指针数组：指针的数组，数组内的每一个元素都是指针 如int \*p[4] 🡪等价于int [4] \*p首先是数组，然后是指针

数组指针：指向数组的指针 如int (\*p)[4]🡪等价于int [4] (\*p)首先是指针然后是int [4]类型

指针函数：程序中定义一个函数，编译时会给函数代码分配一段存储空间，存储空间首地址成为函数的地址，函数名表示就是这个地址，定义一个指针来存放这个地址，这个指针变量叫做函数指针，函数指针如int (\*p)(int , int)指针变量名两端的括号不能省略，省略的话就是声明了一个返回类型为int \* 的函数名为p的输入参数为(int,int)类型的函数，也就是函数指针。

指向函数的指针变量没有++和--运算也就是指针的函数没有++和--操作

但是函数的指针可以有加减操作

函数指针：首先是函数，只不过函数的返回值是地址值，并且对于接收的指针变量，必须是与函数同类型的指针变量，如int \* pfun (int,int)等价于int \* (pfun(int,int));

对数组名取地址代表整个数组，内存偏移量等于整个数组的大小(sizeof(数组名))

struct A{

long nu ;

char \* pc;

short int si;

char c;

short s[5];

}\*p; //\*p申明了一个指针指向这个结构体

int main(){

printf("p+200:=%p\n",p+0x200); //p+0x200是地址偏移 p + 0x200 如果struct A大小是N字节 实际上是移动 0x200 \* N 个字节

printf("p+200:=%p\n",(ulong)p+0x200); //先将p转换为数值，然后进行数值的加减

printf("p+200:=%p\n",(char \*)p+0x200); //先将p转换为char指针，再进行字符指针的偏移，由于char类型的大小为1字节 对char指针的加法都会按照1字节进行

return 0;

}

指针常量、常量指针、指向常量的常量指针区别？//提示可以去掉类型看本质

指针常量：int \* const p; //去掉类型后是 const p 指针说明p不能重新赋值 不能p=&一个新变量的地址 但是可以通过指针改变指向的变量的值

常量指针:const int\* p; 等价于 int const \*p; // const \*p 意味着\*p的值也就是指向的变量值固定，但是可以p=&一个新变量的地址，

指向常量的常量指针: const int \* const p;

指针与引用的异同？如何相互转换？

引用的本质是，对常量指针type \* const p的再次包装

char &rc 等价于char \* const pc

double &rd 等价于double \* const pd

相同：

1. 都是地址的概念，指针指向某一内存、内容是所指内存的地址；引用则是某块内存的别名
2. 从内存分配上看：两者都占内存，程序会为指针分配内存，一般为4字节；引用的本质是指针常量，指向的对象不变，对象的值可以改变。

区别：

1. 指针是实体的，引用是别名
2. 指针和引用的自增(++)运算符意义不同，指针是对内存地址自增，而引用是对值的自增
3. 引用使用时无需解引用(\*)，指针需要解引用；
4. 引用只能在声名时初始化，一经声明，不可变更；指针可以变
5. 引用不能为空，指针可以为空
6. Sizeof 引用得到时所指变量(对象)的大小，而sizeof 指针得到的是指针本身的大小，32位系统指针变量一般占4个字节

指针与引用的转换：

1. 指针转引用的方法与场景：将指针\*（解引用）就可以转换为对象（引用是对象的别名），可以用在函数的引用参数上，如形参是int &p,那么可以使用\*pp(pp为对象的指针)
2. 引用转指针：将引用类型的对象取地址就可以获得指针

野指针是什么？

1. 野指针是指向不可用内存的指针，指针创建时，指针不可能自动指向NULL，默认值是随机的，此时指针成为野指针
2. 当指针被free或者delete释放掉时，如果没有将指针设置为NULL，则会产生野指针，因为释放掉的仅仅是指针指向的内存，并没有将指针本身释放掉。
3. 第三个造成野指针的原因是指针操作超越了变量的作用范围

如何避免野指针？

1. 对指针进行初始化：首先指针初始化为NULL: 类型 指针=NULL；然后用malloc分配内存：类型 指针 = （类型）malloc(size of (类型))；最后用合法的可访问内存地址对指针初始化 ： 类型 变量 = 初始化；(已经开辟内存空间)，；类型 指针 = & 变量地址；
2. 指针用完后释放内存，再将指针赋值NULL : delete(指针)； 指针= NULL；

注意：

1. 检查malloc函数内存分配是否成功，如果成功返回分配的内存的首地址，如果分配不成功那么就会返回NULL,可以用if语句判断
2. 清空内存中的数据可以使用memset函数或者bzero函数

Memset函数三个参数分别是，首地址，值（值为0），和字节数

Bzero函数两个参数分别是置零空间的起始地址，要置零的数据字节个数

C++中的智能指针？

智能指针是一个类，用来存储指针（指向动态分配对象的指针）

C++程序设计中使用智能指针可以提高程序效率，能更好地管理堆内存。

智能指针的内存泄露，怎么解决？

1. **std::unique\_ptr**: 独占所有权，不允许另一个 std::unique\_ptr 共享其所有权
2. **std::shared\_ptr**: 共享所有权，可以有多个 std::shared\_ptr 指向同一个对象。当最后一个 std::shared\_ptr 被销毁时，内存才会被释放。
3. **std::weak\_ptr**: 辅助 std::shared\_ptr，不增加引用计数，避免循环引用。

**弱指针weak\_ptr的构造函数不会修改引用计数的值，从而不会对对象的内存进行管理，类似普通指针，但是不会指向引用计数的共享内存，但是可以监测所管理的对象是否被释放，从而避免非法访问。**

**预处理：**

**预处理标识符#error的目的是？：编译程序时，遇到#error就会生成一个编译错误提示消息，并停止编译，当程序的有些宏定义是在外部定义的，或者在系统头文件中指定的，如果你担心重复定义可以使用#ifdef 与#error结合，这样可以 输出检查宏定义是否已经被定义了**

**预处理定义常量#define 与 const修饰常量区别？**

1. **Define可以替代表达式，常数值，代码段容易出错，define只是单纯文本替换，生命周期止于编译期，不分配内存空间，存在于程序的代码段，而const常量处于程序的数据段，并且在堆栈中分配了空间**
2. **Const 有数据类型，define没有数据类型，const可以对类型安全检查，而define不行**
3. **IDE调试有的不支持调试define定义的常量**

**Typedef 与 define的区别？**

**Typedef进行类型检查，define不进行合理合法检查**

**功能不同，typedef用来定义类型的别名，define可以为类型、变量、常量、编译开关取别名。**

**作用域不同，typedef有自己的作用域有时是全局作用域，有时是函数作用域，define没有作用域的限制**

**对指针的操作不同**

**#include <xx.h>与#incldue “xx.h”有什么区别？**

对于include <xx.h>编译器会从标准库路径开始搜索xx.h对于系统调用比较快，而对于include “xx.h”会从用户工作路径搜索xx.h，然后再去系统路径搜索，对于自定义文件调用比较快

**头文件的作用有哪些？**

1. 通过头文件来调用库功能。
2. 头文件能加强类型安全检查。

**头文件中不可以定义静态变量，为什么？**

不行，在头文件中定义静态变量，会造成资源浪费的问题，也有可能引起程序错误。

**写一个“标准“宏MIN，这个宏输入两个参数并返回两个参数中较小的一个？**

#define MIN（A，B） ((A) < (B))? (A);(B))

**不使用流程控制语句，如何打印出1~1000的整数？**

宏定义多层嵌套(10P\*10L\*10I)，printf多次输出

例如：

#define B P,P,P…..一共十个P

#define P L….一共十个L

#define L I……一共十个I

#define I printf(“%3d”,i++)

Int main(){

Int I = 0;

B;

Return 0;

}

#define A(x) x;x;x;x;x….一共十个

Int main(){

Int n = 1;

A(A(A(printf(“%d”),n++))); //x就是printf(“%d”),n++，嵌套三层

Return 0 ;

}

**变量：**

**全局变量和静态变量的区别？**

1. 全局变量的作用域是整个程序，它可以在任何函数内被访问和修改。静态变量有两种主要类型：局部静态变量和全局静态变量，局部静态变量的作用域是定义它的函数内部。它只能在该函数内被访问，在局部函数结束后，局部静态变量的值也会被保留，直到程序结束；全局静态变量是在函数外部定义的变量，作用域是定义它的文件内部。它只能在该文件中被访问，其他文件无法访问，它在程序开始时被分配内存，并在程序结束时被释放。

全局变量能不能定义在可被多个.c文件包含的头文件中？为什么？

不可以，在不同的C文件中如果用static来声明同名全局变量，会造成每个包含该头文件的 .c 文件都会有一个独立的 static 全局变量实例，违反了全局变量的设计初衷，因为全局变量通常用于在多个文件之间共享状态和数据，static 关键字用于声明全局变量时，会将变量的作用域限制在定义它的文件内。这意味着每个包含该头文件的 .c 文件都会有自己独立的 static 全局变量实例，它们之间互不干扰。

局部变量能否与全局变量重名？

可以，但是同名的局部变量会屏蔽全局变量。

**函数：**

**请写个函数在main函数执行前先运行**

attribute可以设置函数属性、变量属性和类型属性

可以使用\_\_attribute\_\_((constructor))来设置在main函数之前执行

void before() \_\_attribute\_\_((constructor));

void after() \_\_attribute\_\_((deconstructor));在main函数之后执行

**为什么析构函数必须是虚函数？**

将可能会被继承的父类的析构函数设置为虚函数，可以保证当我们new一个子类，然后使用基类指针指向该子类对象，释放基类指针时可以释放子类的空间，防止内存泄漏。(简而言之就是方便管理子类，防止内存泄漏)

**为什么C++默认的析构函数不是虚函数？**

C++默认的析构函数不是虚函数是因为虚函数需要额外的虚函数表和虚函数指针，占用额外的内存。而对于不会被继承的类来说，析构函数是虚函数，就会浪费内存。因此只有当需要当作父类时，才会将析构函数设置为虚函数。

**C++中析构函数的作用？**

用来完成对象被删除前的一些清理工作

**静态函数和虚函数的区别？**

1. 静态函数：与类关联，与具体对象无关；虚函数：与具体对象相关，支持多态
2. 静态函数只能访问静态成员，虚函数可以访问所有的成员
3. 静态函数通过类名调用，虚函数通过对象指针或引用调用

**重载和覆盖(override)有什么区别？**

1. 重载：在同一个类中定义； 覆盖：在派生类中重新定义基类中的虚函数
2. 重载：参数列表必须不同； 覆盖：参数列表必须相同
3. 重载：返回类型可以不同，仅靠返回类型不同无法区分重载函数；覆盖：返回类型必须相同
4. 重载：实现编译时多态性； 覆盖：实现运行时多态
5. 重载：不需要特殊关键字； 覆盖：基类函数必须是虚函数，派生类中可以使用“override”关键字明确覆盖

**虚函数表具体是怎么样实现运行时多态的？**

原理：虚函数表是一个类的虚函数的地址表，每个对象在创建时，都会有一个指针指向这个类虚函数表，每个类的虚函数表，按照函数声明的顺序，会将函数地址存在虚函数表中，当子类对象重写父类的虚函数的时候，父类的虚函数表中对应位置的函数地址会被子类的虚函数地址覆盖。

作用：在用父类的指针调用子类对象的成员函数时，虚函数表会指明要调用的具体函数是哪个

**C语言是如何进行函数调用的？**

大多数CPU上的程序实现使用栈来支持函数调用操作，栈被用来传递函数参数、存储返回信息、临时保存寄存器原有的值以备恢复以及用来存储局部变量

函数调用操作所使用的栈部分叫做栈帧结构，每个函数调用都有属于自己的栈帧结构，栈帧结构有两个指针指定，帧指针(指向起始地址)，栈指针(指向栈顶地址)，函数对大多数数据的访问都是基于帧指针。

图示

描述已自动生成

栈指针和帧指针一般都有专门的寄存器，通常使用ebp寄存器存放帧指针，使用esp寄存器存放栈指针。

帧指针指向栈帧结构的头，存放着上一个栈帧的头部地址，栈指针指向栈顶

**函数：**

**以下代码代表什么意思 \*(a[1]+1)、\*(&a[1][1])、(\*(a+1))[1] ?**

\*(a[1]+1) : 如果a是二维数组 a[1]代表第二行首地址+1，向偏移一个单位(得到第二行第二列地址)，解引用取址得到a[1][1]

\*(&a[1][1]) : 因为[]的优先级高于&，所以先取a[1][1]的地址，再解引用，得到a[1][1]的值

(\*(a+1))[1] : a+1 相当于a[0] +1 得到a[1], []的优先级高于\*,所以最后是取a[1][1]的值

**数组下标可以是负数吗？**

可以，下标负数代表从当前地址向前寻址，因为数组下标只是给出了一个**与当前地址的偏移量，**只要根据这个偏移量能够定位到目标地址即可。

**位操作：**

如何求解整型数的二进制表示中1的个数？

#include <stdio.h>

int func(int x){

    int count1 = 0;

    while (x) //一直循环到x所有二进制的1都去掉

    {

        count1++;

        x= x&(x-1); //当一个数被减1时，最右边的那个值为1的bit将变为0，那个值右边的所有bit都会变成1

//将x对应的二进制中的最后一位1去掉

    }

    return count1;

}

int main() {

    printf("%d\n", func(1));

    return 0;

}

**如何求解二进制中0的个数？**

**#include <stdio.h>**

**#include <string.h>**

**//这个函数对输入值为负数会有问题**

**// int CountZeroBit(int num){**

**//     int count = 0;**

**//     while (num+1)**

**//     {**

**//         /\* code \*/**

**//         count++;**

**//         num |= (num+1);**

**//     }**

**//     return count;**

**// }**

**int CountZeroBit(int num) {**

**int count = 0;**

**int bits = sizeof(num) \* 8; //计算一共多少位bit**

**for (int i = 0; i < bits; i++) {**

**if ((num & (1 << i)) == 0) {**

**count++;**

**}**

**}**

**return count;**

**}**

**int main(int argc,char\*\* argv){**

**int value = 255;**

**int ret = CountZeroBit(value);**

**printf("value%d的二进制中0的个数为%d",value,ret);**

**return 0;**

**}**

**交换两个变量的值，不使用第三个变量(代码实现)**

一种：(注意浮点类型输出精度问题)

    p = p+q;

    q = p-q;

    p = p-q;

一种：（只能对int char类型的变量）

a = a^b;

b = a^b;

b = a^b;

或者a ^= b ^= a;

**给定一个整型变量a，写两段代码，第一个设置a的bit3，第二清除a的bit3。并且都要保持其他位不变。**

#include <stdio.h>

#include <string.h>

#define BIT(N) (0x1 << (N))

static int a = 10;

static int n = 5;

int set\_bit(int n){

    a |= BIT(n);

    return a;

}

int reset\_bit(int n){

    a &= ~BIT(n);

    return a;

}

int main(int argc, char \*\*argv){

    printf("%d\n", a);

    a = set\_bit(n);

    printf("%d\n", a);

    return 0;

}

**容器和算法：**

**Map和set有什么区别？分别又是怎么实现的？**

map和set都是C++的关联容器，其底层实现都是红黑树(RB-Tree)

由于map和set所开放的各种操作接口，RB-Tree也都提供了，所以几乎所有的map和set的操作行为，都只是转调RB-Tree的操作行为。

map与set的区别在于：

map中的元素是key-value(键值对)：**关键字起到索引**的作用，值则表示与索引相关联的数据；set与之相对的就是关键字的简单集合，**set中每个元素只包含一个关键字。**

set的迭代器是const，不允许修改元素的值；map允许修改value，但不允许修改key

其原因因为map和set是根据关键字排序来保证其有序性的，如果允许修改key的话，那么首先需要删除改键(key)，然后调节平衡，再插入修改后的键值，调节平衡，如此一来，就会破坏map和set的结构，导致迭代器(iterator)失效，不知道应该指向改变前的位置，还是指向改变后的位置。所以**STL中将set的迭代器设置成const，不允许修改迭代器的值**；而map则不允许修改key的值，允许修改value的值。

Map支持下标操作，set不支持下标操作。

Map可以用key做下标，map的下标操作符[]作为下标去执行查找，如果关键码不存在，则插入一个具有该关键码和mapped\_type类型的默认值的元素至map中，因此**下标运算符[]在map中应用中需要慎用**,const\_map不能用，只希望确定某一个关键字是否存在而不希望插入元素时也不该使用，mapped\_type类型没有默认值也不应该使用。尽量使用find。

**STL的allocator有什么作用？**

STL的分配器用于封装STL容器在内存管理上的底层细节，在C++中，其内存配置和释放如下：

new运算分为两个阶段：(1)调用::operator new分配内存 (2)调用对象构造函数构造对象内容

delete分为两个阶段：(1)调用对象析构函数 (2)调员工 ::operator delete释放内存

STL allocator将两个阶段操作区分开来：内存配置有alloc::allocate()负责，内存释放由alloc::deallcoate()负责，对象构造由::constructor()负责，对象析构由::destroy()负责

**STL迭代器如何删除元素？**

对于序列容器vector，deque来说，使用erase(iterator)后，后边的每个元素的迭代器都会失效，但是后边每个元素都会往前移动一个位置，但是erase会返回下一个有效迭代器；

对于关联容器map,set来说，使用erase(iterator)，当前元素的迭代器失效，但是其结构是红黑树，删除当前元素，不会影响下一个元素的迭代器，所以在调用erase之前，记录下一个元素的迭代器即可。

对于list来说，它使用了不连续分配的内存，并且它的erase方法也会返回下一个有效的iterator。

**STL中MAP数据如何存放的？**

红黑树，unordered map底层结构是哈希表

**STL中的map与unordered\_map有什么区别?**

map在底层使用了红黑树来实现，unordered\_map是C++标准中新加入的容器，它的底层是使用hash表的形式来完成映射的功能，map是按照operator<比较判断元素是否相同，以及比较元素的大小，然后选择合适的位置插入到树中。所以，如果对map进行遍历(中序遍历)的话，输出的结果是有序的。顺序就是按照operator<定义的大小排序。

而unordered\_map是计算元素的Hash值，根据Hash值判断元素是否相同。所以，对unordered\_map进行遍历，结果是无序的。

使用map时，需要为key定义operator< 。而unordered\_map的使用需要定义hash\_value函数并且重载operator==。对于内置类型，如string，这些不用操心，可以使用默认的。对于自定义的类型做key，就需要自己operator<或者hash\_value()

当不需要结果排好序，最好使用unordered\_map，插入删除和查询的效率要高于map

**vector和list的区别是什么？**

1. vector底层实现是数组；list是双向链表
2. vector支持随机访问，list不支持
3. vector是顺序内存，list不是
4. vector在中间节点进行插入删除会导致内存拷贝，list不会
5. vector一次性分配好内存，不够时才进行2倍扩容；list每次插入新节点都会进行内存申请
6. vector随机访问性能好，插入删除性能差；list随机访问性能差，插入性能好

**STL中迭代器有什么作用？有指针为何还要迭代器？**

1. 迭代器

迭代器模式又称游标模式，用于提供一种方法顺序访问一个聚合对象中各个元素，而又无需暴露对象的内部表示。或者这样说：迭代器模式是运用于聚合对象的一种模式，通过运用该模式，使得我们可以在不知道对象内部表示的情况下，按照一定顺序(利用迭代器提供的方法)访问聚合对象中的各个元素。

由于迭代器模式的以上特性：与聚合对象耦合，在一定程度上限制了它的广泛应用，一般仅用于底层聚合支持类，如STL的list、vector、stack等容器类及iostream\_iterator等扩展迭代器

1. 迭代器与指针的区别？

迭代器不是指针，是类模板，表现得像指针，重载了指针的一些操作，迭代器封装了指针，是一个’可遍历STL容器内全部或部分元素’的对象，本质是封装了原生指针，提供了比指针更高级的行为，相当于一种智能指针，可以根据不同类型的数据结构实现不同的++，--等操作。

**迭代器返回的是对象引用而不是对象的值**，所以**cout只能输出迭代器使用\*取值后的值而不能直接输出其自身**

1. 迭代器产生原因

迭代器类的访问方式就是把不同集合类的访问逻辑抽象出来，使得不用暴露集合的结构而达到循环集合的效果。

**epoll的原理是什么？**

调用顺序：epoll\_create -> epoll\_ctl -> epoll\_wait

首先创建一个epoll对象，然后使用epoll\_ctl对这个对象进行操作，把需要监控的描述添加进去，这些描述将会以epoll\_event结构体的形式组成一个红黑树，接着阻塞在epoll\_wait，进入大循环，当某个fd上有事件发生时，内核将会把其对应的结构体放入一个链表中，返回有事件发生的链表

**STL里resize和reserve的区别是什么？**

改变当前容器内含有元素的数量(size())，也就是vector v; v.resize(len); v的size变为len，如果原来v的size小于len，那么容器新增(len-size)个元素，元素的值默认为0.当v.push\_back(3);之后，3被放在v的末尾，即下标为len，此时容器size为len+1

改变当前容器的最大容量(capacity)，它不会生成元素，只是确定这个容器允许放入多少对象，如果reserve(len)的值大于当前的capacity()，那么会重新分配一块能存len个对象的空间，然后把之前v.size()个对象通过copy constructor复制过来，销毁之前的内存

**类和数据抽象：**

**C++中类成员的访问权限？**

C++通过public、protected、private三个关键字来控制成员变量和成员函数的访问权限，称为成员访问限定符。在类的内部(定义类的代码内部)，无论成员是什么权限，都是可以互相访问的。在类的外部，只能通过对象访问成员，并且通过对象只能访问public属性的成员，不能访问private、protected属性的成员。

**C++中struct和class的区别是什么？**

在C++中，可以用struct和class定义类，都可以继承。区别在于：struct的默认继承权限和默认访问权限是public，而class的默认继承权限和访问权限是private。此外，class还可以定义模板类形参，比如template

**C++类内可以定义引用数据成员吗？**

可以，必须通过成员函数初始化列表初始化

**面向对象与泛型编程是什么？**

1. 面向对象编程简称OOP，是一种程序设计思想。OOP把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数。
2. 面向过程是一组函数的顺序执行，再将大块函数，切割成小块函数来降低系统的复杂度
3. 泛型编程：让类型参数化，方便程序员编码

类型参数化：使得程序可以从逻辑功能上抽象，把被处理对象(数据)的类型作为参数传递

**什么是右值引用，跟左值又有什么区别？**

左值和右值的概念：

左值：能对表达式取地址、或具体名称的对象/变量。一般指表达式结束后依然存在的持久对象

右值：不能对表达式取地址，或匿名对象。一般指表达式结束后就不再存在的临时对象。

右值引用与左值引用的区别：

1. 左值可以寻址，而右值不可以。
2. 左值可以被赋值，右值不可以被赋值，可以用来给左值赋值
3. 左值可变，右值不可变（仅对基本类型有效，用户自定义类型的右值引用可以通过成员函数改变）

**析构函数可以为virtual型，构造函数则不能，为什么？**

构造函数不能声明为虚函数，虚构函数可以声明为虚函数，而且有时是必须声明为虚函数。不建议在析构函数和构造函数里面调用虚函数。

构造函数不能声明为虚函数的原因是：

虚函数的主要作用：在被派生类继承从而产生多态。派生类的构造函数中，编译器会加入构造函数基类的代码，如果基类的构造函数用到参数，则派生类在其构造函数的初始化列表中必须为基类给出参数，就是这个原因。虚函数的意思就是开启动态绑定，程序会根据对象的动态类型来选择要调用的方法。然而在构造函数运行的时候，这个对象的动态类型还不完整，没有办法确定函数类型，无法完成动态绑定。（动态绑定是根据对象的动态类型而不是函数名，在调用构造函数之前，这个对象根本不存在，怎么动态绑定？）

**C++中空类默认产生那些类成员函数？**

C++中空类默认会产生以下六个函数：默认构造函数、复制构造函数、析构函数、赋值运算符重载函数、取址运算法重载函数、const取地址运算符重载函数等。

class Empty{

public：

Empty();//缺省构造函数

Empty( const Empty&);//拷贝构造函数

~Empty();//析构函数

Empty& operator=( const Empty&);//赋值运算符

Empty\* operator&();//取地址运算符

const Empty\* operator&() const;//取址运算符 const

}

**面向对象：**

面向对象与面向过程有以下四个方面的不同：

1. 出发点不同

面向对象使用符合常规思维的方式来处理，将解决的动作映射到对象的接口上。而面向过程则强调的是过程的抽象化与模块化，是以过程为中心构造或处理客观世界问题。

1. 层次逻辑关系不同

面向对象以对象的集合类作为处理问题的单位，面向对象使用类的层次结构来体现类的之间的继承和发展。面向过程处理问题的基本单位是表达过程的模块，用模块的层次结构概括模块或模块间的关系与功能。

1. 数据处理方式与控制程序方式不同

面向对象将代码和数据封装成一个整体，原则上其他对象不能修改其数据，及对象的修改只能由自身的成员函数完成，控制程序方式是通过事件驱动。面向过程是直接通过来处理数据，处理完毕后即可显示处理的结果，在控制方式上是按照设计调用或返回程序。

1. 分析设计与编码转换方式

面向对象贯穿于软件生命周期的分析、设计及编码中，从分析到设计再到编码采用一致性的模型表示，实现是一种无缝连接。面向对象强调分析、设计及编码之间按规则进行转换，是一种有缝的连接。

**面向对象的基本特征？**

面向对象的编程四个特性：

* 抽象：过程抽象，数据抽象
* 继承：
* 封装：
* 多态：

**什么是深拷贝？什么是浅拷贝？**

系统提供默认的拷贝构造器，一经定义不再提供。但是系统提供的是浅拷贝。如果类中包含的数据元素全部在栈上，浅拷贝也可以满足，但是在堆上的数据，则会发生多次析构行为。

**浅拷贝**：只复制对象的基本属性，如果有指针或引用，则只复制其地址，新旧对象共享指向的资源

**深拷贝**：不仅复制对象的基本属性，还复制指针或引用所指向的实际数据，新旧对象各自拥有独立的资源。

**什么是友元？**

有成员只能在类的成员函数内部访问，如果想在别处访问私有成员，只能通过类提供的接口(成员函数)间接地进行。

友元提供了一种普通函数或者类成员函数访问另一个类中的私有或者保护成员的机制，有两种形式的友元：

* 友元函数：普通函数对一个访问某一个类中的私有或保护成员
* 友元类：类A中的成员函数访问类B中的私有或保护成员

**基类的构造函数/析构函数是否能被派生类继承？**

基类的构造函数，析构函数都不能被派生类继承

基类的构造函数不能被派生类继承，派生类中需要声明自己的构造函数。设计派生类的构造函数时，不仅要考虑派生类所增加的数据成员初始化，也要考虑基类的数据成员的初始化。声明构造函数时，只需要对本类中新增成员进行初始化，**对继承来的基类成员的初始化，需要调用基类构造函数完成**。

基类的析构函数也不能被派生类继承，派生类需要自行声明析构函数。声明方法与一般（无继承关系）类的析构函数相同，不需要显式调用，系统会自动隐式调用。析构函数的调用次序与构造函数相反。

**初始化列表和构造函数初始化的区别？**

构造函数初始化列表以一个冒号开始，接着是以逗号分隔的数据成员列表，每个数据成员后面跟一个放在括号中的初始格式。例如：

Example::Example() : Ival(0) ,dval(0.0) {} //Ival 和 dval是类的两个数据成员

ExampleL::Example()

{

ival = 0;

dval = 0.0；

}

上面的构造函数(使用初始化列表的构造函数)显示的初始化类的成员；而没使用初始化列表的构造函数是对类的成员赋值，并没有进行显示的初始化。

有时候必须用带有初始化列表的构造函数：

1. 成员类型是没有默认构造函数的类。若没有提供显示初始化式，则编译器隐式使用成员类型的默认构造函数，若没有默认构造函数，则编译器尝试使用默认构造函数将会失败。
2. const成员或者引用类型的员。因为const对象或引用类型只能初始化，不能对它们赋值。

**C++中有哪些情况只能用初始化列表，而不能用赋值？**

C++中赋值和初始化列表的原理不一样，赋值是删除原址，赋予新值，初始化列表开辟空间和初始化是同时完成的，直接给予一个值。

在C++中，赋值和初始化列表的使用情况也不一样，只能用初始化列表而不能用赋值的情况：

* 当class中含有const、（type）&成员变量时只能初始化，不能对他们进行赋值。常量不能被赋值，只能被初始化，所以必须在初始化列表中完成，C++的引用也一定要初始化，所以必须在初始化列表中完成。
* 派生类在构造函数中要对自身成员初始化，也要对继承过来的基类成员进行初始化当基类没有默认构造函数的时候，通过在派生类的构造函数初始化列表中调用基类的构造函数实现。
* 如果成员类型是没有默认构造函数的类，也只能使用初始化列表。若没有提供显式初始化时，则编译器隐式使用成员类型的默认构造函数，此时编译器尝试使用默认构造函数将会失败。

**类的成员变量的初始化顺序是什么？**

* 成员变量在使用初始化列表初始化时，与构造函数中初始化成员列表的顺序无关，只与定义成员变量的顺序有关。
* 如果不使用初始化列表初始化，在构造函数内初始化，此时与成员在构造函数中的位置有关
* 注意：类成员在定义时，是不能初始化的
* 注意：类中const成员常量必须在构造函数初始化列表中初始化
* 注意：类中static成员变量，必须在类外初始化
* 静态变量进行初始化顺序是基类的静态变量先初始化，然后是它的派生类。直到所有的静态变量都被初始化。注意：全局变量和静态变量的初始化时不分次序的。

**当一个类为另一个类的成员变量时，如何对其进行初始化？**

例子：

#include <iostream>

using namespace std;

class **ABC**{

public:

**ABC**(int x,int y, int z);//显式带参的构造函数

private:

int a;

int b;

int c;

};

class **Myclass**{

public:

**Myclass**():abc(12,2,4) //初始化列表

{

}

private:

ABC abc;//对象作为私有成员

};

int **main**()

{

Myclass mc;

cout << "555" << endl;

return 0;

}

**C++能设计实现一个不能被继承的类吗？**

在Java中定义了关键字final，被final修饰的类不能继承，但是在C++中没有final这个关键字，要实现需要设计。

首先是在C++中，子类的构造函数会自动调用父类的构造函数。同样子类的析构函数也会自动调用父类的析构函数。要想一个类不能被继承，我们只要把他的构造函数和析构函数都定义为私有函数，那么当一个类试图继承它时，必然由于试图调用构造函数和析构函数导致编译错误。

当这个类的构造函数和析构函数都是私有函数，如何得到类的实例？

可以通过定义静态来创建和释放类的实例

**构造函数没有返回值，如何得知对象是否构造成功？**

这里的“构造”不单指分配对象本身的内存，而是指在建立对象时做的初始化操作(如打开文件、连接数据库等)

因为构造函数没有返回值，所以通知对象的构造失败的唯一方法就是在构造函数中抛出异常。构造函数中抛出异常将导致对象析构函数不被执行，当对象发生部分构造时，已经构造完毕的子对象会逆序地被析构。

**Public继承、protected继承、private继承的区别？**

1. 公有继承

对于子类对象，采用公有继承时，基类成员对子类的可见性与一般类成员对对象的可见性相同，公有成员可见，其他成员不可见。

对于子类而言，基类的公有成员和保护成员可见；基类的公有成员和保护成员作为派生类的成员时，它都维持原有的可见性（基类的public成员在子类中依然是public，其他的相同）；基类的私有成员不可见，基类的私有成员依然是私有的，子类不可访问。

1. 保护继承

保护继承的特点：基类的共有成员和保护成员都成为派生类的保护成员，并且只能被基类的派生类成员函数或友元访问。基类的私有成员仍然是私有的，由此看出基类的所有成员对子类的对象是不可见的。

1. 私有继承

私有继承的特点：基类的公有成员和保护成员都作为派生类的私有成员，并且不能被这个派生类的子类所访问。

**C++提供默认参数的函数吗？**

C++可以给函数定义默认参数值。在函数调用时没有指定与形参相对应的实参时，就自动使用默认参数。

默认参数的语法与使用：

(1)在函数声明或定义时，直接对参数赋值，这就是默认参数

(2)在函数调用时，省略部分或全部参数。这是可以使用默认参数来替代

通过调用函数时，腰围函数的每个参数给定指定实参，例如

void **delay**(int loops = 1000);

void **delay**(int loops){

if(loops == 0){

return;

}

for(int i=0;i<loops;i++){

;

}

}

默认参数在函数声明中提供，当又有声明又有定义时，定义中不允许默认参数。如果函数只有定义，则默认参数才可出现在函数定义中。例如：

void **point**(int=3 ,int=4);//声明中给出默认值

void **point**(int x,int y){//定义中不允许再给出默认值

cout<< x <<endl;

cout<< y << endl;

}

**如果一组重载函数（可能带有默认参数）都允许相同实参个数的调用**，将会引起调用的二义性，例如：

void **func**(int);//重载函数之一

void **func**(int, int =4);//重载函数之二，带有默认参数

void func(int=3, int=4);//重载函数之三，带有默认参数

**虚函数：**

**什么是虚函数？**

指向基类的指针在**操作它的多态对象时**，可以根据**指向的不同类对象调用其相应的函数，这个函数就是虚函数**。

虚函数的作用：在基类定义了虚函数之后，可以在派生类中对虚函数重新定义，并且可以通过基类指针或引用，在程序的运行阶段动态选择调用基类和不同派生类中的同名函数。（如果在派生类中没有对虚函数重新定义，则它继承其基类的虚函数），下面是一个实例：

#include <iostream>

#include "string.h"

using namespace std;

class Base{

public:

virtual void Print() //父类虚函数

{

printf("this is class base\n");

}

};

class son1: public Base{

virtual void Print(){

printf("this is class son1\n");

}

};

int main()

{

Base base;

son1 son;

son1 son2;

return 0;

}

使用虚函数注意：

* 只需要在声明函数的类中使用关键字virtual将函数声明为虚函数，而定义函数时不需要使用关键字virtual
* 当将基类中的某一成员声明为虚函数后，派生类中的同名函数自动成为虚函数
* 非类中的成员函数不能定义为虚函数，全局函数以及类的成员函数中静态成员函数和构造函数也不能定义为虚函数，但可以将析构函数定义为虚函数。
* 基类的析构函数应该定义为虚函数，否则会造成内存泄漏，基类析构函数未声明virtual，基类指针指向派生类时，delete指针不调用派生类析构函数。有virtual，则先调用派生类析构函数析构在调用基类析构函数。

**C++如何实现多态？**

C++中通过虚函数实现多态。虚函数的本质就是通过基类指针访问派生类定义的函数。每个含有虚函数的类，其实例对象内部都有一个虚函数表指针。该虚函数表指针被初始化为本类的虚函数表的内存地址。所以，在程序中吗，不管对象类型如何转换，该对象内部的虚函数表指针都是固定的，这样才能实现动态地对对象函数调用，这就是C++多态性的原理。

**C++纯虚函数指的是什么？**

纯虚函数是一种特殊的虚函数，格式一般如下：

class <类名>

{

virtual () //函数返回值类型 虚函数名(形参表) = 0

. …

};

class <类名>

由于在很多情况下，基类中不能对虚函数给出有意义的实现，只能把函数的实现留给派生类。例如，动物可以作为一个基类派生出老虎、孔雀等子类，但是动物本身生成对象不合情理，此时就可以将动物类中的函数定义成纯虚函数，如果基类中有虚函数，那么在子类中必须实现这个虚函数，否则子类将无法被实例化，也无法实现多态。

含有纯虚函数的类称为抽象类，抽象类不能生成对象。纯虚函数永远不会被调用，它们主要用来统一管理子类对象。

**什么函数不能声明成虚函数？**

常见的不能声明为虚函数的有：普通函数(非成员函数)；静态成员函数；内联成员函数；构造函数；友元函数。

1. 为什么C++不支持普通函数为虚函数？

普通函数(非成员函数)只能被overload，不能被override，声明为虚函数也没有意义

1. 为什么C++不支持构造函数为虚函数？

主要从语义上考虑，所以不支持。因为构造函数本来就是为了明确初始化对象成员才产生的，然而virtual functions主要是为了再不完全了解细节的情况下也能正确处理对象。另外，virtual函数是在不同类型的对象产生不同的动作，现在对象还没有产生，如何使用virtual函数完成你想要的动作

1. 为什么C++不支持内联成员函数为虚函数？

内敛函数为了在代码中直接展开，减少函数调用花费的代价，虚函数是为了在继承后对象能够准确的执行自己的动作，这是不可能统一的。（inline函数在编译时被展开，虚函数在运行时才能动态地绑定函数）

4.为什么C++不支持静态成员函数为虚函数？

静态成员函数对于每一个类来说只有一份代码，所有地对象都共享这一份代码，他也没有动态绑定的必要性

5.为什么C++不支持友元函数为虚函数？

因为C++不支持友元地继承，对于没有继承特性的函数没有虚函数的说法

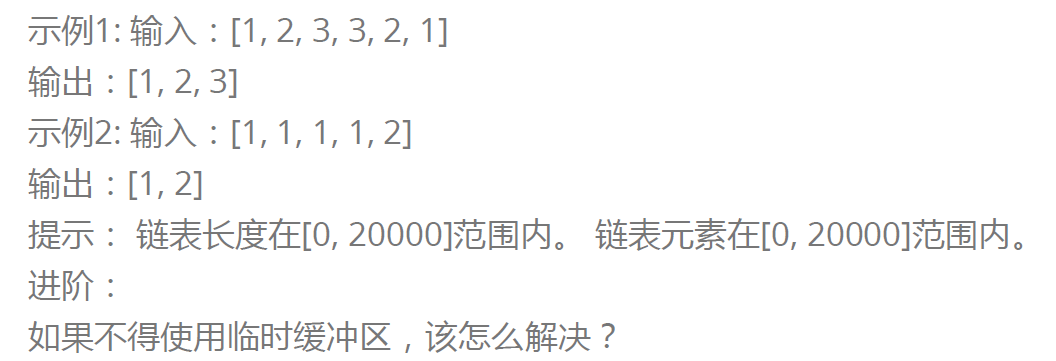
**C++中如何阻止一个类被实例化？**

C++中可以通过使用抽象类，或者将构造函数声明为private阻止一个类被实例化。抽象类之所以不能被实例化，是因为抽象类不能代表一类具体的实物，它是对多种具有相似性的具体事物的的共同特征的一种抽象。例如，动物作为一个基类可以派生出老虎、孔雀等子类，但是由动物本生生成对象不合情理。

**数据结构与算法：**

**链表：代码题**

**删除单链表的重复节点：移除未排序链表中的重复节点。保留最开始出现的节点**



思路一：定义两个指针current和p来逐个遍历链表，current元素依次和p比较，直到p为NULL，current向后移动一个

思路二：利用标记数组，标记当前值是否出现过

**如何找出链表中的倒数第k个元素？**

文本

描述已自动生成

思路：快慢指针，先让快指针先走k步，然后两个指针同步走，当快指针走到头时，慢指针就是链表倒数第k个字节

**如何找出链表中的中间节点？**

文本

描述已自动生成

思路：双指针问题，定义一个快指针，一个慢指针。快指针一次走两步，慢指针一次走一步。当快指针走到结尾时，慢指针指向的就是中间节点。（两个中间节点的时候，返回第二个中间节点）：快指针前进条件是当前快指针和当前快指针的下一个节点都是非空 （如果两个中间节点，返回第一个中间节点）：快指针前进条件是：当前快指针的下一个节点和当前快指针的下下一个节点都是非空

**反转链表？**

文本

描述已自动生成

思路：反转链表实际就是重新指向结构体中的next指针，我们需要修改下一个节点的next

**环形链表？**

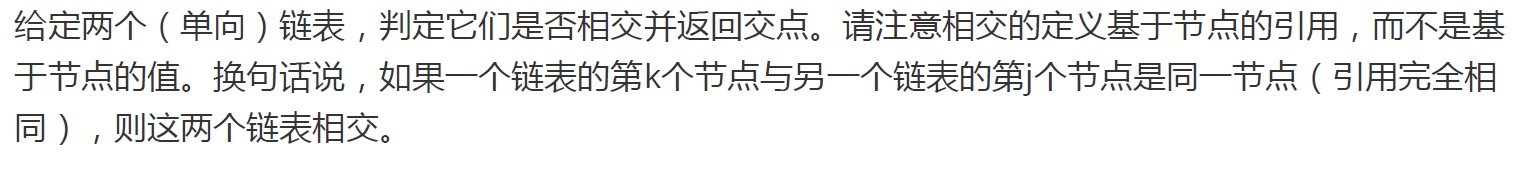
图示

描述已自动生成

图示

描述已自动生成

**单链表相交，如何求交点？**



**判断是否是回文链表？**

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

**用普通算法实现两个有序链表的合并？**

图形用户界面, 文本, 应用程序

描述已自动生成

**树：代码题**

**二叉树先序遍历(递归和非递归)**

**先序遍历规则：**核心思想：

* 访问根节点
* 访问当前节点的左子树
* 若当前节点无左子树，则访问当前节点的右子树；即考察到一个节点，即刻输出该节点的值，并继续遍历其左右子树。(根左右)

形状

描述已自动生成

先序访问这颗二叉树的详细过程为：

* 访问该二叉树的根节点，找到1
* 访问节点1的左子树，找到2
* 访问节点2的左子树，找到4
* 由于访问节点4的左子树失败(4没有左子树)，且也没有右子树，因此以节点4为根节点的子树遍历完成。但节点2还没有遍历其右子树，因此现在开始遍历，即访问节点5
* 由于节点5也没有左右子树，因此节点5也遍历完成，以2为根节点的子树也遍历完成。回到节点1，由于节点1有右子树，所以访问节点3
* 接下来同理，先访问到6，后回到3，访问节点7
* 根节点为3的遍历访问完毕，回到节点1，节点1左右子树全部遍历完成，因此整个二叉树遍历完成
* 因此先序遍历得到序列为：1，2，4，5，3，6，7

用代码表示：

**二叉树中序遍历(递归和非递归)**

中序遍历规则：二叉树中序遍历的实现思想：

1. 访问当前节点的左子树
2. 访问根节点
3. 访问当前节点的右子树，即考察到一个节点后，将其暂存，遍历完左子树后，再输出该节点的值，然后遍历右子树。（左根右）

形状

描述已自动生成

中序访问这颗二叉树的详细过程为：

* 访问二叉树的根节点，找到1，但是不输出
* 遍历节点1的左子树，找到节点2，但是不输出
* 遍历节点2的左子树，找到节点4，但是先不输出
* 由于节点4没有左子树，所以左子树找不到，因此回到根节点也就是节点4，并遍历节点4的右子树，也没有右子树所以输出4
* 回到节点4的根节点，也就是节点2，输出2，遍历右子树5
* 右子树5没有左子树，也没有右子树，输出5
* 节点1的左子树遍历完毕，回到根节点1，输出1，找到右子树节点3
* 遍历节点3的左子树，也就是节点6
* 节点6没有左右子树，输出6，回到节点3，输出节点3，再遍历节点3的右子树，输出节点7
* 中序二叉树序列输出：4，2，5，1，6，3，7

用代码表示：

**二叉树后序遍历(递归和非递归)**

后序遍历规则

二叉树后序遍历的实现思想是：

1. 访问左子树
2. 访问右子树
3. 完成该节点的左右子树的访问后，在访问该节点，即考察一个节点之后，将其暂存，遍历完左右子树后，在输出该节点的值(左右根)

**后续遍历举例**

形状

描述已自动生成

如上图中，对此二叉树进行后序遍历的操作过程为：

从根节点1开始，遍历该节点的左子树(以节点2为根节点)

1. 遍历节点2的左子树(以节点4为根节点)
2. 由于节点4既没有左子树，又没有右子树，此时访问该节点中的元素4，输出4，回退到节点2，遍历节点2的右子树(以5为根节点)
3. 由于节点5无左右子树，访问根节点5，输出5，因此节点2的左右子树遍历完毕，访问根节点2，输出2
4. 回退到节点1，遍历节点1的右子树（以节点3为根节点）
5. 遍历节点3的左子树6（以节点6为根节点）
6. 由于节点6没有左右子树，所以访问并输出根节点6
7. 回退到节点3，遍历节点3的右子树，（以节点7为根节点）
8. 由于节点7无左右子树，因此访问节点7，输出7，并且节点3的左右子树也遍历完成，访问节点3，输出3，回退到节点1
9. 由于节点的左右子树遍历完成，所以可以访问节点1，输出1
10. 所以二叉树后序遍历输出序列为：4，5，2，6，7，3，1

代码表示：

层次遍历

层次遍历规则：

按照二叉树中的层次从左到右依次遍历每层中的节点。通过使用队列的数据结构，从树的根节点开始，依次将其左子树和右子树入队。而后每次队列中一个节点出队，都将其左子树和右子树入队，直到树中所有节点都出队，出队节点的先后顺序就是层次遍历的最终结果。

**层次遍历举例**

图示, 形状

描述已自动生成

例如，层次遍历如上图中的二叉树：

* 根节点1入队
* 根节点1出队，出队的同时，将左子树2和右子树3分别入队
* 队头节点2出队，出队的同时，将节点2的左子树4和右子树5依次入队
* 队头节点3出队，出队的同时，将节点3的左子树6和右子树7依次入队
* 所以层次遍历输出序列：1，2，3，4，5，6，7

代码表示：

**求二叉树的深度？**

**树的深度：从根节点到叶节点依次经过的节点（含根、叶节点）形成树的一条路径，最长路径的长度为树的深度**

**思路：**

**如果二叉树为空，深度为0**

**如果二叉树只有根节点，深度为1**

**如果二叉树的根节点只有左子树，深度为左子树的深度＋1**

**如果二叉树的根节点只有右子树，深度为右子树的深度+1**

**如果二叉树的根节点既有左子树又有右子树，深度为左右子树深度的最大者再+1**

**如何判断二叉树是否相等？**

二叉树是否相等地判断条件：如果两个树在结构上相同，并且节点具有相同地值，则认为它们是相同的。

思路：采用前序遍历方法遍历二叉树，并把遍历的节点值存储在数组中，并且也把每个节点的空子树也储存下来（用0表示某个节点的空子树）

**如何判断二叉树是否是平衡二叉树？**

平衡二叉树：如果某一个二叉树中任意节点的左右子树的深度相差不超过1，那么就是一棵平衡二叉树。

思路：

1. 理解平衡树是什么？平衡树是一棵树的左右子树的高度差距小于等于1
2. 使用递归的算法可以求出二叉树的高度
3. 先求根的左右高度差距是不是满足小于等于1，然后递归查询，用与（&&）的方法

**数组**

**最大子序和：代码题**

给定一个整数数组nums，找到一个具有最大和的连续子数组(子数组最少包含一个元素)，返回其最大和

图形用户界面, 文本

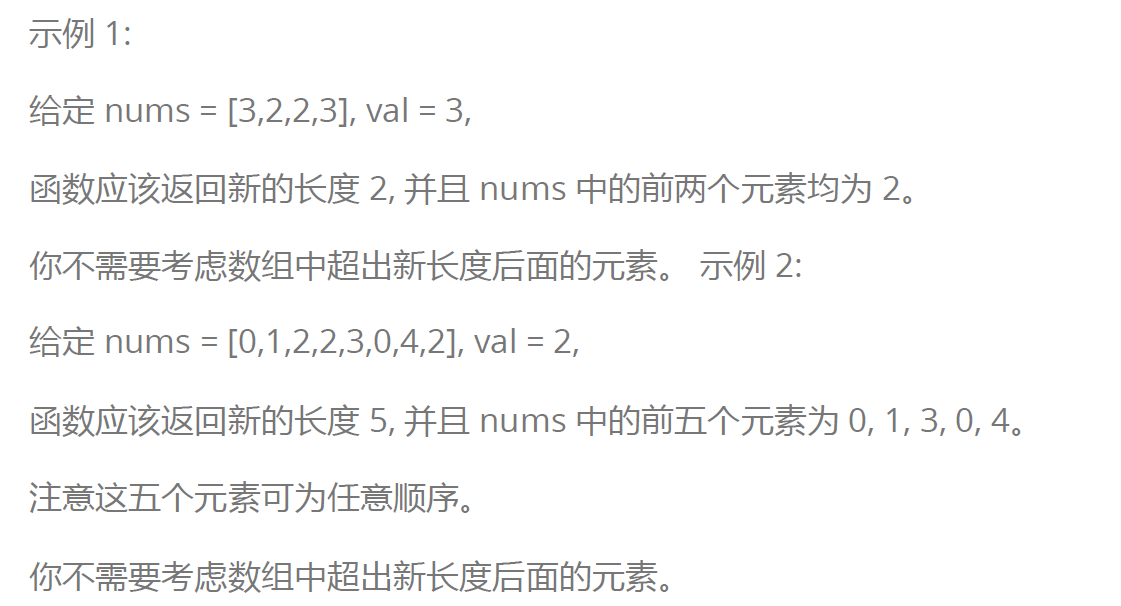
描述已自动生成

**原地移除元素：代码题**

给你一个数组nums和一个值val，你需要原地移除所有数值等于val的元素，并返回移除后数组的新长度。

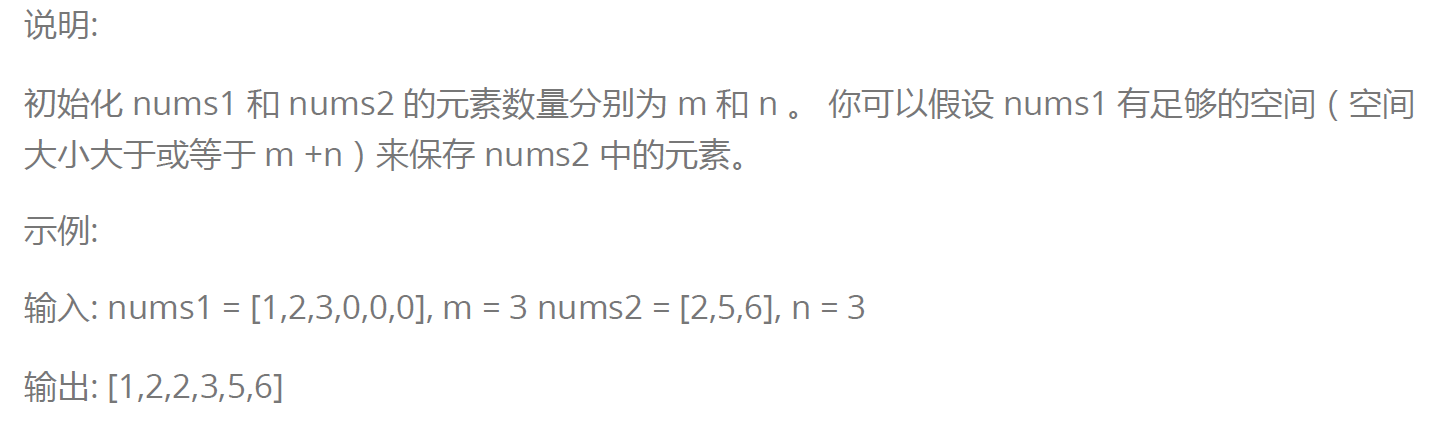
不要使用额外的数组空间，你必须仅使用O(1)额外空间并原地修改输入数组

元素的顺序可以改变，不需要考虑数组中超出新长度后面的元素。



**合并两个有序数组：代码题**

给你两个有序整数数组nums1和nums2，请将nums2合并nums1中，使nums1成为一个有序数组。



**查找常用字符：代码题**

给定仅有小写字母组成的字符串数组A，返回列表中的每个字符串中都显示的全部字符(包括重复字符)组成的列表，例如，如果一个字符在每一个字符串中出现3次，但不是4次，则需要在最终答案中包含该字符3次，可以按任意顺序返回答案。

图形用户界面, 文本, 应用程序

描述已自动生成

**寻找数组的中心索引：代码题**

图片包含 文本

描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

**数组中数字出现的次数**

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

**数组中数字出现的次数||**

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

**数组中缺失的元素**

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

**按奇偶排序数组**

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

**数组是否存在重复元素**

图形用户界面, 文本, 应用程序

描述已自动生成

**有序数组出现次数超过25%的元素**

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

**有效的山脉数组**

文本

描述已自动生成

图示

描述已自动生成

**最长连续递增序列**

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

**字符串：代码题**

**有效的括号**

文本, 信件

描述已自动生成

文本, 信件

描述已自动生成

**排序算法及其改进方法**

**快速排序：**

我们从数组中选择一个元素，我们把这个元素成为中轴元素，然后把数组中所有小于中轴元素的元素放在其左边，所有大于或等于中轴元素的元素放在其右边，显然，此时中轴元素所处的位置的是有序的，也就是说，我们无需在移动中轴元素。

从中轴元素那里开始把大的数组切割成两个小的数组（两个数组都不包含中轴元素），接着我们通过递归的方式，让中轴元素左边的数组和右边的数组也重复同样的操作，直到数组的大小为1，此时每个元素都处于有序的位置。

最好时间复杂度：O(NlogN)

最坏时间复杂度：O(N2)

最坏时间复杂度：O(NlgN)

空间复杂度：O(logN)

稳定排序：否

原地排序：是

**冒泡排序：**

把第一个元素与第二个元素比较，如果第一个比第二个大，则交换他们的位置。接着继续比较第二个与第三个元素，如果第二个比第三个大，则交换他们的位置。

我们对每一个对相邻元素做同样的工作，从开始第一对到结尾的最后一对，这样一趟比较交换下来之后，排在最右的元素就会是最大的数。除去最大的元素，我们对剩余的元素做同样的工作，如此重复下去，直到排序完成。

最好时间复杂度：O(N)

最坏时间复杂度：O(N2)

平均时间复杂度：O(N2)

空间复杂度：O(1)

稳定排序：是

原地排序：是

**堆排序：**

将待排序的序列构造成一个大顶堆。此时，整个序列的最大值就是堆顶的根节点。将它移走(其实就是将其与堆数组的末尾元素交换，此时末尾元素就是最大值)，然后将剩余的n-1序列重新构造成一个堆，这样就会得到n个元素中的次大值，如此返回执行，便得到一个有序序列

最好时间复杂度：O(NlogN)

最坏时间复杂度：O(NlogN)

平均时间复杂度：O(NlogN)

空间复杂度：O(1)

稳定排序：否

原地排序：是

**插入排序：**

每一步将一个待排序的元素，插入到前面已经排好序的有序序列中去，直到插完所有元素为止。

最好时间复杂度：O(N)

最坏时间复杂度：O(N2)

平均时间复杂度：O(N2)

空间复杂度：O(1)

稳定排序：是

原地排序：是

**选择排序：**

首先找到数组中最小的那个元素，其次，将它和数组的第一个元素交换位置(如果第一个元素就是最小元素，那么它就和自己交换)。其次，在剩下的元素中找到最小的元素，将它与数组的第二个元素交换位置。如此往复，直到将整个数组排序。

最好时间复杂度：O(N2)

最坏时间复杂度：O(N2)

平均时间复杂度：O(N2)

空间复杂度：O(1)

稳定排序：否

原地排序：是

**归并排序：**

实际上就是数据序列化为越来越小的半子表，再对半子表排序，最后再用递归方法将排好序的半子表合并成越来越大的有序序列。

最好时间复杂度：O(NlogN)

最坏时间复杂度：O(NlogN)

平均时间复杂度：O(NlogN)

空间复杂度：O(N)

稳定排序：是

原地排序：是

**希尔排序：**

也称递减增量排序算法，是插入排序的一种更高效的改进版本，但希尔排序是非稳定排序算法。

希尔排序是基于插入排序的以下两点性质而提出改进方法的：

* 插入排序在对几乎已经排好序的数据操作时，效率高，即可以达到线性排序的效率
* 但插入排序一般来说是低效的，因为插入排序每次只能将数据移动一位

希尔排序的基本思想：先将整个待排序的记录序列分割成若干子序列分别进行直接插入排序，待整个序列中的记录“基本有序时“，再对全体记录进行依次直接插入排序

最好时间复杂度：O()

最坏时间复杂度：O()

平均时间复杂度：O()

空间复杂度：O(1)

稳定排序：否

原地排序：是

**ARM体系与架构：**

**硬件基础：**

**NAND Flash和Nor Flash异同？**

|  |  |  |
| --- | --- | --- |
| 类别 | Nor | nand |
| 读 | 快 像访问sram一样，可以随机访问任意地址的数据 | 快，有严格的时序要求，需要通过一个函数才能读取数据，先发送读命令->发送地址->判断nandflash是否就绪->读取一页数据。读命令，发送地址，判断状态，读数据都是通过操作寄存器实现的，如数据寄存器NFDATA |
| 写 | 慢，写之前需要擦除，因为写只能使1->0，擦除可以使0->1 | 快，写之前需要擦除，因为写只能使1->0，擦除可以使0->1 |
| 擦除 | 非常慢(5s) | 快(3ms) |
| Xip（处理器可以直接访问外部储存器） | 代码可以直接在在NoR FLASH上运行 | NO |
| 可靠性 | 比较高，位反转的比例小于NAND FLASH的10% | 比较低，位反转比较常见，必须有校验措施 |
| 接口 | 与RAM接口相同，地址和数据总线分开 | I/O接口 |
| 可擦除次数 | 10000~100000 | 100000~1000000 |
| 容量 | 小1MB~32MB | 大，16MB~512MB |
| 主要用途 | 常用于保存代码和关键数据 | 用于保存数据 |
| 价格 | 高 | 低 |

两个flash擦除单元都是以块为单位

注意：nand flash和nor flash的0地址是不冲突的，norflash占用bank地址，而nandflash不占用bank地址，它的0地址是内部的

**CPU，MPU，MCU，SOC，SOPC联系与差别？**

1. CPU是一台计算机的运算核心和控制核心。CPU由运算器、控制器和寄存器及实现它们之间联系的数据、控制和状态的总线构成。cpu运行原理可分为四个阶段：提取，解码，执行，写回。CPU从存储器或高速缓冲存储器中取出指令，放入指令寄存器，并对指令译码，并执行指令。
2. MPU，微处理器，增强型cpu，但是不是为任何已有的特定计算目的而设计的芯片，这种芯片往往是个人计算机和高端工作站的核心cpu。最常见的微处理器是摩托罗拉的68k系列和因特尔的x86系列
3. MCU。微控制器，是指随着大规模集成电路的出现及其发展，将计算机的cpu，ram，rom，定时计数器，和多种i/o接口集成在一片芯片上，形成芯片级的芯片，比如51，avr这些芯片除了cpu之外还有ram，rom可以直接加简单的外围器件(电阻，电容)就可以运行代码了，而MPU如x86，arm这些就不能直接放代码了，得加**RAM，ROM**

MCU MPU最主要的区别就在于能否直接运行代码。MCU有内部的RAM ROM，而MPU是增强版的CPU，需要添加外部RAM ROM才能运行代码

1. SOC指的是片上系统，MCU只是芯片级的芯片，而**SOC是系统级的芯片**，它既像MCU(51，avr)那样有内置RAM，ROM同时又像MPU(arm)那样强大，不单单是放简单的代码，可以放系统级的代码，也就是**可以运行操作系统**(就认为是MCU集成化与MPU强处理力二合一了)
2. SOPC，可编程片上系统，上面4点的硬件配置是固化，就是说51单片机就是51单片机，不能变成avr，而avr就是avr不能变成51单片机，他们的硬件是一次性掩膜成型的，能改的就是软件配置，也就是说原本流水灯，改成数码管显示，而SOPC则是**硬件配置，软件配置都可以修改，**软件修改与上面是一样的，至于硬件是可以根据自己构建的，这颗芯片我们叫“白片”，什么芯片都不是，把硬件配置信息下载进去了，他就是相应的芯片了，可以让它变成51，也可以是avr，甚至arm，同时SOPC是在SOC基础上来的，所以也是系统级的芯片，所以记得把他变成arm时还得加外围ROM，RAM之类，不然就是MPU了，不能直接跑代码

**什么是交叉编译？**

**在一种计算机环境中运行**的编译程序，能编译出在**另一种环境下**运行的代码，我们就称这种编译器**支持交叉编译。**这个**编译过程**叫交叉编译。简单地说，**就是在一个平台上生成另一个平台上的可执行代码。**

注意所谓平台，实际上包含两个概念：体系架构(x86，arm，)、操作系统。同一个架构可以运行不同的操作系统；同样，同一个操作系统也可以在不同的架构上运行。举例来说，我们常说的x86 linux平台实际上是intel x86体系架构和linux for x86操作系统的统称。

**为什么需要交叉编译？**

有时是因为目的平台上不允许或不能够安装我们所需要的编译器，而我们又需要这个编译器的某些特征；有时是因为目的平台上的资源贫乏，无法运行我们所需要的编译器；有时是因为目的平台还没有建立，连操作系统都没有，根本谈不上运行什么编译器。

**描述一下嵌入式基于ROM的运行方式和基于RAM的运行方式有什么区别？：**

**基于RAM：**

* 需要把硬盘和其他介质的代码先加载到RAM中，加载过程中一般有重定位的操作
* 速度比基于ROM的快，可用RAM比基于ROM的少，因为所有的代码，数据都必须存放在RAM中、

**基于ROM：**

* 速度较基于RAM的慢，因为会有一个把变量，部分代码等从存储器(硬盘，flash)搬移到RAM的过程
* 低内存占用，可用内存资源比基于RAM多

**ARM处理器：**

**什么是哈佛结构和冯诺伊曼结构？**

冯诺依曼结构：采用指令和数据**统一编址**，使用**同条总线传输，CPU读取指令和数据的操作无法重叠**

哈佛结构：采用指令和数据独立编址，使用两条独立的总线传输，CPU读取指令和数据操作可以重叠

利弊：冯诺依曼结构主要用于通用计算机领域，需要对存储器中的代码和数据频繁的进行修改，统一编址有利于节约资源

哈佛结构主要用于嵌入式计算机，程序固化在硬件中，有较高的可靠性、运算速度和较大的吞吐

**什么是ARM流水线技术？**

流水线技术通过**多个功能部件并行工作**来缩短程序执行时间，提高处理器核的效率和吞吐率，从而成为微处理器设计中最为重要的技术之一。ARM7处理器核使用了典型三级流水线的冯诺依曼结构，ARM9系列则采用了基于五级流水线的哈佛结构。通过增加流水线级数简化了流水线各级的逻辑，进一步提高了处理器的性能。

PC代表程序计数器，流水线使用三个阶段，因此指令分为三个阶段执行：1.取址（从存储器装载一条指令）2.译码（识别将要被执行的指令）3.执行（处理指令并将结果写回寄存器）。而PC总是指向“正在取址”的指令，而不是指向“正在执行指令”或正在“译码”的指令。PC始终指向该指令地址加8字节的地址：即PC值=当前程序执行位置+8

ARM指令是三级流水线，取址，译址，执行，三步同时执行，现在**PC指向的是正在取址的地址（也就是下一条指令）**，那么**cpu正在译址**的**指令地址是PC-4**（假设在ARM状态下，一个指令占4个字节），**cpu正在执行的指令地址是PC-8，**也就是说PC所指向的地址和现在所执行的指令地址相差8

当突然发生中断的时候，保存的是PC的地址（PC-8+4=PC-4 cpu下一条指令的地址）

图形用户界面, 文本, 应用程序

描述已自动生成

**ARM有几种工作模式？**

1. 用户模式(USR)

用户模式是用户程序的工作模式，**它运行在操作系统的用户态**，它没有权限去操作其他硬件资源，只能执行处理自己的数据，也不能切换到其他模式下，要想访问硬件资源或切换到其他模式只能通过软中断或产生异常。

1. 系统模式(SYS)

系统模式是特权模式，不受用户模式的限制。用户模式和系统模式共用一套寄存器，操作系统在该模式下可以方便的访问用户模式的寄存器，而且操作系统的一些特权任务可以使用这个模式访问一些受控资源

1. 一般中断模式(IRQ)
2. 快速中断模式(FIQ)
3. 管理模式(SVC)
4. 终止模式(ABT)
5. 未定义模式(UND)

除了用户模式之外的其他六种模式成为特权模式

**ARM有多少32位寄存器？**

ARM处理器共有37个寄存器，包含31个通用寄存器和6个状态寄存器

**ARM2440和6410有什么区别？**

文本

描述已自动生成

**ARM指令集分为几类？**

2类，分别为Thumb指令集，ARM指令集。指令长度为32位，Thumb指令长度为16位。这种使得ARM既能执行16位指令，又能执行32位指令，从而增强了ARM内核的功能。

**通用寄存器包括R0~R15，可以分为具体哪三类？**

通用寄存器包括R0~R15,可以分为3类：

1. 未分组寄存器R0~R7

在所有运行模式下，未分组寄存器都指向同一个物理寄存器，他们未被系统用作特殊用途。因此在中断或异常处理进行异常模式进行异常模式转换时，由于不同处理器运行模式均使用相同的物理寄存器，所以可能造成寄存器中数据的破坏。

1. 分组寄存器R8-R14

对于分组寄存器，他们每次所访问的物理寄存器都与当前的处理器运行模式相关。

**R13常用作存放堆栈指针**，用户也可以使用其他寄存器存放堆栈指针，但在Thumb指令集下，某些指令强制要求使用R13存放堆栈指针。

R14成为链接寄存器（LR，link Register），当执行子程序时，R14可得到R15（PC）的备份，执行完子程序后，又将R14的值复制回PC，即使用R14保存返回地址

1. 程序计数器PC（R15）

程序计数器R15用作程序计数器(PC)，在ARM状态下，位[1:0]为0，位[31:2]用于保存PC；在Thumb状态下，位[0]为0，位[31:1]用于保存PC

**ARM处理器有几种工作状态？**

从编程角度看，ARM微处理器的工作状态一般有ARM和Thumb两种，并可在两种状态之间切换。

1. ARM状态：此时处理器执行32位的字对齐ARM指令，绝大部分工作在此状态。
2. Thumb状态：此时处理器执行16位的半字对齐的Thumb指令

**ARM系统中，在函数调用的时候，参数是通过哪种方式传递的？**

当参数小于等于4的时候是通过**R0~R3寄存器**来进行传递的，当参数大于4的时候是通过**压栈**方式进行传递

**ARM协处理器指令包括哪3类，请描述它们的功能？**

ARM协处理器指令包括以下3类：

1. 用于ARM处理器初始化ARM协处理器的数据处理操作
2. 用于ARM处理器的寄存器和ARM协处理器的寄存器间的数据传送操作
3. 用于在ARM协处理器的寄存器和内存单元之间传送数据

**什么是PLL（锁相环）？**

简单来说，输入时钟的存在是作为“参考源”，锁相环不是为了单纯产生同频同相信号，而是一般集成进某种“频率综合电路”，产生一个不同频，但锁相的信号

比如：某参考晶振10MHz，频率综合器A使用该参考源产生了900MHz时钟，而频率综合器B产生了1GHz时钟。虽然两路频率不同，但由于使用的同一个参考源，他们两仍然是同源信号。PLL通过不断调整VCO的输出频率，使得反馈信号的相位与输入信号的相位相匹配，最终达到**相位锁定的状态**（相位差为0）

**中断与异常：**

**中断与异常有何区别？**

中断是指**外部硬件**产生的一个电信号从CPU的中断引脚进入，打断CPU的运行

异常是指软件运行过程中出现了一些**必须做出处理**的事件，CPU自动产生一个陷入来打断CPU的运行。异常在处理的时候必须考虑与处理器的**时钟同步**，实际上异常也称为**同步中断，**在处理器执行到因编译错误而导致错误指令时，或者在执行期间出现特殊错误，必须依靠内核处理的时候，处理器就会产生一个异常。

**中断与DMA有何区别？**

**DMA：**是一种无需CPU的参与，就可以让外设与系统内存之间进行双向数据传输的硬件机制，使用DMA可以使系统CPU从实际的I/0数据传输过程中摆脱出来，从而大大提高系统的吞吐率。

**中断**:是指cpu在执行程序的过程中，出现了某些**突出事件**时，CPU必须**暂停执行**当前程序，转去**处理突发事件**，处理完毕后CPU又**返回源程序**被中断的位置并继续执行。

所以中断和DMA的区别就是：**DMA不需CPU参与，而中断是需要CPU参与的**。

**中断能不能睡眠，为什么？下半部能不能睡眠？**

* 中断处理的时候不应该发生进程切换。因为在中断上下文中，**唯一能打断当前中断handler的只有更高优先级的中断**，它**不会被进程打断**。如果在中断上下文中休眠，则没有办法唤醒它，因为所有的**wake\_up\_xxx都是针对某个进程而言的**，而在中断上下文中，没有进程的概念，没有一个task\_struct(这点对于softirq和tasklet一样)。因此真的休眠了，比如调用了会阻塞的例程，内核肯定会崩溃。
* schedule()在切换进程时，**保存当前的进程上下文**(cpu寄存器的值、进程的状态以及堆栈中的内容)，以便以后恢复此进程运行。中断发生后，内核会**先保存当前被中断的进程上下文**(在调用中断处理程序后恢复)。

但在中断处理程序中，CPU寄存器的值肯定已经变化了(最重要的程序计数器PC,堆栈SP等)，如果此时因为睡眠或阻塞操作调用了schedule()，则保存的进程上下文就不是当前的进程上下文了。所以，**不可以在中断处理程序中调用schedule()**。

* 2.4内核中schedule()函数本身在进来的时候**判断是否处于中断上下文**

if(unlikely(in\_interrupt())) BUG();

因此，强行调用schedule()的结果就是内核BUG，但看其他版本的内核却没有这句，改掉了。

* 中断handler会使用被中断的进程的内核堆栈，但是不会对它有任何影响，因为handler使用完后会完全清除它使用的那部分堆栈，恢复被中断前的原貌。
* 处于**中断上下文**时候，**内核是不可抢占**的。因此，如果休眠，则内核一定挂起。

**中断的响应执行流程是什么？**

中断的响应流程：cpu接收中断->保存中断上下文跳转到中断处理历程->执行中断上半部->执行中断下半部->恢复中断上下文

**当一个异常出现以后，ARM微处理器会执行哪几步操作？**

* 将下一条指令的地址存入相应连接寄存器LR，以便程序在处理异常返回时能从正确的位置重新开始执行。若异常是从**ARM状态**进入，则LR寄存器中保存的是**下一条指令的地址**(当前PC+4或PC+8，与异常的类型有关)；若异常是从**Thumb状态**进入，则**在LR寄存器中保存当前PC的偏移量**，这样，异常处理程序既不需要确定异常是从何种状态进入的。例如：在软件中断异常SWI，指令MOV PC，R14\_svc总是返回到下一条指令，不管SWI是在ARM状态执行，还是在Thumb状态执行。
* 将CPSR复制到相应的SPSR中
* 根据异常类型，强制设置CPSR的运行模式位
* 强制PC从相关的异常向量地址取下一条指令执行，从而跳转到相应的异常处理程序处。

**写一个中断服务需要注意哪些？如果中断产生之后要做比较多的事情你是怎么做的？**

1. 写一个中断服务程序要注意**快进快出**，在中断服务程序里面尽量**快速采集信息**，包括硬件信息，然后退出中断，要做其他事情可以使用**工作队列或者tasklet方式**。也就是中断上半部和下半部。
2. 中断服务程序**不能有阻塞操作**。应为中断期间是完全占用CPU（即不存在内核调度），中断被阻塞住，其他进程将无法操作
3. 中断服务程序**注意返回值**，要用操作系统定义的宏作为返回值，而不是自己定义的
4. 如果要做的事情比较多，应将这些任务放在后半段(tasklet，等待队列等)处理。

**为什么FIQ比IRQ要快？**

1. ARM的FIQ模式提供了**更多的banked寄存器，r8到r14还有SPSR,**而IRQ模式就没有这么多，这就意味着在ARM的IRQ模式下，中断处理程序自己要保存R8到R12这几个寄存器，然后退出中断处理时程序要恢复这几个寄存器，而FIQ模式由于这几个寄存器都有banked寄存器，模式切换时cpu自动保存这些值到banked寄存器，退出FIQ模式时自动恢复，所以这个过程FIQ比IRQ快。
2. FIQ比IRQ有更高的优先级
3. 在symbian系统里，当CPU处于FIQ模式处理FIQ中断的过程中，预取指令异常，未定义指令异常，软件中断全被禁止，所有的中断被屏蔽。所以FIQ就会很快执行，不会被其他异常或者中断打断，所以它又比IRQ快了。
4. 另外FIQ的入口地址是0x1c，IRQ的入口地址是0x18

**中断和轮询哪个效率高？怎样决定是采用中断还是采用轮询方式去实现驱动？**

中断是cpu处于**被动状态**下来接收设备的信号，而轮询是**CPU主动去查询**该设备是否有请求。

如果是请求设备是一个**频繁请求**cpu的设备，或者有**大量数据**请求的网络设备，那么轮询的效率是比中断高，如果是一般设备，并且该设备请求cpu的效率比较低，则使用中断的效率要高一些。主要看请求的频率。

**通信协议：**

**什么是异步传输和同步传输？**

异步传输：是一种典型的基于字节的输入输出，数据按每次一个字节进行传输，其传输速度低。

同步传输：需要外界的时钟信号进行通信，是把数据字节组合起来一起发送，这种组合称之为帧，其传输速度比异步传输快。

**RS232和RS485通讯接口有什么区别？**

1. **传输方式不同**。RS232采用不平衡传输方式，即所谓**单端通讯**。而RS485则采用平衡传输，即**差分传输方式**。
2. **传输距离不同**。RS232适合本地设别之间通信，传输距离一般**不超过20m。**而RS485的传输距离为**几十米到几千米。**
3. **设备数量。**RS232只允许**一对一通信，**而RS485接口在总线中允许连接**多达128个收发器**。
4. **连接方式。**RS232，规定**用电平表示数据**，因此线路就是**单线路的**，用**两根线**才能达到**全双工**的目的；而RS485，使用**差分电平表示数据**，因此用**两根线**才能达到传输数据的基本要求，要实现**全双工**，必须用**四根线**。

总结：从某种意义上，可以说，线路上存在的仅仅是电流，RS232/RS485规定了这些电流在什么样的线路上流动和流动的样式。

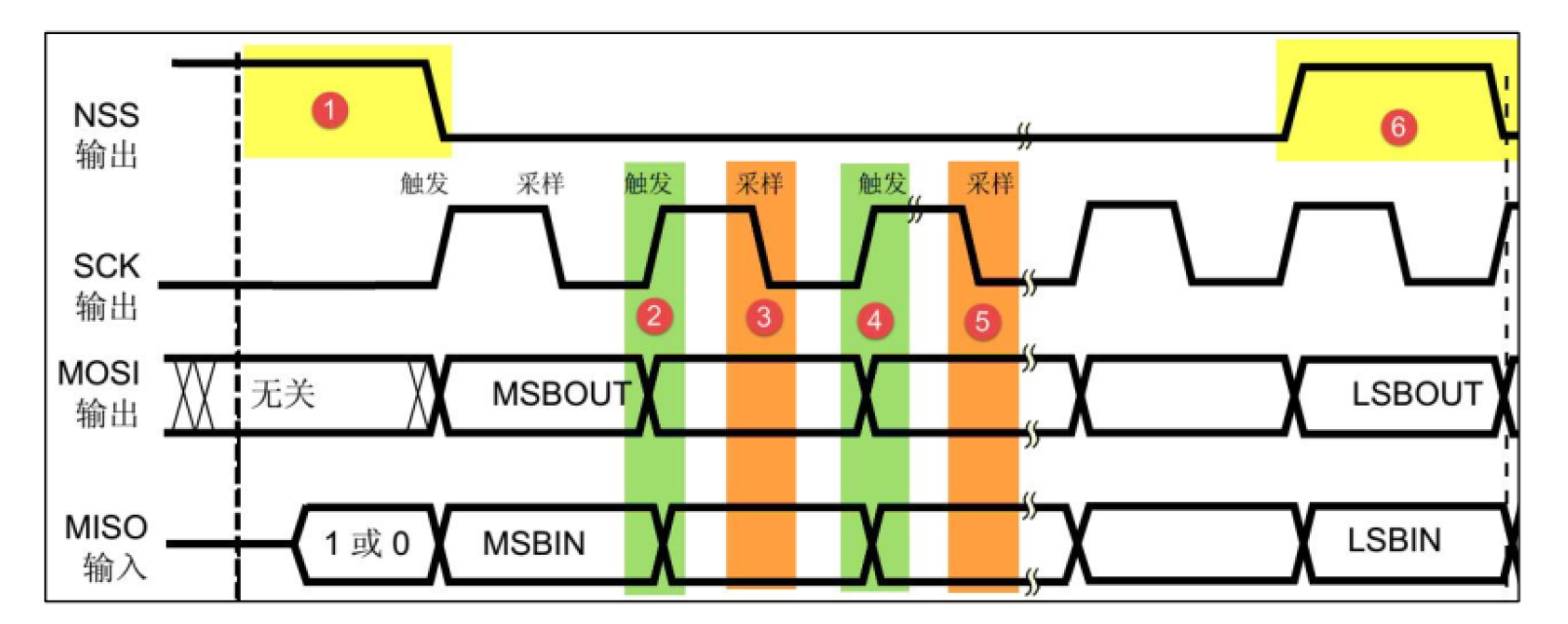
**SPI协议**

SPI的应用：SPI即**串行外围设备接口**，是一种**高速全双工**的通信总线。SPI总线系统是一种同步串行外设接口，它可以使用MCU与各种外围设备以串行方式进行通信以交换信息。SPI总线可直接与各个厂家生产的多种标准外围器件相连，包括FLASH、RAM、网络控制器、LCD显示驱动器、A/D转换器和MCU等

**接口：**

1. MOSI（Master Output , Slave Input）:主设备输出/从设备输入引脚。主机的数据从这条信号线输出，从机由这条信号线读入主机发送的数据，即从这条线上的数据的方向为主机到从机。
2. MISO（Master Input , Slave Output）:主设备输入/从设备输出引脚。主机从这条信号线读入数据，从机的数据由这条信号线输出到主机，即在这条线上数据的方向为从机到主机。
3. SCLK（Serial Clock）：时钟信号线，用于通讯数据同步，**它由通讯主机产生**，决定了通讯的频率，不同的设备支持的最高时钟频率不一样，如STM32的SPI时钟频率最大为fpclk/2，两个设备之间通讯时，**通讯速率受限于低速设备**。
4. SS（Slave Select）：从设备选择信号线，常称为片选信号线，也称为NSS、CS、以下用NSS表示。当有多个SPI从设备与SPI主机相连时，设备的其它信号线SCLK，MOSI及MISO同时并联到相同的SPI总线上，即无论有多少从设备，都共同使用这三条总线；而每个从设备都有独立的这一条NSS信号线，本信号线独占主机的一个引脚，即有多少个从设备，就有多少条片选信号线。
5. I2C协议中通过设备地址来寻址、选中总线上的某个设备并与其进行通讯；而SPI协议中没有设备地址，它使用NSS信号线来寻址，当主机要选择从设备时，把该从设备的NSS信号线设置为低电平，该从设备即被选中，即片选有效，接着主机开始与被选中的从设备进行SPI通讯。所以SPI通讯以NSS线置低电平为开始信号，以NSS线被拉高作为结束信号。

**SPI的通信时序：**



1.通讯的起始和停止信号

在图中的标号1处，NSS信号线从高变低，是SPI通讯的起始信号。NSS是每个从机各自独占的信号线，当从机在自己的NSS线检测到起始信号后，就知道自己被主机选中了，开始准备与主机通讯。在途中的标号6处，NSS由低变高，是SPI通讯的停止信号，表示本次通讯结束，从机的选中状态被取消。

2.数据有效性

SPI使用MOSI及MISO信号线来传输数据，使用SCLK信号线进行数据同步。MOSI及MISO数据线在SCLK的每一个时钟周期传输一位数据，且数据输入输出是同时进行的。数据传输时，MSB先行(高位先行)或LSB(低位先行)先行并没有作硬性规定，但要保证两个SPI通讯设备之间使用同样的协定，一般都会采用上图中的MSB先行（高位先行）模式。

观察图中的2345标号处，MOSI及MISO的数据在SCLK的上升沿期间变化输出，在SCLK的下降时被采样。即在SCLK的下降沿时刻，MOSI及MISO的数据有效，高电平是表示数据“1”，为低电平是表示数据“0”。在其他时刻，数据无效，MOSI及MISO为下一次表示数据做准备。SPI每次数据传输可以8位或16位为单位，每次传输的单位数不受限制。

1. CPOL(时钟极性)/CPHA(时钟相位)及通讯模式

上面描述的图中的时序只是SPI中的一种通讯模式，SPI一共有四种通讯模式，它们的主要区别是:总线空闲时SCLK的时钟状态以及数据采样时刻。

时钟极性CPOL：SPI通讯设备处于空闲时，SCLK信号线的电平信号(SPI通讯开始前，CS线为高电平时SCLK的状态)。CPOL=0时，SCLK在空闲时为低电平，CPOL=1时，则相反。

时钟相位CPHA是指数据的采样的时刻，当CPHA=0时，MOSI或MISO数据线上的信号将会在SCLK时钟线的“奇数边沿”被采样。当CPHA=1时，数据线在SCLK的“偶数边沿”采样。

**IIC协议**

IIC协议是由数据线SDA和时钟SCLK构成的串行总线，可发送和接收数据，是一个多主机的**半双工通信方式，**每个挂接在总线上的器件都有个**唯一的地址**。位速在标准模式下可达100kbit/s，在快速模式下可达400kbit/s，在高速模式下可达3.4Mbit/s。

I2C总线系统结构，如下所示：

箱线图

中度可信度描述已自动生成

**I2C时序介绍**

1. **空闲状态**

当总线上的SDA和SCL两条信号线同时处于高电平，便是空闲状态，如上面的硬件图所示，当我们不传输数据时，SDA和SCL被上拉电阻拉高，即进入空闲状态。

1. **起始信号**

当**SCLK为高**期间，**SDA由高到低的跳变，**便是总线的启动信号，**只能由主机发起**，**且在空闲状态下才能启动该信号**，如下图所示：

形状

描述已自动生成

1. **停止信号**

当**SCL为高期间**，**SDA由低到高的跳变**；**便是总线的停止信号**，表示数据已传输完成，如下图所示

形状

描述已自动生成

1. **传输数据格式**

当发了起始信号后，就开始传输数据，传输的数据格式如下图所示：

当SCLK为高电平时，就会获取SDA数据值，其中**SDA数据必须是稳定的**(如果SDA不稳定就会变成起始/停止信号)

当SCLK为低电平时，I²C 总线上的数据线（SDA）可以改变状态

若主从机在传输数据期间，需要完成其他功能(例如一个中断)，可以主动拉低SCLK，使I2C进入等待状态，直到处理结束再释放SCLK，数据传输会继续。

形状

描述已自动生成

1. **应答信号ACK**

I2C总线上的数据都是以8位数据(一个字节)进行的，当发送了8个数据后，发送方会在第9个**时钟脉冲期间释放SDA数据**，当接收方接收该字节成功，便会输出一个ACK应答信号，当SDA为高电平，表示为非应答信号NACK，当SDA为低电平，表示为有效应答信号ACK。

PS：当主机为接收方时，收到最后一个字节后，主机可以不发送ACK，直接发送停止信号来结束传输。

当从机为接收方时，没有发送ACK，则表示从机可能在忙其他事、或者不匹配地址信号和不支持多主机发送，主机可以发送停止信号，再次发送起始信号启动新的传输。

图示

描述已自动生成

1. **完整的数据传输**

如下图所示，发送起始信号后，便发送一个8位的设备地址，其中第8位是对设备的读写标志，后面紧跟着就是数据了，直到发送停止信号终止。

PS：当我们第一次是都读操作，然后想换成写操作时，可以再次发送一个起始信号，然后发送读的设备地址，不需要停止信号便能实现不同的地址转换。

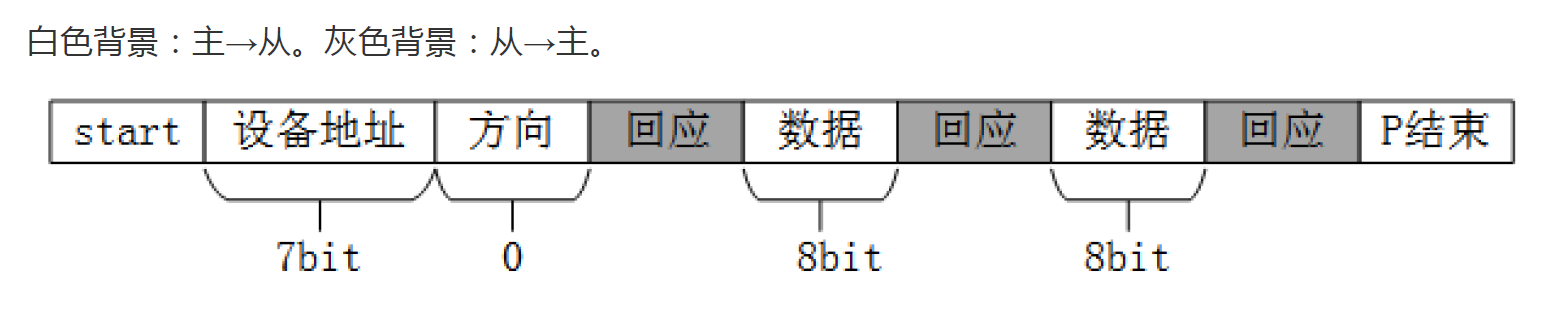
形状, 多边形

描述已自动生成

IIC传输数据的格式

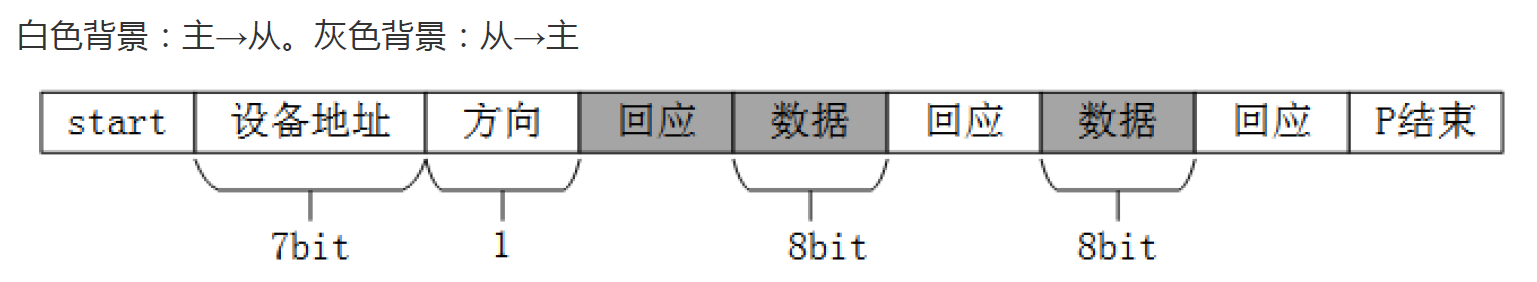
1. 写操作

刚开始主芯片要发出一个start信号，然后发出一个(用来确定是哪一个芯片写数据)，方向（读/写，0表示写，1表示读）。回应（用来确定这个设备是否存在），然后就可以传输数据，传输数据后，要有一个回应信号(确定数据是否接收完成)，然后在传输下一个数据。每传输一个数据，接收方都会有一个回应信号，数据发送完之后，主芯片就会发送一个停止信号。



2.读操作

刚开始主芯片要发出一个start信号，然后发出一个设备地址(用来确定是从哪一个芯片读取数据)，方向(读/写，0表示写，1表示读)。回应（用来确定这个设备是否存在），然后就可以传输数据，传输数据之后，要有一个回应信号(确定数据是否接收完成)，然后再传输下一个数据。每传输一个数据，接收方都会有一个回应信号，数据发送完之后，主芯片就会发送一个停止信号。



**编程**

**嵌入式编程中，什么是大端？什么是小端？**

**大端模式：低位字节存在高地址上，高位字节存在低地址上。**

**小端模式：高位字节存在高地址上，低位字节存在低地址上。**

图示

描述已自动生成

STM32属于小端模式，简单地说，比如u32 temp=0x12345678;假设temp地址在0x2000 0010,那么在内存里面的就变成了：

地址：Ox2000 0010 HEX 78 56 43 12

采用小段模式的CPU对操作数的存放方式是从低字节到高字节，而大端模式对操作数的存放方式是从高字节到低字节。例如，16位宽的数0x1234在小端模式CPU内存中的存放方式(假设从地址0x4000开始存放)见表1，大端模式见表2



表格

描述已自动生成

**X86机内存是小端模式，单片机一般为大端模式，STM32是小端模式**

**如何判断计算机处理器是大端，还是小端？**

**#include <stdio.h>**

**int checkCpu(){**

**{**

**union w**

**{**

**int a;**

**char b;**

**}c;**

**c.a = 1;**

**return (c.b == 1);**

**}**

**}**

**int main()**

**{**

**if(checkCpu()){**

**printf("small end\n");**

**}else{**

**printf("big end\n");**

**}**

**return 0;**

**}**

**联合体union的存放顺序是所有成员都从低地址开始存放，如果能够通过改代码直到CPU对内存采用小端模式读写，采用大端模式读写。**