一、C 和 C++的区别是什么？

C 是面向过程的语言，C++是在 C 语言的基础上开发的一种面向对象编程语言，应用广泛。

C 中函数不能进行重载，C++函数可以重载

C++在 C 的基础上增添类，C 是一个结构化语言，它的重点在于算法和数据结构。C 程序的设计首要考虑的是如何通过一个过程，对输入（或环境条件）进行运算处理得到输出（或实现过程（事务）控制），而对于 C++，首要考虑的是如何构造一个对象模型，让这个模型能够契合与之对应的问题域，这样就可以通过获取对象的状态信息得到输出或实现过程（事务）控制。

C++中 struct 和 class 除了默认访问权限外，别的功能几乎都相同。

Struct 默认访问权限是公共，class 默认访问权限是私有

二、关键字 static、const、extern 作用

static 和 const 的作用在描述时主要从类内和类外两个方面去讲：

static 关键字的作用：

（1）函数体内 static 变量的作用范围为该函数体，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）在模块内的 static 全局变量和函数可以被模块内的函数访问，但不能被模块外其它函数访问；

（3）在类中的 static 成员变量属于整个类所拥有，对类的所有对象只有一份拷贝；

（4）在类中的 static 成员函数属于整个类所拥有，这个函数不接收 this 指针，因而只能访问类的 static 成员变量。

const 关键字的作用：

（1）阻止一个变量被改变

（2）声明常量指针和指针常量

（3）const 修饰形参，表明它是一个输入参数，在函数内部不能改变其值

（4）对于类的成员函数，若指定其为 const 类型，则表明其是一个常函数，不能修改类的成员变量(const 成员一般在成员初始化列表处初始化)

（5）对于类的成员函数，有时候必须指定其返回值为 const 类型，以使得其返回值不为”左值”。

extern 关键字的作用：

（1）extern 可以置于变量或者函数前，以标示变量或者函数的定义在别的文件中，提示编译器遇到此变量和函数时在其他模块中寻找其定义。

（2）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。

三、sizeof 和 strlen 的区别

（1）sizeof 是运算符，而 strlen 是函数；

（2）sizeof 的用法是 sizeof(参数)，这个参数可以是数组，指针，类型，对象，甚至是函数，其值在编译的时候就计算好了，而 strlen 的参数必须是字符型指针（char*）,其值必须在函数运行的时候才能计算出来；

（3） sizeof 的功能是获得保证能容纳实现的建立的最大对象的字节的大小，而 strlen 的功能是返回字符串的长度，切记这里的字符串的长度是包括结束符的；

（4）当数组作为参数传递给函数的时候，传的是指针，而不是数组，传递数组的首地址；

```
char str[20] = "0123456789";
int a = strlen(str);  //10
int b = sizeof(str);//20
```

四、指针和引用的区别

（1）指针：指针是一个变量，只不过这个变量存储的是一个地址，指向内存的一个存储单元；而引用跟原来的变量实质上是同一个东西，只不过是原变量的一个别名而已。

（2）指针可以有多级，但是引用只能是一级（int **p；合法 而 int &&a 是不合法的）

（3）指针的值可以为空，但是引用的值不能为 NULL，并且引用在定义的时候必须初始化

（4）指针的值在初始化后可以改变，即指向其它的存储单元，而引用初始化后就不会再改变。

（5）"sizeof 引用"得到的是所指向的变量(对象)的大小，而"sizeof 指针"得到的是指针本身的大小。

（6）作为参数传递时，二者有本质不同：指针传参本质是值传递，被调函数的形参作为局部变量在栈中开辟内存以存放由主调函数放进来的实参值，从而形成实参的一个副本。而引用传递时，被调函数对形参的任何操作都会通过一个间接寻址的方式影响主调函数中的实参变量。

如果想通过指针参数传递来改变主调函数中的相关变量，可以使用指针的指针或者指针引用。

五 、指针数组、数组指针、函数指针

指针数组：首先它是一个数组，数组的元素都是指针，数组占多少个字节由数组本身的大小决定，每一个元素都是一个指针，在 32 位系统下任何类型的指针永远是占 4 个字节。它是“储存指针的数组”的简称。

数组指针：首先它是一个指针，它指向一个数组。在 32 位系统下任何类型的指针永远是占 4 个字节，至于它指向的数组占多少字节，不知道，具体要看数组大小。它是“指向数组的指针”的简称。

一个小栗子：

```
int arr[] ={1,2,3,4,5};
int *ptr =(int *)(&arr+1);  //2  5
int *ptr =(int *)(arr+1);   //2  1
cout<<*(arr+1)<<" "<<*(ptr-1)<<endl;
```

//数组名 arr 可以作为数组的首地址，而&a 是数组的指针。

//arr 和&arr 指向的是同一块地址，但他们+1 后的效果不同，arr+1 是一个元素的内存大小（增加 4）

//而&arr+1 增加的是整个数组的内存

数组指针(行指针)

```
int a[2][3] = {{1,2,3},{4,5,6}};
int (*p)[3];
p = a;
p++;
cout<<**p<<endl;  //4  the second rank
```

六、C++内存布局

C/C++程序编译时内存分为 5 大存储区

（1）栈区（stack）：由编译器自动分配释放，存放函数的参数值，局部变量值等，其操作方法类似数据结构中的栈。

（2）堆区（heap）：一般由程序员分配释放，与数据结构中的堆毫无关系，分配方式类似于链表。

（3）全局/静态区（static）：全局变量和静态变量的存储是放在一起的，在程序编译时分配。

（4）文字常量区：存放常量字符串。

（5）程序代码区：存放函数体（类的成员函数、全局函数）的二进制代码

```
int a=0; //全局初始化区
char *p1; //全局未初始化区
void main()
{
   int b; //栈
   char s[]="bb"; //栈
   char *p2; //栈
   char *p3="123"; //其中，“123\0”常量区，p3 在栈区
   static int c=0; //全局区
   p1=(char*)malloc(10); //10 个字节区域在堆区
   strcpy(p1,"123"); //"123\0"在常量区，编译器 可能 会优化为和 p3 的指向同一块区域
}
```

C/C++内存分配有三种方式：

（1）从静态存储区域分配。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。例如全局变量，static 变量。

（2）在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。

栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

（3）从堆上分配，亦称动态内存分配。程序在运行的时候用 malloc 或 new 申请任意多少的内存，程序员自己负责在何时用 free 或 delete 释放内存。

动态内存的生存期由程序员决定，使用非常灵活，但如果在堆上分配了空间，就有责任回收它，否则运行的程序会出现内存泄漏。

另外频繁地分配和释放不同大小的堆空间将会产生堆内碎块。

七、堆和栈的区别

（1）申请方式

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

heap:

需要程序员自己申请，并指明大小，在 c 中 malloc 函数

如 p1 = (char *)malloc(10);

在 C++中用 new 运算符

如 p2 = (char *)malloc(10);

但是注意 p1、p2 本身是在栈中的。

（2）申请后系统的响应

栈：只要栈的剩余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。

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，当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 结点链表中删除，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 delete 语句才能正确的释放本内存空间。另外，由于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。

（3）申请大小的限制及生长方向

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（也可能是 1M，它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 。

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。

（4）申请效率的比较：

栈由系统自动分配，速度较快。但程序员是无法控制的。

堆是由 new 分配的内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便.

另外，在 WINDOWS 下，最好的方式是用 VirtualAlloc 分配内存，他不是在堆，也不是在栈是直接在进程的地址空间中保留一快内存，虽然用起来最不方便。但是速度快，也最灵活。

（5）堆和栈中的存储内容

栈：在函数调用时，第一个进栈的是主函数中后的下一条指令（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。

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。

八、malloc/free 、new/delete 区别

（1）malloc 与 free 是 C++/C 语言的标准库函数，new/delete 是 C++的运算符。它们都可用于申请动态内存和释放内存。

（2）对于非内部数据类型的对象而言，光用 maloc/free 无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。

由于 malloc/free 是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于 malloc/free。因此 C++语言需要一个能完成动态内存分配和初始化工作的运算符 new，以一个能完成清理与释放内存工作的运算符 delete。注意 new/delete 不是库函数。

（3）C++程序经常要调用 C 函数，而 C 程序只能用 malloc/free 管理动态内存。

（4）new 可以认为是 malloc 加构造函数的执行。new 出来的指针是直接带类型信息的。而 malloc 返回的都是 void 指针。

九 、常见的内存错误及对策

（1）内存尚未分配成功，却使用了它；

解决办法：在使用内存之前检查指针是否为 NULL。如果指针 p 是函数的参数，那么在函数的入口使用 assert(p != NULL) 进行检查，如果是用 malloc 或者 new 来申请的，应该用

if (p == NULL)或者 if (p != NULL)来进行防错处理。

（2）内存分配虽然成功，但是尚未初始化就引用它；

错误原因：一是没有初始化的观念，二是误以为内存的缺省初值全为零，导致引用初值错误（如数组）。

解决办法：内存的缺省初值是什么并没有统一的标准，尽管有些时候为零值，但是宁可信其有，不可信其无，无论以何种方式创建数组，都要赋初值。

（3）内存分配成功并初始化，但是超过了内存的边界；

这种问题常出现在数组越界，写程序是要仔细。

（4）忘记释放内存，造成内存泄露；

含有这种错误的函数每次被调用都会丢失一块内存，开始时内存充足，看不到错误，但终有一次程序死掉，报告内存耗尽。

（5）释放了内存却继续使用它

产生原因：1.程序中的对象调用关系过于复杂，难以搞清楚某个对象究竟是否已经释放了内存，此时应该重新设计数据结构，从根本上解决对象管理的混乱局面。

2.函数 return 语句写错了，注意不要返回指向“栈内存”的指针或者引用，因为该内存在函数体结束时理论上被自动销毁。

3.使用 free 或者 delete 释放了内存后，没有将指针设置为 null,导致产生野指针。

解决办法：小心仔细。

内存管理需要遵循的规则

（1）用 malloc 或者 new 申请内存之后，应该立即检查指针值是否为 NULL ，防止使用指针值为 NULL 的内存；

（2）不要忘记数组和动态内存赋初值，防止未被初始化的内存作为右值使用；

（3）避免数组或者指针下标越界，特别要当心“多 1”或者“少 1”的操作；

（4）动态内存的申请与释放必须配对，防止内存泄露；

（5）用 free 或者 delete 释放了内存之后，立即将指针设置为 NULL，防止产生“野指针”；

十、字节对齐问题

为什么要使用字节对齐？

字节对齐是 C/C++编译器的一种技术手段，主要是在可接受空间浪费的前提下，尽可能地提高对相同元素过程的快速处理。（比如 32 位系统，4 字节对齐能使 CPU 访问速度提高）

需要字节对齐的根本原因在于 CPU 访问数据的效率问题。

字节对齐的原则

（1）结构体中每个成员相对于结构体首地址的偏移量都是成员大小的整数倍，如有需要编译器会填充字节

（2）结构体的总大小为结构体最宽基本类型成员大小的整数倍，如有需要，编译器会填充字节。

当然这里还要考虑#pragma pack(n)伪指令的影响，如果有取较小值。

```
// 用于测试的结构体
typedef struct MemAlign
{
   char a[18];   // 18 bytes
   double b; // 08 bytes
   char c;       // 01 bytes
   int d;     // 04 bytes
   short e;   // 02 bytes
}MemAlign;

//大小为：48 字节
```

对于 union：sizeof(union)，以结构里面 size 最大元素为 union 的 size，因为在某一时刻，union 只有一个成员真正存储于该地址。

十一、0 的比较判断，浮点数存储

1. int 型变量

if (n == 0)
if (n != 0)
2. bool 型
if (value == 0)
if (value != 0)
3. char*型
if(p == NULL) / if(p != NULL)
5. 浮点型
const float EPSINON = 0.0000001;
if ((x >= - EPSINON) && (x <= EPSINON)

十二、内联函数有什么优点？内联函数和宏定义的区别

优点：函数会在它所调用的位置上展开。这么做可以消除函数调用和返回所带来的开销（寄存器存储和恢复），而且，由于编译器会把调用函数的代码和函数本身放在一起优化，所以也有进一步优化代码的可能。

内联函数使用的场合：对于简短的函数并且调用次数比较多的情况，适合使用内联函数。

内联函数和宏定义区别：

1)内联函数在编译时展开，而宏在预编译时展开

2)在编译的时候，内联函数直接被嵌入到目标代码中去，而宏只是一个简单的文本替换。

3)内联函数可以进行诸如类型安全检查、语句是否正确等编译功能，宏不具有这样的功能。

4)宏不是函数，而 inline 是函数

十三、调用惯例及 printf 变参实现

函数在被调用时，函数的调用方和被调用方对于函数时如何调用的必须有一个明确的规定。只有双方同时遵循同样的规定，函数才能够被正确调用。这样的规定被称为：调用惯例。

函数的调用惯例包含两个方面：

1.函数参数的传递顺序和方式

函数的传递有很多种方式，最常见的是通过栈传递。函数的调用方将参数压入栈中，函数自己再从栈中将参数取出。对于有多个参数的函数，调用惯例要规定函数调用方将参数压栈的顺序，是从左往右压栈，还是从右往左压栈。

2.栈的维护方式

在函数将参数压入栈中之后，函数体会被调用，此后需要将被压入的参数全部弹出，使得栈在函数调用前后保持一致。这个弹出的工作可以由函数调用方来完成，也可以由函数本身来完成。在不指定调用惯例的情况下，默认采用 cdecl 惯例。

在这里插入图片描述

十四 、覆盖、重载、隐藏的区别

（1）重载：重载翻译自 overload，是指同一可访问区内被声明的几个具有不同参数列表（参数的类型，个数，顺序不同）的同名函数，根据参数列表确定调用哪个函数，重载不关心函数返回类型。

（2）重写：重写翻译自 override，是指派生类中存在重新定义的函数。其函数名，参数列表，返回值类型，所有都必须同基类中被重写的函数一致，只有函数体不同。

1.成员函数被重载的特征：

（1）相同的范围（在同一个类中）；

（2）函数名字相同；

（3）参数不同；

（4）virtual 关键字可有可无。

2.覆盖是指派生类函数覆盖基类函数，特征是：

（1）不同的范围（分别位于派生类与基类）；

（2）函数名字相同；

（3）参数相同；

（4）基类函数必须有 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，规则如下：

（1）如果派生类的函数与基类的函数同名，但是参数不同。此时，不论有无 virtual 关键字，基类的函数将被隐藏（注意别与重载混淆）。

（2）如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有 virtual 关键字。此时，基类的函数被隐藏（注意别与覆盖混淆）。

十五、四种强制类型转换

static_cast(expression)

用于数值类型之间的转换，也可以用于指针之间的转换，编译时已经确定好，效率高，但需要保证其安全性。

a) 指针要先转换成 void 才能继续往下转换。

b) 在基类和派生类之间进行转换（必须有继承关系的两个类）

子类对象可以转为基类对象(安全)，基类对象不能转为子类对象(可以转换，但不安全，dynamic_cast 可以实现安全的向下转换)。

static_cast 不能转换掉 expression 的 const、volatile、或者 __unaligned 属性

dynamic_cast < type-id> (expression)

只能用于对象的指针和引用之间的转换，需要虚函数。

dynamic_cast 会检查转换是否会返回一个被请求的有效的完整对象，否则返回 NULL;

Type-id 必须是类的指针、类的引用或者 void *，用于将基类的指针或引用安全地转换成派生类的指针或引用。

const_cast < type-id> (expression)

这个转换类型操纵传递对象的 const 属性，或者是设置或者是移除。

reinterpret_cast < type-id> (expression)

用在任意指针类型之间的转换；以及指针与足够大的整数类型之间的转换，从整数到指针，无视大小。

隐式类型转换

两种常用的实现隐式类类型转换的方式：

a、使用单参数的构造函数或 N 个参数中有 N-1 个是默认参数的构造函数。

b、使用 operator 目标类型（） const

避免隐式转换：前面加 explicit。