**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。

文件类型函数：

**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

复制代码 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：

复制代码 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：

复制代码 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：

复制代码 代码如下:

typedef char \* va\_list; //字符串指针  
#define \_INTSIZEOF(n) ( (sizeof(n) + sizeof(int) - 1) & ~(sizeof(int) - 1) )  
#define va\_start(ap,v) ( ap = (va\_list)&v + \_INTSIZEOF(v) )  
#define va\_arg(ap,t) ( \*(t \*)((ap += \_INTSIZEOF(t)) - \_INTSIZEOF(t)) )  
#define va\_end(ap) ( ap = (va\_list)0 )

使用宏\_INTSIZEOF是为了按照整数字节对齐指针，因为c调用协议下面，参数入栈都是整数字节（指针或者值）。  
函数官方说明，如果你看到英文就烦，可以自行忽略以下说明。  
va\_start()  
       The  va\_start() macro initializes ap for subsequent use by va\_arg() and  
       va\_end(), and must be called first.  
       The argument last is the name of the last argument before the  variable  
       argument list, that is, the last argument of which the calling function  
       knows the type.  
       Because the address of this argument may  be  used  in  the  va\_start()  
       macro,  it should not be declared as a register variable, or as a func‐  
       tion or an array type.  
va\_arg()  
       The va\_arg() macro expands to an expression that has the type and value  
       of  the  next  argument in the call.  The argument ap is the va\_list ap  
       initialized by va\_start().  Each call to va\_arg() modifies ap  so  that  
       the  next  call returns the next argument.  The argument type is a type  
       name specified so that the type of a pointer to an object that has  the  
       specified type can be obtained simply by adding a \* to type.  
       The  first use of the va\_arg() macro after that of the va\_start() macro  
       returns the argument after last.   Successive  invocations  return  the  
       values of the remaining arguments.  
       If  there  is  no  next argument, or if type is not compatible with the  
       type of the actual next argument (as promoted according to the  default  
       argument promotions), random errors will occur.  
       If  ap is passed to a function that uses va\_arg(ap,type) then the value  
       of ap is undefined after the return of that function.  
va\_end()  
       Each invocation of va\_start() must be matched by a corresponding  invo‐  
       cation of va\_end() in the same function.  After the call va\_end(ap) the  
       variable ap is undefined.  Multiple traversals of the list, each brack‐  
       eted  by va\_start() and va\_end() are possible.  va\_end() may be a macro  
       or a function.  
ＧＮＵ给出的一个实例：

复制代码 代码如下:

#include <stdio.h>  
#include <stdarg.h>  
void  
foo(char \*fmt, ...)  
{  
  va\_list ap;  
  int d;  
  char c, \*s;  
　va\_start(ap, fmt);  
　while (\*fmt)  
     switch (\*fmt++) {  
     case 's': /\* string \*/  
     s = va\_arg(ap, char \*);  
         printf("string %s\n", s);  
         break;  
     case 'd': /\* int \*/  
         d = va\_arg(ap, int);  
         printf("int %d\n", d);  
         break;  
     case 'c': /\* char \*/  
/\* need a cast here since va\_arg only takes fully promoted types \*/  
        c = (char) va\_arg(ap, int);  
        printf("char %c\n", c);  
        break;  
   }  
   va\_end(ap);  
}

说明：  
va\_start(ap, fmt);用于根据fmt初始化可变参数列表。  
va\_arg(ap, char \*);用于从参数列表中取出一个参数，其中的char \*用于指定所取的参数的类型为字符串。每次调用va\_arg后，参数列表ａｐ都会被更改，以使得下次调用时能得到下一个参数。  
va\_end(ap);用于对参数列表进行一些清理工作。调用完va\_end后，ap便不再有效。  
以上程序给了我们一个实现printf函数的是思路，即：通过调用va\_start函数，来得到参数列表，然后我们一个个取出参数来进行输出即可。  
3.实例  
例如：对于printf(“a=%d,b=%ｓ,c=%c”,a,b,c)语句;fmt的值为a=%d,b=%ｓ,c=%c，调用va\_start函数将参数a,b,c存入了ap中。注意到：fmt中的%为特殊字符，紧跟%后的参数指明了参数类型．  
因此我们的简易printf函数如下：

复制代码 代码如下:

#include <stdio.h>  
#include <stdarg.h>  
void  
myprintf(char \*fmt, ...)  
{  
  va\_list ap;  
  int d;  
  double f;  
  char c;   
  char \*s;  
  char flag;  
  va\_start(ap,fmt);  
  while (\*fmt){  
  　flag=\*fmt++;  
  　if(flag!='%'){  
 putchar(flag);  
 continue;  
　　}  
　　flag=\*fmt++;//记得后移一位  
    switch (flag)   
　　{   
　　　case 's':  
 s=va\_arg(ap,char\*);  
 printf("%s",s);  
 break;  
　　　case 'd': /\* int \*/           
 d = va\_arg(ap, int);           
 printf("%d", d);           
 break;       
　　　case 'f': /\* double\*/           
 d = va\_arg(ap,double);           
 printf("%d", d);           
 break;  
　　　case 'c': /\* char\*/     
 c = (char)va\_arg(ap,int);          
 printf("%c", c);          
 break;  
　　　default:  
 putchar(flag);  
 break;  
　　}     
  }  
  va\_end(ap);  
}  
int main(){  
　　char str[10]="linuxcode";  
　　int i=1024;  
　　double f=3.1415926;  
　　char c='V';  
　　myprintf("string is:%s,int is:%d,double is:%f,char is :%c",str,i,f,c);  
}

从上面我们可以知道可变参数函数的编写，必须要传入一个参数fmt，用来告诉我们的函数怎样去确定参数的个数。我们的可变参数函数是通过自己解析这个参数来确定函数参数个数的。  
比如，我们编写一个求和函数，其函数实现如下：

复制代码 代码如下:

int sum(int cnt,...){  
    int sum=0;  
int i;  
    va\_list ap;  
    va\_start(ap,cnt);  
for(i=0;i<cnt;++i)  
 sum+=va\_arg(ap,int);  
    va\_end(ap);  
return sum;  
}

总结一下就是：通过va\_start初始化参数列表(也就能得到具体的参数个数了)，然后使用va\_arg函数从参数列表中取出你想要的参数，最后调用va\_end执行清理工作。

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：

复制代码 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：

复制代码 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。

复制代码 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

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

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

# c语言中函数声明的作用

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。

那么函数声明到底有声明作用呢？

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。

为什么如果不声明函数，编译器发现不了错误？

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。

**C语言中函数名的意义深究**

**一、通常的函数调用**

一个通常的函数调用的例子：  
/\* 自行包含头文件 \*/  
void MyFun(int x); /\* 此处的声明也可写成：void MyFun(int) \*/  
int main(int argc, char\* argv[])  
{  
   MyFun(10); /\* 这里是调用MyFun(10) 函数 \*/  
   return(0);  
}  
void MyFun(int x) /\* 这里定义一个MyFun函数 \*/  
{  
   printf("%d\n",x);  
}  
这个MyFun函数是一个无返回值的函数，它并不“完成”什么事情。这种调用函数的格式你应该是很熟悉的吧！看主函数中调用MyFun函数的书写格式：  
   MyFun(10);  
我们一开始只是从功能上或者说从数学意义上理解MyFun这个函数，知道MyFun函数名代表的是一个功能（或是说一段代码）。直到——学习到函数指针概念时。我才不得不在思考：函数名到底又是什么东西呢？  
  
（不要以为这是没有什么意义的事噢！呵呵，继续往下看你就知道了。）

**二、函数指针变量的声明**

就象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。）

**三、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……

**四、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。

**五、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;  
   /\* . . . \*/

**六、函数指针作为某个函数的参数**

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：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。