

# 计算机程序设计基础(1)

## ---C语言程序设计(10B)

孙甲松

[sunjiasong@tsinghua.edu.cn](mailto:sunjiasong@tsinghua.edu.cn)

电子工程系 信息认知与智能系统研究所

罗姆楼6-104

电话: 13901216180/62796193

2022.11.

# 第10章 指针 第2部分

## 第2部分 目录

### 10.4 动态内存的申请与释放

#### 10.4.1 malloc( )函数

#### 10.4.2 calloc( )函数

#### 10.4.3 realloc( )函数

#### 10.4.4 free( )函数

### 10.5 字符串与指针

#### 10.5.1 字符串指针

#### 10.5.2 字符串指针作为函数参数

#### 10.5.3 strstr( )函数

### 10.6 函数与指针

#### 10.6.1 用函数指针变量调用函数

#### 10.6.2 函数指针数组

#### 10.6.3 函数指针变量作为函数参数

#### 10.6.4 返回指针值的函数

### 10.7 main函数的形参

### 10.8 变步长梯形求积法

## 10.4 动态内存的申请与释放

- 在函数内定义的局部数组，是在内存栈上分配内存空间，通常不能开太大的局部数组。要想开大数组有两种方法：
  - ① 在函数外定义外部数组或静态数组。
  - ② 在内存堆上申请动态内存，建立大数组。
- 在函数外定义外部数组或静态数组，数组将在程序运行时先在静态存储区开辟。而且一旦定义了外部大数组或静态大数组，即使不用也会一直占用相应内存空间不释放，这会造成不能同时定义多个外部大数组或静态大数组。
- 能否在使用时申请内存块开辟大数组，而一旦使用完毕即释放，便于内存重复使用（内存复用）和再申请呢？可以通过动态内存的申请与释放来实现。
- C语言提供了malloc()、calloc()、realloc()等函数来进行动态内存申请，要使用这些函数必须包含头文件 stdlib.h 或 malloc.h，下面简要介绍一下如何使用这些函数。

从高  
向低



从低  
向高



内存堆(heap)

内存栈(stack)

静态数据区

程序区

操作系统区

内存地址: FFFFFFFF

内存地址: 00000000

计算机内存分布示意图

## 10.4.1 malloc()函数

- ✓ malloc()函数的功能是从内存堆（heap）上申请指定字节数的内存块，并返回该内存块的首地址。malloc函数的原型是：

void \*malloc(申请内存的字节数)

- ✓ 调用malloc函数时，必须进行强制类型转换：(类型\*)malloc  
把返回的内存首地址转换成相应指针的类型。例如：

```
#include <stdlib.h>
```

```
char *p;
```

```
double **q;
```

```
int (*a)[4];
```

```
p=(char *)malloc(sizeof(char)*20);
```

```
q=(double **)malloc(sizeof(double *)*10);
```

```
a=(int (*)[4])malloc(sizeof(int)*4*5);
```

### 10.4.1 malloc()函数

- ✓ `p=(char *)malloc(sizeof(char)*20);` 函数malloc从内存堆上动态申请一块20个char型大小的内存块给p，形成一个长度为20的动态char型数组。（20个字节）
- ✓ `q=(double **)malloc(sizeof(double *)*10);` 函数malloc从内存堆上动态申请一块10个double指针类型大小的内存块给q，形成一个长度为10的动态double型指针数组。（40个字节）
- ✓ `a=(int (*)[4])malloc(sizeof(int)*4*5);` 函数malloc从内存堆上动态申请一块20个int型大小的内存块给a，形成一个 $5 \times 4$ 的动态int型二维数组。（80个字节）
- ✓ 函数名前()内强制类型转换的类型即为：指针定义格式去掉指针变量名，例如：

`int (*a)[4];`

去掉指针名a后的类型为: `int (*)[4]`

### 10.4.1 malloc()函数

- ✓ 若内存申请不成功，malloc 将返回空指针（NULL），因此在执行malloc后，应该首先通过判断p是否为NULL，来判断内存申请是否成功，以防止对空指针进行操作产生致命错误。例如：

```
char *p;  
p=(char *)malloc(sizeof(char) * 10);  
if (p == NULL)  
{ printf("Can't get memory!\n");  
  exit(1); /* 强制终止当前程序的执行 */  
}  
free(p);
```

- ✓ 需要强调，每一个malloc调用应该对应有一个free释放相应内存块到动态内存堆上。（有借有还.....）



- ✓ 在编程时，总有人想先读入数组长度，然后按读入的长度定义数组：

```
int n;  
scanf("%d", &n);  
double a[n];
```

- ✓ 但这是绝对不允许的，即使C++也不允许这样定义静态数组。
- ✓ 但可以利用指针，生成动态一维数组，数组长度n由键盘输入。这样可以存取一维动态数组  $p[i]$ ， $0 \leq i < n$ 。例如有程序段：

```
int n; double *p;  
scanf("%d", &n);  
p = (double *)malloc(sizeof(double)*n);
```

上面的语句为p申请了n个double单元的动态内存块，可以把p当做一个长度为n的double型数组来使用。读写  $p[i]$ ， $0 \leq i < n$

```

#include <stdio.h>
#include <stdlib.h>
void main( )
{ double *p;
  int n, i;
  scanf("%d", &n);
  p = (double *)malloc(sizeof(double)*n);
  if (p == NULL)
  { printf("Can't get memory!\n");
    exit(1);
  }
  for (i=0; i<n; i++)
    p[i] = i;
  for (i=0; i<n; i++)
    printf("%3.0f",p[i]);
  free(p);
}

```

程序的运行结果为:

20

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

一件很有趣的事:

```
#include <stdio.h>
#include <stdlib.h>
void main( )
{ char *p, q[20];
  int n;
  scanf("%d", &n);
  p = (char*)malloc(sizeof(char)*n);
  if (p == NULL)
  { printf("Can't get memory!\n");
    exit(1);
  }
  printf("q=%s\n", q);
  printf("p=%s\n", p);
  free(p);
  printf("p=%s\n", p);
}
```

在栈上定义的局部字符数组  
如果未初始化，打印出来的  
字符串是：烫烫烫

每个字符的值是0xcc

在堆上申请的动态字符数组  
如果未初始化，打印出来的  
字符串是：屯屯屯

每个字符的值是0xcd

而堆上申请的动态字符数组  
无论是否赋值过，释放内存  
后再打印出来的字符串是：  
葺葺葺

每个字符的值是0xdd

在VS编译系统上程序的运行结果为:

20

q=烫烫烫烫烫烫烫烫烫烫烫烫烫烫?烫烫#?

p=屯屯屯屯屯屯屯屯屯屯 榆陞

p=葺葺葺葺葺葺葺葺葺葺葺葺葺葺葺 榆陞

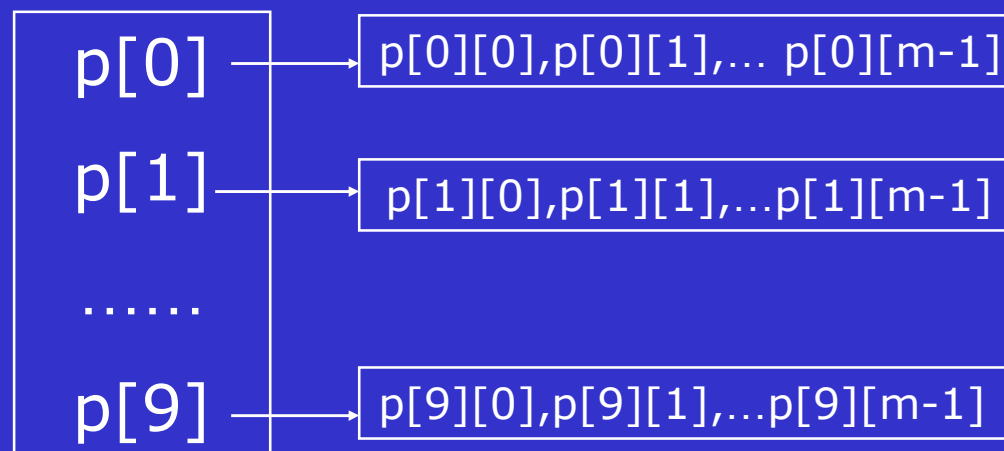
请按任意键继续...

- ✓ 可以利用指针数组，生成一个 $N \times m$ 的动态二维数组， $m$ 由键盘输入，但第一维 $N$ 是固定的一个常量，是指针数组的长度。可以存取二维数组元素  $p[i][j]$ ,  $0 \leq i < N, 0 \leq j < m$ 。例如有程序段：

```
double *p[10];  
int k, m;  
scanf("%d", &m);  
for (k=0; k<10; k++)  
    p[k]=(double *)malloc(sizeof(double)*m);
```

循环为指针数组 $p$ 中每一个指针 $p[k]$ 申请一块 $m$ 个 $\text{double}$ 单元的动态内存，可以把 $p$ 当做一个长度为 $10 \times n$ 的 $\text{double}$ 型二维数组来使用。 $p[i][j]$ ,  $0 \leq i < 10$ ,  $0 \leq j < m$

- ✓ 但实际上， $p$ 是由10块长度为 $m$ 个 $\text{double}$ 单元的动态内存块拼起来的二维数组。



```
#include <stdio.h>
#include <stdlib.h>
void main( )
{ double *p[10];
  int k, m, i, j;
  scanf("%d", &m);
  for (k=0; k<10; k++)
  { p[k]=(double *)malloc(sizeof(double)*m);
    /* 指针数组中的每个指针指向一块m个double型单元的内存块 */
    if (p[k] == NULL)
    { printf("Can't get memory!\n");
      exit(1);
    }
  }
  for (i=0; i<10; i++)
    for (j=0; j<m; j++)
      p[i][j] = i*m+j;
```

```

for (i=0; i<10; i++)
{ for (j=0; j<m; j++)
    printf("%3.0f ",p[i][j]);
  printf("\n");
}
for (k=9; k>=0; k--) /*内存释放顺序最好跟申请的顺序相反, */
    free(p[k]);      /* 以防止内存堆上产生内存碎片 */
}

```

程序的运行结果为:

8

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79

- ✓ 还可以利用行指针，生成一个 $m \times N$ 的动态二维数组， $m$ 由键盘输入，但第二维 $N$ 是固定的一个常量，是行指针中的行长度。例如，下面的程序生成了一个 $m \times 10$ 的int动态二维数组， $m$ 由键盘输入。可以存取二维数组元素  $p[i][j]$   $0 \leq i < m$ ,  $0 \leq j < 10$  :

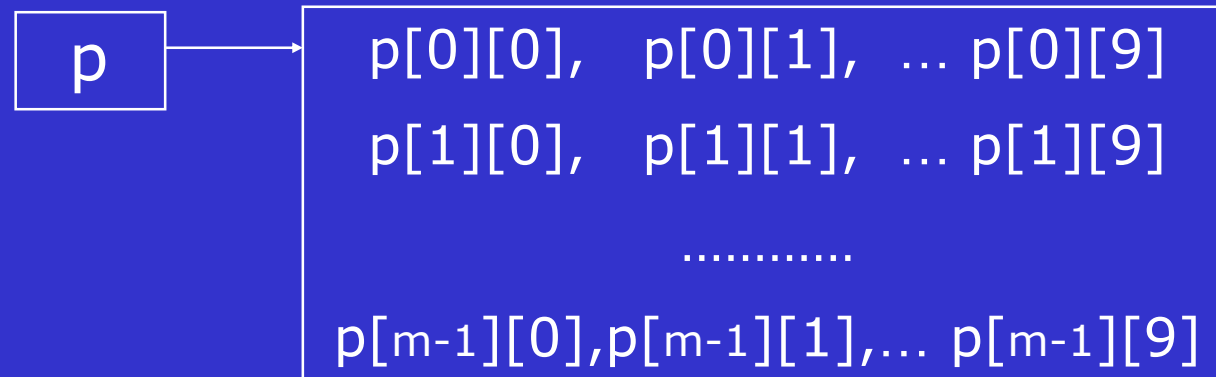
```
int (*p)[10];
```

```
int m;
```

```
scanf("%d", &m);
```

```
p = (int (*)[10])malloc(sizeof(int)*m*10);
```

为行指针 $p$ 申请了一块 $m \times 10$ 个int单元的动态内存，可以把 $p$ 当做一个长度为 $m \times 10$ 的int型二维数组来使用。



```

#include <stdio.h>
#include <stdlib.h>
void main( )
{ int (*p)[10];
  int m, i, j;
  scanf("%d", &m);
  p = (int (*)(10))malloc(sizeof(int)*m*10);
  /* 指针p指向了一块m*10个int型单元的内存 */
  if (p == NULL)
  { printf("Can't get memory!\n");
    exit(1);
  }
  for (i=0; i<m; i++)
    for (j=0; j<10; j++)
      p[i][j] = i*10+j;
  for (i=0; i<m; i++)
  { for (j=0; j<10; j++)
    printf("%3d ", p[i][j]);
    printf("\n");
  }
  free(p);
}

```

程序的运行结果为:

```

8
 0  1  2  3  4  5  6  7  8  9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79

```



- ✓ 同理可以产生动态三维数组:

```
int (*b)[10][10], m, i, j, k;
```

```
scanf("%d", &m);
```

```
b = (int (*)(10)[10])malloc(sizeof(int)*m*10*10);
```

- ✓ 用malloc申请一块 $m \times 10 \times 10$ 个int型单元的内存，构成一个 $m \times 10 \times 10$ 的int型动态三维数组，可以访问数组元素:

$b[i][j][k], \quad 0 \leq i < m, 0 \leq j < 10, 0 \leq k < 10$

- ✓ 也可以这样产生动态三维数组:

```
int *b[10][10], m, i, j, k;
```

```
scanf("%d", &m);
```

```
for (i=0; i<10; i++)
```

```
    for (j=0; j<10; j++)
```

```
        b[i][j]=(double *)malloc(sizeof(int)*m);
```

- ✓ 循环用malloc申请10\*10块m个int型单元的内存, 100个(每块有m个int单元)内存块拼成了一个 $10 \times 10 \times m$ 的int型动态三维数组, 可以访问数组元素:

$b[i][j][k], \quad 0 \leq i < 10, 0 \leq j < 10, 0 \leq k < m$

- ✓ 还可以这样产生动态三维数组:

```
int (*b[10])[10], m, i, j, k;
```

```
scanf("%d", &m);
```

```
for (i=0; i<10; i++)
```

```
    b[i]=(double *)malloc(sizeof(int)*m*10);
```

- ✓ 循环用malloc申请10块m\*10个int型单元的内存, 10个(每块有m\*10个int单元)内存块拼成了一个 $10 \times m \times 10$ 的int型动态三维数组, 可以访问数组元素:

```
b[i][j][k], 0≤i<10, 0≤j<m, 0≤k<10
```

- ✓ 利用指向指针的指针,可以生成 $n*m$ 的变长动态二维数组。例如, 下面的程序生成了一个 $n*m$ 的double型动态二维数组,  $n$ 和 $m$ 都由键盘输入。可以存取二维数组元素:

```
p[i][j] 0≤i<n , 0≤j<m  
double **p;  
int k, i, j, n, m;  
scanf("%d", &n);  
p=(double **)malloc(sizeof(double *)*n); /* 注意是double * */  
scanf("%d", &m);  
for (k=0; k<n; k++)  
    p[k]=(double *)malloc(sizeof(double)*m);
```

- ✓ 先申请一块 $n$ 个double指针单元的内存块, 让 $p$ 指向此内存块, 然后循环 $n$ 次为每一个指针 $p[k]$ 申请一块 $m$ 个double的内存块。 $n$ 个(每块有 $m$ 个double单元)内存块拼成了一个 $n*m$ 的double型动态二维数组:  $p[i][j]$   $0\leq i<n, 0\leq j<m$

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

```
#include <stdlib.h>
```

```
void main( )
```

```
{ double **p;
```

```
  int k, i, j, n, m;
```

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

```
  p=(double **)malloc(sizeof(double *)*n);
```

```
    /* 申请一块有n个double指针单元的内存块 */
```

```
  scanf("%d", &m);
```

```
  for (k=0; k<n; k++)
```

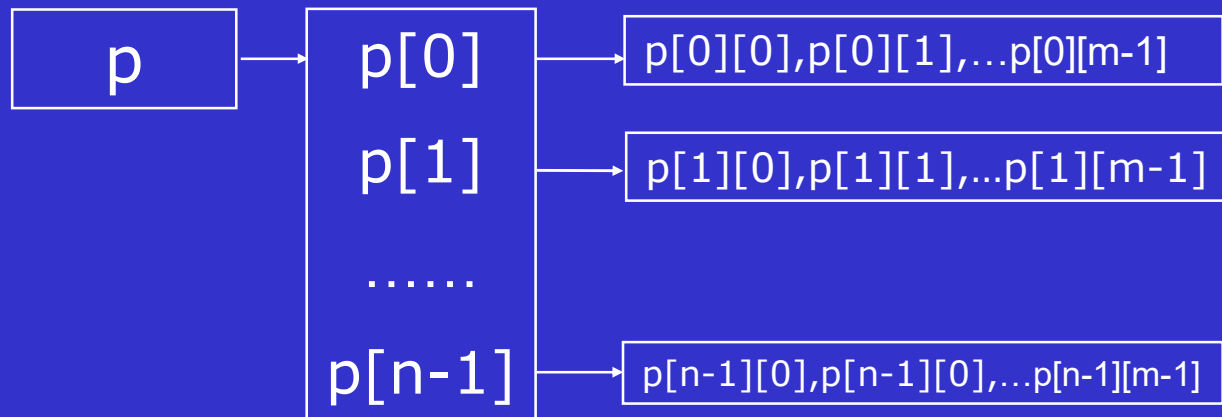
```
    p[k]=(double *)malloc(sizeof(double)*m);
```

```
    /* 循环为每个p[k]申请一块m个double单元的内存块 */
```

```
  for (i=0; i<n; i++)
```

```
    for (j=0; j<m; j++)
```

```
      p[i][j] = i*m+j;
```



```

for (i=0; i<n; i++)
{ for (j=0; j<m; j++)
    printf("%3.0f ",p[i][j]);
  printf("\n");
}
for (k=n-1; k>=0; k--) /* 释放每一个m个double的内存块,顺序最好与 */
    free(p[k]);        /* 申请的顺序相反,防止内存堆上产生内存碎片 */
free(p); /* 释放n个double指针单元的内存块 */
}

```

程序的运行结果为:

```

4 5
0  1  2  3  4
5  6  7  8  9
10 11 12 13 14
15 16 17 18 19

```

同理利用指向指针的指针的指针 `int ***p`, 可以生成 $n*m*k$ 的变长动态三维数组, 时间所限不再赘述。

## 10.4.2 calloc()函数

- ✓ 函数原型为: `void *calloc(size_t n, size_t size);`
- ✓ 其中`size_t`是一个与机器相关的`unsigned`类型, 其大小足以保证存储内存中对象的大小。32位编译器为4字节 (`unsigned long`), 64位编译器上为8字节 (`unsigned long long`); `size`为数据类型的长度。
- ✓ `calloc`的功能是在内存的动态存储区中分配`n`个长度为`size`的连续空间内存块, 函数返回一个指向分配起始地址的指针; 如果申请不成功, 返回`NULL`。
- ✓ 与`malloc`的区别在于, `calloc`在动态分配完内存后, 自动初始化该内存空间为0, 而`malloc`不初始化, 所分配的内存空间里数据是随机的垃圾数据。
- ✓ `calloc`可以比`malloc`申请更大的动态数组。

## 10.4.2 calloc( )函数

✓ 例如，有如下程序段：

```
#include <stdlib.h>
```

```
char *p;
```

```
double **q;
```

```
int (*a)[4];
```

```
p=(char *)calloc(20, sizeof(char));
```

```
q=(double **)calloc(10, sizeof(double *));
```

```
a=(int (*)[4])calloc(4*5, sizeof(int));
```

- ✓ 用calloc函数分别为p、q、a申请了一块相应大小的内存空间。
- ✓ calloc的使用与malloc大同小异，时间和篇幅所限不再举例。



### 10.4.3 realloc( )函数

✓ realloc的原型是：

```
void *realloc(void *mem_address, size_t newsize);
```

✓ 一般的调用格式为：

指针名p = (类型 \*)realloc(指针名p, 新的内存长度)

- ✓ 其中新的内存长度可大可小，如果新的内存长度大于原内存长度，则新分配部分不会被初始化；如果新的内存长度小于原内存长度，可能会导致数据丢失。
- ✓ realloc的功能是：在原来已经申请内存块的基础上，再重新申请一块更长（或更短的）内存块，便于内存块的动态增长或缩减。
- ✓ realloc执行时先判断p内存块后面是否有足够的连续空间，如果有，扩大p所指向地址的内存块，并且将p返回；

### 10.4.3 realloc( )函数

- ✓ 如果空间不够，先按照newsize指定的大小重新分配一块新的内存空间，将原有内存块上的数据从头到尾复制到新分配的内存块上，然后释放原来p所指内存块（注意：原来内存块是隐含自动释放，不需要用户使用free去释放），同时返回新分配的内存块的首地址。如果申请不到新的内存空间，则返回空指针NULL。

- ✓ 例如，有如下程序：

```
#include <stdio.h>
#include <stdlib.h>
main( )
{ int i;
  int *pn=(int *)malloc(5*sizeof(int));
  if (pn==NULL)
  { printf("malloc fail! \n");
    exit(-1);
  }
}
```

```
printf("malloc %p\n",pn);
for(i=0;i<5;i++)
    pn[i]=i;
pn=(int *)realloc(pn, 10*sizeof(int));
if (pn==NULL)
{ printf("realloc fail\n");
  exit(-1);
}
printf("realloc %p\n", pn);
for(i=5;i<10;i++)
    pn[i]=i;
for(i=0;i<10;i++)
    printf("%3d",pn[i]);
free(pn);
}
```

程序运行结果为:

malloc 004C1BE8

realloc 004C1BE8

0 1 2 3 4 5 6 7 8 9

**解释：**程序是在先为pn申请了5个int内存块的基础上，又用realloc申请了一块10个int的新内存块。从运行结果来看，realloc获得的动态内存块首地址与原来malloc内存块的首地址相同，很明显是在原来5个int内存块的基础上又增加了5个int，使得pn所指的内存块长度自动从5个int扩大到了10个int。

## 10.4.4 free()函数

- ✓ free()函数原型是： `void free(void *ptr);`
- ✓ 无论是用malloc、calloc还是realloc申请的动态内存块，都对应用free释放回内存堆中。free函数调用应该与malloc、calloc或realloc函数调用一一对应，以防止内存泄露(Memory Leak)。
- ✓ free释放内存块的顺序最好跟malloc、calloc或realloc申请的顺序相反，以防止内存堆上产生内存碎片（内存碎片化），影响随后的内存申请。
- ✓ 例如，有如下程序：

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h> /* 或 #include <malloc.h> */
main( )
{  char *str;
    /* allocate memory for string */
    str = (char *)malloc(10*sizeof(char));
    /* copy "Hello" to string */
    strcpy(str, "Hello");
    /* display string */
    printf("String is %s\n", str);
    /* free memory */
    free(str);
    str=NULL;
}
```

程序运行结果为:  
String is Hello

## 10.4.4 free()函数

使用free函数需要注意的几个问题:

- ✓ ① 即使某个函数执行完毕, 如果没有用free释放相应所申请的动态内存, 函数内局部指针变量所指的动态内存块也不会自动释放回动态内存堆中, 会造成所谓内存泄漏。这不同于局部数组变量。
- ✓ ② 对于free(p);语句, 如果p是NULL指针, 那么free对p无论操作多少次都不会出问题。但如果p不是NULL指针, 那么free对p连续操作两次就会导致程序运行产生致命错误。因此建议free(p);后立即将相应指针p置为NULL。
- ✓ ③ 对于每个free(p);语句, 其中p指针必须指向所申请动态内存块的首地址。如果进行过p++等运算改变了p的值, p不再指向动态内存块的首地址, 那么执行free(p);会导致程序运行产生致命错误。

## 10.5 字符串与指针

### 10.5.1 字符串指针

在C语言中，表示一个字符串有以下两种形式：

1) 用字符数组存放一个字符串

2) 用字符指针指向一个字符串

例如，

```
#include <stdio.h>
main( )
{ char s[ ]="How do you do!";
  printf("%s\n", s);
}
```

程序输出结果为：  
How do you do!

例如，

```
#include <stdio.h>
main( )
{ char *s="How do you do!";
  printf("%s\n", s);
}
```

程序输出结果为：  
How do you do!

字符数组和字符指针变量都能实现字符串的存储与运算。

### 联系与区别

- ① 字符数组由元素组成，每个元素中存放一个字符，而字符指针变量中存放的是地址，也能作函数参数。

● 对数组赋初值, 如

```
char s[]="How do you do!"; /* 数组长度是字符串长度加1*/
```

而对字符指针变量初始化, 如

```
char *s="How do you do!";
```

- ② 只能对字符数组中的各个元素赋值，而不能用赋值语句对整个字符数组赋值。

例如，以下程序段是错误的：

```
char s[20]; s="How do you do!";
```

● 而对字符指针变量赋的是字符串首地址。

例如，下列程序段是合法的：

```
char *s; s="How do you do!";
```



## 特别说明

- ③ 字符数组名虽然代表地址，但数组名是常量，其值不能改变。

例如,下列用法是错误的:

```
char s[]="How do you do!";  
s=s+4; /* s 不能被修改 */  
printf("%s\n", s);
```

会导致编译错误。

- 但字符指针变量的值可以改变。

例如，下列用法是合法的:

```
char *s="How do you do!";  
s=s+4;  
printf("%s\n", s);
```

输出结果是: do you do!

④ 可以用下标形式引用指针变量所指向的字符串中的字符。

例如，对于程序段：

```
char *s="How do you do!";
```

```
printf("%c ", s[4]);
```

```
printf("%c\n", *(s+7) );
```

其运行结果为：

d y

● 每个字符串会自动有一个首地址指针。

例如,下列用法是合法的：

```
printf("%s\n", "How do you do!" + 4);
```

输出结果为： do you do!

而 

```
printf("%c\n", "How do you do!"[4]);
```

输出结果为： d

可以用下标形式直接引用字符串中的字符。

⑤ 可以随意修改字符数组中元素的值，但指针所指的字符串编译后存放在程序的常量区内，是常量字符串，因此不能修改其中的字符。

例如：

```
char s[]="How do you do!";
```

```
s[0]='W';
```

是正确的,可以改变数组中元素的值。

但 

```
char *s="How do you do!";
```

```
*s='W';
```

是错误的。字符指针所指的字符串是存放在常量区的常量字符串，不能被修改。

⑥ 可以通过输入字符串的方式为字符数组输入字符元素；但不能通过输入函数让字符指针变量指向一个字符串，因为由键盘输入的字符串，系统是不分配存储空间的。

例如，在下列程序中，首先定义了两个字符数组 **str1** 与 **str2**；然后分别通过键盘以字符串形式分别为这两个数组赋以新值。最后以字符串形式输出这两个数组中存放的字符串：

```
#include <stdio.h>
main( )
{ char str1[20], str2[20];
  scanf("%s", str1);
  scanf("%s", str2);
  printf("str1=%s\n", str1);
  printf("str2=%s\n", str2);
}
```

程序运行结果为：（带有下划线的为键盘输入的数据）

asdfghjkl

1234567890

str1=asdfghjkl

str2=1234567890

- 如果将上述程序中定义的数组改成字符指针，即程序变为：

```
#include <stdio.h>
main( )
{ char *str1, *str2;
  scanf("%s", str1);
  scanf("%s", str2);
  printf("str1=%s\n", str1);
  printf("str2=%s\n", str2);
}
```

如果对这个程序进行编译，则会出现警告错误信息：

**warning C4700:** 使用了未初始化的局部变量 "str1"

提醒你这两个字符指针变量没有初始化（没有赋初值）就使用了。在这种情况下，由于指针变量中没有正确指向合理的内存区地址，则从键盘输入的字符串无存储空间可存放，会产生致命错误。

- 如果在上述程序中对定义的两个字符指针变量赋以初值，即程序变为：

```
#include <stdio.h>
main( )
{ char *str1="ccc1", *str2="ccc2";
  scanf("%s", str1);
  scanf("%s", str2);
  printf("str1=%s\n", str1);
  printf("str2=%s\n", str2);
}
```

程序运行的结果为：（带有下列划线的为键盘输入数据）

asdf



● 从运行出现致命错误可以看出，虽然两个字符指针变量中都指向了存放字符串的内存区，但由于字符指针所指的字符串是存放在常量区的常量字符串，常量字符串只能读而不能改写，因此scanf试图向常量区写操作产生致命错误。

● 如果再将上述程序中的由键盘输入字符串改成由赋值语句给字符指针赋新值（即新地址），程序变为：

```
#include <stdio.h>
main()
{ char *str1="cccc", *str2="cccc";
  str1="asdfghjkl";
  str2="1234567890";
  printf("str1=%s\n", str1);
  printf("str2=%s\n", str2);
}
```

程序运行结果为：

str1=asdfghjkl

str2=1234567890      没有任何问题！



- 在这种情况下，程序中虽然在定义字符指针变量的同时给这两个指针均赋了初值地址（即字符串"cccc"的首地址），并且其中字符串的长度为4（实际占5个字节，还有1个字符串结束符），但在执行赋值语句：

```
str1="asdfghjkl"; str2="1234567890";
```

时，编译系统分别为两个字符串常量"asdfghjkl"与"1234567890"分配了内存空间，存放在常量区，并分别将这两个字符串的首地址赋给字符指针变量str1与str2。此时在这两个指针变量中获得了新的地址值，分别指向新的字符串"asdfghjkl"与"1234567890"。

- 由此可知，字符指针只能存放字符串的首地址；并且，对于赋值语句中的字符串，编译系统会给予分配存储空间，而对于通过键盘输入的字符串，系统是不分配存储空间的。

- 字符数组之所以能通过输入字符串的方式为字符数组元素输入字符，是因为在定义字符数组时已经为数组分配了存储空间。

- ⑦ 可以用指针变量所指向的字符串表示程序中的任何字符串，如printf函数中的输出格式字符串。

例如：

```
int a=12345;  
double b=1234.5678;  
char *format;  
format = "a=%10d, b=%12.6f\n";  
printf(format, a, b);
```

等价于：

```
int a=12345;  
double b=1234.5678;  
printf("a=%10d, b=%12.6f\n", a, b);
```

- 用字符数组也能实现上述功能，例如，

```
int a=12345;  
double b=1234.5678;  
char format[ ]="a=%10d, b=%12.6f\n";  
printf(format, a, b);
```

或者：

```
int a=12345; double b=1234.5678;  
char format[100];  
sprintf(format, "a=%%%dd, b=%%%d.%%df\n", 10, 12, 6);  
printf(format, a, b); /* 先生成输出格式，变场宽输出整数和浮点数 */  
puts(format);
```

输出结果：

```
a= 12345, b= 1234.567800  
a=%10d, b=%12.6f
```

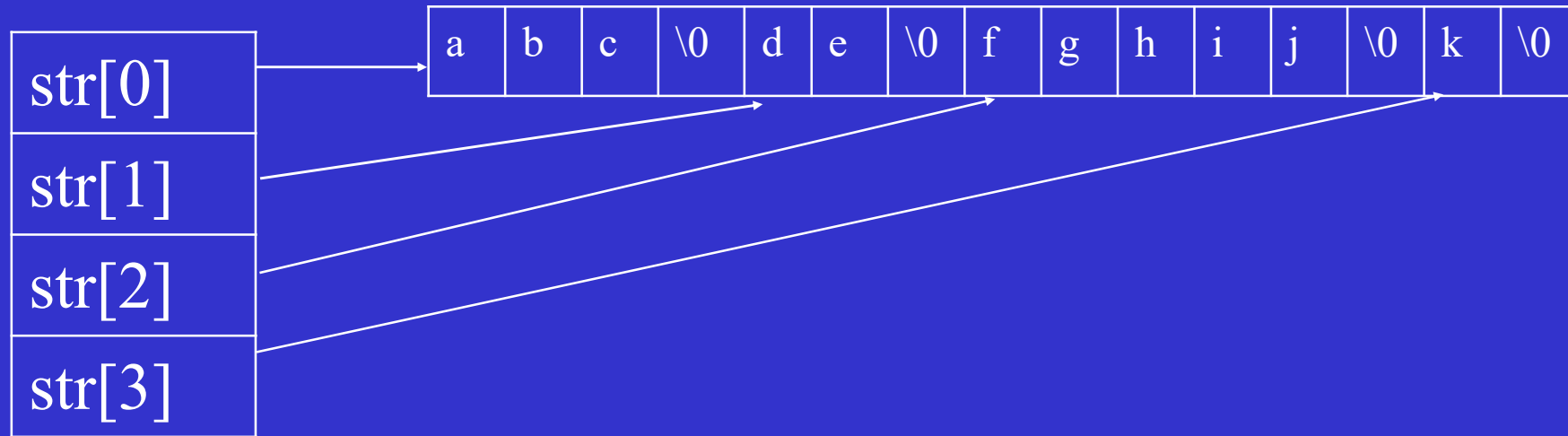
思考题：char \*p; printf(p, p="p=%s"); 输出结果是什么？

输出结果是：p=p=%s 等价于：printf("p=%s", "p=%s");

⑧ 字符型指针数组可构造紧凑型字符串数组，例如：

```
char *str[]={"abc", "de", "fghij", "k"};
```

/\* str是一个字符型指针数组,长度为4,通过字符串个数决定 \*/



str[1]=str[2];不出错！

但 strcpy(str[1], str[2]);出错！

因为str[0], str[1], str[2], str[3]指针可以交换,但不能相互复制。因为字符型指针所指的字符串都是常量，不能修改。

- 也可以用字符型二维数组来存放字符串数组，例如：

```
char str[][6]={"abc", "de", "fghij", "k"};
```

`str[0],str[1],str[2],str[3]`所指字符串可以互相复制交换，  
`strcpy(str[1],str[2]);` 正确。

但 `str[1]=str[2];` 出错，因为`str[0],str[1],str[2],str[3]`都是常量指针。

str字符数组：

a	b	c	\0		
d	e	\0			
f	g	h	i	j	\0
k	\0				

下面举例说明紧凑型字符串数组的应用。

**【例10-13】** 编写程序，根据年月日判断这一天是星期几。

```
#include <stdio.h>
main( )
{ char *Weeks[ ]={"Sunday", "Monday", "Tuesday",
                  "Wendesday", "Thursday", "Friday", "Saturday"};
  int year, month, day, weekday, i;
  int months[12]={0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30};
  scanf("%d%d%d", &year, &month, &day);
  if (year%4==0 && year%100!=0 || year%400==0)
    months[2]++;
  for (i=2; i<12; i++)
    months[i] += months[i-1];
  year--;
  weekday=year+year/4-year/100+year/400+months[month-1]+day;
  weekday %= 7;
  printf("%s\n", Weeks[weekday]);
}
```

2022 11 18

Friday

请按任意键继续...

⑨ 可以用 `sizeof` 操作符求字符串所占内存的大小。

例如：

```
printf("%d\n", sizeof("a"));
```

是求字符串 "a" 所占内存的大小，结果是： 2

而：

```
printf("%d\n", sizeof("abcd"));
```

结果是： 5

而：

```
printf("%d ", sizeof("ab\0cd"));
```

```
printf("%d\n", strlen("ab\0cd"));
```

结果是： 6 2

## 10.5.2 字符串指针作为函数参数

将一个字符串从一个函数传递到另一个函数，可以采用地址传递的方法，即将字符数组名或字符指针变量作为函数参数。采用以下四种形式：

1) 实参与形参都用字符数组名

2) 实参用字符数组名，形参用字符指针变量

3) 实参与形参都用字符指针变量

4) 实参用字符指针变量，形参用字符数组名

`char *s, int n`

`char s[ ], int n`

<pre>main( ) { char str[10];   ...   f(str,10);   ... }</pre>	<pre>void f(char s[ ], int n) { ...   ...   ... }</pre>
---	---

<pre>main( ) { char str[10];   char *p=str;   ...   f(p,10); }</pre>	<pre>void f(char *s, int n) { ...   ...   ... }</pre>
--	---



【例10-14】编写一个能实现字符串复制以及计算字符串长度功能的函数。

C程序如下（包括主函数）：

```
int str_copy(char *str1, char *str2)
{   int k=0;
    while(str1[k]!='\0')
    {   str2[k]=str1[k];
        k=k+1;
    }
    str2[k]='\0';
    return(k);
}
```

```
#include <stdio.h>
main( )
{ char str1[20], str2[20];
  int k;
  printf("input str1: ");
  scanf("%s", str1);
  printf("str1=%s\n", str1);
  k=str_copy(str1, str2);
  printf("str2=%s\n", str2);
  printf("k=%d\n", k);
}
```

程序运行结果为（带有下列划线的为键盘输入）

```
input str1: qwertyuiop
str1=qwertyuiop
str2=qwertyuiop
k=10
```

【例10-14】 str\_copy 可改写为:

```
int str_copy(char *str1, char *str2)
{   int k=0;
    while( str2[k] = str1[k] )
        k=k+1;
    return k;
}
```

【例10-14】 str\_copy 还可改写为:

```
int str_copy(char *str1, char *str2)
{   int k=0;
    while( *str2++ = *str1++ )
        k=k+1;
    return k;
}
```

【例10-14】 str\_copy 还可改写为:

```
int str_copy(char *str1, char *str2)
{   char *k=str1;
    while( *str2++ = *str1++ )
        ;
    return str1 - k - 1;
}
```

在上述程序中,函数**str\_copy()**中的两个形参均定义为字符指针变量。也可以定义为字符数组,即函数**str\_copy()**变为:

```
int str_copy(char str1[ ], char str2[ ])
{ int k=0;
  while(str1[k]!='\0')
  { str2[k]=str1[k];
    k=k+1;
  }
  str2[k]='\0';
  return k;
}
```

需要说明的是,在主函数中定义的两个字符数组的长度均为20,因此,被复制的字符串长度最大为19。但在函数**str\_copy()**中,是不限制字符串长度的,它是用字符串结束符来判断是否结束复制工作。

### 10.5.3 strstr( )函数

#### ● strstr(字符串1,字符串2)

功能

确认字符串2是否在字符串1中出现过，是则返回第一次出现的位置，否则返回NULL

典型的字符串查找搜索函数。

函数原型： `char *strstr(char *str1, char *str2)`

### 10.5.3 strstr ( )函数

```
程序: #include <stdio.h>
      #include <string.h>
      main()
      { char *s1="abcdeabcdeabcde";
        char *s2="dea", *s3;
        s3=strstr(s1, s2);
        printf("%s\n", s3);
      }
```

结果是: **deabcdeabcdeabcde**

找到第一个"dea"在字符串s1中出现的起始位置并返回。  
而如果继续执行:

```
        s3=strstr(s3+strlen(s2), s2);
        printf("%s\n", s3);
```

也就是利用s3+strlen(s2)跳过上次找到的"dea", 继续找到下一个"dea"在字符串中出现的起始位置并返回, 并从此起始位置开始打印字符串。

结果是: **deabcdeabcde**

## 10.6 函数与指针

### 10.6.1 用函数指针变量调用函数

● 一般来说，程序中的每一个函数经编译链接后，其目标代码在计算机内存中是连续存放的，该代码的首地址就是函数执行时的入口地址。在C语言中，函数名本身就代表该函数的入口地址。所谓指向函数的指针，就是指向函数的入口地址。

指向函数的指针变量其定义形式如下：

● 类型标识符 (\*指针变量名)(参数类型)；

其中最后的()即使无参数也不能省略。

● 定义了函数指针变量后，就可以通过它间接调用它所指向的函数。但同其他类型的指针一样，首先必须将一个函数名（代表该函数的入口地址，即函数的指针）赋给函数指针变量，然后才能通过函数指针间接调用该函数。

**【例10-15】** 从键盘输入一个大于1的正整数n,当n为偶数时,计算 $1+1/2+1/4+\dots+1/n$  当n为奇数时, 计算 $1+1/3+1/5+\dots+1/n$

当n为偶数时计算值的函数even() :

```
double even(int n)
{ int k;
  double sum=1.0;
  for (k=2; k<=n; k=k+2)
    sum=sum+1.0/k;
  return sum;
}
```

n为奇数时计算值的函数odd() :

```
double odd(int n)
{ int k;
  double sum=1.0;
  for (k=3; k<=n; k=k+2)
    sum=sum+1.0/k;
  return sum;
}
```

```
#include <stdio.h>
main( )
{ int n,k;
  double even(int), odd(int);
  printf("input n;");
  scanf("%d",&n);
  if (n>1)
  { if (n%2==0) /*n为偶数*/
      printf("even=%09.6f\n",even(n));
    else /*n为奇数*/
      printf("odd=%09.6f\n",odd(n));
  }
  else printf("ERR!\n");
}
```

```
#include <stdio.h>
main( )
{ int n,k;
  double even(int),odd(int);
  double (*p)(int);
  printf("input n;");
  scanf("%d",&n);
  if (n>1)
  { if (n%2==0) p=even;
    /*n为偶数,指针变量p指向函数even( )*/
    else p=odd;
    /*n为奇数,指针变量p指向函数odd( )*/
    printf("sum=%09.6f\n",(*p)(n));
  }
  else printf("ERR!\n");
}
```

用函数指针来解决



下面对指向函数的指针作几点说明:

- 在给函数指针变量赋值时, 只需给出函数名, 不必给出参数。

如上例中的 `p=even;` 与 `p=odd;`

- 可以通过指向函数的指针变量来调用函数, 其调用形式为:

`(*函数指针变量名)(实参表)`

如上例中的`(*p)(n)`。但在调用前必须给函数指针变量赋值。

- 也可以用下面的调用形式:

`函数指针变量名(实参表)`

如上例中的 `(*p)(n)` 可以改为 `p(n)`。

- 对函数指针变量运算是没有意义的。若`p`为函数指针变量, 则`p=p+1`是没有意义的。

## 10.6.2 指向函数的指针数组

- 除了定义函数指针，还可以定义函数指针数组，例如，  
`int (*p[10])(), (*proc[5][5])();`

用处: 方便函数调用，简化程序书写。

例如:

```
int (*proc[5][5])() = { { f11, f12, f13, f14, f15 },  
                        { f21, f22, f23, f24, f25 },  
                        { f31, f32, f33, f34, f35 },  
                        { f41, f42, f43, f44, f45 },  
                        { f51, f52, f53, f54, f55 } };  
  
for (i=0; i<5; i++)  
    for (j=0; j<5; j++)  
        proc[i][j](参数); /* (*proc[i][j])(参数); */
```

循环分别调用不同的函数f11(), f12(), f13(),.....

用指向函数的指针数组改写【例10-15】的主函数为：

```
#include <stdio.h>
main( )
{ int n,k;
  double even(int), odd(int);
  double (*p[2])(int)={even, odd};
  printf("input n;");
  scanf("%d", &n);
  if (n>1)
    printf("sum=%9.6f\n", p[n%2](n)); /* (*p[n%2])(n) */
  else
    printf("ERR!\n");
}
```

其中，函数指针数组元素p[0]指向even， p[1]指向odd

### 10.6.3 函数指针变量作为函数参数

● 当函数指针作为某函数的参数时，可以实现将函数指针所指向的函数入口地址传递给该函数。在这种情况下，当函数指针指向不同函数的入口地址时，在该函数中就可以调用不同的函数，且不需要对该函数体作任何修改。

【例10-16】 用迭代法求下列方程的实根。

1)  $x - 1 + \arctan(x) = 0$

2)  $x - 0.5\cos(x) = 0$

3)  $x^2 + x - 6 = 0$

C程序如下:

```
#include <math.h>

int root(double *x, int m, double eps, double (*f)(double) )
{ double x0;
  do
  { x0=*x;
    *x= f(x0); /* 也可以写作: *x= (*f)(x0); */
    m=m-1;
  } while((m!=0)&&(fabs(*x-x0)>=eps));
  if (m==0)
    return(0); /* 无解 */
  return(1); /* 有解 */
}
```

函数root能对于任意 $f(x)=0$ 求根。

```

#include <stdio.h>
main( )
{double x,(*p)(double),
  f1(double), f2(double),
  f3(double);
  x=1.0;  p=f1;
  if (root(&x,50,0.00001,p))
    printf("x1=%f\n",x);
  x=1.0;  p=f2;
  if (root(&x,50,0.00001,p))
    printf("x2=%f\n",x);
  x=1.0;
  if (root(&x,50,0.00001,f3))
    printf("x3=%f\n",x);
}

```

```

/* f(x)=1+arctan(x) */
double f1(double x)
{ return(1.0+atan(x)); }
/* f(x)=0.5cos(x) */
double f2(double x)
{ return(0.5*cos(x)); }
/* f(x)=(6+3x-x2)/4 */
double f3(double x)
{ return((6.0+3*x-x*x)/4); }

```

这个程序的运行结果为:

x1=2.132267 // 方程 $x=1+\arctan(x)$ 的根  
 x2=0.450183 // 方程 $x=0.5\cos(x)$ 的根  
 x3=2.000000 // 方程 $x=(6+3x-x^2)/4$ 的根

利用函数指针数组调用函数，【例10-16】主函数可改写为：

```
#include <stdio.h>
#include <math.h>
main( )
{ double x, f1(double), f2(double), f3(double);
  double (*p[3])(double)={f1, f2, f3};
  int k;
  for (k=0; k<3; k++)
  { x=1.0;
    if (root(&x, 50, 0.00001, p[k]))
      printf("x%d=%f\n", k+1, x);
  }
}
```

这个程序的运行结果为：

x1=2.132267

x2=0.450183

x3=2.000000

## 10.6.4 返回指针值的函数

- 在C语言中，一个函数不仅可以返回整型、字符型、实型等数据，也可以返回指针类型的数据，即C语言中还允许定义返回指针值的函数，其形式如下：

```
类型标识符 *函数名(形参表)
{ 函数体 }
```

```
例如, int *fun()
{ int *p;
  ...
  return p;
}
```

定义的函数fun()将返回一个指向整型量的指针值。

实际上，在C语言中，可以定义返回任何类型指针的函数。



利用函数指针重写改写9.1.3小节最后一个例子的程序如下：

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

```
int *f(int *p)
```

```
{    p += 3;
```

```
    return p;
```

```
}
```

```
main( )
```

```
{    int a[]={10,20,30,40,50,60}, *q=a;
```

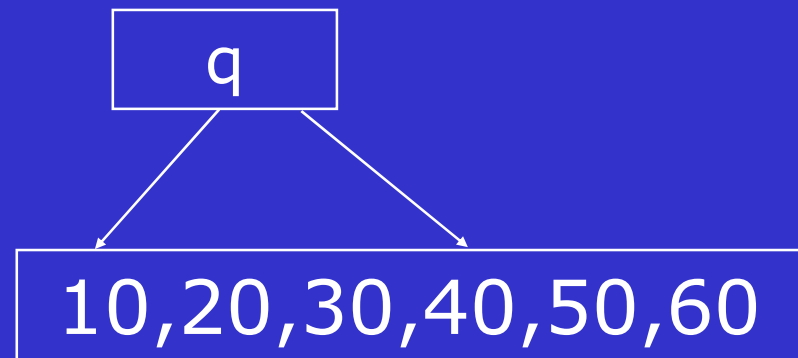
```
    q = f(q);
```

```
    printf("%d\n", *q);
```

```
}
```

输出结果： 40

调用f后，q的值改变了。指针函数通过函数名返回了修改后的指针值。



补充:

```
#include <stdio.h>
int f1(int m)
{ printf("f1!\n");
  return 2*m;
}
int (*f(int n))(int)
{ int (*q)(int)=f1;
  printf("f! %d\n", n);
  return q;
}
main( )
{ int (*p)(int), k;
  p = f(1);
  k = p(3);
  printf("OK! %d\n", k);
}
```

函数f可改写为:

```
int (*f(int n))(int)
{ printf("f! %d\n", n);
  return f1;
}
```

直接返回一个函数名

运行结果: f! 1

f1!

OK! 6

指针函数f通过函数名返回

“指向函数名的指针”。

或者说: f函数返回一个指向函数的指针。

## 10.7 main函数的形参

C语言中的主函数是可以有参数的。带参数main()函数的一般形式如下：

【例10-17】 编写一个命令行程序，其命令符为file，用以输出命令行中除命令符外以空格分隔的所有字符串（一行输出一个字符串）。其C程序如下：

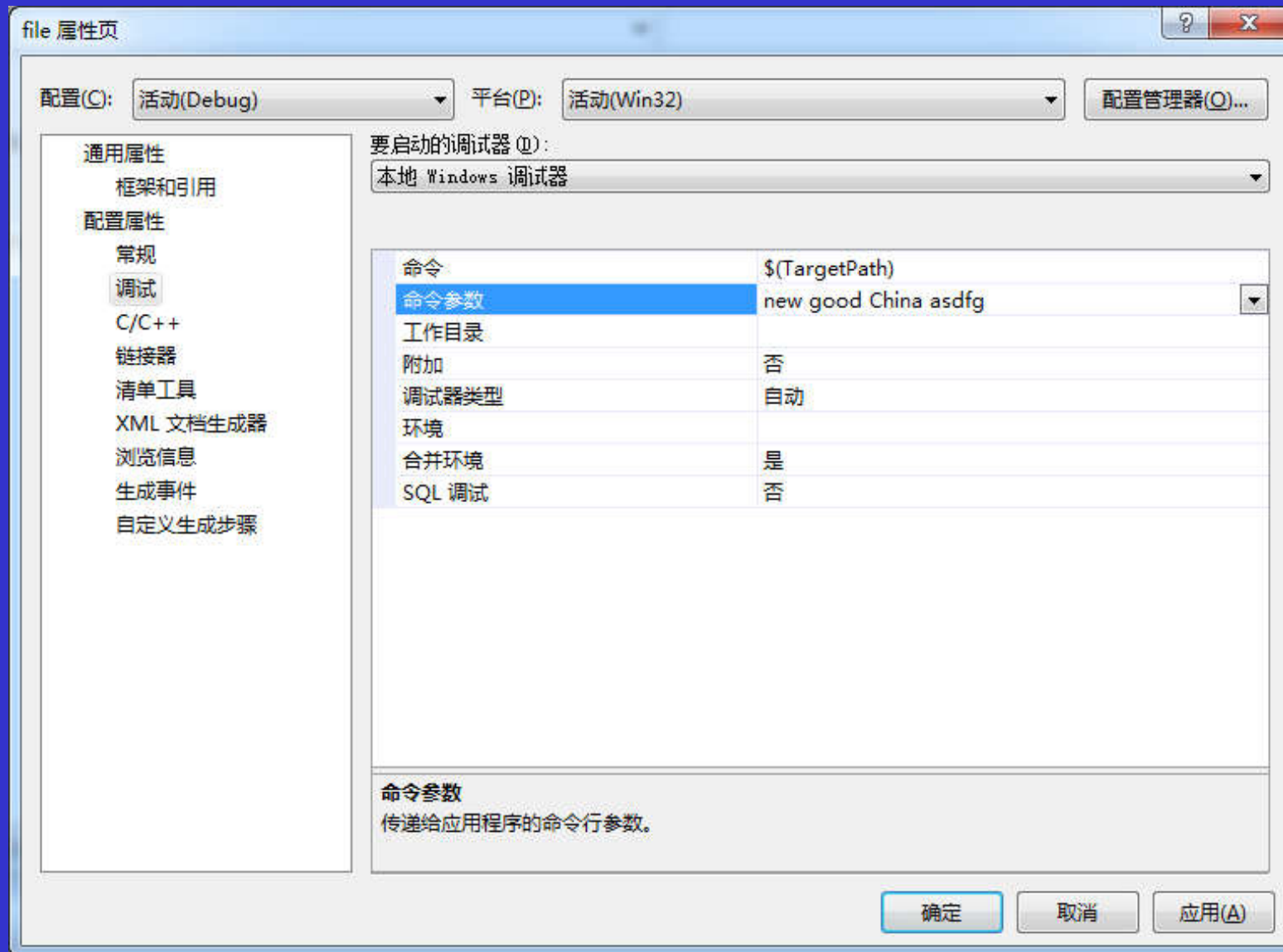
```
main(int argc, char *argv[ ])  
{ ... }
```

其中：argc的值是命令行参数个数+1  
argv是字符型指针数组，指向每一个参数字符串

```
/* file.c */  
#include <stdio.h>  
main(int argc, char *argv[ ])  
{ int k;  
  for (k=1; k<=argc-1; k++)  
    printf("%s\n", argv[k]);  
}
```

在VS2008下设置命令行:

项目 → file属性 → 配置属性 → 调试 → 命令参数



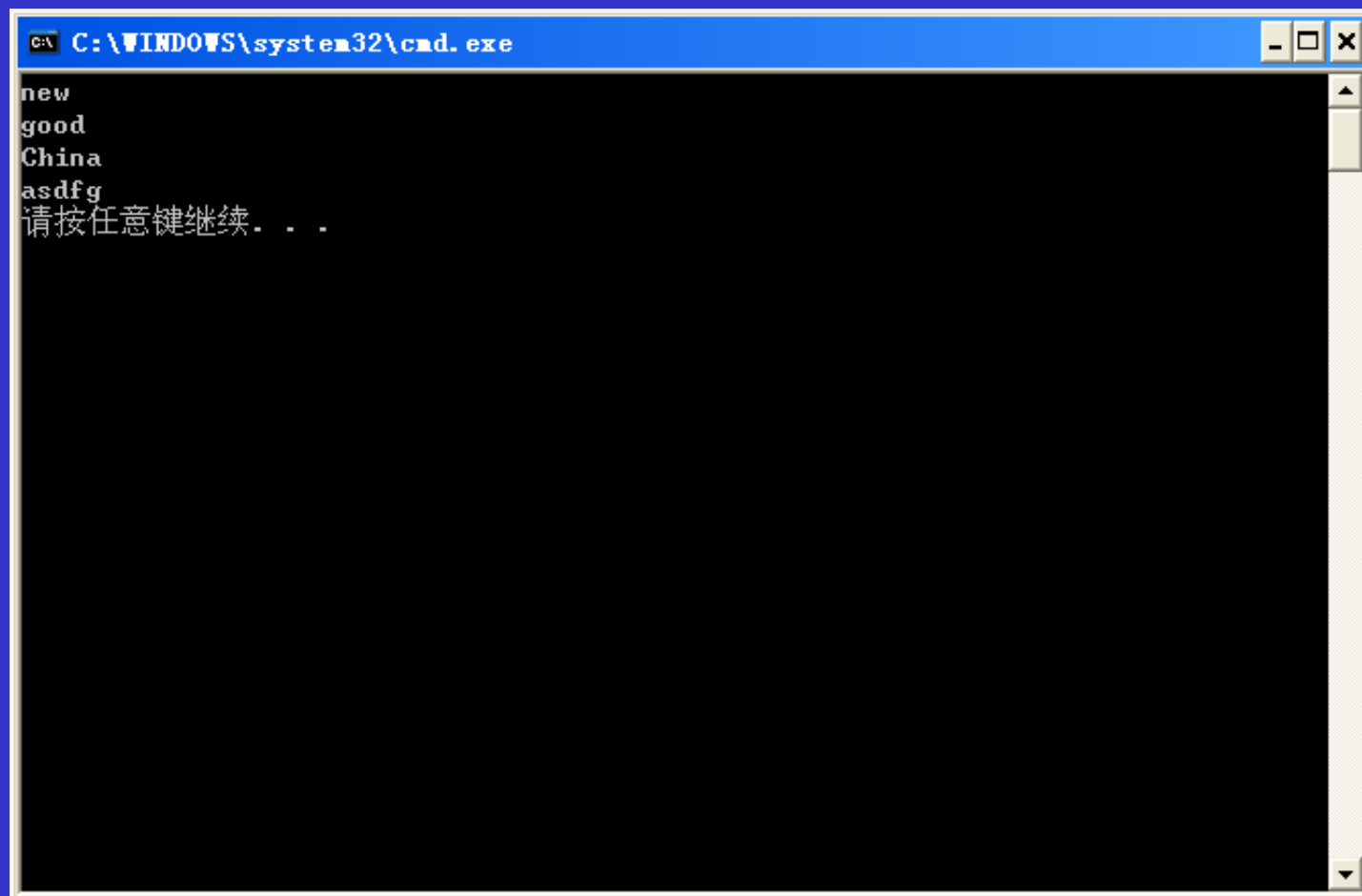
在命令参数处输入：new good China asdfg 并按下"确定"  
编译连接后，程序执行结果是：

new

good

China

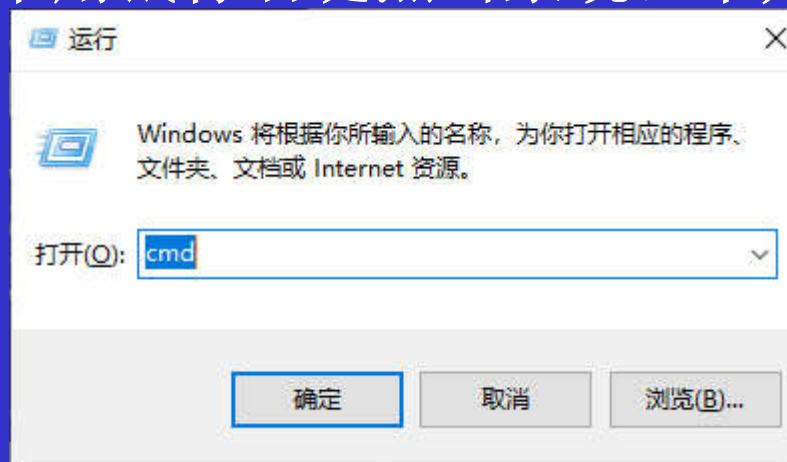
asdfg



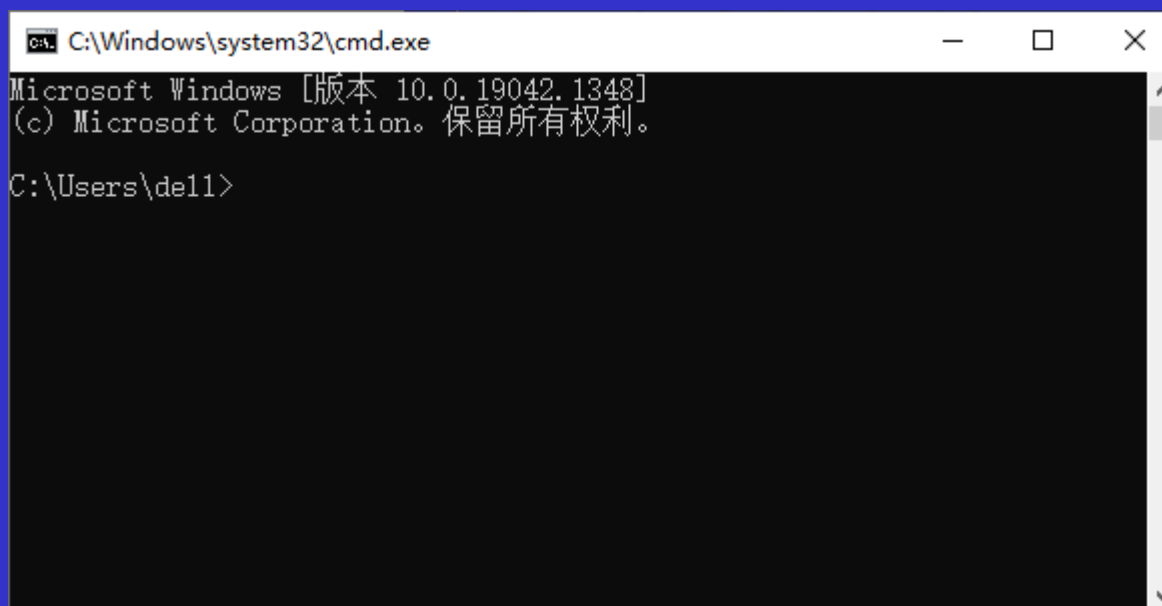
A screenshot of a Windows command prompt window. The title bar shows the path 'C:\WINDOWS\system32\cmd.exe'. The window has a black background with white text. The text displayed is: 'new', 'good', 'China', 'asdfg', and '请按任意键继续. . .'. The window includes standard Windows window controls (minimize, maximize, close) in the top right corner and a vertical scrollbar on the right side.

```
C:\WINDOWS\system32\cmd.exe
new
good
China
asdfg
请按任意键继续. . .
```

●也可以另打开一个命令提示符窗，可以在Win7的“附件”中选择“命令提示符”或Win10中用鼠标右键点击系统左下角图案，选择“运行”会弹出对话框：



在打开中输入cmd并确定，将弹出命令提示符窗：



用cd命令转到file.exe所在的相应的目录下，输入：

C:\file\Debug> file new good China asdfg<回车>

可以得到与上面同样的运行结果：



```
C:\Windows\system32\cmd.exe
Microsoft Windows [版本 10.0.19042.1348]
(c) Microsoft Corporation。保留所有权利。

C:\Users\de11>g:
G:\>cd file
G:\file>cd debug
G:\file\Debug>file new good China asdfg
new
good
China
asdfg
G:\file\Debug>_
```

上面的行被称为命令行，file为可执行命令（可执行文件），后面紧跟的是命令行参数，这也是main命令行参数名称的由来。

●用命令行参数的主要目的是，当一个程序运行时，把需要输入的参数或要读入的数据，通过命令行一次性传入，程序运行过程中不再需要等待输入，这样就可以提高运行效率节省人力。

- “命令提示符”下用命令行执行一个程序ex.exe方式是：

c:> ex 1.dat 100 0.000001

这里1.dat、100和0.000001就是命令行参数。

- C程序中主函数的参数，就是传递命令行参数进入程序中：

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

执行上面的命令 ex 1.dat 100 0.000001

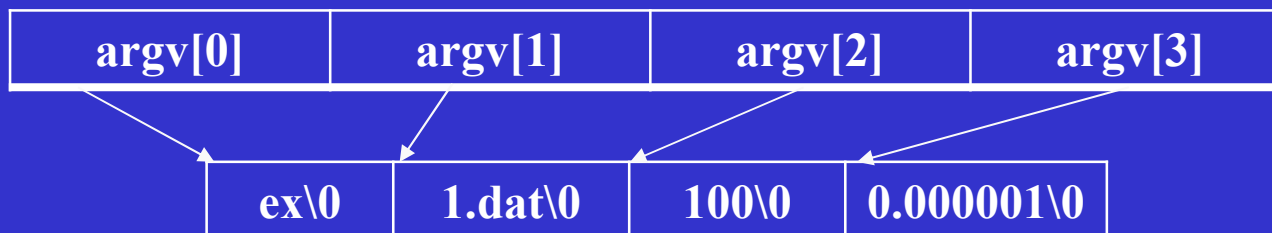
会使 argc的值为4（包括程序名本身），使得指针

argv[0]指向字符串"ex",

argv[1]指向字符串"1.dat",

argv[2]指向字符串"100",

argv[3]指向字符串"0.000001"。





- 如何把整数(例如迭代次数)、浮点数(例如精度)传递进程序呢?

- ✓ 如果要把argv[2]所指的整数字符串"100"当整数使用, 可以用atoi函数转换为整数, 即:

```
int n;
```

```
n = atoi(argv[2]);
```

此时n的值为100。

- ✓ 如果要把argv[3]所指的浮点数数字字符串"0.000001"当浮点数使用, 可以用atof函数转换为浮点数, 即:

```
double eps;
```

```
eps = atof(argv[3]);
```

此时eps的值为0.000001。

- 注意: 要使用atoi和atof函数, 程序前面应该加文件引用:

```
#include <stdlib.h>
```

● 批处理(Batch processing)文件为\*.bat，即文件名后缀为bat的文件，Windows系统可以直接执行。例如：

go.bat内容: (文本文件，可以用任何文本编辑器编辑)

```
ex 1.dat 100 0.000001
```

```
ex 2.dat 100 0.000001
```

```
ex 3.dat 100 0.000001
```

```
ex 4.dat 100 0.000001
```

```
ex 5.dat 100 0.000001
```

---

```
c:> go ↵
```

go命令发出后，将逐个执行go.bat中的ex命令行，每个命令行完成特定的任务。这是科学计算进行大规模数据处理常用的方式。

## 10.8 程序举例

设定积分为  $s = \int_a^b f(x) dx$  变步长梯形求积法的基本步骤如下:

● 利用梯形公式计算积分。

即取  $n=1, h=b-a$  则有:  $T_n = \frac{h}{2} \sum_{k=0}^{n-1} [f(x_k) + f(x_{k+1})]$  其中  $x_k = a + k \cdot h$

● 将求积区间再二等分一次 (即由原来的  $n$  等分变成  $2n$  等分), 在每一个小区间内仍利用梯形公式计算。即有

$$\begin{aligned} T_{2n} &= \frac{h}{2} \sum_{k=0}^{n-1} \left( \frac{f(x_k) + f(x_{k+0.5})}{2} + \frac{f(x_{k+0.5}) + f(x_{k+1})}{2} \right) \\ &= \frac{h}{4} \sum_{k=0}^{n-1} [f(x_k) + f(x_{k+1})] + \frac{h}{2} \sum_{k=0}^{n-1} f(x_{k+0.5}) \\ &= \frac{1}{2} T_n + \frac{h}{2} \sum_{k=0}^{n-1} f(x_{k+0.5}) \end{aligned}$$

- 判断二等分前后两次的积分值之差的绝对值是否小于所规定的误差。若条件  $|T_{2n}-T_n|<\epsilon$  成立,则二等分后的积分值 $T_{2n}$ 即为结果; 否则作如下处理;  $h=h/2, n=2*n, T_n = T_{2n}$  然后重复2)。

由上所述,可以写出其C函数如下:

```
#include <math.h>
double fpts(double a, double b,
double eps, double (*f)(double) )
{ int n,k;
double fa,fb,h,t1,p,s,x,t;
fa=(*f)(a); fb=(*f)(b);
/* 也可以写为:
fa=f(a); fb=f(b); */
n=1; h=b-a;
t1=h*(fa+fb)/2.0;
p=eps+1.0;
```

```
while (p>=eps)
{ s=0.0;
for (k=0; k<=n-1; k++)
{ x=a+(k+0.5)*h;
s=s+(*f)(x);
}
t=(t1+h*s)/2.0; /*计算 $T_{2n}$ */
p=fabs(t1-t); /*计算精度*/
t1=t; n=n*2; h=h/2.0;
}
return t;
}
```

### 【例10-18】

调用函数ffts(a, b, eps, f)  
计算下列三个定积分值;  
其中精度要求 $\varepsilon=0.00001$ 。

```
#include <math.h>
double f1(double x)
{ return(exp(-x*x)); }
double f2(double x)
{ return(1.0/(1+25*x*x)); }
double f3(double x)
{ return(log(1+x)/(1+x*x)); }
```

```
#include <stdio.h>
main( )
{ double f1(double), f2(double), f3(double), (*p)(double);
  p=f1;  printf("s1=%e\n",ffts(0.0, 1.0, 0.00001, p));
  p=f2;  printf("s2=%e\n",ffts(-1.0, 1.0, 0.00001, p));
  p=f3;  printf("s3=%e\n",ffts(0.0, 1.0, 0.00001, p));
}
```

用函数指针数组重写【例10-18】的主程序：

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

```
main( )
```

```
{ double f1(double),f2(double),f3(double);
```

```
double (*p[3])(double)={f1,f2,f3};
```

```
printf("s1=%e\n",ffts(0.0, 1.0, 0.00001, p[0]));
```

```
printf("s2=%e\n",ffts(-1.0, 1.0, 0.00001, p[1]));
```

```
printf("s3=%e\n",ffts(0.0, 1.0, 0.00001, p[2]));
```

```
}
```

运行结果为:

s1=7.468232e-001

s2=5.493573e-001

s3=2.721969e-001

也可以直接用函数名作为函数参数进行调用。程序简化为:

```
#include <stdio.h>
main( )
{ double f1(double), f2(double), f3(double);
  printf("s1=%e\n", ffts(0.0, 1.0, 0.00001, f1));
  printf("s2=%e\n", ffts(-1.0, 1.0, 0.00001, f2));
  printf("s3=%e\n", ffts(0.0, 1.0, 0.00001, f3));
}
```

## 第9次作业（第10章）

p.272-274 习题 1, 2, 4, 11, 12, 13

探究题：试编写函数：

```
char *my_replace(char *s1, char *s2, char *s3)
```

实现如下功能：把字符串s1中所有出现的字符串s2都替换成字符串s3，并通过函数名返回替换后的新字符串，但不得破坏字符串s1。例如，当s1="aabcdabce", s2="abc", s3="ff", 则函数返回的新字符串应该是"affdffe"。编写主程序调用此函数，并以各种输入情况（s2比s3串长、短、相等）验证函数的正确性。

（提示：尽量使用C语言提供的字符串操作函数strcpy, strncpy, strstr, strlen, strcat, strncat来实现）