一、基础知识

说一下C++和C的区别

- 设计思想上：
  C++是面向对象的语言，而C是面向过程的结构化编程语言
- 语法上：
  C++引入类的概念，具有封装、继承和多态三种特性
  C++支持范式编程，比如模板类、函数模板等
  C++增加多许多类型安全的功能，比如强制类型转换

> 相对于C，C++多了重载、内联函数、异常处理等，扩展了面向对象的设计内容：类、继承、虚函数、模板。
> new和delete是对内存分配的运算符，取代了C中的malloc和free；
> 有引用的概念；
> C++随时定义变量随时使用

C++和Java之间的区别？

Java与C++都是面向对象的语言，都使用了面向对象的思想（封装、继承、多态），由于面向对象由许多非常好的特性（继承、组合等），因此二者有很好的可重用性。
主要不同点：

1. Java为解释性语言，其运行过程为：程序源代码经过Java编译器编译成字节码，然后由JVM解释执行。
   而C/C++为编译型语言，源代码经过编译和链接后生成可执行的二进制代码，可直接执行。
   因此Java的执行速度比C/C++慢，但Java能够跨平台执行，C/C++不能。
2. Java是纯面向对象语言，所有代码（包括函数、变量）必须在类中实现，除基本数据类型（包括int、float等）外，所有类型都是类。此外，Java语言中不存在全局变量或者全局函数，而C++兼具面向过程和面向对象编程的特点，可以定义全局变量和全局函数。
3. 与C/C++语言相比，Java语言中没有指针的概念，这有效防止了C/C++语言中操作指针可能引起的系统问题，从而使程序变得更加安全。
4. 与C++语言相比，Java语言不支持多重继承，但是Java语言引入了接口的概念，可以同时实现多个接口。由于接口也有多态特性，因此Java语言中可以通过实现多个接口来实现与C++语言中多重继承类似的目的。
5. 在C++语言中，需要开发人员去管理内存的分配（包括申请和释放），而Java语言提供了垃圾回收器来实现垃圾的自动回收，不需要程序显示地管理内存的分配。

在C++语言中，通常会把释放资源的代码放到析构函数中，Java语言中虽然没有析构函数，但却引入了一个finalize()方法，当垃圾回收器要释放无用对象的内存时，会首先调用该对象的finalize()方法，因此，开发人员不需要关心也不需要知道对象所占的内存空间何时被释放。

C语言函数调用

- C语言怎么进行函数调用？
  每一个函数调用都会分配函数栈，在栈内进行函数执行过程。调用前，先把返回地址压栈，然后把当前函数的esp（栈顶）指针压栈。
- C语言参数压栈顺序
  从右到左
- C++如何处理返回值
  生成一个临时变量，把它的引用作为函数参数传入函数内。

include头文件的顺序以及双引号和尖括号的区别？

- Include头文件的顺序
  对于include的头文件来说，如果在文件a.h中声明一个在文件b.h中定义的变量，而不引用b.h。
  那么要在a.cpp文件中引用b.h文件（并且要先引用b.h），后引用a.h,否则汇报变量类型未声明错误。（因为变量是在b.h定义和初始化的，肯定得先加载）
- 双引号和尖括号的区别：编译器预处理阶段查找头文件的路径不一样。
  - 对于使用双引号包含的头文件，查找头文件路径的顺序为：
    1、当前头文件目录
    2、编译器设置的头文件路径（编译器可使用-I显式指定搜索路径）
    3、系统变量CPLUS_INCLUDE_PATH/C_INCLUDE_PATH指定的头文件路径
  - 对于使用尖括号包含的头文件，查找头文件的路径顺序为：
    1、编译器设置的头文件路径（编译器可使用-I显式指定搜索路径）
    2、系统变量CPLUS_INCLUDE_PATH/C_INCLUDE_PATH指定的头文件路径

什么是预编译 何时需要预编译

- 预编译
  又称为预处理, 是做些代码文本的替换工作;
  处理#开头的指令, 比如拷贝 #include 包含的文件代码, #define 宏定义的替换, 条件编译等, 就是为编译做的预备工作的阶段, 主要处理#开始的预编译指令, 预编译指令指示了在程序正式编译前就由编译器进行的操作, 可以放在程序中的任何位置;
- 何时需要预编译
  1、总是使用不经常改动的大型代码体;
  2、程序由多个模块组成, 所有模块都使用一组标准的包含文件和相同的编译选项; 在这种情况下, 可以将所有包含文件预编译为一个预编译头;
  提供的预处理功能主要有以下三种: 宏定义, 文件包含, 条件编译;

C++源文件从文本到可执行文件经历的过程

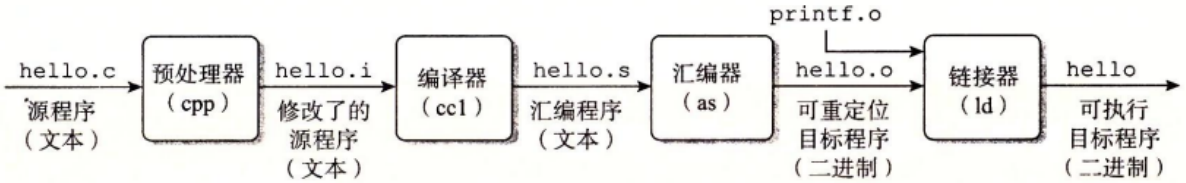


图 1-3　编译系统

- 对于C++源文件，从文本到可执行文件一般需要四个过程：
  预处理阶段：预处理器对源代码文件中文件包含关系（头文件）、预编译语句（宏定义）进行分析和替换，生成预编译文件。
  编译阶段：编译器将经过预处理后的预编译文件转换成特定汇编代码，生成汇编文件
  汇编阶段：汇编器将编译阶段生成的汇编文件转化成机器码，生成可重定位目标文件

链接阶段：链接器将多个目标文件及所需要的库连接成最终的可执行目标文件

字节对齐的原则

对齐的作用和原因：各个硬件平台对存储空间的处理上有很大的不同。一些平台对某些特定类型的数据只能从某些特定地址开始存取。

> 比如有些平台每次读都是从偶地址开始，如果一个int型（假设为32位系统）如果存放在偶地址开始的地方，那么一个读周期就可以读出这32bit，而如果存放在奇地址开始的地方，就需要2个读周期，并对两次读出的结果的高低字节进行拼凑才能得到该32bit数据。显然在读取效率上下降很多。

1. 从0位置开始存储；
2. 变量存储的起始位置是该**变量大小**的整数倍；
3. 结构体总的大小是其**最大元素**的**整数倍**，不足的后面要补齐；
4. 结构体中包含结构体，从结构体中最大元素的整数倍开始存；
5. 如果加入pragma pack(n) ，取n和变量自身大小较小的一个。

```
#pragma pack(push) // 保存对齐状态
#pragma pack(n)  //设定结构体、联合以及类成员变量以 n 字节方式对齐
#pragma pack(pop) // 恢复对齐状态
```

```
struct MyStruct{
  double dda1;
  char dda;
  int type
};
sizeof(MyStruct)=8+1+ 3+4=16，其中有3个字节是VC自动填充的，没有放任何有意义的东西。
```

```
struct MyStruct{
  char dda;
  double dda1;
  int type
};
sizeof(MyStruc)为1+7+8+4+4=24。其中总的有7+4=11个字节是VC自动填充的，没有放任何有意义的东西。
```

用struct关键字与class关键定义类以及继承的区别

- 定义类差别
  struct关键字也可以实现类，用class和struct关键字定义类的唯一差别在于默认访问级别：默认情况下，struct成员的访问级别为public，而class成员的为private。语法使用也相同，直接将class改为struct即可。
- 继承差别
  使用class保留字的派生类默认具有private继承，而用struct保留字定义的类某人具有public继承。其它则没有任何区别。

> 主要点就两个：默认的访问级别和默认的继承级别 class都是private

引用和指针的区别

- 引用
  - 左值引用
    常规引用，一般表示对象的身份。
  - 右值引用
    右值引用就是必须绑定到右值（一个临时对象、将要销毁的对象）的引用，一般表示对象的值。
    右值引用可实现转移语义（Move Sementics）和精确传递（Perfect Forwarding），它的主要目的有两个方面：

- 消除两个对象交互时不必要的对象拷贝，节省运算存储资源，提高效率。
- 能够更简洁明确地定义泛型函数。

引用	指针	
只是别名，不占用具体存储空间，只有声明没有定义	具体变量，需要占用存储空间	存储空间
声明时必须初始化为另一变量	声明和定义可以分开，可以先只声明指针变量而不初始化，等用到时再指向具体变量	初始化
不存在指向空值的引用，必须有具体实体	存在指向空值的指针	值
引用一旦初始化之后就不可以再改变	指针变量可以重新指向别的变量	初始化后

1. 引用是直接访问，指针是间接访问。
2. 引用的创建不会调用类的拷贝构造函数
3. 指针可以有多级指针（**p），而引用只有一级

> 总的来说：引用既具有指针的效率，又具有变量使用的方便性和直观性

- 常引用有什么作用
  常引用的引入主要是为了避免使用变量的引用时, 在不知情的情况下改变变量的值; 常引用主要用于定义一个普通变量的只读属性的别名, 作为函数的传入形参, 避免实参在调用函数中被意外的改变;

> 说明: 很多情况下, 需要用常引用做形参, 被引用对象等效于常对象, 不能在函数中改变实参的值, 这样的好处是有较高的易读性和较小的出错率;

C++四种类型转换

const_cast（小转大）, static_cast（大转小）, dynamic_cast, reinterpret_cast

- const_cast用于将const变量转为非const
  一般它的作用就是把数据类型中的const属性去掉，使之将常指针类型转换成普通指针，将常引用转换为普通引用。

```
void func( const int *p){ int * q = const_cast<int *>p; (*q)++; }
正常情况下，const int *p 要求p指向的地址存储的值不能发生变化，表示常指针，但是此处通过const_cast把const性质去掉，导致对应地址的值变成了普通指针指向的值，从而可以进行修改。
```

- static_cast用的最多，对于各种隐式转换，非const转const，void*转指针等, static_cast用于多态向上转化。向下转能成功的时候，但不安全，转后的结果未知；
  主要用在基本类型转换和类类型转换之间。

> 比如float数赋值给int数时为隐式转换，可以直接进行。而int数转为float数时（低精度转高精度），需要强转，则可以用static_cast。
> 派生类指针可以通过static_cast转成基类指针，而基类指针不可以通过static_cast转成派生类指针。

- dynamic_cast用于动态类型转换。
  只能用于含有虚函数的类，用于类层次间的向上和向下转化。只能转指针或引用。向下转化时，如果是非法的对于指针返回NULL，对于引用抛异常。要深入了解内部转换的原理。
  - 向上转换：指的是子类向基类的转换（安全）
  - 向下转换：指的是基类向子类的转换
    它通过判断在执行到该语句的时候变量的运行时类型和要转换的类型是否相同来判断是否能够进行向下转换。
- reinterpret_cas
  它是用来在任意类型之间进行转换，具有很大的危险性和不确定性，它可以在任意指针之间进行互相转换。比如将int转指针，尽量少用。
- 为什么不使用C的强制转换？
  C的强制转换表面上看起来功能强大什么都能转，但是转化不够明确，不能进行错误检查，容易出错。

有哪些内存泄漏？如何判断内存泄漏？如何定位内存泄漏？

内存泄漏类型：

1. 堆内存泄漏 （ Heap leak ）。对内存指的是程序运行中根据需要分配通过malloc,realloc new等从堆中分配的一块内存，再是完成后必须通过调用对应的 free或者delete 删掉。如果程序的设计的错误导致这部分内存没有被释放，那么此后这块内存将不会被使用，就会产生Heap Leak.
2. 系统资源泄露（ Resource Leak ）.主要指程序使用系统分配的资源比如 Bitmap,handle ,SOCKET等没有使用相应的函数释放掉，导致系统资源的浪费，严重可导致系统效能降低，系统运行不稳定。

检测内存泄漏：（ 选用 ）

1. 在windows平台下通过CRT中的库函数进行检测；
2. 在可能泄漏的调用前后生成块的快照，比较前后的状态，定位泄漏的位置
3. Linux下通过工具valgrind检测

sizeof

sizeof 对数组，得到整个数组所占空间大小。
sizeof 对指针，得到指针本身所占空间大小。

```
char str[]="aaa" ;char*p=str;int n =10; sizeof(str)=?,sizeof(p)=?,sizeof(n)=?
```

sizeof(str) = 4，因为加上一个'\0'
sizeof (p) 的值只取决于多少位的操作系统，若64位则8字节，32位则4字节。
sizeof(n) = 4

- **空结构体的sizeof()返回值**
  答案是1

volatile关键字

访问寄存器要比访问内存要块，因此CPU会优先访问该数据在寄存器中的存储结果，但是内存中的数据可能已经发生了改变，而寄存器中还保留着原来的结果，导致读脏数据。为了避免这种情况的发生将该变量声明为volatile，告诉CPU每次都从内存去读取数据，编译器不再对访问该变量的代码优化，仍然从内存读取，使访问稳定。

> 总结：volatile关键字影响编译器编译的结果，用volatile声明的变量表示该变量随时可能发生变化（操作系统、硬件、其它线程等更改），与该变量有关的运算不再编译优化，以免出错。

- 使用实例如下：
  多线程应用中被几个任务共享的变量（线程1刚处理过共享变量a，寄存器里也有a的值，此时线程2改变了内存a的值，线程1再使用a的值会直接从寄存器拿，导致取的值不对，所以需要使用volatile），就比如有个是否收到app连接的标志位，已连接上就给APP回数据，如果收发是两个线程的话，这个标志位就有必要设置为volatile。
  1、中断服务程序中修改的供其它程序检测的变量需要加volatile；
  2、多任务环境下各任务间共享的标志应该加volatile；
- 一个参数可以即是const又是volatile的吗？
  可以，一个例子是只读状态寄存器，是volatile是因为它可能被意想不到的被改变，是const告诉程序不应该试图去修改他
- 一个指针可以是volatile 吗？解释为什么。**
  可以。尽管这并不很常见。一个例子当中断服务子程序修该一个指向一个buffer的指针时。

变量的声明和定义有什么区别

- 定义
  为变量分配地址和存储空间的称为定义, 只在一个地方定义；
- 声明
  不分配地址的称为声明; 一个变量可以在多个地方声明；

加入extern修饰的是变量的声明, 说明此变量将在文件以外或在文件后面部分定义;

> 说明: 很多时候一个变量, 只是声明不分配内存空间, 直到具体使用时才初始化, 分配内存空间,如外部变量;

static关键字

static关键字的作用?

> 修饰普通变量、普通函数、成员变量、成员函数。

1. 修饰普通变量
   修改变量的存储区域和生命周期，使变量存储在静态存储区（程序运行期间一直存在），在 main 函数运行前就分配了空间，如果有初始值就用初始值初始化它，如果没有初始值系统用默认值初始化它。
   - 修饰全局变量
     全局静态变量在声明他的文件之外是不可见的，准确地说是从定义之处开始，到文件结尾，是一个**本地的全局变量**。
     > 对于函数定义和代码块之外的变量声明，static修改标识符的链接属性，由默认的external变为internal，作用域和存储类型不改变，这些符号只能在声明它们的源文件中访问，而不能通过extern来访问。
   - 修饰局部变量
     作用域仍为局部作用域，当定义它的函数或者语句块结束的时候，作用域结束。但是当局部静态变量离开作用域后，并没有销毁，而是仍然驻留在内存当中，只不过我们不能再对它进行访问，直到该函数再次被调用，并且值不变；
2. 修饰普通函数
   在函数返回类型前加static，就定义为静态函数，仅在定义该函数的文件内才能使用（函数的定义和声明在默认情况下都是extern的）。static函数在内存中只有一份，普通函数在每个被调用中维持一份拷贝。
   > 在多人开发项目时，为了防止与他人命令函数重名，可以将函数加上 static。
   > 全局函数放在头文件声明，只限定在cpp文件内部使用的函数声明加上static修饰。
3. 修饰成员变量
   修饰成员变量使所有的对象只保存一个该变量，而且不需要生成对象就可以访问该成员。
   > 在类中，静态成员可以实现多个对象之间的数据共享，并且使用静态数据成员还不会破坏隐藏的原则，即保证了安全性。
   > 静态成员是类的所有对象中共享的成员，而不是某个对象的成员，所以不能在类内进行定义（声明还是在类内的）。
   > 不占类对象的存储空间（存储在静态变量区）
4. 修饰成员函数
   修饰成员函数使得不需要生成对象就可以访问该函数，但是在 static 函数内不能访问非静态成员。
   > 静态成员函数和静态成员变量都属于类的静态成员，都不是对象成员。（不能直接引用类中说明的非静态成员，不能使用this指针）
   > 静态成员函数对静态成员的引用不需要用对象名。（可以直接引用类中的静态成员）
   > 静态成员函数中要访问非静态成员时，在静态成员函数参数列表中传递类的地址或引用来实现对类对象的非静态成员变量进行访问与操作
   > 调用静态成员函数如下格式：<类名>::<静态成员函数名>(<参数表>);

extern关键字作用

被 extern 限定的函数或变量是 extern 类型的

- 声明一个外部变量

```
//file1.cpp定义并初始化和一个常量，该常量能被其他文件访问
extern const int bufferSize = 5;
//file1.h头文件
extern const int bufferSize; //与file1.cpp中定义的是同一个
```

- extern c 作用

告诉编译器被 extern "C" 修饰的变量和函数是按照 C 语言方式编译和连接的
extern "C" 的作用是让 C++ 编译器将 extern "C" 声明的代码当作 C 语言代码处理，可以避免 C++ 因符号修饰导致代码不能和C语言库中的符号进行链接的问题。

```
#ifdef __cplusplus
extern "C" {
#endif

void *memset(void *, int, size_t);

#ifdef __cplusplus
}
#endif
```

const关键字

作用

- 修饰变量，说明该变量不可以被改变。
  > 修饰局部常量时，存储在栈区，可通过const_cast修改常量在内存的值，但是由于在编译后被替换成立即数，打印的结果不是内存的值。
  > 修饰全局变量时，存储在常量区（只读），通过const_cast修改会报错，即不允许修改。
- 修饰指针，分为指向常量的指针（const int * a）和指针常量（int * const a）
  > 很好理解，正常定义为int * a，const 加在a前即表示指针不可修改
- 常量引用，经常用于形参类型，引用避免了拷贝，常量引用避免了函数对值的修改
- 修饰成员函数，说明该常成员函数内不能修改成员变量
  > 使用关键字mutable可允许修改
  > 指针所指的成员对象值可能会被改变
  > const修饰类的成员变量时不能在声明中初始化，必须在构造函数列表中初始化

const如何做到只读

- C++把const看做常量，编译器会使用常数直接替换掉对i的引用，例如cout<<i; 会理解成cout<<10，不会去访问i的内存地址去取数据
- C++中只是对内置数据类型做常数替换（对结构体和类是不能做常数替换的，需要从内存里取数据）

inline 内联函数

特征

- 相当于宏，却比宏多了类型检查，真正具有函数特性；
- 不能包含循环、递归、switch 等复杂操作；
- 在类声明中定义的函数，除了虚函数的其他函数都会自动隐式地当成内联函数。

使用

```
// 声明（加 inline，建议使用）
inline int functionName(int first, int secend,...);
// 定义
inline int functionName(int first, int secend,...) {/****/};

// 类内定义，隐式内联
class A {
    int doA() { return 0; } // 隐式内联
}
```

```
10
11 // 类外定义，需要显式内联
12 class A {
13     int doA();
14 }
15 inline int A::doA() { return 0; } // 需要显式内联
```

编译器对 inline 函数的处理步骤

1. 将 inline 函数体复制到 inline 函数调用点处；
2. 为所用 inline 函数中的局部变量分配内存空间；
3. 将 inline 函数的的输入参数和返回值映射到调用方法的局部变量空间中；
4. 如果 inline 函数有多个返回点，将其转变为 inline 函数代码块末尾的分支（使用 GOTO）。

优缺点

优点

- 内联函数同宏函数一样将在被调用处进行代码展开，省去了参数压栈、栈帧开辟与回收，结果返回等，从而提高程序运行速度。
- 内联函数相比宏函数来说，在代码展开时，会做安全检查或自动类型转换（同普通函数），而宏定义则不会。
- 在类中声明同时定义的成员函数，自动转化为内联函数，因此内联函数可以访问类的成员变量，宏定义则不能。
- 内联函数在运行时可调试，而宏定义不可以。

缺点

- 代码膨胀。内联是以代码膨胀（复制）为代价，消除函数调用带来的开销（内联函数不能太大）。
- inline 函数无法随着函数库升级而升级。inline函数的改变需要重新编译，不像 non-inline 可以直接链接。
- 是否内联，程序员不可控。内联函数只是对编译器的建议，是否对函数内联，决定权在于编译器。

define/const/inline/typedef/枚举

本质：define只是字符串替换，const参与编译运行，具体的：

1. define不会做类型检查，const拥有类型，会执行相应的类型检查
2. define仅仅是宏替换，不占用内存，而const会占用内存
3. const内存效率更高，编译器通常将const变量保存在符号表中，而不会分配存储空间，这使得它成为一个编译期间的常量，没有存储和读取的操作

本质：define只是字符串替换，inline由编译器控制，具体的：

1. 内联函数在编译时展开，而宏是由预处理器对宏进行展开
2. 内联函数本身是函数，强调函数特性，具有重载等功能（会检查参数类型，宏定义不检查函数参数 ，所以内联函数更安全。）
3. 内联函数可以作为某个类的成员函数，这样可以使用类的保护成员和私有成员。而当一个表达式涉及到类保护成员或私有成员时，宏就不能实现了。
4. 宏在定义时要小心处理宏参数，（一般情况是把参数用括弧括起来）。

typedef和define有什么区别

- 用法不同:
  typedef用来定义一种数据类型的别名, 增强程序的可读性;
  define主要用来定义常量, 以及书写复杂使用频繁的宏;
- 执行时间不同:
  typedef是编译过程的一部分, 有类型检查的功能;
  define是宏定义, 是预编译的部分, 其发生在编译之前, 只是简单的进行字符串的替换, 不进行类型的检查;

- 作用域不同:
  typedef有作用域限定;
  define不受作用域约束, 只要是在define声明后的引用都是正确的;
- 对指针的操作不同:
  typedef和define定义的指针时有很大的区别;

> 注意: typedef定义是语句,因为句尾要加上分号; 而define不是语句, 千万不能在句尾加分号;

枚举与define 宏的区别

1. define 宏常量是在预编译阶段进行简单替换。枚举常量则是在编译的时候确定其值。
2. 可以调试枚举常量，但是不能调试宏常量。
3. 枚举可以一次定义大量相关的常量，而#define 宏一次只能定义一个。

C++中可以重载的运算符

引用大佬的总结，带 . 的都不能被重载
有new/delete、new[]/delete[]、++等。

- 不可以重载的运算符：、.、::、?:、sizeof、typeid、.、**、不能改变运算符的优先级。
- 引申：重载++和–时是怎么区分前缀++和后缀++的？
  例如当编译器看到++a（先自增）时，它就调用operator++(a);
  但当编译器看到a++时，它就调用operator++(a, int)。即编译器通过调用不同的函数区别这两种形式。（解释为什么优先用++a）

new/delete与malloc/free的区别

1. new是运算符，malloc是C语言库函数
2. new可以重载，malloc不能重载
3. new的变量是数据类型，malloc的是字节大小
4. new可以调用构造函数，delete可以调用析构函数，malloc/free不能
5. new返回的是指定对象的指针，而malloc返回的是void*，因此malloc的返回值一般都需要进行类型转化
6. malloc分配的内存不够的时候可以使用realloc扩容，new没有这样的操作
7. new内存分配失败抛出bad_malloc，malloc内存分配失败返回NULL值

> new / new[]：完成两件事，先底层调用 malloc 分了配内存，然后调用构造函数（创建对象）。
> delete/delete[]：也完成两件事，先调用析构函数（清理资源），然后底层调用 free 释放空间。
> new 在申请内存时会自动计算所需字节数，而 malloc 则需我们自己输入申请内存空间的字节数。

深拷贝与浅拷贝

```
 浅拷贝：
char ori[]=“hello”；char *copy=ori；
深拷贝：
char ori[]="hello"; char *copy=new char[]; copy=ori;
```

浅拷贝只是对指针的拷贝，拷贝后两个指针指向同一个内存空间，深拷贝不但对指针进行拷贝，而且对指针指向的内容进行拷贝，经深拷贝后的指针是指向两个不同地址的指针。

- 浅拷贝可能出现的问题：

1. 浅拷贝只是拷贝了指针，使得两个指针指向同一个地址，这样在对象块结束，调用函数析构的时，会造成同一份资源析构2次，即delete同一块内存2次，造成程序崩溃。
2. 浅拷贝使得两个指针都指向同一块内存，任何一方的变动都会影响到另一方。
3. 同一个空间，第二次释放失败，导致无法操作该空间，造成内存泄漏。

数组和指针的区别

数组：数组是用于储存多个相同类型数据的集合，连续的存储空间。
指针：指针相当于一个变量，但是它和不同变量不一样，它存放的是其它变量在内存中的地址。

指针	数组
保存数据的地址	保存数据
间接访问数据，首先获得指针的内容，然后将其作为地址，从该地址中提取数据	直接访问数据
通常用于动态的数据结构	通常用于固定数目且数据类型相同的元素
sizeof(p) = 4（跟操作系统有关）	sizeof(p) = 数组存储长度

- 数组指针（相当于行指针）

```
定义 int (*p)[n];
首先p是一个指针，指向一个整型的一维数组，这个一维数组的长度是n，也可以说是p的步长。执行p+1时，p要跨过n个整型数据的长度。它是“指向数组的指针”的简称，在32 位系统下占4 个字节，至于它指向的数组占多少字节要看数组大小。
```

- 指针数组

```
定义 int *p[n];
int a = 2; p[0] = &a;
首先它是一个数组，数组的元素都是指。由int*说明这是一个整型指针数组，它有n个指向int的指针。
```

- 易误解：如果int a[5], 那么a与&a是等价的，因为两者地址相同。
  解答：一定要注意a与&a是不一样的，虽然两者地址相同，但意义不一样，&a是整个数组对象的首地址，而a是数组首地址，也就是a[0]的地址，a的类型是int[5]，a[0]的类型是int，因此&a+1相当于a的地址值加上sizeof(int) * 5，也就是a[5]，下一个对象的地址，已经越界了，而a+1相当于a的地址加上sizeof(int)，即a[1]的地址。

> 1、一个一维int数组的数组名实际上是一个int* const 类型；
> 2、一个二维int数组的数组名实际上是一个int (*const p)[n]（数组指针）
> 3、数组名做参数会退化为指针，除了sizeof

指针函数与函数指针的区别？

- 指针函数：指带指针的函数，即本质是一个函数，函数返回类型是某一类型的指针。
  > 类型标识符 函数名(参数表)
  > *int* f(x，y);
- 函数指针：指向函数的指针变量，即本质是一个指针变量。

```
    void func(int a){...}
    int (*f) (int x); /*声明一个函数指针 */
   f=func; /* 将func函数的首地址赋给指针f */
    f(2); //相当于调用func(2)

定义一个函数指针，指向的函数有两个int形参并且返回一个函数指针，返回的指针指向一个有一个int形参且返回int的函数？
int (*(*F)(int, int))(int)
//嵌套的函数指针
//一个函数指针，指向的函数有两个int形参，这个就是(*F)(int, int)，这返回的是一个指针
//返回一个函数指针，返回的指针指向一个有一个int形参且返回int的函数；把上面的结果当成一个指针，相当于再做一次上面的步骤，所以结果为：int (*(*F)(int, int))(int)
```

主要的区别是一个是指针变量，一个是函数。在使用是必要要搞清楚才能正确使用

回调函数

回调函数就是一个通过函数指针调用的函数。如果你把函数的指针(地址)作为参数传递给另一个函数，当这个指针被用为调用它所指向的函数时，我们就说这是回调函数。回调函数不是由该函数的实现方直接调用，而是在特定的事件或条件发生时由另外的一方调用的，用于对该事件或条件进行响应。
回调函数实现的机制是：

（1）定义一个回调函数；
（2）提供函数实现的一方在初始化的时候，将回调函数的函数指针注册给调用者；
（3）当特定的事件或条件发生的时候，调用者使用函数指针调用回调函数对事件进行处理。

例子：快速排序调用的比较函数，即把函数指针传入快排程序，每次比较两个数大小的时候调用该函数。

野指针是什么

野指针不是空指针，是一个指向垃圾内存的指针。
其形成的原因有：
（1）指针变量在创建时没有被初始化，其缺省值为随机；
（2）指针被 free 或 delete 掉后没有设置为 NULL，只是把指针所指向的内存释放掉，并没有把指针本身清理掉；
（3）指针操作超越了变量的作用范围，比如指针执行 ++ 操作越界。

智能指针

智能指针定义

是一个存储指向动态分配（堆）对象指针的类，构造函数传入普通指针，析构函数释放指针。栈上分配，函数或程序结束自动释放，防止内存泄露。
使用引用计数器，类与指向的对象相关联，引用计数跟踪该类有多少个对象共享同一指针。

- 创建类的新对象时，初始化指针并将引用计数置为1；
- 当对象作为另一对象的副本而创建，增加引用计数；
- 对一个对象进行赋值时，减少引用计数，并增加右操作数所指对象的引用计数；
- 调用析构函数时，构造函数减少引用计数，当引用计数减至0，则删除基础对象。

> 智能指针主要用于管理在堆上分配的内存，它将普通的指针封装为一个栈对象。当栈对象的生存周期结束后，会在析构函数中释放掉申请的内存，从而防止内存泄漏。C++ 11中最常用的智能指针类型为shared_ptr,它采用引用计数的方法，记录当前内存资源被多少个智能指针引用。该引用计数的内存在堆上分配。当新增一个时引用计数加1，当过期时引用计数减一。只有引用计数为0时，智能指针才会自动释放引用的内存资源。对shared_ptr进行初始化时不能将一个普通指针直接赋值给智能指针，因为一个是指针，一个是类。可以通过make_shared函数或者通过构造函数传入普通指针。并可以通过get函数获得普通指针。

使用智能指针的原因

智能指针的作用是管理一个指针，因为存在以下这种情况：申请的空间在函数结束时忘记释放，造成内存泄漏。使用智能指针可以很大程度上的避免这个问题，因为智能指针就是一个类，当超出了类的作用域是，类会自动调用析构函数，析构函数会自动释放资源。所以智能指针的作用原理就是在函数结束时自动释放内存空间，不需要手动释放内存空间。

常用的智能指针

C++里面的四个智能指针: auto_ptr, shared_ptr, weak_ptr, unique_ptr 其中后三个是c++11支持，并且第一个已经被11弃用。

- auto_ptr（c++98的方案，cpp11已经抛弃）
  采用所有权模式。对于特定的对象，只能有一个智能指针可拥有，这样只有拥有对象的智能指针的析构函数会删除该对象。然后让赋值操作转让所有权。

```
#include <memory>
auto_ptr< string> p1 (new string ("I reigned lonely as a cloud.”));
auto_ptr<string> p2;
```

```
p2 = p1; //auto_ptr不会报错，p1变为NULL
//cout << "str:" << *p1 << endl; 运行该语句会出现内存奔溃问题
```

编译不会报错，但程序运行时p2剥夺了p1的所有权，但是当访问p1指向的对象时将会出错，因为采用所有权模式的p1为NULL。所以auto_ptr的缺点是：**存在潜在的内存崩溃问题！**

> 缺点主要是所有权模式导致一个对象只能一个智能指针拥有，对失去控制权的指针使用时会造成内存泄漏。
> 所以肯定不支持复制（拷贝构造函数）和赋值（operator = ），且编译不会提示出错。

- unique_ptr（替换auto_ptr）
  unique_ptr实现独占式拥有或严格拥有概念，也采用所有权模式，保证同一时间内只有一个智能指针可以指向该对象。它对于避免资源泄露(例如“以new创建对象后因为发生异常而忘记调用delete”)特别有用。

```
unique_ptr<string> pu1(new string ("hello world"));
unique_ptr<string> pu2;
pu2 = pu1; // #1 not allowed 此时会报错

unique_ptr<string> pu3;
pu3 = unique_ptr<string>(new string ("You")); // #2 allowed 临时右值时允许赋值（比较智能的）
```

> 也不支持复制和赋值，但比auto_ptr好，直接赋值会编译出错。

- shared_ptr
  shared_ptr实现共享式拥有概念。可实现多个智能指针可以指向相同对象，该对象和其相关资源会在“最后一个引用被销毁”时候释放。从名字share就可以看出了资源可以被多个指针共享，它使用计数机制来表明资源被几个指针共享。可以通过成员函数use_count()来查看资源的所有者个数。除了可以通过new来构造，还可以通过传入auto_ptr, unique_ptr,weak_ptr来构造。当我们调用release()时，当前指针会释放资源所有权，计数减一。当计数等于0时，资源会被释放。
  - 成员函数：
    use_count 返回引用计数的个数
    unique 返回是否是独占所有权(use_count 为 1)
    swap 交换两个 shared_ptr 对象(即交换所拥有的对象)
    reset 放弃内部对象的所有权或拥有对象的变更, 会引起原有对象的引用计数的减少
    get 返回内部对象(指针), 由于已经重载了()方法, 因此和直接使用对象是一样的.如 shared_ptr sp(new int(1)); sp 与 sp.get()是等价的
  - 缺点
    引用计数是一种便利的内存管理机制，但它有一个很大的缺点，那就是不能管理循环引用的对象。

> 核心要理解引用计数，什么时候销毁底层指针，还有赋值，拷贝构造时候的引用计数的变化，析构的时候要判断底层指针的引用计数为0了才能真正释放底层指针的内存

- Weak_ptr
  weak_ptr 是一种不控制对象生命周期的智能指针, 它指向一个 shared_ptr 管理的对象. 进行该对象的内存管理的是那个强引用的 shared_ptr. weak_ptr只是提供了对管理对象的一个访问手段。weak_ptr 设计的目的是为配合 shared_ptr 而引入的一种智能指针来协助 shared_ptr 工作, 它只可以从一个 shared_ptr 或另一个 weak_ptr 对象构造, 它的构造和析构不会引起引用记数的增加或减少。
  weak_ptr是用来解决shared_ptr相互引用时的死锁问题,如果说两个shared_ptr相互引用,那么这两个指针的引用计数永远不可能下降为0,资源永远不会释放。它是对对象的一种弱引用，不会增加对象的引用计数，和shared_ptr之间可以相互转化，shared_ptr可以直接赋值给它，它可以通过调用lock函数来获得shared_ptr。

```
class B;
class A{
public:
 shared_ptr<B> pb_;
 ~A(){cout << "A delete\n";}
};
```

```
class B{
public:
 shared_ptr<A> pa_;
 ~B(){ cout << "B delete\n";}
};
void fun(){
 shared_ptr<B> pb(new B());
 shared_ptr<A> pa(new A());
 pb->pa_ = pa;
 pa->pb_ = pb;
 cout << pb.use_count() << endl;
 cout << pa.use_count() << endl;
}
int main()
{
 fun();
 return 0;
}
```

> 开始不太看的懂内部实现机理，做了一个实验，把shared_ptr智能pb和pa分别指向一个int数据，进行pb=pa赋值后，那么智能指针之前指向的数据会被释放，由于无智能指针指向，计数器为0。而上面的代码由于赋值后本该释放的对象的智能指针却指向另一个对象，所以没被释放，感觉是这样的。

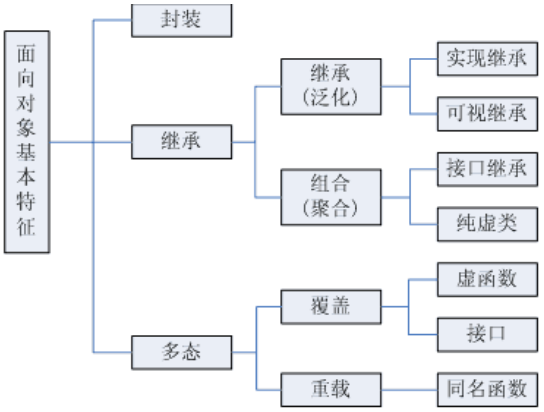
可以看到fun函数中pa ，pb之间互相引用，两个资源的引用计数为2，当要跳出函数时，智能指针pa，pb析构时两个资源引用计数会减一，但是两者引用计数还是为1，导致跳出函数时资源没有被释放（A B的析构函数没有被调用）。
如果把其中一个改为weak_ptr就可以了，我们把类A里面的**shared_ptr pb_;** 改为 **weak_ptr pb_;**（只需要在类中修改）运行结果如下，这样的话，资源B的引用开始就只有1，当pb析构时，B的计数变为0，B得到释放，B释放的同时也会使A的计数减一，同时pa析构时使A的计数减一，那么A的计数为0，A得到释放。

> 为了更好的避免内存泄漏。智能指针是一个类，当超出了类的作用域，类将会自动调用析构函数，析构函数会自动释放资源。
> 1、auto_ptr已经被遗弃，原因是一个对象只能有一个智能指针，易造成内存访问出错，且编译的时候不会报错。接着c++11提出的unique_ptr虽然也是一个对象只能有一个指针，但是编译的时候能对智能指针赋值造成的所有权丢失进行报错。且不能完成对象复制和赋值操作
> 2、以上两个智能指针的对象只能拥有一个智能指针。所以出现了shard_ptr，采用计数法，运行多个指针指向对象，但是在使用引用时对象没增加，计数次数增加了，会出现问题。因此weak_ptr解决了使引用计数次数不增加。这两个智能指针都可用于类对象的复制和赋值曹组。
> 3、综上分析，一个对象一个智能指针的情况用unique_ptr;在对象_ptr需要共享的情况下，使用 shared;需要访问 shared_ptr 对象，而又不想改变其引用计数的情况下，使用weak_ptr

类相关

什么是面向对象？面向对象的三大特性？

面向对象是一种基于对象的、基于类的的软件开发思想。



面向对象的三个基本特征，并简单叙述之？

1. 封装：将客观事物抽象成类，是实现面向对象程序设计的第一步，每个类对自身的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏（通过private, protected,public,friendly），就是将数据或函数等集合在类中。
   > 封装的意义在于保护或者防止代码（数据）被我们无意中破坏，可以隐藏实现细节，使得代码模块化
2. 继承：它可以使用现有类的所有功能, 并在无需重新编写原来类的情况下对功能进行扩展，实现重用代码。
   > 广义的继承有三种实现形式: 实现继承(使用基类的属性和方法而无需额外编码的能力), 可视继承(子窗体使用父窗体的外观和实现代码), 接口继承(仅使用属性和方法,实现滞后到子类实现);
3. 多态：即多种状态，在面向对象语言中，接口的多种不同的实现方式即为多态。不论传递过来的究竟是那个类的对象，函数都能够通过同一个接口调用到适应各自对象的实现方法。（以封装和继承为基础）
   > 一个有趣但不严谨的说法是：继承是子类使用父类的方法，而多态则是父类使用子类的方法
   > 分为两类, 编译时的多态性, 通过重载实现; 运行时的多态,由虚函数来实现;
   > 用子类对象给父类对象赋值之后, 父类对象就可以根据当前赋值给它的子对象的特性以不同的方式运作;声明基类的指针，利用该指针指向任意一个子类对象，调用相应的虚函数，可以根据指向的子类的不同而实现不同的方法。

抽象类、接口类、聚合类

- 抽象类：含有纯虚函数的类
- 接口类：仅含有纯虚函数的抽象类
- 聚合类：用户可以直接访问其成员，并且具有特殊的初始化语法形式。满足如下特点：
  所有成员都是 public
  没有有定于任何构造函数
  没有类内初始化
  没有基类，也没有 virtual 函数

对象组合和继承

- 继承
  继承常常被称为白箱复用(white-box reuse)，因为基类的内部描述与细节通常对子类可见。
- 对象组合
  要求被组合的对象具有良好的接口，并且通过从其他对象得到的引用在运行时运态定义。所以可以将对象组合到其他对象中，以构建更加复杂的功能。
  由于对象的内部细节对其他对象不可见，它们看上去为“黑箱”，这种类型的复用称为黑箱复用(black-box reuse)。

继承的优缺点

- 优点：
  1、类继承简单粗爆，直观，关系在编译时静态定义。
  2、被复用的实现易于修改，sub可以覆盖super的实现。
- 缺点：
  1、基类实现的任何变更都会强制子类也进行变更，因为它们的实现联系在了一起。
  2、派生类的部分实现通常定义在基类中。
  3、派生类直接面对基类的实现细节，因此破坏了封装。
  4、如果在新的问题场景下继承来的实现已过时或不适用，所以必须重写基类或继承来的实现。

对象组合的优缺点

对象组合让我们同时使用多个对象，而每个对象都假定其他对象的接口正常运行。因此，为了在系统中正常运行，它们的接口都需要经过精心的设计。

- 优点
  1、不会破坏封装，因为只通过接口来访问对象；
  2、减少实现的依存关系，因为实面是通过接口来定义的；
  3、可以在运行时将任意对象替换为其他同类型的对象；
  4、可以保持类的封装以专注于单一任务；
  5、类和他的层次结构能保持简洁，不至于过度膨胀而无法管理；

- 缺点：
  1、涉及对象多；
  2、系统的行为将依赖于不同对象间的关系，而不是定义于单个类中；
  3、现成的组件总是不太够用，从而导致我们要不停的定义新对象。

this指针

- this 指针是一个隐含于每一个非静态成员函数中的特殊指针。它指向正在被该成员函数操作的那个对象
- 当对一个对象调用成员函数时，编译程序先将对象的地址赋给 this 指针，然后调用成员函数，每次成员函数存取数据成员时，由隐含使用 this 指针。
- this 指针被隐含地声明为: ClassName *const this*，这意味着不能给 *this* 指针赋值；
  在 *ClassName* 类的 *const* 成员函数中，*this* 指针的类型为：*const ClassName* const，这说明不能对 this 指针所指向的这种对象是不可修改的（即不能对这种对象的数据成员进行赋值操作）；
- this 并不是一个常规变量，而是个右值，所以不能取得 this 的地址（不能 &this）。

重写、重载与隐藏的区别

1. 重载的函数都是在类内的。只有参数类型或者参数个数不同，重载不关心返回值的类型。
2. 覆盖（重写）派生类中重新定义的函数，其函数名，返回值类型，参数列表都跟基类函数相同，并且基类函数前加了virtual关键字。
3. 隐藏是指派生类的函数屏蔽了与其同名的基类函数，注意只要同名函数，不管参数列表是否相同，基类函数都会被隐藏。
   有两种情况：（1）参数列表不同，不管有无virtual关键字，都是隐藏；（2）参数列表相同，但是无virtual关键字，也是隐藏。

关于类占用的内存空间，有以下几点需要注意：

（1）如果类中含有虚函数，则编译器需要为类构建虚函数表，类中需要存储一个指针指向这个虚函数表的首地址，注意不管有几个虚函数，都只建立一张表，所有的虚函数地址都存在这张表里，类中只需要一个指针指向虚函数表首地址即可。
（2）类中的静态成员是被类所有实例所共享的，它不计入sizeof计算的空间
（3）类中的普通函数或静态普通函数都存储在栈中，不计入sizeof计算的空间
（4）类成员采用字节对齐的方式分配空间

类的构造函数

- 类的构造函数调用顺序
  1、首先调用基类的构造函数，
  2、其次调用的是成员类的构造函数，
  3、最后才是自身的构造函数；

  > 1.吸收基类成员 2.改造基类成员 3.添加新成员
  > 子类包含基类的信息，所以首先要调用基类的的构造函数，按照成员类的定义顺序申请空间，调用过成员类的构造函数后在调用自身的构造函数。

- **必须在构造函数初始化式里进行初始化的数据成员有哪些**
  1) 常量成员，因为常量只能初始化不能赋值，所以必须放在初始化列表里面
  2) 引用类型，引用必须在定义的时候初始化，并且不能重新赋值，所以也要写在初始化列表里面
  3) 没有默认构造函数的类类型，因为使用初始化列表可以不必调用默认构造函数来初始化，而是直接调用拷贝构造函数初始化
- **如何让一个类不能实例化？**
  将类定义为抽象基类或者将构造函数声明为private。
- 拷贝构造函数作用及用途？什么时候需要自定义拷贝构造函数？
  一般如果构造函数中存在动态内存分配，则必须定义拷贝构造函数。否则，可能会导致两个对象成员指向同一地址，出现“指针悬挂问题”。
- 谈谈你对拷贝构造函数和赋值运算符的认识
  拷贝构造函数和赋值运算符重载有以下两个不同之处:

1、拷贝构造函数生成新的类对象,而赋值运算符不能;

2、由于拷贝构造函数是直接构造一个新的类对象, 所以在初始化这个对象之前不用检验源对象是否和新建对象相同;而赋值运算符则需要这个操作, 另外赋值运算中如果原来的对象中有内存分配要先把内存释放掉

- 隐式类型转换
  首先，对于内置类型，低精度的变量给高精度变量赋值会发生隐式类型转换，其次，对于只存在单个参数的构造函数的对象构造来说，函数调用可以直接使用该参数传入，编译器会自动调用其构造函数生成临时对象。
- C++中的explicit关键字有何作用？
  解答：禁止将构造函数作为转换函数，即禁止构造函数自动进行隐式类型转换。
  例如CBook中只有一个参数m_price，在构建对象时可以使用CBook c = 9.8这样的隐式转换，使用explicit防止这种转换发生。
- 如何在类中定义常量成员并为其初始化？
  解答：只能在初始化列表里对const成员初始化
- 静态数据成员
  只能在全局区域进行初始化，而不能在类体中进行（构造函数中初始化也不行），且静态数据成员不涉及对象，因此不受类访问限定符的限制。
  例子说明如下：

```
class CBook {
public:
    static double m_price;
};
double CBook::m_price = 8.8; // 只能在这初始化，不能在CBook的构造函数或直接初始化
```

如何定义一个只能在堆上（栈上）生成对象的类

- 只能在堆上创建类对象
  方法1：将析构函数设置为私有，自己写一个destory函数进行析构

> 原因：C++ 是静态绑定语言，编译器管理栈上对象的生命周期，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。若析构函数为私有的，则不可访问，即不能在栈上创建对象。
> 还可以采用把构造函数和析构函数设为protected，使用一个public的static函数来完成构造，和析构。（类似于单例模式）

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

- 定义只能在栈上
  方法：在类中将 new 运算符重载（ 同时还需重载 delete，配对关系 ）

> 原因：在堆上生成对象，使用 new 关键词操作，其过程分为两阶段：
> 第一阶段，使用 new 在堆上寻找可用内存，分配给对象；
> 第二阶段，调用构造函数生成对象。
> 将 new 操作设置为私有，那么第一阶段就无法完成，就不能够在堆上生成对象。

```
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(){}
};
```

有什么类是不能继承的

如果存在子类，那么子类会调用父类的构造函数，那么我们可以将这个类的**构造函数和析构函数**都声明是私有的，那么这样它的子类构造时就会报错，这个类就不能被继承。

> 这个类对象则采用堆上分配的方法进行初始化

在定义类的成员函数时使用mutable关键字的作用是什么？

解答：当需要在const方法中修改对象的数据成员时，可以在数据成员前使用mutable关键字，防止出现编译出错。

```
class CBook {
public:
    mutable double m_price; // 如果不加就会出错
    CBook(double price) :m_price(price) { }
    double getPrice() const; // 定义const方法
};
double CBook::getPrice() const {
    m_price = 9.8;
    return m_price;
}
```

派生类不能继承的有

子类继承父类大部分的资源，不能继承的有：构造函数，析构函数，拷贝构造函数，operator=函数，友元函数等等。

纯虚函数

- **纯虚函数如何定义？**
  纯虚函数是在基类中声明的虚函数，它在基类中没有定义，但要求任何派生类都要定义自己的实现方法。纯虚函数是虚函数再加上= 0。virtual void fun ()=0。
- 含有纯虚函数的类称为什么？
  含有纯虚函数的类称为抽象类，它不能生成对象。

  > 例如，动物作为一个基类可以派生出老虎、孔雀等子类，但动物本身生成对象明显不合常理。

- **纯虚函数的作用和实现方式?**
  纯虚函数是一种特殊的虚函数，在许多情况下，在基类中不能对虚函数给出有意义的实现，而把它声明为纯虚函数，它的实现留给该基类的派生类去做。这就是纯虚函数的作用。

  > 通俗来说，使用纯虚函数的类我们称为抽象类，该类不提供方法只提供接口，也可以很形象的理解为就是一个API。

- **虚函数与纯虚函数区别**
  1、虚函数在子类里面也可以不重载的；但纯虚函数必须在子类去实现
  2、带纯虚函数的类叫虚基类也叫抽象类，这种基类不能直接生成对象，只能被继承，重写虚函数后才能使用，运行时动态绑定！

析构函数的作用

- 析构函数与构造函数对应，当对象结束其生命周期，如对象所在的函数已调用完毕时，系统会自动执行析构函数。
- 析构函数名也应与类名相同，只是在函数名前面加一个位取反符~，例如~stud( )，以区别于构造函数。它不能带任何参数，也没有返回值（包括void类型）。只能有一个析构函数，不能重载。
- 如果用户没有编写析构函数，编译系统会自动生成一个缺省的析构函数（即使自定义了析构函数，编译器也总是会为我们合

成一个析构函数，并且如果自定义了析构函数，编译器在执行时会先调用自定义的析构函数再调用合成的析构函数），它也不进行任何操作。所以许多简单的类中没有用显式的析构函数。

- 如果一个类中有指针，且在使用的过程中动态的申请了内存，那么最好显示构造析构函数在销毁类之前，释放掉申请的内存空间，避免内存泄漏。
- 类析构顺序：1）派生类本身的析构函数；2）对象成员析构函数；3）基类析构函数。

虚函数相关问题

- 为什么对于存在虚函数的类中析构函数要定义成虚函数
  为了实现多态进行动态绑定，将派生类对象指针绑定到基类指针上，对象销毁时，如果析构函数没有定义为析构函数，则会调用基类的析构函数，显然只能销毁部分数据。如果要调用对象的析构函数，就需要将该对象的析构函数定义为虚函数，销毁时通过虚函数表找到对应的析构函数。
- **C++构造函数能抛异常吗？析构函数呢？**
  从语法上来说，构造函数和析构函数都可以抛出异常。但从逻辑上和风险控制上，构造函数可以，析构函数不推荐抛出异常。
  - 构造函数
    动态创建对象要进行两个操作：分配内存和调用构造函数。若在分配内存时出错，会抛出bad_alloc异常；
    构造函数中抛出异常，会导致析构函数不能被调用，但对象本身已申请到的内存资源会被系统释放（已申请到资源的内部成员变量会被系统依次逆序调用其析构函数），不会造成内存泄漏。
  - 析构函数
    (1) 如果析构函数抛出异常，导致异常点之后的程序不会执行，如果异常点之后还有某些必要的动作（比如释放某些资源），则这些动作不会执行，会造成诸如资源泄漏的问题。
    (2) 通常异常发生时，c++的机制会调用已经构造对象的析构函数来释放资源，此时若析构函数本身也抛出异常，则前一个异常尚未处理，又有新的异常，会造成程序崩溃的问题。

    > C++标准指明析构函数不能、也不应该抛出异常。如果对象在运行期间出现了异常，C++异常处理模型会清除由于出现异常所导致失效的对象(即对象超出了它原来的作用域)，并释放对象原来所分配的资源， 这就是调用对象的析构函数来完成释放资源的任务，所以从这个意义上说，析构函数已经变成了异常处理的一部分。

- **构造函数为什么不能定义为虚函数，析构函数呢？**

1. 构造函数
   虚函数的执行依赖于虚函数表。虚函数表需要在构造函数中进行初始化工作，即初始化vptr，让他指向正确的虚函数表。而在构造对象期间，虚函数表还没有被初始化，虚构造函数将无法进行，所以构造函数为什么不能定义为虚函数。
2. 析构函数
   在类的继承中，如果有基类指针指向派生类，那么用基类指针delete时，如果析构函数不是虚函数，那么释放内存时候，编译器会使用静态联编，认为p就是一个基类指针，调用基类析构函数，这样子类对象的内存没有释放，造成内存泄漏。定义成虚函数以后，就会动态联编，先调用子类析构函数，再基类。

- 虚函数（virtual）可以是内联函数（inline）吗
  不可以。
  inline是在编译器将函数内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（不知道复制展开哪个函数），所以不能够inline声明展开，所以编译器会忽略（编译器会按照自己的规则进行编译，比如成员函数会自动优化为内联函数，我定义的inline可能会被忽略）
- **C++中哪些不能是虚函数？**

1. 普通函数只能重载，不能被重写，因此编译器会在编译时绑定函数。
2. 构造函数是知道全部信息才能创建对象，然而虚函数允许只知道部分信息。
3. 内联函数不可以是虚函数
   内联函数在编译时被展开，虚函数在运行时才能动态绑定函数。inline函数是个静态行为，而虚函数是个动态行为。
4. 友元函数因为不可以被继承。
5. 静态成员函数 只有一个实体，不能被继承。父类和子类共有。

- **派生类与虚函数概述**

1. 派生类继承的函数不能定义为虚函数。虚函数是希望派生类重新定义。如果派生类没有重新定义某个虚函数，则在调用的时候会使用基类中定义的版本。
2. 派生类中函数的声明必须与基类中定义的方式完全匹配。
3. 基类中声明为虚函数，则派生类也为虚函数。

- C++的虚函数有什么作用？
  虚函数作用是实现多态，虚函数其实是实现封装，使得使用者不需要关心实现的细节。在很多设计模式中都是这样用法，例如Factory、Bridge、Strategy模式。

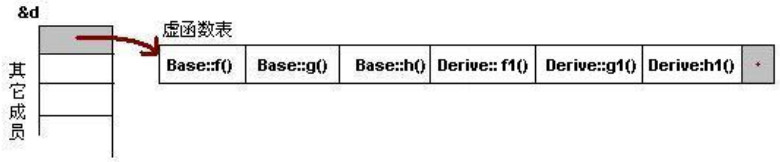
多态实现机制？ 多态作用？两个必要条件？

- C++中多态机制主要体现在：重载和接口的多态
  接口多态指的是“一个接口多种实现”，实现动态的对象函数调用。
  多态的基础是继承，需要虚函数的支持。
- 作用：
  1、 隐藏实现细节，代码能够模块化；
  2、接口重用：为了类在继承和派生的时候正确调用。
- 必要条件：
  1、一个基类的 指针或者引用 指向派生类的对象；
  2、虚函数

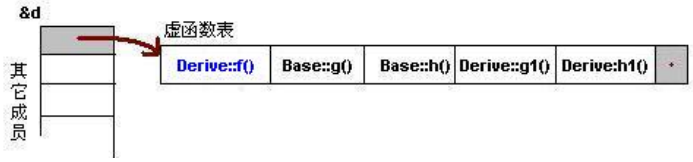
虚函数表

基类

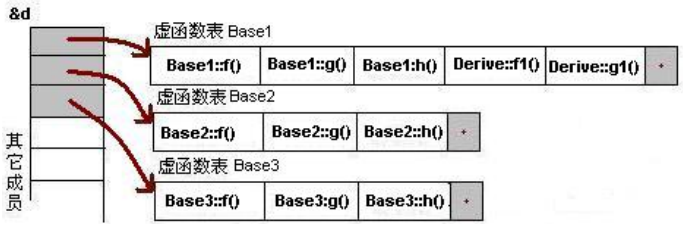
一般继承（无虚函数覆盖）



一般继承（有虚函数覆盖）

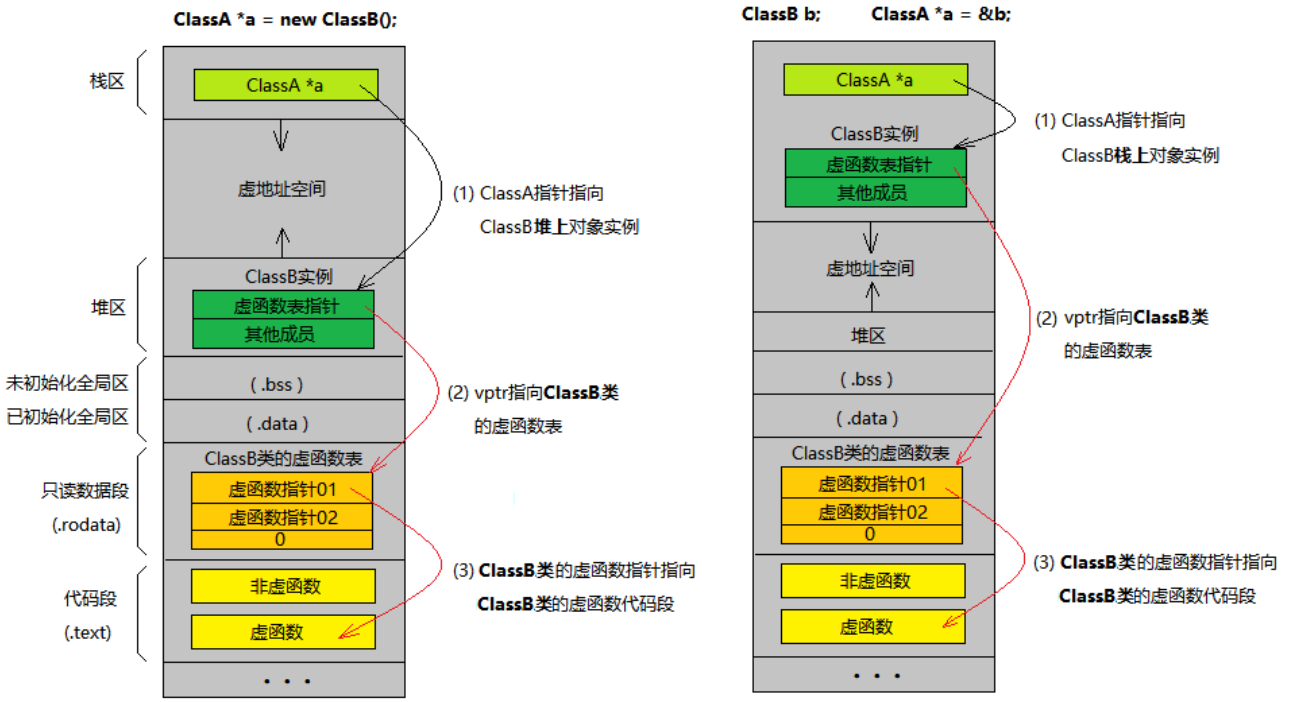


多重继承（无虚函数覆盖）



多重继承（有虚函数覆盖）

> 注意这里是三个虚表指针，占12字节。
> 当子类有多出来的虚函数时，加到第一个虚函数表中。
> 当有多个虚函数表时，虚函数表的结果是0代表没有下一个虚函数表。" * "号位置在不同操作系统中实现不同，代表有下一个虚函数表。（上图全是以小数点结尾，所以不易理解）。按这个讲法，即可按顺序遍历三个虚函数表，找到指定的函数。
> 该例子的基类许多函数没有被重写，实际就没有被调用的意义，而被重写的虚函数会覆盖对应的虚函数表位置



— C++多态虚函数表总体描述01 —

- 虚函数表
  虚表是属于类的，而不是属于某个具体的对象，一个类只需要一个虚表即可。
  虚表是一个指针数组，其元素是虚函数的指针，每个元素对应一个虚函数的函数指针。虚函数指针的赋值发生在编译器的编译阶段，也就是说在代码的编译阶段，创建虚函数表。
  创建派生类虚函数表时，首先新建并复制基类的虚函数表，如果有重写的虚函数，则覆盖表中对应的基类函数，无重写或者新加的虚函数按顺序加在表后面。
- 虚表指针vptr
  每个包含虚表的**类的对象都拥有一个虚表指针**，编译器在类中添加了一个指针，*__vptr，用来指向虚表。这样，当类的对象在创建时（调用构造函数时）便拥有了这个指针，且这个指针的值会自动被设置为指向类的虚表。所以在调用虚函数时，就能够找到正确的函数。
- 虚表指针vptr是如何访问虚函数

```
B bObject;  //A是基类，B是派生类
A *p = & bObject;
p->vfunc1();
```

程序在执行p->vfunc1()时，会发现p是个指针，且调用的函数是虚函数，接下来便会进行以下的步骤。
1、首先，根据虚表指针p->__vptr来访问对象bObject对应的虚表。虽然指针p是基类A类型，但是__vptr也是基类的一部分，所以可以通过p->__vptr可以访问到对象对应的虚表。
2、然后，在虚表中查找所调用的函数对应的条目。由于虚表在编译阶段就可以构造出来了，所以可以根据所调用的函数定位到虚表中的对应条目。对于 p->vfunc1()的调用，B vtbl的第一项即是vfunc1对应的条目。（具体vfunc1()怎么找到的，可能就是根据函

数名吧）
3、最后，根据虚表中找到的函数指针，调用函数。B vtbl的第一项指向B::vfunc1()，所以 p->vfunc1()实质会调用B::vfunc1()函数。

- **原理**
  拥有虚函数的类会有一个虚表，而且这个**虚表存放在类定义模块的数据段中**。模块的数据段通常存放定义在该模块的全局数据和静态数据区，这样我们可以把虚表看作是模块的全局数据或者静态数据
  类的虚表会被**这个类的所有对象所共享**。类的对象可以有很多，但是他们的虚表指针都指向同一个虚表，从这个意义上说，我们可以把**虚表简单理解为类的静态数据成员**。值得注意的是，虽然虚表是共享的，但是虚表指针并不是，**类的每一个对象有一个属于它自己的虚表指针。**
  虚表中存放的是虚函数的地址。
- 安全性问题
  任何使用父类指针想调用子类中的未覆盖父类的成员函数的行为都会被编译器视为非法。
  但是通过指针的方式，指向虚函数表，根据地址运行指定的虚函数，可以非法的访问子类中未覆盖父类的成员函数、派生类访问父类的private虚函数。（不安全的）

虚函数和多态

- 多态的实现主要分为静态多态和动态多态。
  - 静态多态主要是重载（函数重载和运算符重载），在编译的时候就已经确定；
  - 动态多态是用虚函数机制实现的，在运行期间动态绑定。
    举个例子：一个父类类型的指针指向一个子类对象时候，使用父类的指针去调用子类中重写了的父类中的虚函数的时候，会调用子类重写过后的函数，在父类中声明为加了virtual关键字的函数，在子类中重写时候不需要加virtual也是虚函数。
- 什么叫静态关联，什么叫动态关联？
  多态中，静态关联是程序在编译阶段就能确定实际执行动作，程序运行才能确定叫动态关联。
- 虚函数表和虚表指针
  - 编译时若基类中有虚函数，编译器为该类创建一个一维数组的虚表（在程序只读数据段.rodata section），存放是每个虚函数的地址，实际的虚函数在代码段(.text)中。基类和派生类都包含虚函数时，这两个类都建立一个虚表。
  - 在有虚函数的类中，类的最开始部分是一个虚函数表的指针，这个指针指向一个虚函数表。当子类继承了父类的时候也会**继承其虚函数表**，当子类重写父类中虚函数时候，会将其继承的虚函数表中的对应地址替换为重写后的函数地址。
- 虚函数的缺点
  虚函数，会增加访问内存开销（函数指针），降低效率（遍历虚函数表）。
- 多态性与普通函数调用的比较
  - 当用一个指针/引用调用一个普通函数的时候，被调用的函数是取决于这个指针/引用的类型。
    1、如果这个指针/引用是基类对象的指针/引用就调用基类的方法；
    2、如果指针/引用是派生类对象的指针/引用就调用派生类的方法，当然如果派生类中没有此方法，就会向上到基类里面去寻找相应的方法。这些调用在编译阶段就确定了。
  - 涉及到多态性的时候
    采用了虚函数和动态绑定，此时的调用是在运行时确定。不在单独考虑指针/引用的类型，而是看指针/引用的对象的类型来判断函数的调用，根据对象中虚指针指向的虚表中的函数的地址来确定调用哪个函数。
- **简述虚函数实现多态的原理**
  1、编译器发现一个类中有虚函数, 便会立即为此类生成虚函数表 vtable;继承类则先复制基类的虚函数表，再进行覆盖；虚函数表的各表项为指向对应虚函数的指针;
  2、编译器还会在此类中隐含插入一个指针vptr(该指针插在类的第一个位置)指向虚函数表; 调用此类的构造函数时,
  3、在类的构造函数中, 编译器会隐含执行vptr与vtable的关联代码, 将vptr指向对应的vtable, 将类与此类的vtable联系了起来;
  4、另外在调用类的构造函数时, 指向基类的指针此时变成指向具体的类的this指针, 这样依靠此this指针即可得到正确的vtable; 如此才能真正与函数体进行连接, 这就是动态联编,实现多态的基本原理;（相当于基类指着指向了派生类，派生类中含有虚函数表）
  注意: 一定要区分虚函数,纯虚函数, 虚拟继承的关系和区别;牢记虚函数实现原理, 因为多态C++面试的重要考点之一，而虚函

数是实现多态的基础;

虚继承

虚继承用于**解决多继承条件下的菱形继承问题**（浪费存储空间、存在二义性）。

- 底层实现原理与编译器相关，一般通过**虚基类指针**和**虚基类表**实现，每个虚继承的子类都有一个虚基类指针（占用一个指针的存储空间，4字节）和虚基类表（不占用类对象的存储空间）（需要强调的是，虚基类依旧会在子类里面存在拷贝，只是仅仅最多存在一份而已，并不是不在子类里面了）；当虚继承的子类被当做父类继承时，虚基类指针也会被继承。
- 实际上，vbptr 指的是虚基类表指针（virtual base table pointer），该指针指向了一个虚基类表（virtual table），虚表中记录了虚基类与本类的偏移地址；通过偏移地址，这样就找到了虚基类成员，而虚继承也不用像普通多继承那样维持着公共基类（虚基类）的两份同样的拷贝，节省了存储空间。

虚继承的作用是什么？

在多继承中，子类可能同时拥有多个父类，如果这些父类还有相同的父类（祖先类），那么在子类中就会有多份祖先类。例如，类B和类C都继承与类A，如果类D派生于B和C，那么类D中就会有两份A。为了防止在多继承中子类存在重复的父类情况，可以在父类继承时使用虚函数，即在类B和类C继承类A时使用virtual关键字，例如：
class B : virtual public A
class C : virtual public A
注：因为多继承会带来很多复杂问题，因此要慎用。

虚继承与虚函数

- 相同之处：
  都利用了虚指针（均占用类的存储空间）和虚表（均不占用类的存储空间）
- 不同之处：
  虚基类依旧存在继承类中，只占用存储空间。虚基类表存储的是虚基类相对直接继承类的偏移
  虚函数不占用存储空间，虚函数表存储的是虚函数地址

多重继承有什么问题？怎样消除多重继承中的二义性？

1. 增加程序的复杂度，使程序的编写和维护比较困难，容易出错；
2. 继承类和基类的同名函数产生了二义性，同名函数不知道调用基类还是继承类，C++中使用虚函数解决这个问题
3. 继承过程中可能会继承一些不必要的数据，对于多级继承，可能会产生数据很长

> 可以使用成员限定符和虚函数解决多重继承中函数的二义性问题。

return this有关的问题

```
 return *this
返回的是当前对象的克隆或者本身（若返回类型为A， 则是克隆， 若返回类型为A&(引用)， 则是本身 ）。
 return this
返回当前对象的地址（指向当前对象的指针）
```

c++11 新特性

RAII

RAII技术被认为是C++中管理资源的最佳方法，进一步引申，使用RAII技术也可以实现安全、简洁的状态管理，编写出优雅的异常安全的代码。

资源管理

AII全称是“Resource Acquisition is Initialization”，直译过来是“资源获取即初始化”，也就是说在构造函数中申请分配资源，在析构函数中释放资源。因为C++的语言机制保证了，当一个对象创建的时候，自动调用构造函数，当对象超出作用域的时候会自动调用

析构函数。所以，在RAII的指导下，我们应该使用类来管理资源，将资源和对象的生命周期绑定。

- 智能指针（std::shared_ptr和std::unique_ptr）即RAII最具代表的实现，使用智能指针，可以实现自动的内存管理，再也不需要担心忘记delete造成的内存泄漏。毫不夸张的来讲，有了智能指针，代码中几乎不需要再出现delete了。

状态管理

RAII另一个引申的应用是可以实现安全的状态管理。一个典型的应用就是在线程同步中，使用std::unique_lock或者std::lock_guard对互斥量std:: mutex进行状态管理。

> 因为，在互斥量lock和unlock之间的代码很可能会出现异常，或者有return语句，这样的话，互斥量就不会正确的unlock，会导致线程的死锁。所以正确的方式是使用std::unique_lock或者std::lock_guard对互斥量进行状态管理：

总结

RAII的核心思想是将资源或者状态与对象的生命周期绑定，通过C++的语言机制，实现资源和状态的安全管理。理解和使用RAII能使软件设计更清晰，代码更健壮。

lambda表达式

lambda表达式是C++11最重要也最常用的一个特性之一。lambda来源于函数式编程的概念，也是现代编程语言的一个特点。

优点

1)声明式编程风格:就地匿名定义目标函数或函数对象，不需要额外写一个命名函数或者函数对象。以更直接的方式去写程序，好的可读性和可维护性。
2)简洁：不需要额外再写一个函数或者函数对象，避免了代码膨胀和功能分散，让开发者更加集中精力在手边的问题，同时也获取了更高的生产率。
3)在需要的时间和地点实现功能闭包，使程序更灵活。

STL

容器	底层数据结构	时间复杂度	有无序	可不可重复	其他
array	数组	随机读改 O(1)	无序	可重复	支持快速随机访问
vector	数组	随机读改、尾部插入、尾部删除 O(1)、头部插入、头部删除 O(n)	无序	可重复	支持快速随机访问
list	双向链表	插入、删除 O(1)、随机读改 O(n)	无序	可重复	支持快速增删
deque	双端队列	头尾插入、头尾删除 O(1)	无序	可重复	一个中央控制器 + 多个缓冲区，支持首尾快速增删，支持随机访问
stack	deque / list	顶部插入、顶部删除 O(1)	无序	可重复	deque 或 list 封闭头端开口，不用 vector 的原因应该是容量大小有限制，扩容耗时
queue	deque / list	尾部插入、头部删除 O(1)	无序	可重复	deque 或 list 封闭头端开口，不用 vector 的原因应该是容量大小有限制，扩容耗时
priority_queue	vector + max-heap	插入、删除 O(log2n)	有序	可重复	vector容器+heap处理规则
set	红黑树	插入、删除、查找 O(log2n)	有序	不可重复	
multiset	红黑树	插入、删除、查找 O(log2n)	有序	可重复	
map	红黑树	插入、删除、查找 O(log2n)	有序	不可重复	
multimap	红黑树	插入、删除、查找 O(log2n)	有序	可重复	
hash_set	哈希表	插入、删除、查找 O(1) 最差 O(n)	无序	不可重复	
hash_multiset	哈希表	插入、删除、查找 O(1) 最差 O(n)	无序	可重复	
hash_map	哈希表	插入、删除、查找 O(1) 最差 O(n)	无序	不可重复	
hash_multimap	哈希表	插入、删除、查找 O(1) 最差 O(n)	无序	可重复	

- STL里set和map是基于什么实现的。红黑插入、删除、查找 O(1) 最差 O(n)树的特点？
  a. set和map都是基于红黑树实现的。
  b. 红黑树是一种平衡二叉查找树，与AVL树的区别是什么？AVL树是完全平衡的，红黑树基本上是平衡的。
  c. 为什么选用红黑数呢？因为红黑数是平衡二叉树，其插入和删除的效率都是N(logN)，与AVL相比红黑数插入和删除最多只需要3次旋转，而AVL树为了维持其完全平衡性，在坏的情况下要旋转的次数太多。
  红黑树的定义：
  (1) 节点是红色或者黑色；
  (2) 父节点是红色的话，子节点就不能为红色；
  (3) 从根节点到每个页子节点路径上黑色节点的数量相同；
  (4) 根是黑色的，NULL节点被认为是黑色的。

stl各容器的实现原理

1. Vector顺序容器，是一个动态数组，支持随机插入、删除、查找等操作，在内存中是一块连续的空间。在原有空间不够情况下自动分配空间，增加为原来的两倍。vector随机存取效率高，但是在vector插入元素，需要移动的数目多，效率低下。

> 注：vector动态增加大小时是以原大小的两倍另外配置一块较大的空间，然后将原内容拷贝过来，然后才开始在原内容之后构造新元素，并释放原空间。因此，对vector空间重新配置，指向原vector的所有迭代器就都失效了。

2. Map关联容器，以键值对的形式进行存储，方便进行查找。关键词起到索引的作用，值则表示与索引相关联的数据。红黑树的结构实现，插入删除等操作都在O(logn)时间内完成。
3. Set是关联容器，set每个元素只包含一个关键字。set支持高效的关键字检查是否在set中。set也是以红黑树的结构实现，支持高效插入、删除等操作。

emplace_back与push_back区别

emplace_back和push_back都是向容器内添加数据.

> 对于在容器中添加类的对象时, 相比于push_back,emplace_back可以避免额外类的复制和移动操作

```
elections.emplace_back("Nelson Mandela", "South Africa", 1994); //没有类的对象创建，但会调用构造函数
reElections.push_back(President("Franklin Delano Roosevelt", "the USA", 1936)); //会调用类的构造函数，复制构造函数，类的析构函数
```

STL有7种主要容器

vector,list,deque,map,multimap,set,multiset