C语言

目录

[一些感悟： 2](#_Toc4276543)

[C语言的历史： 2](#_Toc4276544)

[术语： 2](#_Toc4276545)

[内存对齐： 3](#_Toc4276546)

[左值和右值： 3](#_Toc4276547)

[32个关键字： 3](#_Toc4276548)

[register： 3](#_Toc4276549)

[Static： 4](#_Toc4276550)

[Sizeof： 4](#_Toc4276551)

[Bool： 4](#_Toc4276552)

[Case： 5](#_Toc4276553)

[Void： 5](#_Toc4276554)

[return： 5](#_Toc4276555)

[Const： 5](#_Toc4276556)

[Volatile： 5](#_Toc4276557)

[Extern： 5](#_Toc4276558)

[Struct： 6](#_Toc4276559)

[Union： 8](#_Toc4276560)

[Enum： 8](#_Toc4276561)

[Typedef： 9](#_Toc4276562)

[Define： 10](#_Toc4276563)

[C语言中关于字符的批处理： 12](#_Toc4276564)

[memcpy： 12](#_Toc4276565)

[menset： 12](#_Toc4276566)

[符号： 13](#_Toc4276567)

[做oj的时候的调试技巧 14](#_Toc4276568)

[c语言数组： 14](#_Toc4276569)

[指针： 15](#_Toc4276570)

[原码，反码，补码： 17](#_Toc4276571)

[链表及其应用： 17](#_Toc4276572)

[双向链表： 18](#_Toc4276573)

[位段/位域： 18](#_Toc4276574)

[内存管理： 19](#_Toc4276575)

[野指针： 19](#_Toc4276576)

[栈，堆和静态区： 19](#_Toc4276577)

[常见的内存错误： 19](#_Toc4276578)

[文件（流）： 20](#_Toc4276579)

[文件与流 20](#_Toc4276580)

[文件的打开模式： 21](#_Toc4276581)

[函数的详解: 24](#_Toc4276582)

[使用C语言中的time函数获取系统时间 42](#_Toc4276583)

[C语言中system()函数的用法总结 42](#_Toc4276584)

[基于C语言sprintf函数的深入理解 45](#_Toc4276585)

[自定义函数 51](#_Toc4276586)

[程序设计中的函数 51](#_Toc4276587)

[推荐的几个词典类型的网站： 52](#_Toc4276588)

[函数： 53](#_Toc4276589)

[函数指针变量的声明 53](#_Toc4276590)

[通过函数指针变量调用函数 53](#_Toc4276591)

[调用函数的其它书写格式 54](#_Toc4276592)

[定义某一函数的指针类型 55](#_Toc4276593)

[函数指针作为某个函数的参数 56](#_Toc4276594)

# 一些感悟：

我发现有一句话说的很对：（在弄明白为什么的时候）少使用printf，应该多使用调试框里面的每个变量变化的情况，了解他们是怎么一步一步变化而来的。过分依赖printf会导致被表面的现象迷惑，仅仅可能是结果是对的，而放弃了深究为什么是这个样子的。事实上，printf也是一个函数，我们得到的只是它的一个返回值。

# C语言的历史：

C语言是AT&T贝尔实验室的D.M.Rotchie在1973年推出的程序开发语言。C语言是高级语言，但是c语言具有能够和汇编语言相媲美的低层处理（内存操作以及位操作）。

# 术语：

定义：创建一个对象，为这个对象分配一个内存空间并给他取上一个名字，这么名字就是我们所说的变量名。

注：这个名字会一直和这块内存区域连在一起，不会改变，且同一个变量只能定义一次。

声明：表示这个变量名的名字已经被预定了。

区别：定义创建了对象并且分配了内存，声明没有分配内存

大端模式(big\_endian）：字数据的高字节存储在低地址中，字数据的低字节存储在高地址中。

小端模式(little\_endian)：字数据的高字节存储在高地址中，字数据的低字节存储在低地址中。

# 内存对齐：

字，双字，四字在自然边界上不需要在内存中对齐，因为他们自然边界分别是偶数地址，可以被4整除的地址，可以被8整除的地址。为了提高程序的性能，数据结构（尤其是栈）应该尽可能的在自然边界上对齐。因为如果为了访问未对齐的内存，处理器需要做两次内存访问，然而，对齐的内存访问只需要一次访问。

一个字后者两个字操作数跨越了4字节边界，或者一个四字操作数跨越了8个字节边界，被认为是未对齐的，从而需要两次总线周期来访问内存。一个字起始地址是奇数但却没有跨越字边界被认为是对齐的，能够在一个总线周期内被访问。某些操作双四字的指令需要内存操作数在自然边界上对齐。如果操作数没有对齐，这些指令将会产生一个通用保护异常。需要额外的内存总线周期来访问内存中未对齐的数据。

如何避免内存对齐的影响：

将每个成员都对应在自然边界上，从而避免了编译器自动对齐。而使用预编译指令#pragma pack(X)//编译器将按照n个字节对齐

# 左值和右值：

简单的来说，出现在=左边的就叫 左值，出现在=右边的就叫右值

可修改的左值：我们能给非只读性的变量赋值

编译器会认为数组名作为左值代表的意思是a的首元素的首地址，但是这个地址开始的一块内存是一个总体，只能对数组的一个个元素进行访问而不能访问数组的整体。所以能够把a【1】当作左值而不能把a当作左值。

# 32个关键字：

auto,int,double,long,char,float,short,signed,unsigned,struct,union,enum,static,switch,case,default,break,register,const(申明只读变量）,volatile（变量在程序执行中可被隐含地改变）,typedef（给数类型取别名）,extern（引用变量，申明变量在其他文件正申明）,return,void,continue,do,while,if,else,goto,sizeof,for

register：

register：最快的关键字

请求编译器将变量（尽可能的）存储在CPU内部寄存器而不是通过内存寻址访问，提高了效率。//因为寄存器能够存储的变量有限

注：寄存器：（一个中转站）

寄存器实际上也是一块一块的存储空间，但是他的读取速度要比内存快上很多。数据从内存取到寄存器，CPU再从寄存器里面拿取数据。

使用register时候的注意要点：

Register的值必须是一个单个的值，而且他的大小必须是小于或者等于整形的长度因为register很可能都在寄存器中，所以不能使用寻地址符&来获取register变量的地址（因为这个是在内存中使用的）。

Static：

使用static修饰变量不是改变它的存储方式，而是表示它的作用域仅限于该文件中，所以又称为内部函数。好处是多个人在编写函数的时候，不用担心变量名字被其他人重复使用（c++）

静态变量分为局部变量和全局变量，他们都存在内存的静态区。

静态全局变量：作用域仅限于变量被定义的文件中

静态局部变量：作用域仅限于被定义的函数中。在该函数外不能被相应的使用，但是再次使用该函数的时候，能够继续被使用。因为是静态函数，所以这个静态函数的值不会被销毁，函数下次使用的时候还会使用到这个函数的值。

数据类型（比如int，long），可以把他们想象成一个个模子，在内存上模出一个一个的内存空间。

对于变量的命名规则：

用英文命名，不要用拼音和数字或者无意义的字母

对于在多个文件共同使用的全局变量或函数要加范围限定符，可以用模块名（缩写）作为范围限定符，比如（GUI-）

一般的命名规则：

模块名缩写 \_ 作用域前缀 数据类型前缀 【指针前缀】 含义标志 数据/结构后缀

比如：标识符型号为Auto static，那么他的作用域前缀为a

下划线常常用于分割单词

Sizeof：

sizeof是关键字不是函数，sizeof在计算变量所占的内存空间的时候，括号可以省略。而计算类型（模子）的时候不可以省略。

当发生溢出的时候，会自动舍去二进制的最高位（补码）

Bool：

关于bool，FLASE的值定义为0，但是TRUE的值有的定义为1，有的编译器定义为-1

Ps：在一些编译器中TAB键的后退格数不一样，所以使用了TAB后有可能会导致不同的编译器打开同一个源文件会凌乱。

Case：

注：在使用case语句后千万不要忘了加break，不然会导致语句重叠（除非想让他们重叠）。

即使程序最后不需要default处理，也最好加上default，以免让别人误以为自己忘了default处理。

Case后面只能是整形或字符型的常量或常量表达式

在多重循环中，如果可能，将最长循环放在最内层，最短循环放在最外层，以减少CPU跨切循环层的次数。即最外层循环次数多，效率低；最外层循环次数低，效率高。

Void：

Void的字面意思是“空类型”，void \*则为“空类型指针”，它可以指向任何形式的数据。Void几乎只有“注释”和限制程序的作用。Void真正发挥作用在于对函数返回的限定和对函数参数的限定。任何指针类型的值都可以直接赋给void类型的指针，无需进行强制类型转换。但是void\*类型的指针不能赋给int\*等

如果一个函数没有返回值，一定要加上void。除此之外，void不能代表一个真实的变量，否则，他的内存占用到底

return：

注意：return语句不可以返回指向“栈内存”的“指针”，因为该内存在函数结束的时候被释放。

Const：

它是constant的缩写（常量），但是这里它核心的意思应该是只读。Const出现的初始目的是取代预编译指令。节省空间，避免不必要的内存分配，同时提高效率。

编译器一般不为const只读变量分配存储空间，而是将他们保存在符号表中，使他们成为一个编译期间的值，没有了存储和读内存的操作，使得其效率变得很高。

关于const修饰指针，const距离谁近就修饰谁

事实上关于const的东西有很多，这里有待补充

Volatile：

Volatile：（易变，不稳定）：

用了这个修饰符表示这个变量可能会随时变化，用了volatile可以对特殊地址进行稳定访问。

Extern：

Extern：（外来的）：

表示变量或者函数的定义在别的文件中。

Struct：

结构体，在一个自定义函数中，如果参数的数量多了，会影响CPU的，这时候可以用结构体来压缩参数个数。

由于编译器为一个结构体至少预留1byte的内存空间，所以即使是空结构体，大小也是1byte。

C中的struct类似于c++中的class，只不过是struct是public，class是private。

结构体类型的优点：能集中不同数据类型于一体的构造类型

申明：struct

结构体类型的定义形式：

struct 结构体名

{

成员项表列

} ;

结构体类型变量的定义的三种形式

1、定义结构体同时定义变量

struct student

{

char name [20];

char sex;

long num;

} stu1,stu2;

2、通过定义无名结构体数据类型定义变量

struct

{

char name [20];

char sex;

long num;

} stu1,stu2;

3、定义结构体后定义再定义变量

struct student

{

char name [20];

char sex;

long num;

};

struct student stu1,stu2;

结构体类型可以嵌套定义

结构体成员的引用

成员访问：域访问运算符"."，结构体变量是通过运算符来访问结构体成员的

形式：struct student stu1,stu2;

stu1.name

stu1.sex//以上这些为调用方式

结构体类型变量的定义和初始化

struct student

{

char name[20];

char sex;

long num;

float score[3];

} ;

struct student stu={"gsy",'m',18901463761,5.0,5.0,5.0};

或者这样子：strcpy(stu,name,"gsy");

stu.sex='m';

结构体变量的赋值

两个类型相同的结构体可以用赋值符号进行相互赋值

stu1=stu2;//这样是合法的，相当于把stu2的成员值赋给stu1对应的成员。但是两者变量不能进行运算，如：stu1==stu2;这些都是不合法的

除本质是赋值完成的操作外，不能通过整体去使用结构体变量

结构体数组于结构体指针

结构体类型数组的定义形式：

struct 结构体类型名 数组名 【数组长度】；

struct student

{

char name [20];

char sex;

long num;

} ；

struct student stu[30];

//定义全班同学30人的信息

引用时，通过引用符"."

stu[0].name,stu[0].num,...

定义结构总结构体名

struct 总结构体名 \*p;

p=&定义的结构分支名;

printf("%d",p->内部名);

printf("%d",(\*p).内部名);//union类同

指向结构体变量的指针

比如：struct student stu,\*p1,\*p2;

p1=&stu;

p=(struct student \*)malloc(sizeof(struct student));

指向结构体变量的指针访问结构体成员

1、指针变量->成员

p1->name;

p2->score[2];

2、（结构体指针变量名）.成员名

(\*p).name;

scanf("%s%c%d",p->name,&p->sex,&p->num);

c语言中，允许用结构变量做函数参数进行传递，函数的返回值可以是结构体类型

Union：

在union中，所有的元素共用一个内存空间。

联合体的访问对于任意一个变量的存取都是从union的首地址开始的。

共用体（联合体）是一种数据类型。一个共用体是几个类型可能不同的成员的组合。

一个共用体变量的所有成员共享同一片存储区，在一个时刻只能保存一个成员的值。

定义形式：

union data

{

int a;

float b;

char c;

}obj;

共用体与结构体表示的含义与存储是完全不同的

union 共用体名

{

成员列表

} ;

总的来说，union是一些变量的集合，他的大小取共同体里面最大变量占据的内存空间，那些变量仅存一个，如果第二次写入，会导致覆盖

Enum：

枚举类型的定义和枚举变量的说明

定义：(在int main()前)

enum 枚举名 {枚举值表};

enum Weekday {sun,mon,tue,wed,thu,fri,sat};//不要加单引号双引号

该枚举名为Weekday，凡被说明为Weekday类型变量的取值只能是七选一

先申明类型，再定义变量

enum Weekday {sun,mon,tue,wed,thu,fri,sat};

enum Weekday a,b,c;

申明类型的同时定义变量

enum {sun,mon,tue,wed,thu,fri,sat}a,b,c;

枚举值是常量，不是变量，所以不能在程序中继续对他赋值。

枚举元素：由系统定义一个表示序号的数值，从0开始顺序定义为0，1，2...

在Weekday中，sun的值是0，mon的值是1...

若a，b均为枚举变量，则a=sum;b=mon;

用强制类型转换：a=(enum weekday)2;

这时候，printf("%d",sum);//输出0

memcpy：

是c和c++使用的内存拷贝函数，memcpy函数的功能是从源内存地址的起始位置开始拷贝若干个字节到目标内存地址中。它的函数原型是void \*memcpy(void \*dest, const void \*src, size\_t n);

menset：

是将某一块内存中的内容全部设置为指定的值，这个函数通常为新申请的内存做初始化工作。它的原型是void \*memset(void \*s, int ch, size\_t n);

函数解释：将s中当前位置后面的n个字节（typedef unsigned int size\_t ）用 ch 替换并返回 s 。

作用是在一段内存块中填充某个给定的值，它是对较大的结构体或数组进行清零操作的一种最快方法 [1]  。

函数原型是extern void \*memset(void \*buffer, int c, int count) buffer：为指针或是数组,c：是赋给buffer的值,count：是buffer的长度.

Typedef：

类型定义符，大致的意思是typerename。给一个已经存在的数据类型取一个别名，取出来的新名字与原来的名字相比，更能表达出想要表达的意思。（原来的名字和别名都能用而且表明同一个意思）

功能：允许用户为数据类型取别名

格式：typedef 原类型名 新类型名

比如：typedef int INTEGER

若：typedef char NAME[20]//定义20个单位

那么NAME a1,a2,s1,s2;

等价于char a1[20],a2[20],s1[20],s2[20];

注：可用宏定义来代替typedef的功能，但是宏定义是由预处理完成的，而typedef则是在编译时完成的。typedef更加灵活

Define：

预处理：在进行编译前，对源程序预先添加和替换一些信息，以便程序能够正常的编译，预处理由预处理器完成

事实上，#也是一个运算符，在预编译中，比如“#x”，可以把符号x直接取地址得到输出值

##：预算符：用于宏函数的替换部分。这个运算符把两个语言符号组成单个语言符号。比如：#define EXAM（n） x##n

如果EXAM（8）

就会被展开成：x8

总的来说，##是一个粘合剂，将前后两部分粘合起来

宏定义：#define

文件包含：#include

条件编译：#if … #else … #endif

撤销已经定义过的宏名：#undef

意义与#else if相同：#elif

关于#ifdef和#ifndef：

这两者都是条件编译，前者是如果有定义，后者是如果没有定义

#error：

只要在编译的时候遇到这个，就会生成一个编译错误的提示信息，并停止编译。

这个东西的一般格式为 #error XXXXX 如果遇到了error的编译，就会跳出编译错误的原因，即在源代码的时候写上的XXXXX，注意，XXXXX不用加双引号

#pragma：

它允许向编译程序传输各种指令（我还不怎么懂，等待补充）

//#pragma的格式有很多种，我不太懂，只能把出现的样子记录一下。

#line （这边似乎能够和内联搭上关系）//改变当前行数和文件名称，一般这么用：#line number 【“文件名”】比如#line 100 XXX 意思是把这个文件放在第30行的位置，这样可以有效避免了中间文件对源代码造成不必要的错误

比如：#pragma message（“你想要输入的东西”）

#pragma code\_seg([“”])

#pragma once

#pragma hdrstop

#pragma resource

Pragma warning

`#pragma comment

#pragma pack

格式：#开头，语句尾无分号

C语言中，允许用一个标识符来表示一个字符串，称为“宏”，标识符称为宏名。

宏代换/宏调用/宏展开：在编译预处理时，程序中所有的“宏”名，都用宏定义中的字符串去代换。

宏分类：有无参数

宏定义可以出现在源代码的任何地方。，使用了宏定义有利于后期维护程序。

在宏定义使用算术公式的时候不要吝啬括号，如果没有使用括号会导致10+1\*10+1这个样子

#include的意思是将已存在文件的内容镶嵌到正在编写的文件中。

在include中，.代表当前目录，..代表上层目录

无参宏定义：

宏名后不带参数

一般形式 #define 宏名（标识符） 宏定义串（常数，表达式，格式串）

比如：#define M(3\*y+y\*y)//特别注意这边有无括号结果不一样

注意：若在宏定义串后加了分号，替换时分号也会被替换上

#define 写在函数外，如果终止其作用域可使用#undef命令

如：#define PI 3.1415927

int main()

{

......

}

#undef PI

表示PI仅在main函数中有效，undef下面的函数都无效

宏定义允许嵌套：但要先定义，再使用，依次替换

比如：#define ONE 1

#define TWO ONE+ONE

如果字符串长于一行，可以再结尾用反斜线"\"续行

比如：#define xx xxxxxxxxxxxxxxxxxxxxxx\

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

#define F "%10s"使用宏定义实现格式控制输出

带参宏定义：

#define 宏名（参数表） 宏定义串

可包含多个参数，参数之间用","分开

调用形式为宏名（实参表）;

#define M（y） y\*y+3\*y//宏定义

...

K=M（5）;//宏调用

注：宏名与形参表之间不能有空格

include指令

事实上，#include <stdio.h> 和#include"stdio.h"这两个均可，一般系统库文件使用<>,用户自定义的文件使用""

功能：将指定文件嵌入到程序的指定位置，一个include命令只能指定一个被包含文件

条件编译：

按照不同的条件去编译不同的程序部分，因而产生不同的目标替代文件

形式1：#ifdef 标识符

程序段1

#else

程序段2

#endif

如果标识符已经被#define命令定义过，则对程序段1进行编译，否则对程序段2编译，如没有程序段2，格式中的#else可以省略

若标志符未被定义，则对程序段1进行编译

使用条件编译的原理：省去对源代码的主函数中if的编译，减去程序内存开销，缩短生成的目标程序，提高程序的执行效率

宏定义中的字符串应加圆括号，字符串中出现的形式参数两边也应加圆括号

特别注意：int\*p【n】；n个指向整型数据的指针变量组成的指针数组p

# C语言中关于字符的批处理：

memcpy：

是c和c++使用的内存拷贝函数，memcpy函数的功能是从源内存地址的起始位置开始拷贝若干个字节到目标内存地址中。它的函数原型是void \*memcpy(void \*dest, const void \*src, size\_t n);

menset：

是将某一块内存中的内容全部设置为指定的值，这个函数通常为新申请的内存做初始化工作。它的原型是void \*memset(void \*s, int ch, size\_t n);

函数解释：将s中当前位置后面的n个字节（typedef unsigned int size\_t ）用 ch 替换并返回 s 。

作用是在一段内存块中填充某个给定的值，它是对较大的结构体或数组进行清零操作的一种最快方法 [1]  。

函数原型是extern void \*memset(void \*buffer, int c, int count) buffer：为指针或是数组,c：是赋给buffer的值,count：是buffer的长度.

举的例子：memset（a，0，sizeof（a））；

Memset函数有三个参数，第一个是要被设置的内存起始位置；第二个参数是要被设置的值；第三个参数是要被设置的内存大小，单位为byte。

如果指针变量没有被初始化会导致if语句或assert宏检验失败。

关于字符串处理的还有：strcat，strcpy，strcmp

# 符号：

符号有28种：

, . ; : ? ‘ “ ( [ { % ^(xor异或) - < > ! | / \ ~ # } ] ) & \* = +

关于/\* \*/：这个东西不能嵌套，因为/\*总是和最近的\*/匹配。注：汇编程序的注释以分号开头。

注释应该采用英文，在不同的编译器中，注释的中文可能导致一团乱码（gtf8标准与gbk标准）

接续符和转义符：

跟在反斜杠后面的字符自动接到前面一行（源代码中），所以\是可以把关键字拆分成两行的。

常见转义字符以及含义：

转义字符 转义字符的意义

\n 回车转行

\t 横向跳到下一制表位置

\v 竖向跳格

\b 退格

\r 回车

\f 走纸换页

\a 蜂鸣

\ddd 1~3位八进制数代表字符

\xhh 1~2位十六进制数代表字符

Ddd和hh分别是八进制和十六进制的ASCII代码，广义的讲，c语言字符集中任何一个字符均可用转义字符来表示。

单引号和双引号：

“”//字符串

’’//字符

逻辑运算符：||与&&

位运算符：

&按位与

|按位或

^按位异或

~按位取反

<<按位左移//高位丢弃，地位补0

>>按位右移//当为负数时，符号位为1，右移的时候符号位也移动

注：左移和右移的位数都不能大于数据的长度，不能小于0

花括号：花括号的作用是打包

同一优先级的运算符，运算次序由结合方向所决定。

# 做oj的时候的调试技巧

对于输入cin/scanf之类的（其实这些都是函数，输入的只是这些函数的返回值），当他们的返回值为-1的时候，就会退出。我们知道，当遇到EOF的时候就会退出，因为在运行的时候EOF就是-1，但是在输入的时候ctrl+z表是的也是-1，也会退出。在oj的时候，测评系统一般会使用ctrl+z表示输入结束。所以在使用while的时候，使用while（scanf（“%d”，&a）！=EOF）可以在合适的时候退出。

特别注意的是：输入流可以是键盘也可以是文件。在文件形式的时候：首先，将输入数据编辑成文本文件，接着找到生成的可执行文件，然后在windows系统提供的命令行窗口中使用系统命令运行程序

举一个data.in的例子(如果生成的文件名为LSC)：

LSC<data.in//运行程序LCS.exe。符号<表示重定向，就是将标准输入设备键盘输入数据的形式重新定向为从LSC.exe文件位于同一个文件夹下的data.in文件中获取，输出的结果出现在屏幕上。

# c语言数组：

事实上，指针和数组几乎没有关系。数组可以看成是一些连续变量的集合，而指针也就是用来存储内存地址的。以指针的形式访问数组和以下标的形式访问数组原理其实一样，知识形式上不同。比如\*（a+4）和a【4】是一样的，前者的4就叫做内存的偏移量。偏移量的单位是元素的个数不是byte数。

如果对数组不初始化，那么数组元素值为随机数

对static（静态）类型的数组不赋初值，系统会自动把他们每一个都赋值为0

拓展：在自定义函数前加static，该自定义函数中的数组变化不会改变主函数中的数组

柔性数组（存在于结构体中）：

柔性数组是结构体中的最后一个（保证结构体中元素大于1个），柔性数组的意思是在结构体中定义了一个数组，但是这个数组可以长短。使用的时候要给它动态分配内存空间。定义形式为：

struct flexible

{

Int a;

int b[ ];

}a;//这时候它占有的内存空间只有4字节，即这时候内存没有给柔性数组分配。

给结构体分配内存：

flexible \*a=（flexible\*）malloc(sizeof(flexible)+100\*sizeof(int));

分配的内存是编外人员，用sizeof测量这个结构体大小还是4。

# 指针：

指针是c/c++的精华（我现在也深有感触，特别是对计算机的内存地址总线等有了初步的了解以后）指针，是存放内存地址的变量。

使用指针的优点：使程序简洁，紧凑，高效

可以有效的表示复杂的数据结构

可以动态分配内存（malloc—memory allocation）

得到多于一个的函数返回值

NULL就是NULL，不是Null。因为#define NULL 0，所以在使用的时候发现NULL的值是0，但是实际上完全没有关系，两者相差很大。

指针类型：地址也是数，这类数据就是数。比如&a在32位系统中就是一个32位的数

指针变量：专门存放地址的变量叫做指针变量

关于&与\*（这两者互为逆运算）

&：取变量的地址，结合性是从右往左。，内容是地址

\*：取指针所指向变量的内存。结合性是从右往左，内容是数据

比如i=3；//直接访问

\*i=3；//间接访问

指针变量的定义：

一般形式：【存储类型】数据类型 \*指针名；（指针名应为合法标志符）

指针变量只能指向定义时所规定类型的变量

指针变量定义后，变量值不确定，应用前必须先赋值，再使用。没有赋初值前为随机地址

指针变量的使用：

指针变量的初始化和指针所指向的变量

一般形式：【存储类型】数据类型 \*指针名=初始地址值；

例如：int i；int \*p=&i；//这是赋给指针变量，不是赋给目标变量。其中变量必须已说明，类型应一致

不允许将地址赋给一般变量，也不允许将数值赋给指针变量

零指针与空类型指针

零指针（空指针）：

定义：指针变量值为0

表示：int \*p=0；

P指向地址为0的单元，系统保证该单元不作他用，表示指针变量值没有意义

如：#define NULL 0

int \*p=NULL;

p=NULL与未对p赋值不同，它的用途在于避免指针变量的非法使用，在程序中常作为状态进行比较。

比如：int \*p;

......

while(p!=NULL)

{

......

}

空类型指针（void类型指针）

表示：void \*p；//表示不确定p是指向哪一种类型数据的指针变量，这种指针在使用的时候要进行强制类型转换。

指针变量作为函数参数

这里注意值传递和地址传递这两种方法即可

返回指针的函数（返回值为某指针类型的函数）

函数定义形式：类型标志符 \*函数名（参数表）；

例如：int \*max （int x，int y）……

int main（）{…a=max（&m，&n）}；

指向函数的指针

函数指针：函数在编译时被分配的入口地址（程序段的存储地址），将之一地址存入指针变量，这个变量就是指向函数的指针变量

函数指针用函数名表示

定义形式：数据类型（\*指针变量名）（）；//指针变量名专门存放函数入口地址，可指向返回值类型相同的不同函数

比如：int(\*p)();//其中（）不能省，int(\*p)()与int\*p()不同

函数指针变量指向的函数必须有函数说明。

对函数指着变量的赋值，如：p=max；

函数调用形式：c=max（a，b）；

C=（\*p）(a，b);

c=p(a，b)；

对函数指针变量 p±n，p++，p—都无意义

关于指针：

int(\*p)(int)//p是一个指向有一个整形参数且返回类型为整形的函数的指针

(\*ptr)++;//加地址，地址加sizeof（类型）

ptr++;//加相应地址的数值

# 原码，反码，补码：

关于原码，反码，补码（特别注意了，原码的第一位都是零）

计算机在运算的时候用的是补码。

正数的原码，反码，补码都是一样的

原码（二进制）负数的反码是把原码的0和1互换，补码是在反码的基础上再加上1

# 链表及其应用：

链表是一种数据结构，其数据之间的顺序关系需要使用者自行维护，它无需像数组那样连续存放

链表结点的定义：

struct node

{

char name[20];

long telephone;

struct node \*next;//结点的指针域

} ;

链表的定义：若存在一组结点类型的数据，在第一个结点的指针域内存入第二个结点的首地址，在第二个结点内又存放着第三个结点的首地址，如此串联下去知道最后一个结点。最后一个结点因无后续结点连接，其指针域可赋为NULL或0，这样一种连接方式称为链表

链表的创建过程

1、定义链表的指针数据结构

2、定义头节点指针，置指针域为空，表示创建了一个空表。

3、申请一个结点存储单元

4、将新结点的指针赋值为空。若是空表，将新节点连接至表头；若是非空表，将指针链接到表尾

链表的输出：

链表的输出过程：

1、找到表头

2、若是非空表，输出结点的成员，是空表则退出

3、转移到下一个结点

链表的删除：

链表结点的删除：

1、定位删除点

2、先将指针接在合适的位置上，再断开需要删除的结点

3、释放被删除结点的存储单元

双向链表：

双向链表是指可以追溯到父节点和子节点的一种链表。

# 位段/位域：

为了节省存储空间，如将开关0和1两种状态存一位二进制而非1字节

位段是把一个字节中的二进制划分为几个不同的区域，并说明每一个区域的位数。

每个域都有一个域名，允许在程序中按域名进行操作。

位域的定义和位域变量的说明

形式为：（结构体，位域，共用体他们的形式很相似，但是用处不同）

struct 位域结构体名

{

类型说明符 位域名：位域长度;

} ;

如：struct bitsec //说明data为bitsec变量，共占两个字节

{

int a:6; //表示位域a占6位（一个字节有8位）

int b:2;

int c:8;

} data;

位域变量的说明与结构体变量说明的方式相同。

可以1、先定义后说明

2、同时定义同时说明

3、直接说明

注：一个位域必须存储在同一个字节中，不能跨两个字节

如一个字节所剩空间不够存放在一个位域的时候，应从下一单元起存放该位置，也可以有意使某位域从下一个单元开始

位域的使用

与结构体成员的形式相同，一般形式：位域变量名，位域名

位域允许用各种格式输出

# 内存管理：

野指针：

野指针又叫悬垂指针。据说千万不能对野指针这个东西掉以轻心，它会毁掉一整个源文件。

拴住野指针的链子就是“NULL”，定义指针变量的时候最好初始化为NULL。用完指针后也将指针变量的值设置为NULL。也就是说除了在使用的时候，都把指针栓在NULL上。

栈，堆和静态区：

一般来说，可以把内存去理解成三个部分：栈（stack），堆（heap），静态区。堆栈就是栈。

内存的三个部分不是什么东西都能够存进去的。

静态区：保存自动全局变量和static变量（包括static全局和局部变量）。静态区的内容在总个程序的生命周期内都存在，由编译器在编译的时候分配。

栈：保存局部变量。栈上的内容只有在函数的范围内存在。当函数运行结束的时候，这些数据会被自动销毁。它的特点是效率高，但是空间大小有限。

堆：由malloc系列函数或new操作符分配的内存。其生命周期由free或delete决定。在没有释放之前一直存在，直到程序结束。他的特点是使用灵活，空间比较大，但是容易出错。

常见的内存错误：

指针没有指向一块合法的内存

结构体成员指针未初始化

没有为结构体指针分配足够的内存

注：不管什么时候我们使用指针之前要保证指针是有效的

为指针分配内存大小的时候一定要谨慎，防止指针越界。

内存分配成功但没有被初始化（这种情况有时候会导致大问题，但是问题不多见而且很难被察觉）

内存越界

内存泄漏：这个很难避免，甚至系统软件都有内存泄漏。这种对于普通的游戏来说重启一次可能问题不大，但是如果是嵌入式系统比如汽车的制动系统或者心脏起搏器就很致命了。

会产生泄露的内存是堆上的内存（资源/句柄等的泄露情况），也就是说由malloc系列函数或new操作符分配的内存。如果用完之后没有即使free或delete，这块内存就无法释放，直达整个程序结束为止。

注意：malloc申请的是一整块连续的内存，如果内存是一小块一小块的，是分配不成功的。

既然malloc函数申请内存存在不可能的情况，那我们在使用指向这块内存的指针时，必须用if（NULL！=p）语句来验证内存确实分配成功了。

注：malloc可以分配0的内存空间，能够有内存地址，但是这块内存大小是0，是不能使用的。

内存释放：

Malloc对应的函数就是free函数了，free函数只有一个参数，就是所要释放的内存块的首地址，比如free（p）；

Free函数的实际意义是把这块内存和p之间的关系斩断。那块内存保存的数据没有变，那个指针保留的数组也没有变，但是那个指针已经没有那个数组的所有权了。

所以free（p）；使用一次后，指针与内存没有了所有权，再多用一次free（p）；就会报错。

Malloc两次free一次会导致内存泄漏，malloc一次free两次就会报错。所以malloc使用的次数一定要与free使用的次数相等，不然一定会出错。内存释放之后，要记得及时p=NULL；

这个NULL就是拴住野指针的绳子

# 文件（流）：

文件的定义

对于输入和输出设备：在程序中，输入设备和输出设备可以是文件，也可以是键盘，显示器，打印机等。

文件与流

c语言把每一个独立的数据流称为文件，每个文件在相应的磁盘目录下，文件用文件名来标识。

定义一个结构体类型FILE，将存储与流操作有关的（与打开的文件有关的）所有信息（与指针相关的）

c程序启动时自动创建三个流（建立三个文件指针并指定值）

流 指针名为

标准输入流 stdin //与操作系统的标准输入链接

标准输出流 stdout //与操作系统的标准输出链接

标准错误流 stderr //直接与显示器链接，这说明stderr不能重新定向

文件逻辑结构：

记录文件：由具有一定记录组成

流式文件：由一个个字符（字节）数据顺序组成

数据的组织形式：

文本文件：ASCII文件，每个文件存放一个字节的ASCII码

二进制文件：数据按其在内存中的存储形式原样存放

FILE定义在stdio.h中，它本身是一个结构体，只不过这个结构体被取小名为FILE

在c程序中使用文件，需要完成以下工作：

声明一个FILE\*类型的文件指针变量

调用fopen函数打开（open）文件（将文件指针和某实际文件相联系）。打开文件时需指定文件名以及使用文件的目的--用于输入还是输出

调用stdio.h中适当的函数完成必要的I/O操作

调用fclose函数表明为你教案操作结束，这一操作称为关闭（close）文件，它断开了文件指针变量与实际文件间的联系

函数fopen的原型：

FILE \*fopen（const char \*filename(希望打开的文件名)，const char \*mode(用于指定文件打开方式)）

文件的打开模式：

Mode 含义

r 打开一个文本文件只读

w 打开一个文本文件只写

a 打开一个文本文件在尾部追加

rb（这边的b表示bit） 打开一个只读的二进制文件

wb 打开一个只写的二进制文件

ab 对二进制文件追加

r+ 打开一个可读/写的文本文件

w+ 创建一个新的可读/写的文本文件

a+ 打开一个可读/写的文本文件

rb+ 打开一个可读/写的二进制文件

wb+ 创建一个新的可读/写的二进制文件

ab+ 打开一个可读/写的二进制文件

打开文件的方法：#include <stdio.h>

FILE \*fp；//定义文件指针

//创建一个只写的新文本文件

if((fp=fopen("text.txt","w"))==NULL)

//判断以写的方式打开文件是否成功

关闭文件是丢掉缓冲区的内容，释放对应的内存

通过函数fclose完成

int fclose(FILE \*fp);

fclose正常完成时返回0，出问题时返回值是EOF

feof函数判断文件是否达到了结束的标志

常用该函数来检查文件中的数据是否都已访问

函数原型：int feof(FILE \*fp);

函数返回值若是非0的数，则说明文件的指针已经指向文件的结尾

while(!feof(fp))//说明当文件没结束时，以输出文件中的内容

读写函数fgetc和fputc

从文件中读取一个字符：int fgetc (FILE \*fp);

fgetc()函数从输入流的当前位置读取一个字符，并将文件指针指示器移到下一个字符处，如果已经到达文件尾，函数返回EOF

读写字符串读写函数fgets和fputs

从文件中读取一个字符串

函数形式为:char \*fgets(char \*str,int num,FILE\*fp);

fgets()函数从流文件stream中读取至多num-1个字符，并把他们放入str指向的字符数组中。读取字符知道遇见回车符或EOF（文件结束符）为止，或读入了所限定的字符数

将一个字符串写入文件

函数形式为：int fputs(char \*str,FILE \*fp);

fputs()函数将str指向的字符串写入流文件。操作成功时，函数返回0值，失败返回非0值

格式化输入输出函数fscanf和fprintf

按照指定的格式对文件内容进行读写

按指定格式将文件内容写入：

int fscanf(FILE \*fp,char \*format,arg\_list);

按指定格式将内容写入文件

int fprintf(FILE \*fp,char \*format,arg\_list);

块读写函数fread和fwrite

对二进制文件无法以整体形式向文件写入或从文件读出。c语言提供成块的读写方式来操作文件，使其数组或结构体等类型可以进行一次性读写

size\_t fread(void \*pointer,size\_t size,size\_t num,FILE \*fp);

size\_t fwrite(void \*pointer,size\_t size,size\_t num,FILE \*fp);

说明，size\_t使c语言系统确定的无符号整形

函数fwrite向流fp输出一批数据，数据的起始位置由指针pointer给定，元素大小时size，共num个

函数fread的功能正好与fwrite对应，它要求读入num个数据元素，每个元素的大小为size，指针参数pointer应指向接受数据的起始存储位置

文件指针的定位

c语言在标准I/O库stdio.h中提供了一组文件指针定位的函数，它的作用是使文件指针移动到所需要的位置。

函数名 函数原型 功能

fseek int fseek(FILE\*fp,long offset,int origin); 根据origin的值移动指针

rewind void rewind(FILE \*fp); 重返文件起始位置

ftell long ftell(FILE \*fp); 返回文件指针的当前位置

文件指针的定位

fseek函数origin参数的定义：

符号常量 值 含义

SEEK\_SET 0 从文件开头计算

SEEK\_CUR 1 从文件指针当前位置计算

SEEK\_END 2 从文件末尾计算

比如：

fseek(fp,5L,0) ;//将文件指针从文件头向下移动5个字节

fseek(fp,-10L,2);//表示将文件从结尾位置向上移动10个字节

出错的检测：

调用各种输入输出函数时，如果出现错误，除了函数返回值有所反应外，还可以用ferror函数来检测

函数的原型：int ferror(FILE \*fp);

如果ferror返回值为0（假），表示文件操作未出错；如果返回一个非零值，则说明文件操作失败

比如：if(ferror(fp))...

文件类型指针变量：比如 FILE \*fp；fp是一个指向FILE类型结构体的指针变量

Buffer缓存（简写为buf）

FILE类型的数组：FILE f【5】；定义了一个结构体数组f，它有5个元素，可以用来存放5个文件的信息。

文件的打开（fopen函数）：

FILE \*fp；//看了库文件就会知道FILE其实是一个结构体，这个结构体被typedef为FILE，定义一个fp的指针，这个指针指向FILE的那个结构体

fp=fopen(“ ”，“ ”)；//双引号里面的部分是要赋给这个fp指针的文件（ACSII文件或者二进制文件）的地址，比如：H：\\gsy.txt ;后面一个双引号里面是输入文件的打开形式，比如只读，只写，添加之类的。

文件的关闭（fclose函数）：

如果一直不关闭那个fp指针，会一直浪费资源。

文件的读写（使用以下函数都要包含头文件stdio.h）：

字符读写函数：fgetc()和fputc()

fputc(ch,fp)；//将ch的值输入到fp指向的文件中去，每写入一个字符，文件内部指针向后移动一个字节。

字符串读写函数：fgets()和fputs()

数据块读写函数：freed和fwrite()

格式化读写函数：fscanf()和fprintf()

函数的详解:

**fopen（打开文件）函数**  
表头文件 #include<stdio.h>  
定义函数 FILE \* fopen(const char \* path,const char \* mode);  
函数说明 参数path字符串包含欲打开的文件路径及文件名，参数mode字符串则代表着流形态。  
**mode有下列几种形态字符串:**r 打开只读文件，该文件必须存在。  
r+ 打开可读写的文件，该文件必须存在。  
w 打开只写文件，若文件存在则文件长度清为0，即该文件内容会消失。若文件不存在则建立该文件。  
w+ 打开可读写文件，若文件存在则文件长度清为零，即该文件内容会消失。若文件不存在则建立该文件。  
a 以附加的方式打开只写文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾，即文件原先的内容会被保留。  
a+ 以附加方式打开可读写的文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾后，即文件原先的内容会被保留。

返回值 ：

文件顺利打开后，指向该流的文件指针就会被返回。若果文件打开失败则返回NULL，并把错误代码存在errno 中。  
附加说明 一般而言，开文件后会作一些文件读取或写入的动作，若开文件失败，接下来的读写动作也无法顺利进行，所以在fopen()后请作错误判断及处理。

比如：

#include<stdio.h>  
main()  
{  
FILE \* fp;  
fp=fopen(“noexist”,”a+”);  
if(fp= =NULL) return;  
fclose(fp);  
}

**1. fprintf**功能：传送格式化输出到一个文件中  
表头文件：#include<stdio.h>  
函数原型：int fprintf(FILE \*stream, char \*format[, argument,...]);  
FILE\* 一个FILE型的指针  
char\* 格式化输入函数，和printf里的格式一样  
返回值：成功时返回转换的字节数，失败时返回一个负数  
fp = fopen("/local/test.c","a+");  
fprintf(fp,"%s\n",str);  
  
**2. fscanf**功能：从一个流中执行格式化输入  
表头文件：#include<stdio.h>  
函数原型：int fscanf(FILE \*stream, char \*format[,argument...]);  
FILE\* 一个FILE型的指针  
char\* 格式化输出函数，和scanf里的格式一样  
返回值：成功时返回转换的字节数，失败时返回一个负数  
fp = fopen("/local/test.c","a+");  
fscanf(fp,"%s",str);  
  
**3. clearerr（清除文件流的错误旗标）**相关函数 feof  
表头文件 #include<stdio.h>  
定义函数 void clearerr(FILE \* stream);  
函数说明 clearerr（）清除参数stream指定的文件流所使用的错误旗标。  
返回值  
　  
**4.fclose（关闭文件）**相关函数 close，fflush，fopen，setbuf  
表头文件 #include<stdio.h>  
定义函数 int fclose(FILE \* stream);  
函数说明 fclose()用来关闭先前fopen()打开的文件。此动作会让缓冲区内的数据写入文件中，并释放系统所提供的文件资源。  
返回值 若关文件动作成功则返回0，有错误发生时则返回EOF并把错误代码存到errno。  
错误代码 EBADF表示参数stream非已打开的文件。  
范例 请参考fopen（）。  
　  
**5.fdopen（将文件描述词转为文件指针）**相关函数 fopen，open，fclose  
表头文件 #include<stdio.h>  
定义函数 FILE \* fdopen(int fildes,const char \* mode);  
函数说明 fdopen()会将参数fildes 的文件描述词，转换为对应的文件指针后返回。参数mode 字符串则代表着文件指针的流形态，此形态必须和原先文件描述词读写模式相同。关于mode 字符串格式请参考fopen()。  
返回值 转换成功时返回指向该流的文件指针。失败则返回NULL，并把错误代码存在errno中。  
**范例**

复制代码 代码如下:

#include<stdio.h>  
main()  
{  
FILE \* fp =fdopen(0,”w+”);  
fprintf(fp,”%s/n”,”hello!”);  
fclose(fp);  
}  
执行 hello!

**6.feof（检查文件流是否读到了文件尾）**相关函数 fopen，fgetc，fgets，fread  
表头文件 #include<stdio.h>  
定义函数 int feof(FILE \* stream);  
函数说明 feof()用来侦测是否读取到了文件尾，尾数stream为fopen（）所返回之文件指针。如果已到文件尾则返回非零值，其他情况返回0。  
返回值 返回非零值代表已到达文件尾。  
　  
**7.fflush（更新缓冲区）**相关函数 write，fopen，fclose，setbuf  
表头文件 #include<stdio.h>  
定义函数 int fflush(FILE\* stream);  
函数说明 fflush()会强迫将缓冲区内的数据写回参数stream指定的文件中。如果参数stream为NULL，fflush()会将所有打开的文件数据更新。  
返回值 成功返回0，失败返回EOF，错误代码存于errno中。  
错误代码 EBADF 参数stream 指定的文件未被打开，或打开状态为只读。其它错误代码参考write（）。  
　  
**8.fgetc（由文件中读取一个字符）**相关函数 open，fread，fscanf，getc  
表头文件 include<stdio.h>  
定义函数 nt fgetc(FILE \* stream);  
函数说明 fgetc()从参数stream所指的文件中读取一个字符。若读到文件尾而无数据时便返回EOF。  
返回值 getc()会返回读取到的字符，若返回EOF则表示到了文件尾。  
**范例**

复制代码 代码如下:

#include<stdio.h>  
main()  
{  
FILE \*fp;  
int c;  
fp=fopen(“exist”,”r”);  
while((c=fgetc(fp))!=EOF)  
printf(“%c”,c);  
fclose(fp);  
}

**9.fgets（由文件中读取一字符串）**相关函数 open，fread，fscanf，getc  
表头文件 include<stdio.h>  
定义函数 har \* fgets(char \* s,int size,FILE \* stream);  
函数说明 fgets()用来从参数stream所指的文件内读入字符并存到参数s所指的内存空间，直到出现换行字符、读到文件尾或是已读了size-1个字符为止，最后会加上NULL作为字符串结束。  
返回值 gets()若成功则返回s指针，返回NULL则表示有错误发生。  
**范例**

复制代码 代码如下:

#include<stdio.h>  
main()  
{  
char s[80];  
fputs(fgets(s,80,stdin),stdout);  
}  
执行 this is a test /\*输入\*/  
this is a test /\*输出\*/

**10.fileno（返回文件流所使用的文件描述词）**相关函数 open，fopen  
表头文件 #include<stdio.h>  
定义函数 int fileno(FILE \* stream);  
函数说明 fileno()用来取得参数stream指定的文件流所使用的文件描述词。  
返回值 返回文件描述词。  
**范例**

复制代码 代码如下:

#include<stdio.h>  
main()  
{  
FILE \* fp;  
int fd;  
fp=fopen(“/etc/passwd”,”r”);  
fd=fileno(fp);  
printf(“fd=%d/n”,fd);  
fclose(fp);  
}  
执行 fd=3

**12.fputc（将一指定字符写入文件流中）**相关函数 fopen，fwrite，fscanf，putc  
表头文件 #include<stdio.h>  
定义函数 int fputc(int c,FILE \* stream);  
函数说明 fputc 会将参数c 转为unsigned char 后写入参数stream 指定的文件中。  
返回值 fputc()会返回写入成功的字符，即参数c。若返回EOF则代表写入失败。  
**范例**

复制代码 代码如下:

#include<stdio.h>  
main()  
{  
FILE \* fp;  
char a[26]=”abcdefghijklmnopqrstuvwxyz”;  
int i;  
fp= fopen(“noexist”,”w”);  
for(i=0;i<26;i++)  
fputc(a,fp);  
fclose(fp);  
}

**13.fputs（将一指定的字符串写入文件内）**相关函数 fopen，fwrite，fscanf，fputc，putc  
表头文件 #include<stdio.h>  
定义函数 int fputs(const char \* s,FILE \* stream);  
函数说明 fputs()用来将参数s所指的字符串写入到参数stream所指的文件内。  
返回值 若成功则返回写出的字符个数，返回EOF则表示有错误发生。  
**范例 请参考fgets（）。**fread（从文件流读取数据）  
相关函数 fopen，fwrite，fseek，fscanf  
表头文件 #include<stdio.h>  
定义函数 size\_t fread(void \* ptr,size\_t size,size\_t nmemb,FILE \* stream);  
函数说明 fread()用来从文件流中读取数据。参数stream为已打开的文件指针，参数ptr 指向欲存放读取进来的数据空间，读取的字符数以参数size\*nmemb来决定。Fread()会返回实际读取到的nmemb数目，如果此值比参数nmemb 来得小，则代表可能读到了文件尾或有错误发生，这时必须用feof()或ferror()来决定发生什么情况。  
返回值 返回实际读取到的nmemb数目。  
**附加说明  
范例**

复制代码 代码如下:

#include<stdio.h>  
#define nmemb 3  
struct test  
{  
char name[20];  
int size;  
}s[nmemb];  
int main(){  
FILE \* stream;  
int i;  
stream = fopen(“/tmp/fwrite”,”r”);  
fread(s,sizeof(struct test),nmemb,stream);  
fclose(stream);  
for(i=0;i<nmemb;i++)  
printf(“name[%d]=%-20s:size[%d]=%d/n”,i,s.name,i,s.size);  
}  
执行  
name[0]=Linux! size[0]=6  
name[1]=FreeBSD! size[1]=8  
name[2]=Windows2000 size[2]=11

**14.freopen（打开文件）**相关函数 fopen，fclose  
表头文件 #include<stdio.h>  
定义函数 FILE \* freopen(const char \* path,const char \* mode,FILE \* stream);  
函数说明 参数path字符串包含欲打开的文件路径及文件名，参数mode请参考fopen()说明。参数stream为已打开的文件指针。Freopen()会将原stream所打开的文件流关闭，然后打开参数path的文件。  
返回值 文件顺利打开后，指向该流的文件指针就会被返回。如果文件打开失败则返回NULL，并把错误代码存在errno 中。  
**范例**

复制代码 代码如下:

#include<stdio.h>  
main()  
{  
FILE \* fp;  
fp=fopen(“/etc/passwd”,”r”);  
fp=freopen(“/etc/group”,”r”,fp);  
fclose(fp);  
}

**15.fseek（移动文件流的读写位置）**相关函数 rewind，ftell，fgetpos，fsetpos，lseek  
表头文件 #include<stdio.h>  
定义函数 int fseek(FILE \* stream,long offset,int whence);  
函数说明 fseek()用来移动文件流的读写位置。参数stream为已打开的文件指针，参数offset为根据参数whence来移动读写位置的位移数。  
参数 whence为下列其中一种:  
SEEK\_SET从距文件开头offset位移量为新的读写位置。SEEK\_CUR 以目前的读写位置往后增加offset个位移量。  
SEEK\_END将读写位置指向文件尾后再增加offset个位移量。  
当whence值为SEEK\_CUR 或SEEK\_END时，参数offset允许负值的出现。  
**下列是较特别的使用方式:**1) 欲将读写位置移动到文件开头时:fseek(FILE \*stream,0,SEEK\_SET);  
2) 欲将读写位置移动到文件尾时:fseek(FILE \*stream,0,0SEEK\_END);  
返回值 当调用成功时则返回0，若有错误则返回-1，errno会存放错误代码。  
附加说明 fseek()不像lseek()会返回读写位置，因此必须使用ftell()来取得目前读写的位置。  
**范例**

复制代码 代码如下:

#include<stdio.h>  
main()  
{  
FILE \* stream;  
long offset;  
fpos\_t pos;  
stream=fopen(“/etc/passwd”,”r”);  
fseek(stream,5,SEEK\_SET);  
printf(“offset=%d/n”,ftell(stream));  
rewind(stream);  
fgetpos(stream,&pos);  
printf(“offset=%d/n”,pos);  
pos=10;  
fsetpos(stream,&pos);  
printf(“offset = %d/n”,ftell(stream));  
fclose(stream);  
}  
执行 offset = 5  
offset =0  
offset=10

**16.ftell（取得文件流的读取位置）**相关函数 fseek，rewind，fgetpos，fsetpos  
表头文件 #include<stdio.h>  
定义函数 long ftell(FILE \* stream);  
函数说明 ftell()用来取得文件流目前的读写位置。参数stream为已打开的文件指针。  
返回值 当调用成功时则返回目前的读写位置，若有错误则返回-1，errno会存放错误代码。  
错误代码 EBADF 参数stream无效或可移动读写位置的文件流。  
**范例 参考fseek()。**　  
**17.fwrite（将数据写至文件流）**相关函数 fopen，fread，fseek，fscanf  
表头文件 #include<stdio.h>  
定义函数 size\_t fwrite(const void \* ptr,size\_t size,size\_t nmemb,FILE \* stream);  
函数说明 fwrite()用来将数据写入文件流中。参数stream为已打开的文件指针，参数ptr 指向欲写入的数据地址，总共写入的字符数以参数size\*nmemb来决定。Fwrite()会返回实际写入的nmemb数目。  
返回值 返回实际写入的nmemb数目。  
**范例**

复制代码 代码如下:

#include<stdio.h>  
#define set\_s (x,y) {strcoy(s[x].name,y);s[x].size=strlen(y);}  
#define nmemb 3  
struct test  
{  
char name[20];  
int size;  
}s[nmemb];  
main()  
{  
FILE \* stream;  
set\_s(0,”Linux!”);  
set\_s(1,”FreeBSD!”);  
set\_s(2,”Windows2000.”);  
stream=fopen(“/tmp/fwrite”,”w”);  
fwrite(s,sizeof(struct test),nmemb,stream);  
fclose(stream);  
}  
**执行 参考fread（）。**

**18.getc（由文件中读取一个字符）**相关函数 read，fopen，fread，fgetc  
表头文件 #include<stdio.h>  
定义函数 int getc(FILE \* stream);  
函数说明 getc()用来从参数stream所指的文件中读取一个字符。若读到文件尾而无数据时便返回EOF。虽然getc()与fgetc()作用相同，但getc()为宏定义，非真正的函数调用。  
返回值 getc()会返回读取到的字符，若返回EOF则表示到了文件尾。  
**范例 参考fgetc()。**　  
**19.getchar（由标准输入设备内读进一字符）**相关函数 fopen，fread，fscanf，getc  
表头文件 #include<stdio.h>  
定义函数 int getchar(void);  
函数说明 getchar()用来从标准输入设备中读取一个字符。然后将该字符从unsigned char转换成int后返回。  
返回值 getchar()会返回读取到的字符，若返回EOF则表示有错误发生。  
附加说明 getchar()非真正函数，而是getc(stdin)宏定义。  
**范例**

复制代码 代码如下:

#include<stdio.h>  
main()  
{  
FILE \* fp;  
int c,i;  
for(i=0li<5;i++)  
{  
c=getchar();  
putchar(c);  
}  
}  
执行 1234 /\*输入\*/  
1234 /\*输出\*/

**20.gets（由标准输入设备内读进一字符串）**相关函数 fopen，fread，fscanf，fgets  
表头文件 #include<stdio.h>  
定义函数 char \* gets(char \*s);  
函数说明 gets()用来从标准设备读入字符并存到参数s所指的内存空间，直到出现换行字符或读到文件尾为止，最后加上NULL作为字符串结束。  
返回值 gets()若成功则返回s指针，返回NULL则表示有错误发生。  
附加说明 由于gets()无法知道字符串s的大小，必须遇到换行字符或文件尾才会结束输入，因此容易造成缓冲溢出的安全性问题。建议使用fgets()取代。  
**范例 参考fgets()**　  
**21.mktemp（产生唯一的临时文件名）**相关函数 tmpfile  
表头文件 #include<stdlib.h>  
定义函数 char \* mktemp(char \* template);  
函数说明 mktemp()用来产生唯一的临时文件名。参数template所指的文件名称字符串中最后六个字符必须是XXXXXX。产生后的文件名会借字符串指针返回。  
返回值 文件顺利打开后，指向该流的文件指针就会被返回。如果文件打开失败则返回NULL，并把错误代码存在errno中。  
**附加说明** 参数template所指的文件名称字符串必须声明为数组，如:  
char template[ ]=”template-XXXXXX”；  
不可用char \* template=”template-XXXXXX”；  
**范例**

复制代码 代码如下:

#include<stdlib.h>  
main()  
{  
char template[ ]=”template-XXXXXX”;  
mktemp(template);  
printf(“template=%s/n”,template);  
}

**22.putc（将一指定字符写入文件中）**相关函数 fopen，fwrite，fscanf，fputc  
表头文件 #include<stdio.h>  
定义函数 int putc(int c,FILE \* stream);  
函数说明 putc()会将参数c转为unsigned char后写入参数stream指定的文件中。虽然putc()与fputc()作用相同，但putc()为宏定义，非真正的函数调用。  
返回值 putc()会返回写入成功的字符，即参数c。若返回EOF则代表写入失败。  
**范例 参考fputc（）。**　  
**23.putchar（将指定的字符写到标准输出设备）**相关函数 fopen，fwrite，fscanf，fputc  
表头文件 #include<stdio.h>  
定义函数 int putchar (int c);  
函数说明 putchar()用来将参数c字符写到标准输出设备。  
返回值 putchar()会返回输出成功的字符，即参数c。若返回EOF则代表输出失败。  
附加说明 putchar()非真正函数，而是putc(c，stdout)宏定义。  
**范例 参考getchar()。**　  
**24.rewind（重设文件流的读写位置为文件开头）**相关函数 fseek，ftell，fgetpos，fsetpos  
表头文件 #include<stdio.h>  
定义函数 void rewind(FILE \* stream);  
函数说明 rewind()用来把文件流的读写位置移至文件开头。参数stream为已打开的文件指针。此函数相当于调用fseek(stream,0,SEEK\_SET)。  
返回值  
**范例 参考fseek()  
  
25.setbuf（设置文件流的缓冲区）**相关函数 setbuffer，setlinebuf，setvbuf  
表头文件 #include<stdio.h>  
定义函数 void setbuf(FILE \* stream,char \* buf);  
函数说明 在打开文件流后，读取内容之前，调用setbuf()可以用来设置文件流的缓冲区。参数stream为指定的文件流，参数buf指向自定的缓冲区起始地址。如果参数buf为NULL指针，则为无缓冲IO。Setbuf()相当于调用:setvbuf(stream,buf,buf?\_IOFBF:\_IONBF,BUFSIZ)  
返回值  
　  
**26.setbuffer（设置文件流的缓冲区）**相关函数 setlinebuf，setbuf，setvbuf  
表头文件 #include<stdio.h>  
定义函数 void setbuffer(FILE \* stream,char \* buf,size\_t size);  
函数说明 在打开文件流后，读取内容之前，调用setbuffer()可用来设置文件流的缓冲区。参数stream为指定的文件流，参数buf指向自定的缓冲区起始地址，参数size为缓冲区大小。  
返回值  
  
**27.setlinebuf（设置文件流为线性缓冲区）**相关函数 setbuffer，setbuf，setvbuf  
表头文件 #include<stdio.h>  
定义函数 void setlinebuf(FILE \* stream);  
函数说明 setlinebuf()用来设置文件流以换行为依据的无缓冲IO。相当于调用:setvbuf(stream,(char \* )NULL,\_IOLBF,0);请参考setvbuf()。  
返回值  
  
**28.setvbuf（设置文件流的缓冲区）**相关函数 setbuffer，setlinebuf，setbuf  
表头文件 #include<stdio.h>  
定义函数 int setvbuf(FILE \* stream,char \* buf,int mode,size\_t size);  
函数说明 在打开文件流后，读取内容之前，调用setvbuf()可以用来设置文件流的缓冲区。参数stream为指定的文件流，参数buf指向自定的缓冲区起始地址，参数size为缓冲区大小，参数mode有下列几种  
\_IONBF 无缓冲IO  
\_IOLBF 以换行为依据的无缓冲IO  
\_IOFBF 完全无缓冲IO。如果参数buf为NULL指针，则为无缓冲IO。  
返回值  
  
**29.ungetc（将指定字符写回文件流中）**相关函数 fputc，getchar，getc  
表头文件 #include<stdio.h>  
定义函数 int ungetc(int c,FILE \* stream);  
函数说明 ungetc()将参数c字符写回参数stream所指定的文件流。这个写回的字符会由下一个读取文件流的函数取得。  
返回值 成功则返回c 字符，若有错误则返回EOF。

复制代码 代码如下:

#include <stdio.h>  
#include <stdlib.h>  
int main()  
{  
     FILE \*fp = NULL;  
     char\* str;  
     char re;   
     int num = 10;  
     str = (char\*)malloc(100);  
     //snprintf(str, 10,"test: %s", "0123456789012345678");  
    // printf("str=%s\n", str);  
     fp = fopen("/local/test.c","a+");  
     if (fp==NULL){  
        printf("Fail to open file\n");  
     }  
//     fseek(fp,-1,SEEK\_END);  
     num = ftell(fp);  
     printf("test file long:%d\n",num);  
     fscanf(fp,"%s",str);  
     printf("str = %s\n",str);  
     printf("test a: %s\n",str);  
     while ((re=getc(fp))!=EOF){//getc可以用作fgetc用  
        printf("%c",re);  
     }  
     //fread(str,10,10,fp);  
     fgets(str,100,fp);  
     printf("test a: %s\n",str);  
     sprintf(str,"xiewei test is:%s", "ABCDEFGHIGKMNI");  
     printf("str2=%s\n", str);  
   //  fprintf(fp,"%s\n",str);  
     fwrite(str,2,10,fp);  
     num = ftell(fp);  
     if(str!=NULL){  
        free(str);  
     }  
     fclose(fp);  
     return 0;  
}

基于typedef的用法详解

也许新手用这个关键字不多，但它却是一个很有用的关键字，可以使代码模块化程度更好（即与其它代码的关联较少），在C++中还是实现Traits技术的基础，也是模板编程的基本语法之一。  
  
若说变量定义是为变量命名，而typedef（或称为类型定义）就是为类型命名。既然都是命名，那就会有很多类似的地方。而变量定义我想大家都会使用，因此类型定义也必然会使用。  
  
类型定义的语法可以归结为一句话：只要在变量定义前面加上typedef，就成了类型定义。这儿的原本应该是变量的东西，就成为了类型。  
**如，下面的变量定义：**int integer;     //整型变量  
int \*pointer;   //整型指针变量  
int array [5]; //整型数组变量  
int \*p\_array [5]; //整型指针的数组的变量  
int (\*array\_pointer) [5];//整型数组的指针的变量  
int function (int param);//函数定义，也可将函数名看作函数的变量  
int \*function (int param);//仍然是函数，但返回值是整型指针  
int (\*function) (int param);//现在就是指向函数的指针了  
**若要定义相应类型，即为类型来起名字，就是下面的形式：**typedef int integer\_t;                      //整型类型  
typedef int \*pointer\_t;     //整型指针类型  
typedef int array\_t [5]; //整型数组类型  
typedef int \*p\_array\_t [5];    //整型指针的数组的类型  
typedef int (\*array\_pointer\_t) [5]; //整型数组的指针的类型  
typedef int function\_t (int param);     //函数类型  
typedef int \*function\_t (int param);    //函数类型  
typedef int (\*function\_t) (int param); //指向函数的指针的类型  
**注意：**上面的函数类型在C中可能会出错，因为C中并没有函数类型，它的函数变量会自动退化成函数指针；在C++中好像是可以的。在这里主要说明的是形式上的相似性.  
   
**typedef的一般形式为：**typedef   类型     定义名;  
在编程中使用typedef目的一般有两个，一个是给变量一个易记且意义明确的新名字，另一个是简化一些比较复杂的类型声明。  
其实，在C语言中声明变量的时候，有个存储类型指示符（storage-class-specifier），它包括我们熟悉的extern、static、auto、register。在不指定存储类型指示符的时候，编译器会根据约定自动取缺省值。另外，存储类型指示符的位置也是任意的（但要求在变量名和指针\*之前），也就是说以下几行代码是等价的：  
static const int i;  
const static int i;  
int const static i;  
const int static i;  
根据C语言规范，**在进行句法分析的时候，typedef和存储类型指示符是等价的**！所以，我们把上述使用static的地方替换为typedef：  
typedef const int i;  
const typedef int i;  
int const typedef i;  
const int typedef i;  
上述代码的语义是：将i定义为一个类型名，其等价的类型为const int。以后如果我们有i   a代码，就等价于const int a。对于有指针的地方也是一样的，比如：  
int const typedef \*t;那么代码t   p。就相当于int const \*p。  
**另外，typedef不能和static等存储类型指示符同时使用，因为每个变量只能有一种存储类型，所以代码：typedef static int i;是非法的。  
使用typedef简化复杂的变量声明**1）、定义一个有10个指针的数组，该指针指向一个函数，该函数有一个整形参数，并返回一个整型？  
第一种方法：int (\*a[10])(int);  
第二种方法：typedef int (\*pfunc)(int);  
             pfunc a[10];  
2）、定义一个有10个指针的数组，该指针指向一个函数，该函数有一个函数指针（不带参数，返回值为空）参数，并返回空。  
第一种方法：void (\*a[10])(void (\*)(void));  
第二种方法：typedef void (\*pfuncParam)(void);  
               typedef void (\*pfunc)(pfuncParam);  
pfunc a[10];  
3）、一个指向有10个函数指针（不带参数，返回值为double）数组的指针  
第一种方法：double (\*)(void) (\*p)[10];  
第二种方法：typedef double (\*pfunc)(void);  
             typedef pfunc (\*pfuncParam)[10];  
             pfuncParam p;  
**总结：  
typedef有两种用法：**一、一般形式，定义已有类型的别名  
　　typedef   类型    定义名;  
二、创建一个新的类型  
     typedef   返回值类型   新类型名(参数列表);

C语言中的回调函数实例

如果函数A的指针作为函数B的参数，在函数B中利用该指针调用函数A，则此时的A就是回调函数。

在C语言中一般用typedef来为回调函数定义别名（参数名）。 别名通过宏定义typedef来实现，不是简单的宏替换。可以用作同时声明指针型的多个对象。

比如：

复制代码 代码如下:

char \*pa，pb；//pa是一个char型指针，但pb是一个char型字符。我们可以这样来实现  
typedef char\* PCHAR；  
PCHAR pa，pb；//pa和pb都是char型指针

先看一个回调函数的例子：

复制代码 代码如下:

#include<stdio.h>

//方法指针的格式为：int (\*ptr)(char \*p) 即：返回值(指针名)(参数列表)  
typedef int (\*CallBackFun)(char \*p);    //为回调函数命名，类型命名为 CallBackFun，参数为char \*p

//方法 Afun，格式符合 CallBackFun 的格式，因此可以看作是一个 CallBackFun     
int Afun(char \*p)  
{  
    printf("Afun 回调打印出字符%s!\n", p);     
    return 0;  
}

// 方法 Cfun，格式符合 CallBackFun 的格式，因此可以看作是一个 CallBackFun  
int Cfun(char \*p)  
{     
    printf("Cfun 回调打印:%s, Nice to meet you!\n", p);     
    return 0;  
}

// 执行回调函数，方式一：通过命名方式，pCallBack可以看做是CallBackFun的别名  
int call(CallBackFun pCallBack, char \*p)  
{     
    printf("call 直接打印出字符%s!\n", p);     
    pCallBack(p);     
    return 0;  
}

// 执行回调函数，方式二：直接通过方法指针      
int call2(char \*p, int (\*ptr)())  //或者是int call2(char \*p, int (\*ptr)(char \*)) 同时ptr可以任意取名  
{  
    printf("==============\n", p);      
    (\*ptr)(p);  
}

int main()  
{     
    char \*p = "hello";  
    call(Afun, p);     
    call(Cfun, p);  
    call2(p, Afun);     
    call2(p, Cfun);  
    return 0;  
}  
再看一个回调函数的例子：

#include <stdio.h>  
typedef void (\*callback)(char \*);  
void repeat(callback function, char \*para)  
{  
    function(para);  
    function(para);  
}

void hello(char\* a)  
{  
     printf("Hello %s\n",(const char \*)a);  
}

void count(char \*num)  
{  
     int i;  
     for(i=1;i<(int)num;i++)  
          printf("%d",i);  
     putchar('\n');  
}

int main(void)  
{  
     repeat(hello,"Huangyi");  
     repeat(count, (char \*)4);  
}

本例中回调函数的参数按什么类型解释由调用者(repeat)规定，实现者(hello,count)就是一个void 指针，实现者只负责将这个指针转交给回调函数，而不关心它到底指向什么数据类型。调用者知道自己传的参数是char 型的，那么在自己提供的回调函数中就应该知道参数要转换成char \*型来解释。

C语言可变参数函数详解示例

一般我们编程的时候，函数中形式参数的数目通常是确定的，在调用时要依次给出与形式参数对应的实际参数。但在某些情况下我们希望函数的参数个数可以根据需要确定，因此c语言引入可变参数函数。典型的可变参数函数的例子有printf()、scanf()等,下面我就开始讲解

先看代码

代码如下:

printf(“hello,world!”);其参数个数为1个。  
printf(“a=%d,b=%ｓ,c=%c”,a,b,c);其参数个数为4个。

如何编写可变参数函数呢？我们首先来看看printf函数原型是如何定义的。  
在linux下，输入man 3 printf，可以看到prinf函数原型如下：

代码如下:

SYNOPSIS  
#include <stdio.h>  
int printf(const char \*format, ...);

后面的三个点...表示printf参数个数是不定的．  
如何实现可变参数函数？  
2. 编写可变函数准备  
为了编写可变参数函数，我们通常需要用到<stdarg.h>头文件下定义的以下函数：

代码如下:

void va\_start(va\_list ap, last);  
type va\_arg(va\_list ap, type);  
void va\_end(va\_list ap);  
void va\_copy(va\_list dest, va\_list src);

其中：  
va\_list是用于存放参数列表的数据结构。  
va\_start函数根据初始化last来初始化参数列表。  
va\_arg函数用于从参数列表中取出一个参数，参数类型由type指定。  
va\_copy函数用于复制参数列表。  
va\_end函数执行清理参数列表的工作。  
上述函数通常用宏来实现，例如标准ANSI形式下，这些宏的定义是：

# 使用C语言中的time函数获取系统时间

可以通过time()函数来获得计算机系统当前的日历时间（Calendar Time），处理日期时间的函数都是以本函数的返回值为基础进行运算。其原型为：  
time\_t time(time\_t \* t);  
如果你已经声明了参数t，你可以从参数t返回现在的日历时间，同时也可以通过返回值返回现在的日历时间，即从一个时间点（例如：1970年1月1日0时0分0秒）到现在此时的秒数。如果参数为空（NULL），函数将只通过返回值返回现在的日历时间，比如下面这个例子用来显示当前的日历时间：

复制代码 代码如下:

#include <SPAN style="FONT-FAMILY: Times New Roman"><stdio.h></SPAN>  
int main(void) {  
    time\_t t;  
    t=time(NULL);  
    printf("The number of seconds since January 1, 1970 is  %d\n",t);  
    return 0;  
}

运行的结果与当时的时间有关，我当时运行的结果是：   
The Calendar Time now is 1266637045  
其中1266637045就是我运行程序时的日历时间。即从1970年1月1日0时0分0秒到此时的秒数。  
第6行中给time函数的参数设置为NULL，可得到具体的秒数。  
可将第6行改写为以下形式:  
time(&t);  
变量t中存放当前的日期和时间(相当于函数返回值);

C语言中system()函数的用法总结

system()函数功能强大，很多人用却对它的原理知之甚少先看linux版system函数的源码：

代码如下:

#include <sys/types.h>  
#include <sys/wait.h>  
#include <errno.h>  
#include <unistd.h>

int system(const char \* cmdstring)  
{  
    pid\_t pid;  
    int status;

    if(cmdstring == NULL){        
         return (1);  
    }

    if((pid = fork())<0){  
            status = -1;  
    }

    else if(pid = 0){  
        execl("/bin/sh", "sh", "-c", cmdstring, (char \*)0);  
        -exit(127); //子进程正常执行则不会执行此语句  
        }

    else{  
           while(waitpid(pid, &status, 0) < 0){  
                if(errno != EINTER){  
                    status = -1;  
                    break;  
                }  
            }  
        }

        return status;  
}

分析一下原理估计就能看懂了：

当system接受的命令为NULL时直接返回，否则fork出一个子进程，因为fork在两个进程：父进程和子进程中都返回，这里要检查返回的pid，fork在子进程中返回0，在父进程中返回子进程的pid，父进程使用waitpid等待子进程结束，子进程则是调用execl来启动一个程序代替自己，execl("/bin/sh", "sh", "-c", cmdstring, (char\*)0)是调用shell，这个shell的路径是/bin/sh，后面的字符串都是参数，然后子进程就变成了一个shell进程，这个shell的参数是cmdstring，就是system接受的参数。在windows中的shell是command，想必大家很熟悉shell接受命令之后做的事了。  
  
再解释下fork的原理：当一个进程A调用fork时，系统内核创建一个新的进程B，并将A的内存映像复制到B的进程空间中，因为A和B是一样的，那么他们怎么知道自己是父进程还是子进程呢，看fork的返回值就知道，上面也说了fork在子进程中返回0，在父进程中返回子进程的pid。

windows中的情况也类似，就是execl换了个又臭又长的名字，参数名也换的看了让人发晕的，我在MSDN中找到了原型，给大家看看：

代码如下:

HINSTANCE   ShellExecute(  
               HWND   hwnd,  
               LPCTSTR   lpVerb,  
               LPCTSTR   lpFile,  
               LPCTSTR   lpParameters,  
               LPCTSTR   lpDirectory,   
               INT   nShowCmd   
   );

用法见下：  
ShellExecute(NULL,   "open",   "c:\\a.reg",   NULL,   NULL,   SW\_SHOWNORMAL);

你也许会奇怪 ShellExecute中有个用来传递父进程环境变量的参数 lpDirectory，linux中的execl却没有，这是因为execl是编译器的函数（在一定程度上隐藏具体系统实现），在linux中它会接着产生一个linux系统的调用 execve, 原型见下：  
int execve(const char \* file,const char \*\*argv,const char \*\*envp);  
  
看到这里就会明白为什么system（）会接受父进程的环境变量，但是用system改变环境变量后，system一返回主函数还是没变。原因从system的实现可以看到，它是通过产生新进程实现的，从我的分析中可以看到父进程和子进程间没有进程通信，子进程自然改变不了父进程的环境变量。

使用了system函数就能执行dos指令。

代码如下:

#include <stdio.h>  
#include <stdlib.h>  
xiaoyu()  
{  
char \*a;  
int n=0;  
FILE \*f;  
f=fopen("file.bat","w+");/\*新建一个批处理\*/  
if(f==NULL)  
exit(1);  
    a="echo"; /\*DOS命令\*/  
    for(n=65;n<=90;n++)/\*大写A-Z\*/  
    fprintf(f,"%s %c\n",a,n);/\*利用ASCII码输出A-Z，写出批处理\*/  
    fclose(f);  
    system("file.bat");/\*运行批处理\*/  
}  
main()  
{  
    char \*string;  
    xiaoyu();  
    string="echo C语言的system函数\n";/\*输出中文\*/  
    system(string);  
    system("pause");/\*程序暂停\*/  
}

C中可以使用DOS命令，以后编程通过调用DOS命令很多操作就简单多了。

# 基于C语言sprintf函数的深入理解

printf 可能是许多程序员在开始学习C语言时接触到的 第二个函数（我猜第一个是main），说起来，自然是老朋友了，可是，你对这个老朋友了解多吗？你对它的那个孪生兄弟sprintf了解多吗？在将各种类 型的数据构造成字符串时，sprintf的强大功能很少会让你失望。  
由于sprintf跟printf在用法上几乎一样，只是打印的目的地不同而已，前者打印到字符串中，后者则直接在命令行上输出。这也导致sprintf比printf有用得多。所以本文着重介绍sprintf，有时也穿插着用用 pritnf。  
**sprintf是个变参函数，定义如下：**int sprintf( char \*buffer, const char \*format [, argument] … );  
除了前两个参数类型固定外，后面可以接任意多个参数。而它的精华，显然就在第二个参数：格式化字符串上。  
printf 和sprintf都使用格式化字符串来指 定串的格式，在格式串内部使用一些以“%”开头的格式说明符（format specifications）来占据一个位置，在后边的变参列表中提供相应的变量，最终函数就会用相应位置的变量来替代那个说明符，产生一个调用者想要 的字符串。  
**1. 格式化数字字符串**sprintf最常见的应用之一莫过于把整数打印到字符串中，所以，spritnf在大多数场合可以替代itoa。如：  
//把整数123打印成一个字符串保存在s中。  
sprintf(s, “%d”, 123);   //产生“123″  
可以指定宽度，不足的左边补空格：  
sprintf(s, “%8d%8d”, 123, 4567); //产生：“    123    4567″  
当然也可以左对齐：  
sprintf(s, “%-8d%8d”, 123, 4567); //产生：“123         4567″  
也可以按照16进制打印：  
sprintf(s, “%8x”, 4567); //小写16进制，宽度占8个位置，右对齐  
sprintf(s, “%-8X”, 4568); //大写16进制，宽度占8个位置，左对齐  
这样，一个整数的16进制字符串就很容易得到，但我们在打印16进制内容时，通常想要一种左边补0的等宽格式，那该怎么做呢？很简单，在表示宽度的数字前面加个0就可以了。  
sprintf(s, “%08X”, 4567); //产生：“000011D7″  
上面以”%d”进行的10进制打印同样也可以使用这种左边补0的方式。  
这里要注意一个符号扩展的问题：比如，假如我们想打印短整数（short）-1的内存16进制表示形式，在Win32平台上，一个 short型占2个字节，所以我们自然希望用4个16进制数字来打印它：  
short si = -1;  
sprintf(s, “%04X”, si);  
产生“FFFFFFFF”，怎么回事？因为 spritnf是个变参函数，除了前面两个参数之外，后面的参数都不是类型安全的，函数更没有办法仅仅通过一个“%X”就能得知当初函数调用前参数压栈时 被压进来的到底是个4字节的整数还是个2字节的短整数，所以采取了统一4字节的处理方式，导致参数压栈时做了符号扩展，扩展成了32位的整数-1，打印时 4个位置不够了，就把32位整数-1的8位16进制都打印出来了。如果你想看si的本来面目，那么就应该让编译器做0扩展而不是符号扩展（扩展时二进制左 边补0而不是补符号位）：  
sprintf(s, “%04X”, (unsigned short)si);  
就可以了。或者：  
unsigned short si = -1;  
sprintf(s, “%04X”, si);  
sprintf和printf还可以按8进制打印整数字符串，使用”%o”。注意8进制和16进制都不会打印出负数，都是无符号的，实际上也就是变量的内部编码的直接的16进制或8进制表示。  
**2. 控制浮点数打印格式**浮点数的打印和格式控制是sprintf的又一大常用功能，浮点数使用格式符”%f”控制，默认保留小数点后6位数字，比如：  
sprintf(s, “%f”, 3.1415926);    //产生“3.141593″  
但有时我们希望自己控制打印的宽度和小数位数，这时就应该使用：”%m.nf”格式，其中m表示打印的宽度，n表示小数点后的位数。比如：  
sprintf(s, “%10.3f”, 3.1415626);   //产生：“     3.142″  
sprintf(s, “%-10.3f”, 3.1415626); //产生：“3.142     ”  
sprintf(s, “%.3f”, 3.1415626); //不指定总宽度，产生：“3.142″  
注意一个问题，你猜  
int i = 100;  
sprintf(s, “%.2f”, i);  
会打出什么东东来？“100.00”？对吗？自己试试就知道了，同时也试试下面这个：  
sprintf(s, “%.2f”, (double)i);  
第一个打出来的肯定不是正确结果，原因跟前面提到的一样，参数压栈时调用者并不知道跟i相对应的格式控制符是个”%f”。而函数执行时函数本身则并不知道当年被压入栈里的是个整数，于是可怜的保存整数i的那4个字节就被不由分说地强行作为浮点数格式来解释了，整个乱套了。  
不过，如果有人有兴趣使用手工编码一个浮点数，那么倒可以使用这种方法来检验一下你手工编排的结果是否正确。J  
字符/Ascii码对照  
我们知道，在C/C++语言中，char也是一种普通 的scalable类型，除了字长之外，它与short，int，long这些类型没有本质区别，只不过被大家习惯用来表示字符和字符串而已。（或许当年 该把这个类型叫做“byte”，然后现在就可以根据实际情况，使用byte或short来把char通过typedef定义出来，这样更合适些）  
于是，使用”%d”或者”%x”打印一个字符，便能得 出它的10进制或16进制的ASCII码；反过来，使用”%c”打印一个整数，便可以看到它所对应的ASCII字符。以下程序段把所有可见字符的 ASCII码对照表打印到屏幕上（这里采用printf，注意”#”与”%X”合用时自动为16进制数增加”0X”前缀）：  
for(int i = 32; i < 127; i++) {  
printf(”[ %c ]: %3d 0x%#04X/n”, i, i, i);  
}  
**3. 连接字符串**sprintf的格式控制串中既然可以插入各种东西，并最终把它们“连成一串”，自然也就能够连接字符串，从而在许多场合可以替代strcat，但sprintf能够一次连接多个字符串（自然也可以同时在它们中间插入别的内容，总之非常灵活）。比如：  
char\* who = “I”;  
char\* whom = “CSDN”;  
sprintf(s, “%s love %s.”, who, whom); //产生：“I love CSDN. ”  
strcat 只能连接字符串（一段以'/0'结尾的字 符数组或叫做字符缓冲，null-terminated-string），但有时我们有两段字符缓冲区，他们并不是以'/0'结尾。比如许多从第三方库函 数中返回的字符数组，从硬件或者网络传输中读进来的字符流，它们未必每一段字符序列后面都有个相应的'/0'来结尾。如果直接连接，不管是sprintf 还是strcat肯定会导致非法内存操作，而strncat也至少要求第一个参数是个null-terminated-string，那该怎么办呢？我们 自然会想起前面介绍打印整数和浮点数时可以指定宽度，字符串也一样的。比如：  
char a1[] = {'A', ‘B', ‘C', ‘D', ‘E', ‘F', ‘G'};  
char a2[] = {'H', ‘I', ‘J', ‘K', ‘L', ‘M', ‘N'};  
如果：  
sprintf(s, “%s%s”, a1, a2); //Don't do that!  
十有八九要出问题了。是否可以改成：  
sprintf(s, “%7s%7s”, a1, a2);  
也没好到哪儿去，正确的应该是：  
sprintf(s, “%.7s%.7s”, a1, a2);//产生：“ABCDEFGHIJKLMN”  
这可以类比打印浮点数的”%m.nf”，在”%m.ns”中，m表示占用宽度（字符串长度不足时补空格，超出了则按照实际宽度打印），n才表示从相应的字符串中最多取用的字符数。通常在打印字符串时m没什么大用，还是点号后面的n用的多。自然，也可以前后都只取部分字符：  
sprintf(s, “%.6s%.5s”, a1, a2);//产生：“ABCDEFHIJKL”  
在许多时候，我们或许还希望这些格式控制符中用以指定 长度信息的数字是动态的，而不是静态指定的，因为许多时候，程序要到运行时才会清楚到底需要取字符数组中的几个字符，这种动态的宽度/精度设置功能在 sprintf的实现中也被考虑到了，sprintf采用”\*”来占用一个本来需要一个指定宽度或精度的常数数字的位置，同样，而实际的宽度或精度就可以 和其它被打印的变量一样被提供出来，于是，上面的例子可以变成：  
sprintf(s, “%.\*s%.\*s”, 7, a1, 7, a2);  
或者：  
sprintf(s, “%.\*s%.\*s”, sizeof(a1), a1, sizeof(a2), a2);  
实际上，前面介绍的打印字符、整数、浮点数等都可以动态指定那些常量值，比如：  
sprintf(s, “%-\*d”, 4, ‘A'); //产生“65 ”  
sprintf(s, “%#0\*X”, 8, 128);    //产生“0X000080″，“#”产生0X  
sprintf(s, “%\*.\*f”, 10, 2, 3.1415926); //产生“      3.14″  
**4. 打印地址信息**有时调试程序时，我们可能想查看某些变量或者成员的地址，由于地址或者指针也不过是个32位的数，你完全可以使用打印无符号整数的”%u”把他们打印出来：  
sprintf(s, “%u”, &i);  
不过通常人们还是喜欢使用16进制而不是10进制来显示一个地址：  
sprintf(s, “%08X”, &i);  
然而，这些都是间接的方法，对于地址打印，sprintf 提供了专门的”%p”：  
sprintf(s, “%p”, &i);  
我觉得它实际上就相当于：  
sprintf(s, “%0\*x”, 2 \* sizeof(void \*), &i);  
**5. 利用sprintf的返回值**较少有人注意printf/sprintf函数的返回值，但有时它却是有用的，spritnf返回了本次函数调用最终打印到字符缓冲区中的字符数目。也就是说每当一次sprinf调用结束以后，你无须再调用一次strlen便已经知道了结果字符串的长度。如：  
int len = sprintf(s, “%d”, i);  
对于正整数来说，len便等于整数i的10进制位数。  
下面的是个完整的例子，产生10个[0, 100)之间的随机数，并将他们打印到一个字符数组s中，以逗号分隔开。

复制代码 代码如下:

#include <stdio.h>  
#include <time.h>  
#include <stdlib.h>  
int main() {  
srand(time(0));  
char s[64];  
int offset = 0;  
for(int i = 0; i < 10; i++) {  
offset += sprintf(s + offset, “%d,”, rand() % 100);  
}  
s[offset - 1] = ‘/n';//将最后一个逗号换成换行符。  
printf(s);  
return 0;  
}

设想当你从数据库中取出一条记录，然后希望把他们的各 个字段按照某种规则连接成一个字符串时，就可以使用这种方法，从理论上讲，他应该比不断的strcat效率高，因为strcat每次调用都需要先找到最后 的那个'/0'的位置，而在上面给出的例子中，我们每次都利用sprintf返回值把这个位置直接记下来了。  
**6. 使用sprintf的常见问题**sprintf是个变参函数，使用时经常出问题，而且只要出问题通常就是能导致程序崩溃的内存访问错误，但好在由sprintf误用导致的问题虽然严重，却很容易找出，无非就是那么几种情况，通常用眼睛再把出错的代码多看几眼就看出来了。  
?? 缓冲区溢出  
第一个参数的长度太短了，没的说，给个大点的地方吧。当然也可能是后面的参数的问题，建议变参对应一定要细心，而打印字符串时，尽量使用”%.ns”的形式指定最大字符数。  
?? 忘记了第一个参数  
低级得不能再低级问题，用printf用得太惯了。//偶就常犯。：。（  
?? 变参对应出问题  
通常是忘记了提供对应某个格式符的变参，导致以后的参数统统错位，检查检查吧。尤其是对应”\*”的那些参数，都提供了吗？不要把一个整数对应一个”%s”，编译器会觉得你欺她太甚了（编译器是obj和exe的妈妈，应该是个女的，:P）。  
**7. strftime**sprintf还有个不错的表妹：strftime，专门用于格式化时间字符串的，用法跟她表哥很像，也是一大堆格式控制符，只是毕竟小姑娘家心细，她还要调用者指定缓冲区的最大长度，可能是为了在出现问题时可以推卸责任吧。这里举个例子：

复制代码 代码如下:

time\_t t = time(0);  
//产生“YYYY-MM-DD hh:mm:ss”格式的字符串。  
char s[32];  
strftime(s, sizeof(s), “%Y-%m-%d %H:%M:%S”, localtime(&t));

sprintf在MFC中也能找到他的知音：CString::Format，strftime在MFC中自然也有她的同道：CTime::Format，这一对由于从面向对象哪里得到了赞助，用以写出的代码更觉优雅。  
**8. 后记**本文介绍的所有这些功能，在MSDN中都可以很容易地查到，笔者只是根据自己的使用经验，结合一些例子，把一些常用的，有用的，而可能为许多初学者所不知的用法介绍了一点，希望大家不要笑话，也希望大家批评指正。  
有人认为这种带变参的函数会引起各种问题，因而不提倡使用。但笔者本人每每还是抵挡不了它们强大功能的诱惑，在实际工作中一直在使用。实际上，C#.NET 从开始就支持变参，刚发布不久的Java5.0也支持变参了。  
①获取System时间: void GetSystemTime(LPSYSTEMTIME lpSystemTime); 下面是例子:

复制代码 代码如下:

#include <windows.h>  
#include <stdio.h>  
#include <stdlib.h>  
void main() {  
SYSTEMTIME st; //定义存放时间的结构体  
char strTime[256];  
int n=0;  
GetSystemTime(&st);  
n = sprintf(strTime,”Year:/t%d/n”,st.wYear);  
n += sprintf(strTime+n,”Month:/t%d/n”,st.wMonth);  
n += sprintf(strTime+n,”Day:/t%d/n”,st.wDay);  
n += sprintf(strTime+n,”Date:/t%d/n”,st.wDayOfWeek);  
n += sprintf(strTime+n,”Hour:/t%d/n”,st.wHour);  
n += sprintf(strTime+n,”Minute:/t%d/n”,st.wMinute);  
n += sprintf(strTime+n,”Second:/t%d/n”,st.wSecond);  
n += sprintf(strTime+n,”MilliSecond:/t%d/n”,st.wMilliseconds);  
printf(”%s”,strTime);  
system(”pause”);  
}

自定义函数：

　函数是C源码程序中最基本的功能单位，是一个可以从程序其它地方调用执行的语句块

　　C语言是一种结构化程序设计语言，结构化程序设计思想是“分解”大问题，依次解决小问题，通过小问题解决实现大问题的解决，描述“小问题”解决方法的工具即是函数。

　　函数的定义格式如下：

|  |
| --- |
|  |

|  |
| --- |
| type name ( argument1, argument2, ...) statement |

　　说明：

　　type 是函数返回的数据的类型

　　name 是函数被调用时使用的名

　　argument 是函数调用需要传入的参量(可以声明任意多个参量)。每个参量(argument)由一个数据类型后面跟一个标识名称组成，就像变量声明中一样(例如，int x)。参量仅在函数范围内有效，可以和函数中的其它变量一样使用， 它们使得函数在被调用时可以传入参数，不同的参数用逗号(comma)隔开.

　　statement 是函数的内容。它可以是一句指令，也可以是一组指令组成的语句块。如果是一组指令，则语句块必须用花括号{}括起来，这也是我们最常见到情况。其实为了使程序的格式更加统一清晰，建议在仅有一条指令的时候也使用花括号，这是一个良好的编程习惯。

程序设计中的函数  
许多程序设计语言中，可以将一段经常需要使用的代码封装起来，在需要使用时可以直接调用，这就是程序中的函数。比如在C语言中:  
int max(int x,int y)  
{  
return(x>y?x:y;);  
}  
就是一段比较两数大小的函数，函数有参数与返回值。C++程序设计中的函数可以分为两类：带参数的函数和不带参数的函数。这两种参数的声明、定义也不一样。  
带有（一个）参数的函数的声明：  
类型名标示符+函数名+（类型标示符+参数）  
{  
}  
不带参数的函数的声明：  
void+函数名（）  
{  
}  
花括号内为函数体。  
带参数的函数有返回值，不带参数的没有返回值。  
C++中函数的调用：函数必须声明后才可以被调用。调用格式为：函数名（实参）  
调用时函数名后的小括号中的实参必须和声明函数时的函数括号中的形参个数相同。  
有返回值的函数可以进行计算，也可以做为右值进行赋值。  
#include <iostream>  
using namespace std;  
int [f1](https://www.baidu.com/s?wd=f1&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)(int x, inty)  
{int z;  
return x+y;  
}  
void main()  
{cout<<[f1](https://www.baidu.com/s?wd=f1&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)(50,660)<<endl  
}  
C语言中的部分函数  
main（主函数）  
max（求最大数的函数）  
scanf（输入函数）  
printf（输出函数）

通俗的来理解说就是：在程序中需要c语言函数进行简易化操作，有了函数大大减少了庞大程序中的错误和不清楚问题，例如：我要把加好的数放到函数中去打印，这样就可用一两句c语言语句来编写，否则会相当麻烦要写上许多条语句

推荐的几个词典类型的网站：

<http://c.biancheng.net/ref/>

<https://www.jb51.net/article/75765.htm>

# 函数：

事实上：函数本身是没有类型的，只有函数的返回值有类型。以前一直误以为的函数类型其实是函数返回值的类型。

在递归中调用函数的开销比循环要大很多，递归的效率很低。递归的深度太大可能会出现错误（栈溢出）

在c语言中函数声明不是必须的，即使没有声明函数，gcc编译器也只是会提示警告。但是函数声明却是很有必要的。

函数声明的作用：

其实函数声明的作用是让编译器帮你检查你调用函数时有没有错误。比如参数的数量是否正确，如果调用函数时候少

传入一个参数，并且没有声明该函数，编译器无法知道你调用是否正确，只会提示一个警告。很多人会忽略警告，导致最后程序运行时出现异常。

如果不声明函数，编译器就发现不了错误：

编译器在编译过程中依次生成对应源文件的可重定位目标文件（.o），每个源文件中调用的函数在链接前都是以符号的

形式体现在.o文件中。在编译过程中不会去检查某个函数的形式，因为函数参数是通过寄存器和压栈来处理的，直接把函数翻译成符号，编译器是不知道关于函数参数的信息的，最后交给连接器把符号翻译成地址。所以链接的时候只要能找到对应得符号就不会报错。

函数指针变量的声明

就像某一数据变量的内存地址可以存储在相应的指针变量中一样，函数的首地址也以存储在某个函数指针变量里的。这样，我就可以通过这个函数指针变量来调用所指向的函数了。  
在C系列语言中，任何一个变量，总是要先声明，之后才能使用的。那么，函数指针变量也应该要先声明吧？那又是如何来声明呢？以上面的例子为例，我来声明一个可以指向MyFun函数的函数指针变量FunP。下面就是声明FunP变量的方法：  
   void (\*FunP)(int) ; /\* 也可写成void (\*FunP)(int x)\*/  
你看，整个函数指针变量的声明格式如同函数MyFun的声明处一样，只不过——我们把MyFun改成“(\*FunP)”而已，这样就有了一个能指向MyFun函数的指针FunP了。（当然，这个FunP指针变量也可以指向所有其它具有相同参数及返回值的函数了。）

通过函数指针变量调用函数

有了FunP指针变量后，我们就可以对它赋值指向MyFun，然后通过FunP来调用MyFun函数了。看我如何通过FunP指针变量来调用MyFun函数的：  
/\* 自行包含头文件 \*/  
void MyFun(int x); /\* 这个声明也可写成：void MyFun( int )\*/  
void (\*FunP)(int ); /\*也可声明成void(\*FunP)(int x)，但习惯上一般不这样。 \*/  
int main(int argc, char\* argv[])  
{  
   MyFun(10); /\* 这是直接调用MyFun函数 \*/  
   FunP = &MyFun; /\* 将MyFun函数的地址赋给FunP变量 \*/  
   (\*FunP)(20); /\* （★）这是通过函数指针变量FunP来调用MyFun函数的。 \*/  
}  
void MyFun(int x) /\* 这里定义一个MyFun函数 \*/  
{  
   printf("%d\n",x);  
}  
请看（★）行的代码及注释。运行看看。嗯，不错，程序运行得很好。哦，我的感觉是：MyFun与FunP的类型关系类似于int 与int \* 的关系。函数MyFun好像是一个如int的变量（或常量），而FunP则像一个如int \* 一样的指针变量。  
   int i,\*pi;  
   pi = &i; /\* 与FunP = &MyFun比较。\*/  
（你的感觉呢？）呵呵，其实不然……

调用函数的其它书写格式

函数指针也可如下使用，来完成同样的事情：  
/\* 自行包含头文件 \*/  
void MyFun(int x);  
void (\*FunP)(int );/\* 声明一个用以指向同样参数，返回值函数的指针变量。 \*/  
int main(int argc, char\* argv[])  
{  
   MyFun(10); /\* 这里是调用MyFun(10)函数 \*/  
   FunP = MyFun; /\* 将MyFun函数的地址赋给FunP变量 \*/  
   FunP(20); /\* （★）这是通过函数指针变量来调用MyFun函数的。\*/  
   return 0;  
}  
void MyFun(int x) //这里定义一个MyFun函数  
{  
   printf("%d\n",x);  
}  
我改了（★）行（请自行与之前的代码比较一下）。运行试试，啊！一样地成功。咦？  
   FunP = MyFun;  
可以这样将MyFun值同赋值给FunP，难道MyFun与FunP是同一数据类型（即如同的int 与int的关系），而不是如同int 与int\*的关系了？（有没有一点点的糊涂了？）看来与之前的代码有点矛盾了，是吧！所以我说嘛！  
  
请容许我暂不给你解释，继续看以下几种情况（这些可都是可以正确运行的代码哟！）：  
代码之三：  
int main(int argc, char\* argv[])  
{  
   MyFun(10); /\* 这里是调用MyFun(10)函数 \*/  
   FunP = &MyFun; /\* 将MyFun函数的地址赋给FunP变量 \*/  
   FunP(20); /\* 这是通过函数指针变量来调用MyFun函数的。 \*/  
   return 0;  
}  
  
代码之四：  
int main(int argc, char\* argv[])  
{  
   MyFun(10); /\* 这里是调用MyFun(10)函数 \*/  
   FunP = MyFun; /\* 将MyFun函数的地址赋给FunP变量 \*/  
   (\*FunP)(20); /\*这是通过函数指针变量来调用MyFun函数的。\*/  
   return 0;  
}  
真的是可以这样的噢！（哇！真是要晕倒了！）还有呐！看——  
int main(int argc, char\* argv[])  
{  
   (\*MyFun)(10); /\*看，函数名MyFun也可以有这样的调用格式\*/  
   return 0;  
}  
你也许第一次见到吧：函数名调用也可以是这样写的啊！（只不过我们平常没有这样书写罢了。）那么，这些又说明了什么呢？  
  
呵呵！依据以往的知识和经验来推理本篇的“新发现”，我想就连“福尔摩斯”也必定会由此分析并推断出以下的结论：  
1）其实，MyFun的函数名与FunP函数指针都是一样的，即都是函数指针。MyFun函数名是一个函数指针常量，而FunP是一个函数数指针变量，这是它们的关系。  
2）但函数名调用如果都得如(\*MyFun)(10)这样，那书写与读起来都是不方便和不习惯的。所以C语言的设计者们才会设计成又可允许MyFun(10)这种形式地调用（这样方便多了并与数学中的函数形式一样，不是吗？）。  
3）为统一起见，FunP函数指针变量也可以FunP(10)的形式来调用。  
4）赋值时，即可FunP = &MyFun形式，也可FunP = MyFun。  
  
上述代码的写法，随便你爱怎么着！请这样理解吧！这可是有助于你对函数指针的应用喽！最后 ——  
  
补充说明一点，在函数的声明处：  
   void MyFun(int); /\*不能写成void (\*MyFun)(int)。\*/  
   void (\*FunP)(int); /\*不能写成void FunP(int)。\*/  
（请看注释）这一点是要注意的。

定义某一函数的指针类型

就像自定义数据类型一样，我们也可以先定义一个函数指针类型，然后再用这个类型来声明函数指针变量。  
我先给你一个自定义数据类型的例子。  
typedef int\* PINT; /\* 为int\* 类型定义了一个PINT的别名\*/  
int main()  
{  
   int x;  
   PINT px = &x; /\* 与“int \*px=&x;”是等价的。PINT类型其实就是int \* 类型 \*/  
   \*px = 10; /\* px就是int\*类型的变量 \*/  
   return 0;  
}  
根据注释，应该不难看懂吧！（虽然你可能很少这样定义使用，但以后学习Win32编程时会经常见到的。）下面我们来看一下函数指针类型的定义及使用：（请与上对照！）  
/\* 自行包含头文件 \*/  
void MyFun(int x); /\*此处的声明也可写成：void MyFun( int )\*/  
typedef void (\*FunType)(int); /\*（★）这样只是定义一个函数指针类型\*/  
FunType FunP; /\*然后用FunType类型来声明全局FunP变量\*/  
int main(int argc, char\* argv[])  
{  
   FunType FunP; /\*函数指针变量当然也是可以是局部的 ，那就请在这里声明了。 \*/  
   MyFun(10);  
   FunP = &MyFun;  
   return 0;  
}  
void MyFun(int x)  
{  
   printf("%d\n",x);  
}  
看（★）行：  
首先，在void (\*FunType)(int)前加了一个typedef 。这样只是定义一个名为FunType函数指针类型，而不是一个FunType变量。  
然后，“FunType FunP;”这句就如“PINT px;”一样地声明一个FunP变量。  
  
其它相同。整个程序完成了相同的事。这样做法的好处是：  
有了FunType类型后，我们就可以同样地、很方便地用FunType类型来声明多个同类型的函数指针变量了。如下：  
   FunType FunP2;  
   FunType FunP3;  
   /\* . . . \*/

函数指针作为某个函数的参数

既然函数指针变量是一个变量，当然也可以作为某个函数的参数来使用的。所以，你还应知道函数指针是如何作为某个函数的参数来传递使用的。  
  
给你一个实例：  
要求：我要设计一个CallMyFun函数，这个函数可以通过参数中的函数指针值不同来分别调用MyFun1、MyFun2、MyFun3这三个函数（注：这三个函数的定义格式应相同）。  
实现：代码如下：  
/\* 自行包含头文件 \*/  
void MyFun1(int x);  
void MyFun2(int x);  
void MyFun3(int x);  
typedef void (\*FunType)(int ); /\* ②. 定义一个函数指针类型FunType,与①函数类型一致 \*/  
void CallMyFun(FunType fp,int x);  
int main(int argc, char\* argv[])  
{  
   CallMyFun(MyFun1,10); /\* ⑤. 通过CallMyFun函数分别调用三个不同的函数 \*/  
   CallMyFun(MyFun2,20);  
   CallMyFun(MyFun3,30);  
}  
void CallMyFun(FunType fp,int x) /\* ③. 参数fp的类型是FunType。\*/  
{  
   fp(x);/\* ④. 通过fp的指针执行传递进来的函数，注意fp所指的函数是有一个参数的。 \*/  
}  
void MyFun1(int x) /\* ①. 这是个有一个参数的函数，以下两个函数也相同。 \*/  
{  
   printf("函数MyFun1中输出：%d\n",x);  
}  
void MyFun2(int x)  
{  
   printf("函数MyFun2中输出：%d\n",x);  
}  
void MyFun3(int x)  
{  
   printf("函数MyFun3中输出：%d\n",x);  
}  
输出结果：略分析：看我写的注释。你可按我注释的①②③④⑤顺序自行分析。