,实际上分配给你的是32bytes（debug版以32bytes为单位分配）， 而在release版，分配给你的就是24bytes（release版以8bytes为单位），所以在debug版中如果你写ele[6],可能不会有什么问题，而在release版中，就有ACCESS VIOLATE。

Release好处：
它往往进行了各种优化，以期达到代码最小和速度最优。为用户的使用提供便利。

讲一下指针传递和引用传递

指针从本质上讲是一个变量，变量的值是另一个变量的地址，指针在逻辑上是独立的，它可以被改变的，包括指针变量的值（所指向的地址）和指针变量的值对应的内存中的数据（所指向地址中所存放的数据）。

引用从本质上讲是一个别名，是另一个变量的同义词，它在逻辑上不是独立的，它的存在具有依附性，所以引用必须在一开始就被初始化（先有这个变量，这个实物，这个实物才能有别名），而且其引用的对象在其整个生命周期中不能被改变，即自始至终只能依附于同一个变量（初始化的时候代表的是谁的别名，就一直是谁的别名，不能变）。

指针参数传递本质上是值传递，它所传递的是一个地址值。值传递过程中，被调函数的形式参数作为被调函数的局部变量处理，会在栈中开辟内存空间以存放由主调函数传递进来的实参值，从而形成了实参的一个副本（替身）。值传递的特点是，被调函数对形式参数的任何操作都是作为局部变量进行的，不会影响主调函数的实参变量的值（形参指针变了，实参指针不会变）。

引用参数传递过程中，被调函数的形式参数也作为局部变量在栈中开辟了内存空间，但是这时存放的是由主调函数放进来的实参变量的地址。被调函数对形参（本体）的任何操作都被处理成间接寻址，即通过栈中存放的地址访问主调函数中的实参变量（根据别名找到主调函数中的本体）。因此，被调函数对形参的任何操作都会影响主调函数中的实参变量。

引用传递和指针传递是不同的，虽然他们都是在被调函数栈空间上的一个局部变量，但是任何对于引用参数的处理都会通过一个间接寻址的方式操作到主调函数中的相关变量。而对于指针传递的参数，如果改变被调函数中的指针地址，它将应用不到主调函数的相关变量。如果想通过指针参数传递来改变主调函数中的相关变量（地址），那就得使用指向指针的指针或者指针引用。

从编译的角度来讲，程序在编译时分别将指针和引用添加到符号表上，符号表中记录的是变量名及变量所对应地址。指针变量在符号表上对应的地址值为指针变量的地址值，而引用在符号表上对应的地址值为引用对象的地址值（与实参名字不同，地址相同）。符号表生成之后就不会再改，因此指针可以改变其指向的对象（指针变量中的值可以改），而引用对象则不能修改。

总结

相同点：

都是地址的概念

不同点：

指针是一个实体（替身）；引用只是一个别名（本体的另一个名字）

引用只能在定义时被初始化一次，之后不可改变，即“从一而终”；指针可以修改，即“见异思迁”；

引用不能为空（有本体，才有别名）；指针可以为空；

sizeof 引用，得到的是所指向变量的大小；sizeof 指针，得到的是指针的大小；

指针 ++，是指指针的地址自增；引用++是指所指变量自增；

引用是类型安全的，引用过程会进行类型检查；指针不会进行安全检查；

矩阵乘法代码

```
void mul(){
    for(int i=0;i<a_m;i++){
        for(int j=0;j<b_n;j++){
            for(int k=0;k<a_n;k++){
                ans[i][j]+=a[i][k]*b[k][j];
            }
        }
    }
}
```

局部性原理

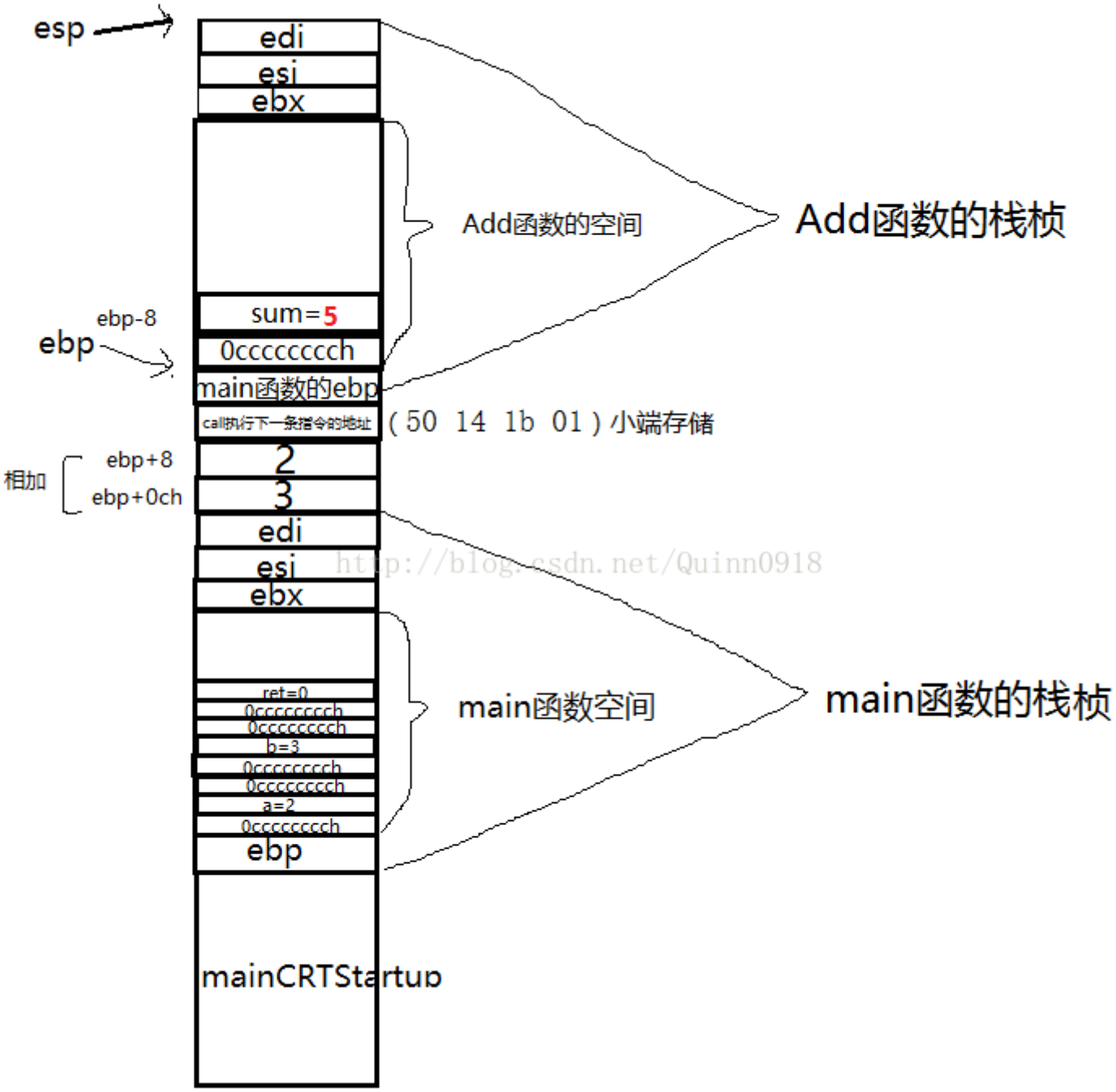
如何在一个函数通过调用地址修改另一个函数的临时变量

```
#include<stdio.h>
void swap(int *p1, int *p2)  // 交换两个整数—交换形参值（地址）
{  // 实际上是改变p1与p2所指向的地址
    int *temp;
    temp = p1;
    p1 = p2;
    p2 = temp;
    printf("交换后：x=%d   y=%d\n", *p1, *p2);
}
void main()
{
    int x, y;
    scanf("%d%d", &x, &y);
    swap(&x, &y);  // 调用swap函数的时候，是将变量x和y的地址传递进去的，也就是使形参p1和p2分别指向变
    printf("交换前：x=%d   y=%d\n", x, y);
}
```

在**main**函数中定义了**a**和**b**两个**int**变量，调用**sum**函数求其和，说一下其压栈过程

```c
#include <stdio.h>
int Add(int x, int y)
{
	int sum = 0;
	sum = x + y;
	return sum;
}
int main()
{
	int a = 2;
	int b = 3;
	int ret = 0;
	ret = Add(a, b);
	return 0;
}
```

首先main函数先保存之前的函数（在执行到main之前的初始化函数）
先保存之前的ebp(帧指针)，也就是将ebp压栈
然后ebp指向esp(栈指针)
为main开辟一段空间(栈空间由高地址往低地址)，esp指向栈顶
依次把ebx、esi、edi压进去，esp指向栈顶
将刚刚开辟的空间初始化为0xcccccccc
将局部变量压入栈中(放在之前初始化的main函数空间)
调用Add函数之前，将函数的参数从右往左压入栈中
然后将call执行下一条的地址压栈(返回值)
进入Add函数，将main函数的ebp压栈
然后ebp指向esp(栈指针)
为Add开辟空间
依次把ebx、esi、edi压进去，esp指向栈顶
将刚刚开辟的空间初始化为0xcccccccc
将局部变量压入栈中(放在之前初始化的Add函数空间)
ebp+8，此时ebp+8指向形参a；
ebp+0ch，指向形参b，把a+b放到eax里
ebp-8就是sum所在的位置，把eax（a+b）的值给sum，sum=5；
把sum的值再放到eax里，那么eax里存放的就是我们的返回值



pop就是出栈的意思，esp此时指向ebx下面的空间，这三个地址相当于被回收了
把ebp的值给esp
ebp就是我们所存储的main函数的ebp，那么此时ebp指向main函数里面的ebp
ret指令要返回值，首先把栈顶call执行下一条指令的地址出栈，然后紧接着跳到下面这一行的地址
esp+8直接把定义的形参跳过去，到这一步的时候，我们就是Add的栈桢已经！！！被销毁了！！！
eax里存放的是Add函数里sum的值，把eax的值给ebp-20h（ret）就把sum的值返回了

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