C/C++/算法复习

liz*

April 21, 2009

Contents

*Email: shengyan1985@gmail.com

1 C基础知识

1.1 数组和字符数组

1. 数组声明和初始化 float b[1][2][3] = 0.0;
2. 字符数组 char str[] = "china"; // 字符串常量尾部都有字符\0作为字符串结束符
3. getchar()和putchar() putchar(getchar())
4. puts(s) 输出字符串s // stdio.h
5. gets(s) 从标准输入中断接受以\n结束的串转换为\0后存入s,并返回串s的指针. // stdio.h
6. strcat(s1, s2) 将s1和s2合并为s1, 并返回s1地址 //string.h
7. strcpy(s1, s2) 将s2拷入s1中, 并返回s1地址 //string.h
8. strcpy(s1, s2, n) 将s2的前n个字符拷入s1中, 并返回s1地址 //string.h
9. strcmp(s1, s2) 比较s1, s2大小, 若相等, 返回0; 若不等, 返回第一个不等字符的ASCII码差值
10. strlen(s) 测试s的实际长度

1.2 函数

1. 函数参数从右到左计算, 形实结合是值传递.
2. 函数返回值类型(函数类型)缺省为整型, void定义无(空)类型, 表示函数不返回值.
3. 如果return返回类型于函数类型不一致, 以函数类型为准.
4. 但数组是地址传递, 因为数组名本质是一个地址. 这里的形参数组长度可缺省说明, 形实数组长度可不等, 但形参数组长不能大于实参数组长.
5. 函数允许嵌套调用, 不允许嵌套定义.

1.3 function相关算法整理

```c
#include <stdio.h>
#include <string.h>

/*******************
模拟strcat(s1, s2)功能
*******************/
char * mystrcat(char s1[], char s2[])
{
    int i = 0, j = 0;
    while (s1[i] != '\0') i++; //while (s[i]) i++;
    while ((s1[i]=s2[j]) != '\0')
    {
        i++;
        j++;
    }
    return s1;
}

/*******************
模拟strcpy(s1, s2)功能
*******************/
char * mystrcpy(char s1[], char s2[])
{
    int i;
    for (i=0; (s1[i]=s2[i]) != '\0'; i++); // for (i=0; (s1[i]=s2[i]); i++);
```

```
    return s1;
}

/*******************
字符串反转功能
*******************/
char * reverse(char ss[])
{
    int i, j, c;
    for (i=0, j=strlen(ss)-1; i<j; i++, j--)
    {
        c = ss[i];
        ss[i] = ss[j];
        ss[j] = c;
    }
    return ss;
}

/*******************
统计字符串中单词个数，以空格为分隔符
碰到空格后的第一个非空格字符，形如 _ A，表示出现一个单词
*******************/
int get_words_count(char ss[])
{
    int i, num=0, word=0;
    for (i=0; ss[i]; i++)
    {
        if (ss[i] == '\40') word = 0;
        else if (word == 0)
        {
            word = 1;
            num++;
        }
    }
    return num;
}

int main(void)
{
    char s1[10], s2[20];
    //scanf("%s", s1);
    //scanf("%s", s2);
    //puts(mystrcat(s1, s2));
    //puts(mystrcpy(s1, s2));
    //puts(reverse(s2));
    //printf("%d\n", get_words_count(gets(s2)));
    printf("%d %d\n", 12/7, 12%7);
    return 0;
}
```

1.4 变量存储

1. 变量生存期： 变量存储单元存在的时效.
2. 变量作用域： 变量值可被访问的范围.
3. 全局变量(extern)： 在任何函数体外定义的变量. 生存期为整个程序执行期，作用域为从定义点到本源文件结束(隐含规则). 如果是同一源文件中访问，在定义点前访问,需使用extern属性说明外部变量；在定义点后访问，可缺省extern说明. 如果是其他文件中访问，在引用外部变量的源文件开头处用extern说明外部变量.
4. 静态变量： 在变量定义前使用static属性说明
5. 静态局部变量： 在局部变量类型定义前用static定义. 作用域仅限于定义它的函数内访问(同auto)，生存期同整个程序执行期. 于是乎，在多次函数调用中，静态局部变量保持值的连续性.
6. 静态全局变量： 在全局变量定义处用static说明，生存期： 同整个程序执行期，作用域： 仅限于本编译单位(源文件，避免冲突??).

7. 寄存器变量： 使用register定义的变量，仅使用auto变量，不适用全局变量和静态变量． 它的值存防御cpu通用寄存器中，不在普通内存中，提高速度．

8. 内部函数： 在函数类型定义前加static属性说明，使得该函数的作用域为本文件内(编译单位内，避免冲突)

9. 外部函数： 在函数类型定义前加extern属性说明或缺省． 一个文件想要调用另一个文件中的函数，需对所调外部函数进行说明．

10. 静态函数： 使某个函数只在一个源文件中有效，不能被其他源文件所用，可以在函数前面加上static．

11. 变量的定义形式，完整为： 存储类别 数据类型 变量列表; //存储类别为： auto，static，register，extern
    函数的定义形式，完整为： 存储类别 数据类型 函数名(形参列表) 函数体
    静态变量或外部变量如未初始化，系统自动使其初值为0(对数值类型)或空(对字符型数据)，而自动变量或寄存器变量，则其初值为随机数据．

1.5 宏

1. # define 宏名标识符 字符串
   编译预处理包括的工作： 宏定义，文件包含，条件编译．

2. # undef 宏名 用于终止宏名作用域，宏的默认作用域是从定义点到本源文件结束．
   带参宏,,,#define 宏名(参数表) 含参数字符串 ,,, 宏调用时提供实际参数． 宏展开用实参替代宏名中形参，对#define行中的字符串从左到右进行置换，将置换后的串在宏调用处宏展开...

1.6 条件编译

```
1) # ifdef 标识符 //若标识符已用#define定义过，则只编译段1
2)       程序段1
3) [# else
4)       程序段2]
5) # endif
```

```
1) # if 表达式 //表达式值为真(非零)，则编译段1，否则编译段2
2)       程序段1
3) [# else
4)       程序段2]
5) # endif
```

附上所有预处理命令:

```
# define 宏名 字符串
# define 宏名(参数1, 参数2, ..., 参数n) 字符串
# undef 宏名
# include "文件名" (或<文件名>)
# if 常量表达式
# ifdef 宏名
# ifndef 宏名
# else
# endif
```

1.7 指针

1. 变量指针与直接访问
   变量指针： 变量的内存起始地址

直接访问： 按变量指针存取变量值的方式.

2. 指针变量和间接访问
指针变量： 存放指针值的变量
间接访问： 通过访问指针变量获得所访变量指针再访问变量的方式.
类型说明符 * 指针变量名1, *指针变量名2; 指针变量的类型特指其所指向变量的类型.
取址运算符&： 取出变量(常量是没有地址的)地址; 不能用整型量和任何非地址类型的数给指针变量赋值.
间接访问(指针)运算符* ,, *和&满足有结合性. 严禁使用未初始化指针变量.

3. 数组和指针
数组名代表数组指针, 是指针常量.
++p, p++, --p, p--, 若p=&a[i], 则++p, p++等价于&a[i+1], 表示指针变量加(减)所指元素内存字节数.
p+i等价于p=p+i*sizeof(arraytype), 即指针变量向前/后移i个数组元素.
arraytype a[n], *p=a; 则p+i=a+i=&a[i],,,*(p+i)=*(a+i)=a[i]=*&a[i],,,[]又称为变址运算符.
若二指针变量指向同一数组, 则可进行减法(获得相差个数)和比较运算.

4. 数组元素的访问

(a) 下标法,,a[i]形式 &a[i]等价于a+i,, a[i]等价于*(a+i)

(b) 指针变量法,,*(a+i)或*(p+i)形式 同上

(c) 带下标的指针变量,,p[i]形式 (a+i)获得数组a的第i个元素地址,,,p[i]虽然p是指针, 但仍然可以使用[]取第i个元素值.

(d) 指针变量增1运算 如程序中...

(e) 数组名作为函数参数 它传递的是地址, 实参和形参可以是数组名和指针变量...

(f) 多维数组和指针 对于二维数组, 数组名a代表的是首行首地址, a+1代表第一行的首地址, 因为现在的首元素是一个由4个整型元素所组成的一维数组. a+1的含义是a[1]的地址, 即a+4*2=...而a[0],a[1]既然是一维数组名, 则它们代表一维数组a[0]中第0列元素的地址, 即&a[0][0],&a[1][0], a[0]等价于*(a+0), a[0]+1等价于*(a+0)+1, 它们值为&a[0][1], a[i][j]可以通过*(a[i]+j), *(*(a +i)+j)取值.,,一定要记住*(a+i)和a[i]是等价的. a[i]从形式上看是a数组的第i个元素, 如果a是一维数组名, 则a[i]代表a数组第i个元素所占内存单元的内容, a[i]有有物理地址的, 占内存单元. 但如果a是二维数组, a[i]代表一维数组名, 它只是一个地址,a,a+i, a[i], *(a+i), *(a+i)+j, a[i]+j都是地址.
a为二维数组名, 指向一维数组a[0], 即第0行首地址. a[0], *(a+0), *a为第0行第0列元素a[0][0]的地址. 纵向的 a+i, &a[i]为第1行首地址,,,横向的.
所以a+1和*(a+1)是相同的地址,,,但一个行上的, 一个列上看的.
一维数组名是指向列元素的,,在指向行的指针前面加一个*, 就转换为指向列的指针. 反之, 在指向列的指针前面加上&就称为指向行的指针.
不要把&a[i]简单得理解为a[i]单元的物理地址, 因为不存在a[i]这样一个实际的变量, 他只是一种地址的计算方法, 能得到第i行的首地址, &a[i]和a[i]的值是一样的,但它们的含义不同. &a[i]或a+i指向行, 而a[i]或*(a+i)指向列.

5. 函数与指针
int (*p)(); // 声明p是一个指向函数的指针变量, 此函数带回整型的返回值, 不能写成int *p(); 这样就是声明了一函数了.
p = max; 赋给p为一个函数, 在使用时可以c=(*p)(a,b); 这相当于c=max(a,b);
对指向函数的指针变量进行p+n, p++等运算是无意义的.

6. 用指向函数的指针作为函数参数
例子在代码中.

7. 返回指针值的函数
类型名 * 函数名(参数列表),,,一样用, 只是返回的指针不要是局部的, 也就是不要过期了.

8. 指针数组和指向指针的指针
   类型名 * 数组名[数组长度]; 主要用在字符串数组中,如:
   char * name[] = {"follow me", "Basic", "other"}; // name就是为char **, 这和main函数的参数类表(int argc, char * argv[])的argv是一个道理.
   char ** p;// *的从右到左的结合性, 可以看成是*(*p) 表示指针变量p是指向一个字符指针变量, p = name+2, 表示name[2].

1.8 array相关算法整理

```c
#include <stdio.h>
#include <string.h>

/*******************
统计s数组中串的实际长度
*******************/
int get_string_real_count(char *ss)
{
    char * ps=ss;
    while (*ps != '\0') ps++;
    return ps-ss;
}

/*******************
字符串与指针
*******************/
int string(void)
{
    char * string="some string"; // 这个只是将"some string"的首字符地址赋给string指针, 其指向的是一字符类型.
    char * string1;
    char str[20]={"some string"}; // 数组初始化, 之后就能整体赋值了. str[]="some string"是不对的.
    //*string1 = "some string"; // 这才是真正的将字符串赋给string1指针. 不过测试出来不对阿...
    printf("%s\n", string);
    printf("%d\n", sizeof(string));
    //printf("%s\n", string1);

    char * a, str2[10];
    a = str2;
    scanf("%s", a);  //这样做比较安全, 比不加a = str2;好, 因为光光char * a中a的值是随机的, 有可能指向已用的内存段.

}

// 调用时可以sub(f1, f2);这在每次调用sub函数时, 要调用的函数不是固定的, 就很方便了.
void sub(int (* x1)(int), int (* x2)(int, int)) //x1指向的函数有一个整型形参, x2指向的函数有两个整型形参
{
    int a, b, i, j;
    a = (* x1)(i);
    b = (* x2)(i,j);
}

//在举个函数指针的例子
int max(int, int); //两个数的较大者 先作声明, 因为在process中使用max作为形参, 表明max是函数, 而不是普通变量.
int min(int, int); //两个数的较小者
int add(int, int); //两个数的和

//现在定义一个process, 要求可以实现不同功能...调用时process(10, 10, max), process(10, 10, add);
void process(int x, int y, int (* func)(int, int))
{
    int result;
    result = (*func)(x,y);
    printf("%d\n", result);
}

void mainmain(int argc, char * argv[]) // 这个argv值可变, 而不是像之前的数组名常量, 因为形参是变量
{
    while (argc-->1) printf("%s\n", *++argv);
    while (--argc>0) printf("%s%c", *++argv, (argc>1)?' ':'\n');
```

```
}
int main(void)
{
    char s1[10];
    int a[10], i, *p;

    /*scanf("%s", s1);
    printf("%d\n", get_string_real_count(s1));

    //指针变量增1运算,,, 运行速度会比(a+i)等快
    for (i=0; i<10; i++) scanf("%d", (a+i));
    for (p=a; p<(a+10); p++) printf("%d", *p);
    p = a;
    for (i=0; i<10; i++) printf("%d", *p++); */
    string();
    printf("\n");
    return 0;
}
```

1.9 一些关于指针的注意点

1. 指针类型变量和整型变量不能相互赋值
2. 指针变量可以为空值，即该指针变量不指向任何变量，p=NULL，这个NULL在stdio.h中符号常量为0，可以与NULL值进行相等性比较
3. 两个指针变量比较，同一数组中的两个指针p，进行<，>比较． 不在同一数组中的两个指针比较是没有意义的．
4. 关于void *，就是不指定它是指向哪一种类型数据的． "ANSI C标准规定用动态存储分配函数时返回void指针，它可以用来指向一个抽象的类型的数据，在将它的值赋给另一指针变量时要进行强制类型转换使之适合于被赋值的变量的类型．"
   p1 = (char *)p2; or p2 = (void *)p1; 强制转换成需要的类型．

1.10 结构体

struct 结构体名
{ 成员列表 };
struct 结构体名 // 或者结构体名直接省略
{ 成员列表 } 变量名表列;
指向结构体变量的指针 struct student *p = &stu; 使用p->name，或者(*p).name，或者stu.name都可以访问成员变量．

1.11 共用体

现在想起来，这个东西是多么稀有的东西．
union 共用体名
{ 成员表列 } 变量表列;
共用体是使几种不同类型的变量存放到同一段内存单元中． 变量所占内存的字节数不同，但都是从同一地址开始，使用覆盖技术，几个变量相互覆盖，使几个不同的变量共占同一段内存的结构．
a.i访问共用体a中的变量i． 多个变量在一个时间点上只有一个有意义，比如，a.f= 0.0，将会冲掉之前的a.i值，所以这时访问a.i是没有多大意义的．

1.12 枚举类型

声明: enum weekday { sun, mon, tue, wed, thu, fri, sat }; {}中的为枚举常量，顺序从0开始
定义枚举变量: enum weekday myweekday=mon;//赋为mon值

也可以直接: enum { sun, mon, tue, wed, thu, fri, sat } myweekday;
改变枚举元素的值: enum { sun=7, mon=1, tue, wed, thu, fri, sat } sw;
if (sw == mon) ...
if (sw < sun) ...
sw = (enum weekday)2; //将序号为2的枚举元素赋给sw, 相当于sw = tue, sw = (enum weekday)(5-3);

1.13 用typedef定义类型

typedef int INTEGER;
typedef struct ... DATE;
typedef int NUM[100]; NUM n;
typedef char * STRING; STRING p, s[10];
typedef int (* POINTER)(); POINTER p1;
typedef和#define的不同: typedef是编译时处理的, #define是在预编译时处理的,作简单字符串替换; 前者是语句, 后者是宏命令.

1.14 位运算

&: 位与
|: 位或
^: 位异或
: 取反
<<: 左移
>>: 右移
相关组合, 可以形式循环移位等.

1.15 文件

1. fopen: FILE * fp; fp = fopen(文件名, 使用文件方式); 文件打开方式: r(只读), w(只写), a(追加)/文本文件, rb, wb, ab/二进制文件, r+/读写代开一个文本文件, w+/读写建立一个新的文本文件, a+/读写打开一个文本文件, rb+, wb+, ab+
   打开失败, fopen函数带回一个出错信息, fopen返回的是一个NULL.

2. fclose: fclose(文件指针);

3. fputc: 把一个字符写到磁盘文件上去.
   一般调用形式: fputc(ch, fp);
   作用: 将字符ch输出到fp所指向的文件中去. 如果输出成功, 返回这个输出的字符, 失败, 返回EOF(stdio.h中的符号常量, 值为-1).
   之前的putchar是由fputc派生出来的. #define putchar(c) fputc(c, stdout) 这将c输出到标准输出.

4. fgetc: 从指定文件读入一个字符, 该文件必须是以读或读写方式打开.
   调用形式: ch = fgetc(fp); 如果在执行fgetc读字符时遇到文件结束符, 函数返回一个文件结束标志EOF(-1), 这可作为判断文件结束条件.

5. fread和fwrite: 读写一个数据块
   调用形式: fread(buffer, size, count, fp); fwrite(buffer, size, count, fp); buffer是一个指针, 对fread来说,是读入数据的存放地址, 对fwrite来说, 是要输出数据的地址. size为读写的字节数, count要进行读写多个size字节的数据项. 如果文件以二进制形式打开, 用fread和fwrite就可以读写任何类型的信息. 如fread(f, 4, 2, fp); 表示从fp所指向的文件读入2个2个字节的数据并存储到数组f中. 对于复杂的结构体类型, 同样可以使用这种方式整个读写一个结构体.
   fread和fwrite调用成功, 返回值为count的值, 即输入或输出数据项的完整个数.

6. fprintf和fscanf: 和printf, scanf函数作用相仿, 都是格式化读写函数, 只是前者的读写对象不是终端而是磁盘文件.
   调用形式: fprintf(文件指针, 格式字符串, 输出列表); fscanf(文件指针, 格式字符串, 输入列表);
   使用fprintf和fscanf函数, 由于在输入时要将ASCII码转换为二进制形式, 在输出时又要将二进制形式转换成字符(--难道只能是读写文本文件, 好像是的哦), 花费时间较多. 因此在内存与磁盘频繁交换数据的情况下,最好是使用fread和fwrite函数.

7. putw和getw: 用来对磁盘文件读写一个字(整数, 一般为2个字节, 但根据系统不同又所差别).
   putw(10, fp); 将整数10输出到fp指向的文件. 而i = getw(fp); 从磁盘文件读一个整数到内存, 赋给整型变量i.

8. fgets和fputs:从指定文件中读入一个字符串/向指定文件中输出一个字符串.
   fgets(str, n, fp); n为要求得到的字符, 但只从fp指向的文件输入n-1个字符, 最后加一个'\0', 放于字符数组str中, 如果在读完n-1个字符之前遇到换行符或EOF, 读入马上结束. fgets返回值为str的首地址.
   fputs("sring", fp); 字符串末尾的'\0'不输出到文件, 若输出成功, 函数值为0, 失败时, 为EOF.

9. rewind:使位置指针重新返回文件的开头, 没有返回值.
   rewind(fp); 它使文件的位置指针重新定位于文件开头, 并使feof函数的值恢复为0(假).

10. fseek: 对流式文件可以进行顺序读写, 也可以进行随机读写. 关键在于控制文件的位置指针, 如果位置指针是按照字节位置顺序移动的, 就是顺序读写. 如果将位置指针按需要移动到任意位置, 就可以实现随机读写, 读写完上一个字符(字节)后, 并不一定要读写其后续的字符(字节), 可以读写文件中任意位置上所需要的字符(字节). fseek正是实现了改变文件的位置指针.
    调用形式: fseek(文件类型指针, 位移量, 起始点); 起始点有 文件开始(SEEK_SET, 0), 文件当前位置(SEEK_CUR, 1), 文件末尾(SEEK_END, 2); 位移量以起始点为基点, 向前移动的字节数, long型数据.
    fseek函数一般用于二进制文件, 因为文本文件要发生字符转换, 计算位置时往往会发生混乱.
    fseek(fp, i*sizeof(struct student_type), o); 然后fread(&stud[i], sizeof(struct student_type), 1, fp);

11. ftell: 获得流式文件中的当前位置, 用相对于文件开头的位移量来表示.
    i = ftell(fp); 若返回-1L表示出错.

12. ferror和clearerr: 在调用上述输入输出函数时, 如果出现错误, 除了返回数值反映外, 还可以用ferror来检查
    ferror(fp); 如果返回值为0(假), 表示未出错. 如果返回一个非零值, 表示出错. 对同一文件每一次调用输入输出函数, 均产生一个新的ferror函数值, 所以应当在调用一个输入输出函数立即检查ferror函数的值, 否则会丢失信息. 在执行fopen函数时, ferror函数的初始值置为0;
    clearerr使文件错误标志和文件结束标志置为0. 只要出现错误标志, 就一直保留, 直到对同一文件调用clearerr或rewind函数.

1.16 文件相关算法整理

```c
#include <stdio.h>
#include <string.h>

/*******************
读取一个文本文件
*******************/
void read_text_file(void)
{
    char ch;
    FILE * fp;
    fp = fopen("test", "r");
    ch = fgetc(fp);
    while (ch != EOF)
    {
```

```
        putchar(ch);
        ch = fgetc(fp);
    }
}

/********************
顺序读取一个二进制文件
********************/
void read_text_file(void)
{
    char ch;
    FILE * fp;
    fp = fopen("test", "rb");
    while (!feof(fp)) //feof值为0时表示未到文件结束，读入一个字节的数据赋给整型变量c，遇到文件结束时，
       feof为1. 这种方法也适用于文本文件.
    {
        ch = fgetc(fp);
        //...
    }
    fclose(fp);
}

/********************
自定义的getw函数
********************/
int getw(FILE * fp)
{
    char * ch;
    int i;
    s = (char * )&i;
    s[0] = getc(fp);
    s[1] = getc(fp);
    return i;
}

/********************
自定义的putw函数
********************/
int putw(int i, FILE * fp)
{
    char * s;
    s = (char * )&i;
    putc(s[0],fp);
    putc(s[1],fp);
    return i;
}

/********************
在系统不提供fread和fwrite时，编写任何类型的读写文件函数，这里putfloat，将一个浮点数写入文件
********************/
int putfloat(float num, FILE * fp)
{
    char * s;
    int count;
    s = (char * )&num;
    for (count=0; count<4; count++) putc(s[count], fp);
    return 1;
}
```

2 C++基础知识

2.1 基础之基础

1. 常量: const float pi = 3.14; const常量和#define是有区别的. const是程序中的常量, define定义的宏在编译预处理时已被替换掉, 不是一个具有一定类型的常量名.
2. I/O格式: I/O流的常用控制符, 用于控制输出,,, iomanip.h中, 较多略. page 24.
3. 赋值表达式也具有值, 具体为赋值符左边表达式的值
4. 不同类型间的转换, 转换总是朝表达数据能力更强的方向, 并且转换总是逐个运算符进行的.
5. do ... while
   do
   sth
   while (条件);

6. switch
   switch (表达式)
   case 常量表达式1: 语句组1
   ...
   default: 语句组n+1

   case中不加break, 会依次执行下面的case直到整个switch退出.
7. break&continue

2.2 程序的内存区域

程序内存空间可以分为代码区, 全局数据区(程序全局数据和静态数据), 堆区(动态分配的数据), 栈区(函数的局部数据, 函数调用现场)
全局变量由编译器建立, 初始化默认为0.
局部变量, auto修饰(一般省略), 在栈中分配空间.
静态局部变量, 存放在内存全局数据区, 始终驻留在全局数据区, 直到程序运行结束.

2.3 递归函数

包含一个递归终止条件, 递归内容相同.

2.4 内联函数

主要是解决程序的运行效率, 函数调用需要建立内存环境, 进行参数传递, 并产生程序执行转移, 这些工作需要一些时间开销. 将函数声明为inline, 编译器为该函数创建一段代码, 以便在后面每次碰到该函数的调用都用相应的一段代码来替换. 需要在声明时使用inline才能使其为内联函数. 内联函数不能含有负责的结构控制语句, 如switch和while, 如果内联函数有这些语句, 则编译将该函数视同普通函数那样产生函数调用代码. 递归函数不能做内联函数. 一般实际使用的很少.

2.5 重载函数

对于在不同类型上作不同运算而又用同样的名字.
匹配顺序: 1) 寻找一个严格的匹配, 2) 通过内部类型转换寻找一个匹配, 3) 通过用户定义的转换,,,查找到的需要是唯一性, 不然会出现二义性编译错误.
重载函数至少在参数个数, 参数类型或参数顺序上有所不同.
重载函数的内部实现, C++使用名字粉碎方法, 来改变函数名,,使得各个重载函数名还是不一样.

2.6 默认参数

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，那么默认参数才可出现在函数定义中． 声明时的默认参数，如 void point(int = 3, int = 4); 比较怪异，所以还是只用光光定义吧
多个默认参数时，形参分布中，默认参数应从右至左逐渐定义，当调用函数时，只能向左匹配参数．
默认参数和参数个数的重载函数可以相互替换．
默认值可以是全局变量，全局常量，或者是一个函数． 但默认值不可以是局部变量，因为默认参数的函数调用是在编译时确定的． 而局部变量的位置与值在编译时均无法确定....像这样类型的东西,,,考虑时要分清是运行时，还是编译时．

2.7 作用域与可见性

可见性从另一角度表现标识符的有效性，标识符在某个位置可见，表示该标识符可以被引用． 可见性和作用域是一致的，作用域指的标识符有效的范围，而可见性是分析在某一位置标识符的有效性．
如果被隐藏的是全局变量，则可用符号::来引用该全局变量．

2.8 生命期

静态生命期与程序的运行期相同，随程序开始而开始，结束而结束． 主要有全局变量，静态全局变量，静态局部变量．
局部生命期，始于声明点，结束于其作用于结束处，栈区． 具有局部生命期的变量也具有局部作用域，反之不成立．
动态生命期，堆区． 开始于malloc()或new()，结束于free()或delete()．

2.9 头文件

包含： 类型声明，函数声明，内联函数定义，常量定义，数据声明，枚举，包含指令(可嵌套)，宏定义．
不宜包含： 一般函数定义，数据定义，常量聚集定义(如 const int c[]={1,2,3})．

2.10 指针

一定要记住指针不是整型数．
不要将int * iPtr = &iCount; 理解为 * iPtr = &iCount; 后者是错误的，*iPtr是指针指向的变量． 地址和整型虽然都是4个字节,但是不能随意转换．
指针是有类型的，给指针赋值，不但必须是一个地址，而且应该是一个与该指针类型相符的变量或常量地址． C/C++是强类型语言．
指针具有一定类型，它是值为地址的变量，该地址是内存中另一个该类型变量的存储位置． 或者说指针是具有某个类型的地址．

2.11 堆内存分配

堆允许程序在运行时(而不是在编译时)，申请某个大小的内存空间． 程序编译时的数组大小一定是已知的． 而在编译和连接时不予确定这种在运行中获取的内存空间，这种内存环境随着程序运行的进展而时大时小，这种内存就是对内存，所以堆内存是动态的．
C++中的新的，new与delete操作符． 如： array = new int[arraysize]; 失败为NULL，成功为指针; delete [] array; 具体： new的操作数为数据类型，它可以带初始化值表或单元个数．new返回一个具有操作数之数据类性的指针． delete的操作数是new返回的指针，当返回的是new分配的数组时，应该带[]．

2.12 const指针

- 指向常量的指针(常量指针), const int * pi = &a; //指针变量指向的是个常量, 不能修改, 但指针变量本身可变. 这主要用于作为原数据的数组遭到破坏, 特别是函数参数时, 申明形参为常量, 使得在函数内部不能修改这个常量形参, 如 void fun(const char * source), 指明* source不能作为左值.
- 指针常量, char * const pc = "asdf"; 表示指针本身是常量, 在定义指针常量时必须初始化. 但该指针指向的变量是不受指针常量的约束, 即*pc = 'b';
- 指向常量的指针常量, const int * const cpc = &ci; 不允许修改指针值cpc, 也不允许修改* cpc的值.

2.13 函数与指针

- 数组作为实参, 其对应的形参是一个指针, 可以作为左值. 也就是两个指针地址是一样的, 指向的是同一块内存区域, 这就是所谓的址传递.

- 返回指针的函数称为指针函数, 注意: 指针函数不能把在它内部说明的具有局部作用域的数据地址作为返回值. 可以返回堆地址, 可以返回全局或静态变量的地址, 但不要返回局部变量的地址.
- void指针, 空类型指针, 不指向任何类型, 仅仅是一个地址, 不能进行指针运算, 不能间接引用. 空类型指针和非空类型指针间的相互转换也一定要指明类型....一定要提醒自己, C/C++是强类型语言, 访问时必须明确类型, 不管是隐式(非空类型可隐式转换为空类型)还是显式转换(空类型到非空类型得强制转换)都得明确.

2.14 字符与指针

"join" == "join" 是比较的两个字符串常量地址, 地址是不同的, 字符串常量存放在内存全局数据区的常量const区.
char * pc = "hihihi"; *(pc+1) = 'h';
指针可强制转换为整型输出.

2.15 NULL指针值

NULL是空指针值, 它不指向任何地方. 不同的操作系统, 会使编译去不同的NULL值. 因此NULL是个不确定值.

2.16 函数指针

int (*func) (char a, char b); // 指向函数地址的指针.
调用时: func(a, b); 或(* func)(a, b);
函数代码也占内存, 存放在代码区, 所以每个函数具有地址. 一个函数不能赋给 不一致的函数指针.
函数指针与其他数据类型的指针尽管都是地址, 但在类型上有很大的差别. 两类指针之间的不允许互相赋值, 显式转换也不行. 因为函数指针指向程序的code区, 是程序运行的指令代码, 而数据指针指向data数据区, stack栈区和heap堆区.
typedef int (* FUNC)(int a, int b);
FUNC funp; // 声明一个函数指针.
double sigma(double (*func)(double), double d1, double d2) ... // 函数指针作为函数参数.
调用时, 使用sigma(sin, 0.1, 1.0); //奇怪函数指针所代表的形参是哪边传入的呢?

分析一下下面这个例子:
typedef int (* SIG)();
typedef void (* SIGARG)();
SIG sinal(int, SIGARG); // 定义返回函数指针的函数. 第二个参数为一函数指针.

2.17 引用

引用就是给变量取个别名,,,外号 , 当建立引用时, 程序用另一个变量或对象(目标)的名字初始化它.
int someInt; int & rInt = someInt;
引用不是值, 不占存储空间, 声明引用时, 目标的存储状态不会改变, 所以引用仅是声明, 不需要定义.
引用在声明时必须被初始化, 否则会产生编译错误.
若一个变量声明为T&, 它必须用T类型的变量或对象, 或是能够转换成T类型的对象进行初始化. 如果引用类型T的初始值不是一个左值, 那么将建立一个T类型的目标并用初始值初始化, 那个目标的地址变成引用的值. 关于引用的类型转换: 首先作必要的类型转换, 将结果置于临时变量, 把临时变量的地址作为初始化的值.
对void进行引用是不允许的. 不能建立引用的数组, int & ra[10] = a; 是错误的. 引用本身不是一种数据类型, 所以没有引用的引用, 也没有引用的指针. 引用不能用类型来初始化. 有空指针, 但无空引用...这么多条规定, 其主要还是由于引用不是真正的占内存单元, 仅是一个别名而已.
传递引用作为函数参数可以达到和传递指针一样的效果. 但这里当在函数重载时, 有时会出现歧义. fn(a)是匹配void fn(int s); 还是void fn(int & s);呢??
函数返回值时, 要生成一个值的副本, 而用引用返回值时, 不生成值的副本. float & fn(float r) return temp;;
使用引用可以让函数返回多个值.
函数调用作为左值,,,只要保证引用对象可用, 避免将栈中变量的地址返回..这里还有和变量作用域, 生成期有关的. 只要保证被引用的变量没有在函数退出后消失, 栈空间消失, 该变量就可以被使用.
传递实参不被修改的办法是传递const指针和引用. 这使得在函数内不能改变实参. double & fn (const double & pd);
对引用的初始化, 可以是变量, 可以是常量, 也可以是一定类型的堆空间变量. 但注意, 引用不能为NULL, 所以new 空间需要验证是否为NULL. 对使用堆的引用, 有注意: 必要时使用值传递参数, 必要时返回值, 不要返回有可能退出作用域的引用, 不要引用空目标.

2.18 数组和结构体

声明一个结构并不分配内存, 内存分配发生在定义这个新数据类型的变量中.
数组是一个数据类型的聚集, 它本质上不是数据类型. 不同结构名的变量是不允许相互赋值的, 即使两者包含有同样的成员.
不要返回一个局部结构变量的引用或指针.

2.19 类

1. ::为作用域区分符, 指明一个函数属于哪个类或一个数据属于哪个类,, ::可以不根类名, 表示全局数据或全局函数(即非成员函数).
2. 名空间, 指某名字在其中必须唯一的作用域.
3. 构造函数: 没有返回类型, 函数体中也不允许返回值, 但可以无值返回语句return; 专门用于创建对象和为其初始化, 所以不要随意被调用. 如果一个类对象是另一个类的数据成员, 则在那个类的对象创建所调用的构造函数中, 对该成员(对象)自动调用其构造函数. 构造函数可以重载.
   Student(char * pName="no name", int ssID=0):id(ssID)... 冒号表示后面要对类的数据成员的构造函数进行调用.
4. 析构函数: 也是特殊的类成员函数, 没有返回类型, 没有参数, 不能随意调用, 也没有重载, 只是在类对象生命期结束的时候, 由系统调用. 析构函数以调用构造函数相反的顺序被调用.

5. 构造对象的顺序
   局部(块作用域)和静态对象(文件作用域)，以声明的顺序构造.
   静态对象只被构造一次，文件作用域的静态对象在主函数开始运行 全部构造完毕. 块作用域中的静态对象，则在首次进入到定义该静态对象的函数时，进行构造.
   所有全局对象都在主函数main()之前被构造，构造时无特殊顺序.

2.20 堆与拷贝构造函数

全局变量，静态数据，常量存放在全局数据区，所有类成员函数和非成员函数代码存放在代码区，为运行函数而分配的局部变量，函数参数，返回数据，返回地址等存放在栈区，余下的空间都被作为堆区.
堆对象的作用域是整个程序生命期，所以除非程序运行完毕，否则堆对象作用域不会到期. 堆对象析构是在释放堆对象语句delete执行之时.
拷贝构造函数，Student(Student & s) ...
深拷贝与浅拷贝: 主要区别是对于所占资源的占用情况，深拷贝是复制一份资源(堆内存，硬件资源等)，浅拷贝只复制相关引用(可以说是引用或地址，资源还是只有一份).

2.21 临时对象和无名对象

略

2.22 静态成员

加上static前缀，int Student::noofStudents = 0; 在类外初始化静态数据成员.
静态数据成员用得比较多的场合一般为:

- 用来保存流动变化的对象个数;
- 作为一个标志，指示一个特定的动作是否发生，如可能创建几个对象，每个对象要对某个磁盘文件进行写操作，但显然在同一时间里只允许一个对象写文件，在这种情况下，用户希望说明一个静态数据成员指出文件何时正在使用，何时处于空闲状态.;
- 一个指向一个链表第一个成员或最后一个成员的指针.

静态成员函数，属于类，不存在this指针，静态成员函数只能访问静态数据成员或全局变量，不能对类的非静态成员进行访问.

2.23 友元

类中，friend 函数类型 函数名(参数); 友元函数，也可以是友元成员，友元类，只要加friend即可声明某个类的友元是什么. 友元类的话，友类的每个成员函数都可以访问另一个类中的所有(保护或私有)数据成员.

2.24 继承

class Student: public People ...
多态，在运行时，能依据其类型确认调用哪个函数的能力，成为多态性，或称迟后联编; 而在编译时就能确定哪个重载函数被调用的，称为先期联编.
虚函数，virtual关键字 ，只有类的成员函数才能说明为虚函数，因为仅适用于有继承关系的类对象; 静态成员函数不能是虚函数，因为静态成员函数不受限于某个对象; 内联函数不能是虚函数; 构造函数不能是虚函数; 析构函数可以是虚函数.
抽象类(abstract class)，至少具有一个纯虚函数，如virtual void Withdrawal(float amount) = 0; //纯虚函数 ... 一个抽象类不能有实例对象，即不能由该类抽象来制造一个对象.

不能创建一个抽象类的对象，但可以声明一个抽象类的指针或引用.
纯虚函数最主要的作用就是，在基类中为子类保留一个位置，以便子类用自己的实在函数定义来重载它，如果在基类中没有保留位置，则就没有重载.

2.25 多继承

虚拟继承： 如果某个类继承的两个基类都继承了同一个最基类，使用虚拟继承就不会用两个最基类对象.
多继承的构造顺序： 任何虚拟基类的构造函数按照它们被继承的顺序构造；任何非虚拟基类的构造函数按照它们被继承的顺序构造；任何成员对象的构造函数按照它们声明的顺序调用；类自己的构造函数.

2.26 C++中相关代码

```
#include <iostream.h>

void fn()
{
    int & a = new int(2); // a不是指针
    int * pInt = new int;
    if(pInt == NULL){
        return;
    }
    int & rInt = *pInt;
    //...
    delete &pInt;
    delete &a;
}

class SleeperSofa: public Bed, public Sofa
{
    public:
        SleeperSofa():Sofa(), Bed(){}
};

class Sofa: virtual public Furniture //虚拟继承
{
    public:
        Sofa(){}
};
void main()
{
    int a, b;
    cin>>a>>b;
    cout<<"hi~"<<endl;
}
```

3 C/C++库函数使用

3.1 malloc函数

函数原型: void * malloc(unsigned int size);
作用: 在内存的动态存储区中分配一个长度为size的连续空间, 返回的是一个指向该分配域起始地址的指针(类型为void), 如果分配失败(如内存空间不足), 则返回空指针(NULL). 返回的void * 常被强制转换为自己需要的指针类型.
C++: void * malloc(size_t size); // size_t即为unsigned long. alloc.h
例如: array = (int *)malloc(arraysize * sizeof(int));

3.2 calloc函数

函数原型: void * calloc(unsigned n, unsigned size);
作用: 在内存动态存储区中分配n个长度为size的连续空间, 返回的是一个指向该分配域起始地址的指针(类型为void), 如果分配失败(如内存空间不足), 则返回空指针(NULL). 用calloc函数可以为一维数组开辟动态存储空间, n为数组元素个数, 每个元素长度为size.

3.3 free函数

函数原型: void free(void * p);
作用: 释放由p指向的内存区, p是最近一次调用calloc或malloc函数时返回的值. free函数无返回值.

3.4 memset函数

函数原型: void * memset(void *, int, unsigned);
作用: 可以一字节一字节地把整个数组设置为一个指定的值, mem.h中, 参数依次为数组的起始地址, 设置数组每个字节的值, 数组长度(字节数, 不是元素个数). memset(ia1, 0, 50*sizeof(int));

3.5 memcpy函数

函数原型: void * memcpy(void * d, const void * s, size_t n); //size_t 为unsigned int. mem.h中, 将s指向的内容拷贝到d指向的内容中,

3.6 strcmp函数

函数原型: int strcmp(const char * str1, const char * str2);

3.7 strcpy函数

函数原型: char * strcpy(char * dest, const char * src); // 不能直接对字符数组赋予一个字符串, 因为数组名是常量指针, 不是指针.
strcpy仅能对以'\0'作结束符的字符数组进行操作. 若要对其他类型的数组赋值可调用函数memcpy();

3.8 qsort函数

函数原型: void qsort(void *, size_t nelem, size_t width, int(*fcmp)(const void *, const void *));

作用： 标准库函数qsort()可对任何类型的数组排序． 在stdlib.h． 第一个参数为待排序数组，nelem是数组元素个数，widht是元素类型的长度，fcmp是函数指针，比较两个参数，如果相等，则返回0，如果参数1大于参数2，则返回值为正，否则返回值为负．