# 纯C

## 实现n个元素的全排列

void permutation(T list[], int k, int m)

{

if (k == m)

{

copy(list, list + m + 1, ostream\_iterator<T>(cout, "")); //将当前list排序

cout << endl;

}

else{

for (int i = k; i <= m; i++)

{

swap(list[i], list[k]); //将下标为i的元素交换到k位置，类似从list[k:m]中剔除操作

permutation(list, k + 1, m);

swap(list[i], list[k]);

}

}

}

int main(int argc, char\* argv[])

{

char arr[3] = { 'a', 'b', 'c' };

cout << "排序结果如下：" << endl;

permutation(arr, 0, 2);

return 0;

}

## longlong型数据操作

表示unsigned long long：

printf("num = %llu\n", num);

表示long long int：

printf("num = %lld\n", num);

## 3.接收

7

5 -3 6 5 -5 -1 6

-6 1 4 -2 0 -2 0

这种格式的方法

**scanf**("%d", &n);

**for** (**int** i=0; i<n; i++) **scanf**("%d", &a[i]);

**for** (**int** i=0; i<n; i++) **scanf**("%d", &b[i]);

用getchar()可以获取一个字符就可以读走换行符。

如果读取

3，5

则用scanf("%d,%d",&N,&M);以什么间隔scanf就写什么

此时缓冲区还有一个\n如果不读入则会影响后面的输入

## 4.C语言字符串分割

#include<string.h>

分解字符串为一组字符串。s为要分解的字符串，delim为分隔符字符串。

strtok()用来将字符串分割成一个个片段。参数s指向欲分割的字符串，参数delim则为分割字符串中包含的所有字符。当strtok()在参数s的字符串中发现参数delim中包涵的分割字符时,则会将该字符改为\0 字符。在**第一次调**用时，strtok()必需给予参数s字符串，往后的调用则将参数s设置**成NULL**，则**函数保存的指针SAVE\_PTR在下一次调用中将作为起始位置**。每次调用成功则返回指向被分割出片段的指针s开头开始的一个个被分割的串。当没有被分割的串时则返回NULL。所有delim中包含的字符都会被滤掉，并将被滤掉的地方设为一处分割的节点。

例子：

#include <stdio.h>

#include <conio.h>

int main(void)

{

char input[16]="abc,d,yuwen";

char \*p;

p=strtok(input,",");//此时input里面只有abc三个字符了

while(p!=NULL)

{

printf("%s\n",p);

p=strtok(NULL,",");

}

return 0;

}

输出：

abc

d

yuwen

## 4.while (x=0)与while (x==0)两条语句的循环条件有何区别

x=0是赋值，可以拆成两句，int x,x=0,while（x），x为零，条件为假，while结束。  
x==0是判断语句，当x为零时，条件为真，while继续。

## 5.n++和++n的区别

无论前置还是后值，++ 都会使其操作数的值增 1。不同的是，++ 前置时，自增表达式（如 ++n）的值等于其操作数自增后的值；++ 后置时，自增表达式（如 n++）的值等于其操作数自增前的值。例如：知int n = 5, post = 1, pre = 1;pre = ++n + pre;    // 运算结束后 pre 为 7，n 为 6n = 5;post = n++ + post;  // 运算结束后 post 为 6，n 为 6上例中，++n 的值等于 n 自增后的值，道即 6，所以 pre 最后得到的值为 7。n++ 的值等于 n 自增前的值，即 5，所以 post 最后得到的值为 6。

## scanf会返回接收正确值的个数

## 时间复杂度就是语句执行次数的最高次

空间复杂度指的来是运行时**临时占**用的存储空间大小.， 主要是看运行程序时临时变量的数目和数据规模之间的关系。举例：无论数据规模多大，我都定义100个变量，这就叫做源数据规模和临时变量数目无关。就是说空间复杂度zhidao是O(1).int i;  
for(i=0;i<n;i++)  
i++;  
时间复杂度是O(n),空间复杂度是O(1);  
int i;  
int j;  
int k;  
for(i=0; i<n; i++)  
for(j=0; j<i; j++)  
for(k=0; k<j; k++)  
;  
时间复杂度是O(n^3),空间复杂度还是O(1)；

## C语言fork()函数

一个进程调用fork（）函数后，系统先给新的进程分配资源，例如存储数据和代码的空间。然后把原来的进程的所有值都复制到新的新进程中，只有少数值与原来的进程的值不同。相当于克隆了一个自己。

fork调用的一个奇妙之处就是它仅仅被调用一次，却能够返回两次，它可能有三种不同的返回 值：  
    1）在父进程中，fork返回新创建子进程的**进程ID**；  
    2）在子进程中，fork返回0；  
    3）如果出现错误，fork返回一个负值；

通过判断**返回值**来确定当前是父还是子进程。

## C语言指针详解：**变量先与[]结合,因为其优先级比\*高**

**判断变量含义方法：从变量开始，按照先后顺序进行解析，指针指向的内容是（数组，int等等）数组的内容是。。。**

**说出含义的时候先判断类型，可以按照从低到高的顺序来解释：**

比如int (\*p)[3] p是一个指针，然后开始从低到高读，指向int型大小为三的数组指针。

int p; //这是一个普通的整型变量    
 int \*p; //首先从P 处开始,先与\*结合,所以说明P 是一个指针,然后再与int 结合,说明指针 所指向的内容的类型为int 型.所以P是一个返回整型数据的指针    
 int p[3]; //首先从P 处开始,先与[]结合,说明P 是一个数组,然后与int 结合,说明数组里的 元素是整型的,所以P 是一个由整型数据组成的数组    
 int \*p[3]; //首先从P 处开始,先与[]结合,因为其优先级比\*高,所以P 是一个数组,然后再与\* 结合,说明数组里的元素是指针类型,然后再与int 结合,说明指针所指向的内容的类型是整型 的,所以P 是一个由返回整型数据的指针所组成的数组    
 int (\*p)[3]; //首先从P 处开始,先与\*结合,说明P 是一个指针然后再与[]结合(与"()"这步可 以忽略,只是为了改变优先级),说明指针所指向的内容是一个数组,然后再与int 结合,说明数 组里的元素是整型的.所以P 是一个指向由整型数据组成的数组的指针    
 int \*\*p; //首先从P 开始,先与\*结合,说是P 是一个指针,然后再与\*结合,说明指针所指向的元 素是指针,然后再与int 结合,说明该指针所指向的元素是整型数据.由于二级指针以及更高级 的指针极少用在复杂的类型中,所以后面更复杂的类型我们就不考虑多级指针了,最多只考虑一 级指针.    
 int p(int); //从P 处起,先与()结合,说明P 是一个函数,然后进入()里分析,说明该函数有一 个整型变量的参数,然后再与外面的int 结合,说明函数的返回值是一个整型数据    
 Int (\*p)(int); //从P 处开始,先与指针结合,说明P 是一个指针,然后与()结合,说明指针指向 的是一个函数,然后再与()里的int 结合,说明函数有一个int 型的参数,再与最外层的int 结 合,说明函数的返回类型是整型,所以P 是一个指向有一个整型参数且返回类型为整型的函数的 指针    
 **int \*(\*p(int))[3]; //可以先跳过,不看这个类型,过于复杂从P 开始,先与()结合,说明P 是 一个函 数,然后进入()里面,与int 结合,说明函数有一个整型变量参数,然后再与外面的\*结 合,说明函数返回的是一个指针,,然后到最外面一层,先与[]结合,说明返回的指针指向的 是 一个数组,然后再与\*结合,说明数组里的元素是指针,然后再与int 结合,说明指针指向的 内 容是整型数据.所以P 是一个参数为一个整数据且返回一个指向由整型指针变量组成的数 组 的指针变量的函数.**

**指针的访问**

一个指针ptrold 加(减)一个整数n 后，结果是一个新的指针ptrnew，ptrnew 的类型和ptrold 的类型相同，ptrnew 所指向的类型和ptrold所指向的类型也相同。ptrnew 的值将比ptrold 的值增加(减少)了n 乘sizeof(ptrold 所指向的类型)个字节。就是说，ptrnew 所指向的内存区将比ptrold 所指向的内存区向高(低)地址方向移动了n 乘sizeof(**ptrold 所指向的类型比如int \*p p指向的就是int ， int （\*p）[5] 这里p指向的就是int[5]**)个字节。指针和指针进行加减：两个指针不能进行加法运算，这是非法操作，因为进行加法后，得到的结果指向一个不知所向的地方，而且毫无意义。两个指针可以进行减法操作，但必须类型相同，一般用在数组方面，不多说了。

**指针例题**

int main()

{

char \*c[] = { "ENTER", "NEW", "POINT", "FIRST" };

char \*\*cp[] = { c + 3, c + 2, c + 1, c };

char \*\*\*cpp = cp;

printf("%s\n\r", \*\*++cpp);

printf("%s\n\r", \*--\*++cpp + 3);

printf("%s\n\r", \*cpp[-2] + 3);

printf("%s\n\r", cpp[-1][-1] + 1);

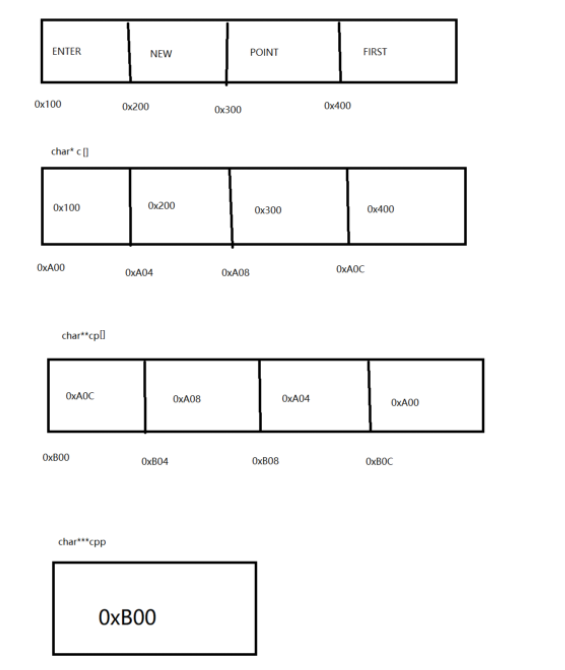
system("pause");

return 0;

}

分析：

理出空间分布



第一个输出printf("%s\n\r", \*\*++cpp);

先进行 ++cpp 当前cpp为0xB00 步长为sizeof(char \*\*)所以结果为 0xB04

然后进行 \*++cpp 得到结果为0xA08

最后计算 \*\*++cpp 得到结果为0x300

打印出来为 **POINT**

第二个输出 printf("%s\n\r", \*--\*++cpp + 3)

先进行 ++cpp 当前cpp为0xB04 步长为sizeof(char \*\*)所以结果为 0xB08

然后进行 \*++cpp 得到结果为0xA04

接着计算 --\*++cpp 当前 \*++cpp为 0xA04 步长为sizeof(char \*)所以结果为 0xA00

计算 \*--\*++cpp 得到结果为 0x100

最后计算 \*--\*++cpp + 3 步长为sizeof(char) 最后结果为**ENTER**该字符串的第四个字符

第三个输出 printf("%s\n\r", \*cpp[-2] + 3);

当前 cpp 为0xB08 cpp[-2] 代表往后移动两个步长 结果为 \*0xB00 即0xA0C

再计算 \*cpp[-2] 得到 0x400

最后计算 \*cpp[-2] + 3 结果为**FIRST**该字符串的第四个字符

第四个输出printf("%s\n\r", cpp[-1][-1] + 1);

当前 cpp为0xB08 cpp[-1] 结果为 \*0xB04 即 0xA08

再计算 cpp[-1][-1] 结果为 \*0xA04 即0x200

最后加上字符偏移 为**NEW**字符串的第2个字符

## C语言易错点

int a = 10;

char b[a];                    //编译不报错

//char b[a] = "jfkd"; //报错，**使用变量定义长度时，不可在定义时同时进行初始化赋值**， 需要在之后进行赋值

## 各种变量比较

1. static： **（被修饰者不能被其它文件访问）**

修饰的在**编译期初**（**类也是**在编译期初始化的）

存放位置：不管是在全局还是局部声明的static变量**都**存放于程序的**全局变量区域**

生命周期：是从程序开始到程序结束（和存放位置相关）。

作用域：决定于它被定义的位置，**作用域为包含该定义的代码块**，**不能被其它文件访问**。

1. 全局变量：

存放位置：程序的**全局变量区域**

生命周期：是从程序开始到程序结束（和存放位置相关）。

作用域：整个程序，也就是所有的源文件，**能被其它文件访问（使用extern声明）**。

1. 局部变量

存放位置：程序的**栈区域**

生命周期：是从**函数**开始到程序结束（和存放位置相关）。

作用域：决定于它被定义的位置，**作用域为包含该定义的代码块**也就是**当前函数**。

1. 申请的地址空间：

存放位置：程序的**堆区域**

生命周期：是从**申请成功**开始到**地址释放**结束（和存放位置相关）。

作用域：只要不释放，他就一直在，你只要有地址就能访问。

Static修饰的在**编译期初**（**类也是**在编译期初始化的）始化，所以在没有对象时也能通过类名调用在类初次被加载的时候，会按照static块的顺序来执行每个static块，并且**只会执行**一次。在**内存中只有一个拷贝**（节省内存）不同对象的static变量**指向的地址**相同。

Final表示不能改变，变量被修饰后就是常量了。

一个程序的几个阶段

（1）程序的设计期，主要设计整体框架以及每个功能模块；大家常用UML语言来设计软件框架；

（2）软件的编译时期，这个时间主要是将用C++语言写的代码编译成机器码或者是过渡时间的语言形式；这个时期的任务是由编译器负责；

（3）软件的链接期，这个时期是在编译之后，编译后的各个模块（以文件为单位）需要相互之间以及和一些必须的库之间进行连接，只有链接后才生成可执行格式的文件。链接期的功能由连接器负责；

（4）程序要运行，必须先进行加载，只有加载到系统中才能运行，这就是程序的加载期，加载工作是由系统的加载器负责的，在加载的过程中会提供初始化一些变量（全局和静态变量）或对象的操作；

（5）运行期，这时期就是程序加载后一条条执行程序指令的时期，这个时期也会初始化一些零时变量的工作等

## C语言递归函数

函数不会执行变量的声明操作比如int i,j;但会执行初始化比如 int i=0;则会执行

递归函数返回**未申请空间（相应初始化）**的地址时会返回空。

## 函数中参数传值

参数传递有两个目的

1. 函数需要使用相应的数据，此时调用者把数据放入参数传递，在这个情况下函数中的参数都被转存为相应指针，具体转换规则为：**函数中的参数名代表的是指向该参数的指针**
2. 通过传入参数来改变该参数，用于想要返回多个数据的情况，传入规则为：**传入想要数据的指针，比如想要 int 那就传入指向 int 的指针 int\*，想要 int \*那就传入指向 int \*的指针 int\*\*。**

## C语言格式化输出

|  |  |
| --- | --- |
| **格式控制符** | **说明** |
| %c | 输出一个单一的字符 |
| %hd、%d、%ld | 以十进制、有符号的形式输出 short、int、long 类型的整数 |
| %hu、%u、%lu | 以十进制、无符号的形式输出 short、int、long 类型的整数 |
| %ho、%o、%lo | 以八进制、不带前缀、无符号的形式输出 short、int、long 类型的整数 |
| %#ho、%#o、%#lo | 以八进制、带前缀、无符号的形式输出 short、int、long 类型的整数 |
| %hx、%x、%lx %hX、%X、%lX | 以十六进制、不带前缀、无符号的形式输出 short、int、long 类型的整数。如果 x 小写，那么输出的十六进制数字也小写；如果 X 大写，那么输出的十六进制数字也大写。 |
| %#hx、%#x、%#lx %#hX、%#X、%#lX | 以十六进制、带前缀、无符号的形式输出 short、int、long 类型的整数。如果 x 小写，那么输出的十六进制数字和前缀都小写；如果 X 大写，那么输出的十六进制数字和前缀都大写。 |
| %f、%lf | 以十进制的形式输出 float、double 类型的小数 |
| %e、%le %E、%lE | 以指数的形式输出 float、double 类型的小数。如果 e 小写，那么输出结果中的 e 也小写；如果 E 大写，那么输出结果中的 E 也大写。 |
| %g、%lg %G、%lG | 以十进制和指数中较短的形式输出 float、double 类型的小数，并且小数部分的最后不会添加多余的 0。如果 g 小写，那么当以指数形式输出时 e 也小写；如果 G 大写，那么当以指数形式输出时 E 也大写。 |
| %s | 输出一个字符串 |

printf() 格式控制符的完整形式如下：

%[flag][width][.precision]type

[ ] 表示此处的内容可有可无，是可以省略的。

1. type 表示输出类型，比如 %d、%f、%c、%lf，type 就分别对应 d、f、c、lf；再如，%-9d中 type 对应 d。type 这一项必须有，这意味着输出时必须要知道是什么类型。  
   2) width 表示最小输出宽度，也就是至少占用几个字符的位置；例如，%-9d中 width 对应 9，表示输出结果**最少**占用 9 个字符的宽度。  
   当输出结果的宽度不足 width 时，以空格补齐（如果没有指定对齐方式，默认会在左边补齐空格）；当输出结果的**宽度超过 width 时，width 不再起作用**，按照数据本身的宽度来输出。

3) .precision 表示输出精度，也就是小数的位数。

* 当小数部分的位数大于 precision 时，会按照四舍五入的原则丢掉多余的数字；
* 当小数部分的位数小于 precision 时，会在后面补 0。

1. flag 是标志字符。例如，%#x中 flag 对应 #，%-9d中 flags 对应-。下表列出了 printf() 可以用的 flag：

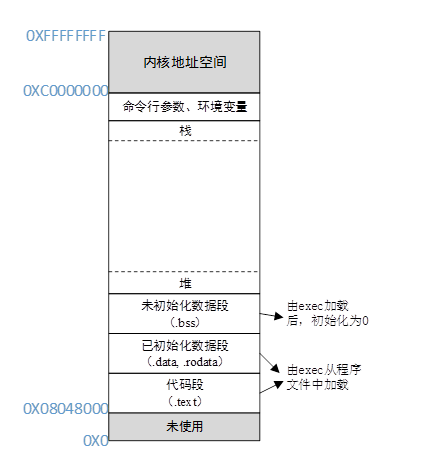
## c++源文件从文本阶段到可执行文件的过程

1. 预处理：就是把代码中的预处理指令换成相应的值。**文本替换 宏命令展开**
2. 编译：生成汇编文件
3. 汇编：生成机器指令
4. 链接：将独立文件连接成一个可执行文件，

静态链接（在执行前就生成了完整程序，同一函数可能出现多次在代码占用空间）

动态链接（在执行时，才找相应的库，如果没有则加入内存，后面再用就不用调入内存了）

## 16.C语言程序空间组成 及程序运行阶段



**每个进程自认为的地址空间**

1）正文段（代码段）：存储内容：**CPU执行的机器指令部分**；一个程序只有一个副本；只读，防止程序由于意外事故而修改自身指令；**字符串常量**也属于这部分。  
 2）初始化数据段（数据段）：存储内容：在程序中所有赋了初值的**全局变量**，存放在这里。  
 3）非初始化数据段（bss段）：存储内容：在程序中没有初始化的全局变量；内核将此段初始化为0。（这就是为什么全局内置类型变量会初始化，而局部变量就为未初始化的未知值）  
 4）栈——：存储内容：自动变量以及每次函数调用时所需要保存的信息（**返回地址，局部变量，环境信息，活动过程记录结构体**）增长方向：自顶向下增长；。  
 5）堆——动态存储分区，用于**malloc等函数的地址分配**。

17.#include <>和#include “”区别 **所以说 ”” 的作用范围大于<>**

#include <>用于包含标准库文件，会在VC++的Include文件夹寻找相应文件。

#include “”用于自己编写的文件，先在**当前目录**搜索是否有符合的文件，如果没有**再到**Include文件夹里去找对应的文件

## 17.define 和 typedef 区别

define本质上就是给一段**字符串**起个别名，在预处理阶段就会被替换回来，只要前面出现define语句（不管调用与否），后面相应都会被替换

typedef本质 它的作用就是给**类型**起别名，是一个**关键字，**编译时处理，它定义的东西 **不会被替换**

1. 功能不同：typedef只能给类型起别名，而define能给任何合法语句取别名
2. 作用域：typedef 是关键字，所以只能在当前代码段生效，而define**定义后（只要之前有定义：不管是在子函数或是没调用）**，后面就都能够使用
3. 封闭性不同：typedef 是给一个类型起了别名，虽然功能上和之前类型一样，但也 新的类型，不能加上其他关键字，因为这样不合法，而define在预处理会被替换， 就可以完成加关键字组合。

## int long 字节占用数

**本质：占用字节由编译器解释和编译器相关**

在64位windows系统中

Int和long字节都为4

在64位linux系统中

Int和long字节分别为4和8

## switch case

1. 一定是遵守 先执行case 的语句，最后执行default 语句的，**不管先后顺序**
2. 从例子可以明显看出没有break的话则会从条件匹配的 case 语句开始运行，然后**不论是否符合条件**继续持续执行下一个 case 语句，直到遇到break或者执行完所有case。

## 越界访问后果

当出现越界时，由于无法得知被访问空间存储的内容及功能，所以会出现不可预知后果。**可能程序崩溃**，**可能运算结果非预期**，**也可能完全没有影响**。

在编程时要避免指针越界访问，对于用变量作为参数访问元素时，需要控制好变量范围。如果变量值由外部因素决定，那么访问前先对合法性做判断，防止越界。

## 函数调用过程中函数栈详解

入栈顺序：

## 22.const详解

1. 修饰代码块：

表示里面的内容不能出现修改的操作，包括调用其它函数也要要求是**被const修饰**的。

1. 修饰变量

表示被修饰后的那一部分不能改变，比如

const int \*p; 修饰int即\*p 所指向的内容无法改变

int const \*p; 修饰int即\*p 所指向的内容无法改变

int \*const p; 修饰int\* 即\*p 不能改变指向

1. const修饰的作用：防止常量的值被直接或间接修改，即防止常量的指针或引用被泄露出去，**被const修饰的变量，被取地址，地址也会是const类型的**

如：

const int a = 10;

//int \*p = &a; //error **const int \*** --> **int \***存在修改a值得风险\*p = 0; **不允许等级变低**

const int \*p1 = &a;

int b = 20;

const int \*p2 = &b; //对于普通常量b可以用任意类型的指针指向,因为b只可以修改

int \*q = p2; //error C2440: “初始化”: 无法从“const int\*”转换为“int \*”

# 2.C++

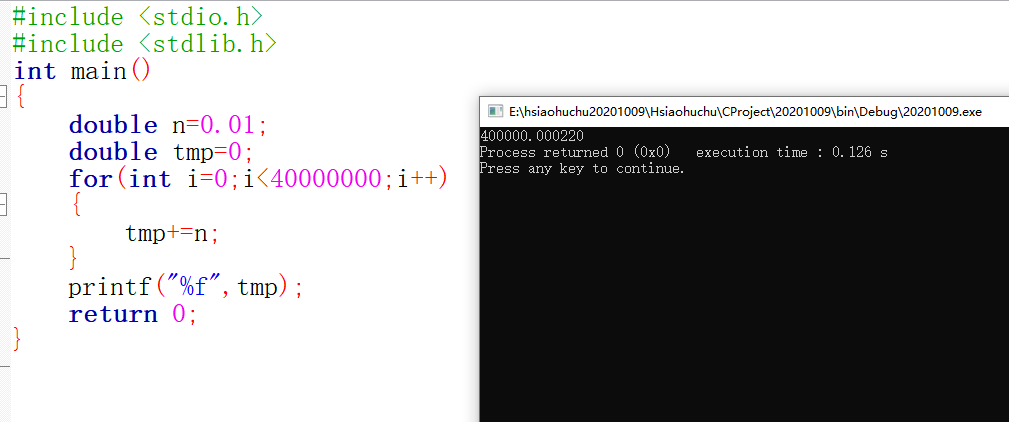
## 1.float a,b,c ,问等式 (a+b)+c==(b+a)+c 和(a+b)+c==(a+c)+b能否成立？

**本质：**浮点数计算存在计算误差

两者都不行。在比较float或double时，不能简单地比较。由于计算误差，相等的概率 很低。应判断两数之差是否落在区间（-e,e)内。

这个e应比浮点数的精度大一个数量级。

举证：



## 枚举类型

**注意**：

1.同一程序中不能定义同类型名的枚举类型；不同枚举类型的枚举元素不能同名。

2.枚举型可以隐式的转换为int型，int型不能隐式的转换为枚举型。

3.枚举元素是常量不是变量，不能在程序中再为枚举元素赋值。

枚举元素值规则：

1.默认第一个元素为0

2.没有赋值的枚举值在为前一个枚举值加1

## 类型转换函数

类型转换函数的作用就是将当前类类型转换为其它类型，它只能以**成员函数**的形式出现，也就是只能出现在类中。

## inline函数

inline仅是一个对编译器的建议，会把函数调用替换成函数的定义，就像宏定义一样

**Inline需要和函数体**在一起才能有效替换

  （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；**

## 友元函数

访问权限：

和**成员函数拥有相同的权限**，可以访问所有成员

## 析构函数

**格式： “~” +”类名”**

**用法**：当对象被销毁时会调用相应的析构函数

#include<iostream>

using namespace std;

class A{

public:

A(int a, int b) : m\_a(a)

{cout <<"A()" <<endl;}

~A()

{cout <<"~A()" << endl;}

void printfP()

{dotest();}

virtual void dotest()

{cout <<"A:dotest( " <<m\_a << endl;}

private:

int m\_a;

};

class B :public A

{

public:

B(int a, int b, int c):A(4, 5), m\_a(a)

{cout <<"B0" << endl;}

~B()

{cout <<"~B0" <<endl;}

virtual void dotest()

{cout <<"B:dotest() " <<m\_a << endl;}

private:

int m\_a;

};

int main()

{

A\*c = new B(1,2,3);// 换成B\*c = new B(1,2,3); 可以发现多了一个B被析构函数的执行

//只有被销毁才能调用析构，当前c是A，销毁c只会调用A的析构，

//当c是B对象时，因为B中有A所以先调用B析构再调用A析构

c->printfP();

delete c;

return 0;

}

## Vector

vector是一个能够存放任意类型的动态数组，能够增加和压缩数据。

扩容时是以**2倍**增加容量的

但因为内存空间是连续的，所以在进行插入和删除操作时，会造成内存块的拷贝，时间复杂度为o(n)。另外，当数组中内存空间不够时，会重新申请一块内存空间并进行内存拷贝（内存空间会变）。

size()函数返回的是已用空间大小，capacity()返回的是总空间大小，capacity()-size()则是剩余的可用空间大小。

**成员函数：**

c.push\_back(elem)在尾部插入一个elem数据。push\_back()函数是在原有size后面向量的末尾增加元素，如果capacity等于size则**新分配内存**，而不是覆盖原有的内存。**以前内存则被释放**。

c.pop\_back()删除末尾（size）的数据。

c.resize(num)重新指定队列的长度。Size也变大了，内容默认填充为**默认构造对象**（往往用来增加vector的长度,小->大 ok **大->小 没用！**）

c.reserve()保留适当的容量。只增加capacity大小**并没有**给相应地址创建相应对象

注意：

Vector内存占用**只增不减**

所有内存空间是在vector**析构时候才能**被系统回收。empty()用来检测容器是否为空的，clear()可以清空所有元素。但是即使clear()，vector所占用的内存空间依然如故，无法保证内存的回收。

## 迭代器

迭代器（iterator）是一种对象，它能够用来遍历标准模板库容器中的部分或全部元素，每个迭代器对象代表容器中的确定的地址，简单地说，迭代器是一种检查容器内元素并遍历元素的可带泛型数据类型。

## extern "C"

在C++源文件中的语句前面加上extern "C"，表明它按照类C的**编译和连接**规约来编译和连接。

**编译差别：**

为了支持函数重载的这个特性，C++编译器实际上将下面这些重载函数：

1    voidprint(inti);  
2    voidprint(charc);  
3    voidprint(floatf);  
4    voidprint(char\* s);

编译为：  
1    \_print\_int  
2    \_print\_char  
3    \_print\_float  
4    \_pirnt\_string

C语言中并没有重载和类这些特性，故并不像C++那样print(int i)，会被编译为\_print\_int，而是直接编译为\_print等。因此如果直接在C++中调用C的函数会失败，因为连接是调用C中的 print(3)时，它会去找\_print\_int(3)。因此extern "C"的作用就体现出来了。

## 10.抽象类

**抽象类：**

**定义：**含有纯虚函数的类被称为抽象类

**纯虚函数**：

**意义**：基类中仅仅给出声明，不对虚函数实现定义，而是在派生类中实现。

**声明格式**：virtual <类型><函数名>(<参数表>)=0;

抽象类之所以**不能被实例化**，是因为它里面是抽象方法，实例化对象调用其里面的方法没有意义，**只要有纯虚函数就不能被实例化**

**子类不用全部实现抽象类**

## 11.继承

**被继承**的成员的访问权限都**不会大于基类**该成员的权限

比如在保护继承中，基类的public和protected 在子类都会变成 protected

**所有成员**都会被子类继承，只是因为权限问题，**有些成员不能访问（可以用sizeof对父类和子类进行判断）**

## 12.重载 重写 重定义

重载(overload):是函数名相同，参数列表不同，override只是在类的内部存在。

## 纯虚函数和虚函数

定义一个函数为虚函数，**不代表函数为不被实现的函数**。

定义他为虚函数是为了允许用基类的指针来调用子类的这个函数。

定义一个函数为**纯虚函数，才代表函数没有被实现**。

重写(override),也叫覆盖。子类重新定义父类中有相同名称和参数的虚函数(virtual)。在继承关系之间。C++利用虚函数实现多态。

重写的特点:

          1 被重写的函数不能是static的。必须是virtual的

          2 重写函数必须有****相同的类型，名称和参数列表****

          3 重写函数的****访问修饰符可以不同****。尽管父类的virtual方法是private的，派生类中重写改写为public,protected也是可以的。

          这是因为被virtual修饰的成员函数，无论他们是private/protect/public的，都会被统一放置到虚函数表中。对父类进行派生时，子类会继承到拥有相同偏移地址的虚函数标(相同偏移地址指的是各虚函数先谷底与VPTR指针的偏移)，因此就允许子类对这些虚函数进行重写

重定义(redefining)，也叫隐藏。子类重新定义父类有**相同名称**的非虚函数(参数列表**可以不同**)。

子类若有和父类相同的函数，那么，这个类将会隐藏其父类的方法。除非你在调用的时候，强制转换成父类类型。在子类和父类之间尝试做类似重载的调用时不能成功的。

# 3.公共知识

## 1.为什么栈比堆快

本质：栈由操作系统管理

1.**工作量不同**：在分配和销毁方面，**修改栈指针**就能完成功能。而堆分配需要寻找空间，而销毁又需要空间合并记录的操作。

**2.使用的变量不同：**栈分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，分配和释放的命令都比堆的简单

## 2.字节对齐

原因：

1. **为了兼容**，有些平台对变量的地址有严格的要求
2. **提高效率**，为了避免对同一个变量多次读取。

**对齐原则：**

每个变量的地址都要是该变量大小的整数倍

建议：

定义结构体时尽量，**从小到大**充分利用空间

## 3.在程序中凡是以“#”开始的语句行都是预处理命令行

大概分为四类：

头文件包含，条件编译，宏替换和布局控制