中国科学技术大学
University of Science and Technology of China

程序设计II

计算机学院 黄章进
zhuang@ustc.edu.cn

课程内容

- C程序设计实践
  - C语言回顾
  - 简单程序设计
    - ✓数制转换、日期和时间处理，等
  - 字符串处理
  - 高精度计算
  - 枚举
- C++语言
  - 从C到C++
  - 类与对象
  - 运算符重载与类型转换
  - 继承与虚函数
- 编码规范

教材

- 李文新, 郭炜, 余华山. 程序设计导引及在线实践,清华大学出版社, 2007
- C99参考书： K. N. King. C Programming: A Modern Approach, 2nd Edition, 2008
  - C语言程序设计:现代方法(第2版)

考核

- 平时成绩：40%
  - 到课
  - 实验：西区电3楼406室 18:00-21:00
- 期末考试：60%
  - 7月29日下午

- 课程主页：
  http://staff.ustc.edu.cn/~zhuang/clp.htm

中国科学技术大学
University of Science and Technology of China

程序设计II

第1讲 C语言回顾

计算机学院 黄章进

zhuang@ustc.edu.cn

内容

- C语言概述
- 输入输出
- 位运算
- 函数指针
- C99的一些新特性
- 命令行参数
- 库函数
- 动态内存分配

C程序

- 程序 = 数据结构 + 算法
- C程序 = 变量/常量 + 函数
  - C程序设计
- C标准
  - K&R C：Kernighan and Ritchie, *The C Programming Language* (1978)
  - C89/C90：ISO/IEC 9899:1990
  - C99：ISO/IEC 9899:1999
  - C11： ISO/IEC 9899:2011

变量与数据类型

- 变量是内存中的一块区域
  - 变量名
  - 变量类型
- 数据类型：内存、取值、操作
  - void
  - 布尔型
    - ✓ _Bool　无符号整型，取值0或1，C99
    - ✓ bool (true/false)　C99 <stdbool.h> 宏
  - 字符型 char
  - 整型
    - ✓ signed char, short, int, long, long long
    - ✓ unsinged char/short/int/long, unsigned long long
  - 浮点型 float, double, long double

数据类型

- 自定义类型
  - 指针
    - ✓NULL，void *
  - 数组
    - ✓字符数组
  - 枚举 enum
    - ✓整型
  - 结构体 struct
  - 共用体 union

函数与语句

- 函数(或程序)的基本单元是语句(statement)
- 函数
  - 自定义函数
  - 库函数
- 语句
  - 空语句 ; 和 {}
  - 声明语句
  - 表达式语句
  - 分支语句 if-else语句, switch-case语句
  - 循环语句 for语句、while语句、do-while语句
  - break, continue, goto语句
  - 语句块

表达式与运算符

- 将变量、常量等用运算符(operator)连接在一起，就构成了表达式(expression)
- 运算符：目数、优先级、结合性
  - sizeof, &, *
  - 算术运算符：++, --, *, /, %, +, -
  - 位运算符：<<, >>, ~, &, ^, |
  - 关系运算符：<, <=, >, >=, ==, !=
  - 逻辑运算符：!, &&, ||
  - 条件运算符：?:
  - 赋值运算符：=, 复合算术/位运算赋值
  - 逗号运算符

Precedence	Operator	Description	Associativity
1	++ -- () [] . –> (*type*){*list*}	Suffix/postfix increment and decrement Function call Array subscripting Structure and union member access Structure and union member access through pointer Compound literal(C99)	Left-to-right
2	++ -- + – ! ~ (*type*) * & sizeof _Alignof	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT Type cast Indirection (dereference) Address-of Size-of Alignment requirement(C11)	Right-to-left
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ –	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <= > >=	For relational operators < and ≤ respectively For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13[note 1]	?:	Ternary conditional[note 2]	Right-to-Left
14	= += –= *= /= %= <<= >>= &= ^= |=	Simple assignment Assignment by sum and difference Assignment by product, quotient, and remainder Assignment by bitwise left shift and right shift Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

输入输出语句

C语言通过库函数来支持输入输出

输入输出

#include <stdio.h>

scanf( ) 将输入读入变量

printf( ) 将变量内容输出

scanf( ) 函数

int scanf(const char * , ...);

输入输出

- 参数可变的函数
- 第一个参数是格式字符串，后面的参数是变量的地址，函数作用是按照第一个参数指定的格式，将数据读入后面的变量

scanf 返回值

输入输出

- >0 成功读入的数据项个数
- 0　没有项被赋值
- EOF 第一个尝试输入的字符是EOF(结束)（对某些题，返回值为EOF可以用来判断输入数据已全部读完）

Number of receiving arguments successfully assigned, or EOF if read failure occurs before the first receiving argument was assigned.

printf( ) 函数

输入输出

int printf(const char * , ...);

- 参数可变的函数
- 第一个参数是格式字符串，后面的参数是待输出的变量，函数作用是按照第一个参数指定的格式，将后面的变量在屏幕上输出
- 返回值:
  - 成功打印的字符数
  - 返回负值为出错

格式字符串里的格式控制符号

输入输出

- %d 读入或输出int变量
- %c 读入或输出char变量
- %f 读入或输出float变量
- %s 读入或输出char * 变量
- %lf 读入或输出double 变量
- %e 以科学计数法格式输出数值
- %x 以十六进制读入或输出 int 变量
- %I64d 读入或输出 __int64 变量(64位整数）
- %p 输出指针地址值
- %.5lf 输出浮点数，精确到小数点后5位

例子

输入输出

```
#include <stdio.h>
int main()
{
        int a;
        char b;
        char c[20];
        double d = 0;
        float e = 0;
        int n = scanf("%d%c%s%lf%f", &a, &b, c, &d, &e);
        printf("%d %c %s %lf %e %f %d", a, b, c, d, e, e, n);
        return 0;
}
```

结果

输入输出

```
int n = scanf("%d%c%s%lf%f", &a, &b, c, &d, &e);
printf("%d %c %s %lf %e %f %d", a, b, c, d, e, e, n);
```

- input:

123a teststring 8.9 9.2

- output:

123 a teststring 8.900000 9.200000e+000 9.200000 5

- input:

123ateststring 8.9 9.2

- output:

123 a teststring 8.900000 9.200000e+000 9.200000 5

- input:

123 a teststring 8.9 9.2

- output:

123 a 0.000000 0.000000e+000 0.000000 3

例子

输入输出

```
#include <stdio.h>
int main()
{
        int a, b;
        char c;
        char s[20];
        __int64 n = 9876543210001111;
        scanf("%d %c,%s%x%I64d", &a, &c, s, &b, &n);
        printf("%d %x %u %s %p %x %d %I64d", a, a, a, s, s, b, b, n);
        return 0;
}
```

- input:

-28 K,test ffee 1234567890123456

- output:

-28 fffffe4 4294967268 test 0012FF60 ffee 65518 1234567890123456

64位整数类型__int64

输入输出

- __int64是有符号 64 位整数类型，范围为 -2^63 (-9,223,372,036,854,775,808) 到 2^63-1 (9,223,372,036,854,775,807)，存储空间占 8 字节
  - 用于整数值可能超过int数据类型支持范围的情况
  - __int64关键字和%I64标号是Microsoft实现专有的
  - 要用64位int的话，标准C(C99)可用long long和%lld

常见错误

输入输出

```
#include <stdio.h>
int main()
{
        char * s;
        scanf( “%s”, s);
}
```

错在何处？

- 错在 s 不知道指向何处，往其指向的地方写入数据，不安全

读取一行

输入输出

char * gets(char * s);

- 从标准输入读取一行到字符串s
- 如果成功，返回值就是 s 地址
- 如果失败，返回值是 NULL
- 可以根据返回值是 NULL判定输入数据已经读完
- 调用时要确保 s 指向的缓冲区足够大，否则可能发生内存访问错误

例子

输入输出

```
#include <stdio.h>
int main()
{
        char s[200];
        char * p = gets(s);
        printf("%s:%s", s, p);
        return 0;
}
```

◆ input:

Welcome to Beijing !

◆ output:

Welcome to Beijing !:Welcome to Beijing !

sscanf 和 sprintf函数

输入输出

int sscanf(const char * buffer, const char * format[, address, ...]);

- 和scanf的区别在于，它是从buffer里读取数据

int sprintf(char *buffer, const char *format[, argument, ...]);

- 和printf的区别在于，它是往buffer里输出数据

例子

输入输出

```
#include <stdio.h>
int main()
{
        int a,b; char c;        char s[20];
        char szSrc[] = "-28 K,test ffee 1234567890123456";
        char szDest[200];
        __int64 n = 9876543210001111;
        sscanf(szSrc, "%d %c,%s%x%I64d", &a, &c, s, &b, &n);
        sprintf(szDest, "%d %x %u %s %p %x %d %I64d",
                        a, a, a, s, s, b, b, n);
        printf("%s", szDest);
        return 0;
}
```

- output:

-28 fffffe4 4294967268 test 0012FF60 ffee 65518 1234567890123456

位运算

- 有时我们需要对某个整数类型变量中的某一位（bit）进行操作
  - 比如，判断某一位是否为1，或只改变其中某一位，而保持其他位都不变。
- 六种位运算符来进行位运算操作：

&	按位与
|	按位或
^	按位异或
~	取反
<<	左移
>>	右移

按位与

位运算

- 按位与运算符“&”是双目运算符
  - 功能是将参与运算的两操作数各对应的二进制位进行与操作，只有对应的两个二进位均为1时，结果的对应二进制位才为1，否则为0。
- 按位与运算通常用来将某变量中的某些位清0或保留某些位不变。

按位与

位运算

- 如果需要将int型变量n的低8位全置成0，而其余位不变，则可以执行

  n = n & 0xffffff00;

- 也可以写成：

  n &= 0xffffff00;

- 如果n是short类型的，则只需执行：

  n &= 0xff00;

- 如何判断一个int型变量n的第7位（从右往左，从0开始数）是否是1？
  - 只需看表达式“n & 0x80”的值是否等于0x80即可

按位或

位运算

- 按位或运算符"|"是双目运算符
  - 功能是将参与运算的两操作数各对应的二进制位进行或操作，只有对应的两个二进位都为0时，结果的对应二进制位才是0，否则为1
- 例如：表达式“21 | 18 ”的值是23

21： 0000 0000 0000 0000 0000 0000 0001 0101

18： 0000 0000 0000 0000 0000 0000 0001 0010

21|18: 0000 0000 0000 0000 0000 0000 0001 0111

按位或

位运算

- 按位或运算通常用来将某变量中的某些位置1或保留某些位不变
- 如果要将int型变量n的低8位全置成1，而其余位不变，则可以执行：

```
n |= 0xff;
```

按位异或

位运算

- 按位异或运算符"^"是双目运算符
  - 功能是将参与运算的两操作数各对应的二进制位进行异或操作，即只有对应的两个二进位不相同时，结果的对应二进制位才是1，否则为0。
- 例如：表达式“21 ^ 18 ”的值是7(即二进制数111)。

21：　0000 0000 0000 0000 0000 0000 0001 0101

18：　0000 0000 0000 0000 0000 0000 0001 0010

21^18: 0000 0000 0000 0000 0000 0000 0000 0111

- 异或运算的特点：如果 a^b=c，那么就有c^b = a以及c^a=b
  - 此规律可以用来进行最简单的加密和解密。

按位非

位运算

- 按位非运算符"~"是单目运算符
  - 功能是将操作数中的二进制位0变成1，1变成0
- 例如，表达式“~21”的值是无符号整型数0xffffffea

21：　0000 0000 0000 0000 0000 0000 0001 0101

~21：　1111 1111 1111 1111 1111 1111 1110 1010

- 而下面的语句：

```
printf("%d,%u,%x",~21,~21,~21);
```

输出结果就是:

-22,4294967274,ffffffea

左移运算符

位运算

- 左移运算符“<<”是双目运算符
  - 计算结果是将左操作数的各二进位全部左移若干位后得到的值，右操作数指明了要左移的位数。
  - 左移时，高位丢弃，低位补0
  - 左移运算符不会改变左操作数的值。

左移运算符

位运算

- 例如，常数9有32位，其二进制表示是：

0000 0000 0000 0000 0000 0000 0000 1001

- 因此，表达式“9<<4”的值，就是将上面的二进制数左移4位，得：

0000 0000 0000 0000 0000 0000 1001 0000

即为十进制的144。

- 实际上，左移1位，就等于是乘以2，左移n位，就等于是乘以2^n。而左移操作比乘法操作快得多。

例子

位运算

```
#include <stdio.h>
int main() {
        int n1 = 15;
        short n2 = 15;
        unsigned short n3 = 15;
        unsigned char c = 15;
        n1 <<= 15;
        n2 <<= 15;
        n3 <<= 15;
        c <<= 6;
        printf( "n1=%x,n2=%d,n3=%d,c=%x,c<<4=%d",
                n1, n2, n3, c, c << 4);
}
```

上面程序的输出结果是:

n1=78000,n2=-32768,n3=32768,c=c0,c<<4=3072

例子

位运算

n1: 0000 0000 0000 0000 0000 0000 0000 1111
n2: 0000 0000 0000 1111
n3: 0000 0000 0000 1111
c: 0000 1111

n1 <<= 15: (变成78000)
  0000 0000 0000 0111 1000 0000 0000 0000
n2 <<= 15: ,(变成-32768)
  1000 0000 0000 0000
n3 <<= 15: (变成 32768)
  1000 0000 0000 0000
c <<= 6; (变成 c0)
  1100 0000
c << 4 这个表达式是先将 c 转换成整型
  0000 0000 0000 0000 0000 0000 1100 0000
然后再左移。 c<<4=3072

右移运算符

位运算

- 右移运算符“>>”是双目运算符
  - 计算结果是把“>>”的左操作数的各二进位全部右移若干位后得到的值，要移动的位数就是“>>”的右操作数。移出最右边的位就被丢弃。
- 对于有符号数，如long,int,short,char类型变量，在右移时，符号位（即最高位）将一起移动，并且大多数C编译器规定，如果原符号位为1，则右移时高位就补充1，原符号位为0，则右移时高位就补充0。

右移运算符

位运算

- 对于无符号数，如unsigned long, unsigned int, unsigned short, unsigned char类型的变量，则右移时，高位总是补0。
- 右移运算符不会改变左操作数的值
- 实际上，右移n位，就相当于左操作数除以2^n，并且将结果往小里取整。

-25 >> 4 = -2

-2 >> 4 = -1

18 >> 4 = 1

例子

位运算

```
#include <stdio.h>
int main()
{
        int n1 = 15;
        short n2 = -15;
        unsigned short n3 = 0xffe0;
        unsigned char c = 15;
        n1 = n1>>2;
        n2 >>= 3;
        n3 >>= 4;
        c >>= 3;
        printf( "n1=%x,n2=%d,n3=%x,c=%x", n1, n2, n3, c);
}
```

上面的程序输出结果是：

n1=3,n2=-2,n3=ffe,c=1

例子

n1: 0000 0000 0000 0000 0000 0000 0000 1111
n2: 1111 1111 1111 0001
n3: 1111 1111 1110 0000
c: 0000 1111

位运算

n1 >>= 2: 变成3
0000 0000 0000 0000 0000 0000 0000 0011
n2 >>= 3: 变成-2
1111 1111 1111 1110
n3 >>= 4: 变成 ffe
0000 1111 1111 1110
c >>= 3; 变成1
0000 0001

思考题

位运算

- 有两个int型的变量a和n(0 <= n <= 31)，要求写一个表达式，使该表达式的值和a的第n位相同

- 答案：

(a >> n) & 1

或：

(a & (1 << n)) >> n

函数指针

- 程序运行期间，每个函数都会占用一段连续的内存空间。而函数名就是该函数所占内存区域的起始地址(也称“入口地址”)。
- 可以将函数的入口地址赋给一个指针变量，使该指针变量指向该函数。然后通过指针变量就可以调用这个函数。这种指向函数的指针变量称为函数指针

定义

函数指针

◆函数指针定义的一般形式为：

类型名 (* 指针变量名)(参数类型1, 参数类型2,…);

- “类型名”表示被指函数的返回值的类型
- “(参数类型1, 参数类型2,……)”中则依次列出了被指函数的所有参数的类型

```
int (*pf)(int, char);
```

- 表示pf是一个函数指针，它所指向的函数，返回值类型应是int，该函数应有两个参数，第一个是int 类型，第二个是char类型

调用

函数指针

- 可以用一个原型匹配的函数的名字给一个函数指针赋值。
- 通过函数指针可调用它所指向的函数，写法为：

  函数指针名(实参表);
- 下面的程序说明了函数指针的用法

例子

函数指针

```
#include <stdio.h>
void PrintMin(int a, int b)
{
        if( a<b )
                printf("%d", a);
        else
                printf("%d", b);
}
int main(){
    void (* pf)(int, int);
    int x = 4, y = 5;
    pf = PrintMin;   pf(x, y);
    return 0;
}
```

上面的程序输出结果是：

4

快速排序库函数qsort

函数指针

```
void qsort(void *base, int nelem, unsigned int width,
    int (*pfCompare)( const void *, const void *));
```

- base是待排序数组的起始地址，
- nelem是待排序数组的元素个数，
- width是待排序数组的每个元素的大小（以字节为单位）
- pfCompare是一个函数指针，它指向一个“比较函数”。该比较函数应是返回值为int,有两个参数为const void * 的函数

比较函数

函数指针

- 排序就是一个不断比较并交换位置的过程。qsort如何在连元素的类型是什么都不知道的情况下，比较两个元素并判断哪个应该在前呢？
  - 答案是，qsort函数在执行期间，通过pfCompare指针调用“比较函数”，调用时将要比较的两个元素的地址传给“比较函数”,然后根据“比较函数”返回值判断两个元素哪个更应该排在前面
- 这个“比较函数”不是C/C++的库函数，而是由使用qsort的程序员编写的。在调用qsort时，将“比较函数”的名字作为实参传递给pfCompare

比较函数

函数指针

- qsort函数的用法规定，比较函数的原型是

int 函数名(const void * elem1, const void * elem2);

  - 该函数的两个参数，elem1和elem2，指向待比较的两个元素。也就是说，* elem1和 * elem2就是待比较的两个元素。
- 该函数必须具有以下行为：
  - 如果 * elem1应该排在 * elem2前面，则函数返回值是负整数（任何负整数都行）。
  - 如果 * elem1和* elem2哪个排在前面都行，那么函数返回0
  - 如果 * elem1应该排在 * elem2后面，则函数返回值是正整数（任何正整数都行）。

例子

下面的程序，功能是调用qsort库函数，将一个unsigned int数组按照个位数从小到大进行排序。比如 8，23，15三个数，按个位数从小到大排序，就应该是 23，15，8

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

int MyCompare( const void * elem1, const void * elem2 )
{
    unsigned int * p1, * p2;
    p1 = (unsigned int *) elem1;
    p2 = (unsigned int *) elem2;
    return  (* p1 % 10)  - (* p2 % 10 );
}
```

例子


```
#define NUM 5
int main()
{
    unsigned int an[NUM] = { 8,123,11,10,4 };
    qsort( an, NUM, sizeof(unsigned int), MyCompare);
    for( int i = 0; i < NUM; i ++ )
        printf("%d ", an[i]);
    return 0;
}
```

上面程序的输出结果是：

10 11 123 4 8

思考题: 如果要将an数组从大到小排序，那么MyCompare函数该如何编写？

C99的一些新特性

- for语句
- 数组初始化
- 变长数组
- main函数

C99中的for语句

- for语句的一般形式:

  for (*expr1* ; *expr2* ; *expr3*) *statement*

- 在C99中第一个表达式可以替换为一个声明，用来声明一个用于循环的变量

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

- 变量i不需要在for语句之前进行声明
  - 如果变量i在之前已经进行了声明，这个语句将创建一个新的i且该值仅用于循环内

C99中的for语句

- for语句声明的变量不可以在循环外访问（在循环外不可见）

```
for (int i = 0; i < n; i++) {
   …
   printf("%d", i);
   /* legal; i is visible inside loop
*/
   …
}
printf("%d", i);    /*** WRONG ***/
```

C99中的for语句

- for语句声明自己的循环控制变量使得程序可读性更强
- 如果在for循环退出之后还要使用该变量，只能使用以前的for语句格式
- for语句可以声明多个变量，只要它们的类型相同：

```
for (int i = 0, j = 0; i < n; i++)
    …
```

C99中的数组

- 数组初始化式最常见的格式是一个用大括号括起来的逗号分隔的常量表达式

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

- 如果初始化式比数组短，那么数组中剩余的元素初始化为0

```
int a[10] = {1, 2, 3, 4, 5, 6};
/* initial value of a is {1, 2, 3, 4, 5, 6, 0, 0, 0, 0} */
```

- 利用这一特性，可以很容易把数组初始化为全0

```
int a[10] = {0};
/* initial value of a is {0, 0, 0, 0, 0, 0, 0, 0, 0, 0} */
```

- 初始化式完全为空是非法的，所以要在大括号内放上一个0。初始化式比数组长也是非法的

C99中的数组

◆ 有时数组中只有较少的元素需要显式初始化，其他元素进行默认初始化

```
int a[15] =
  {0, 0, 29, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 48};
```

◆ 使用指定初始化式可以解决这个问题

```
int a[15] = {[2] = 29, [9] = 7, [14] = 48};
```

- 括号中的数字称为指示符
- 初始化的顺序不再重要

```
int a[15] = {[14] = 48, [9] = 7, [2] = 29};
```

◆ 对多维数组也有效，例如创建2x2的单位阵

```
double ident[2][2] = {[0][0] = 1.0, [1][1] = 1.0};
```

C99中的数组

- 指示符必须是整型常量表达式
  - 如果待初始化的数组长度是n，指示符的值必须在0和n-1之间
  - 如果数组的长度是省略的，指示符可以是任意非负整数
    - ✓编译器根据最大的指示符推断出数组的长度，下面数组有24个元素

```
int b[] = {[5] = 10, [23] = 13, [11] = 36, [15] = 29};
```

- 初始化式中可以同时使用旧方法（逐元素初始化）和新方法（指定初始化）

```
int c[10] = {5, 1, 9, [4] = 3, 7, 2, [8] = 6};
// int c[10] = {5, 1, 9, 0, 3, 7, 2, 0, 6, 0}
```

对数组使用sizeof

- sizeof运算符可以确定数组的大小（字节数）
  - 如果数组a有10个整数，那么sizeof(a)通常为40
- sizeof可以来计算数组元素(如a[0])大小
- 用数组大小除以数组元素的大小可以得到数组的长度

```
sizeof(a) / sizeof(a[0])
```

对数组使用sizeof

- 对数组a清零可以写成如下形式：

```
for (i = 0; i < sizeof(a) / sizeof(a[0]); i++)
    a[i] = 0;
```

  - 使用这种方法，即使数组长度日后改变，也不需要改变循环

- 有些编译器会对表达式i < sizeof(a) / sizeof(a[0])给出警告消息，这是因为i可能为int型，sizeof返回类型为无符号的size_t型

```
for (i = 0; i < (int) (sizeof(a) / sizeof(a[0])); i++)
    a[i] = 0;
```

C99中的变长数组

- 在C89中，数组变量的长度必须用常量表达式进行定义
- 在C99中，有时也可以使用非常量表达式
- 变长数组(variable-length array, VLA)的长度在程序运行时计算，而不是在编译时
  - 程序员不必在声明数组时给定一个长度

C99中的变长数组


```
#include <stdio.h>

int main(void)
{
  int i, n;

  printf("How many numbers do you want to reverse? ");
  scanf("%d", &n);

  int a[n];   /* C99 only - length of array depends on n */

  printf("Enter %d numbers: ", n);
  for (i = 0; i < n; i++)
    scanf("%d", &a[i]);

  printf("In reverse order:");
  for (i = n - 1; i >= 0; i--)
    printf(" %d", a[i]);
  printf("\n");

  return 0;
}
```

C99中的变长数组

- 变长数组的长度可以是任意整型表达式（可以包含运算符）

```
int a[3*i+5];
int b[j+k];
```

- 变长数组也可以是多维数组：

```
int c[m][n];
```

- 变长数组的限制：
  - 没有静态存储期
  - 没有初始化式
  - C99不允许goto语句绕过变长数组的声明
  - POJ不支持变长数组(VC2008/GCC4.4.0)

main函数

- 一般地，main函数的返回类型是int:

```
int main(void)
{
  …
}
```

- 旧式的C程序经常省略main的返回类型，假定默认为int:

```
main()
{
  …
}
```

- 在C99中省略main的返回类型是非法的
- 在main的形参列表中省略void仍然是合法的，但最好加上它

程序终止

- main函数返回的值是状态码，在某些操作系统中程序终止时可以检测到状态码
  - 返回0，程序正常终止
  - 返回非0，表示异常终止
- 尽量保证每个C程序都返回状态码

exit函数

- 在main函数中执行return语句是终止程序的一种方法
- 另一种方法是调用exit函数，该函数属于<stdlib.h>
- 传递给exit函数的实参与main函数的返回值具有相同的含义：两者都说明程序终止时的状态
- 为了表示正常终止，传递0：

```
exit(0);    /* normal termination */
```

exit函数

exit函数

- 既然0含义有些模糊，C语言允许使用EXIT_SUCCESS来代替（效果是相同的）：

```
exit(EXIT_SUCCESS);
```

- 传递EXIT_FAILURE表示异常终止：

```
exit(EXIT_FAILURE);
```

- EXIT_SUCCESS和EXIT_FAILURE是定义在<stdlib.h>中的宏
  - EXIT_SUCCESS和EXIT_FAILURE的值是由实现所定义，通常分别是0和1

exit函数

- main函数中的语句

  `return` *expression*`;`

  等价于

  `exit(`*expression*`);`

- return语句和exit函数之间的差异：
  - 不管哪个函数调用exit函数都会导致程序终止
  - return 语句仅当由main函数调用时才会导致程序终止

命令行参数

- 用户在DOS窗口输入可执行文件名的方式启动程序时，跟在可执行文件名后面的那些字符串，称为命令行参数
  - 命令行参数可以有多个，以空格分隔。
- 比如，在Dos窗口敲：

```
copy file1.txt file2.txt
```

  - “copy”, “file1.txt”, “file2.txt”就是命令行参数

- 如何在程序中获得命令行参数呢？

命令行参数

命令行参数

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

- 参数argc就代表启动程序时，命令行参数的个数
  - C/C++语言规定，可执行程序程序本身的文件名，也算一个命令行参数，因此，argc的值至少是1。
- argv是一个数组，其中的每个元素都是一个char*类型的指针，该指针指向一个字符串，这个字符串里就存放着命令行参数。
  - argv[0]指向的字符串就是第一个命令行参数，即可执行程序的文件名，argv[1]指向第二个命令行参数，argv[2]指向第三个命令行参数……

例子

命令行参数

```
#include <stdio.h>
int main(int argc, char * argv[])
{
    int i;
    for(i = 0; i < argc; i ++ )
        printf( "%s\n",argv[i]);

    return 0;
}
```

将上面的程序编译成sample.exe，然后在控制台窗口敲:

sample para1 para2 s.txt 5 4

◆ 输出结果就是:

sample

para1

para2

s.txt

5

4

C语言标准库函数

- 数学库函数 math.h
- 字符处理函数 ctype.h
- 字符串处理与内存操作函数 string.h
- 字符串转换函数 stdlib.h

数学库函数

标准库函数

数学库函数声明在math.h中，主要有：

abs(x)	求整型数x的绝对值
cos(x)	x(弧度)的余弦
fabs(x)	求浮点数x的绝对值
ceil(x)	求不小于x的最小整数
floor(x)	求不大于x的最小整数
log(x)	求x的自然对数
log10(x)	求x的对数(底为10)
pow(x,y)	求x的y次方
sin(x)	求x(弧度)的正弦
sqrt(x)	求x的平方根

字符处理函数

标准库函数

在ctype.h中声明，主要有：

int isdigit(int c)	判断c是否是数字字符
int isalpha(int c)	判断c是否是一个字母
int isalnum(int c)	判断c是否是一个数字或字母
int islower(int c)	判断c是否是一个小写字母
int isupper(int c)	判断c是否是一个大写字母
int toupper(int c)	如果c是一个小写字母，则返回其大写字母
int tolower (int c)	如果c是一个大写字母，则返回其小写字母

字符串和内存操作函数

标准库函数

字符串和内存操作函数声明在string.h中，常用的有：

char * strchr(char * s, int c)

如果s中包含字符c,则返回一个指向s第一次出现的该字符的指针，否则返回NULL

char * strstr(char * s1, char * s2)

如果s2是s1的一个子串，则返回一个指向s1中首次出现s2的位置的指针，否则返回NULL

char * strlwr(char * s)

将s中的字母都变成小写

char * strupr(char * s)

将s中的字母都变成大写

char * strcpy(char * s1, char * s2)

将字符串s2的内容拷贝到s1中去

char * strncpy(char * s1, char * s2,int n)

将字符串s2的内容拷贝到s1中去，但是最多拷贝n个字节。如果拷贝字节数达到n，那么就不会往s1中写入结尾的\0

字符串和内存操作函数

标准库函数

char * strcat(char * s1, char * s2)

将字符串s2添加到s1末尾

int strcmp(char * s1, char * s2)

比较两个字符串，大小写相关。如果返回值小于0，则说明s1按字典顺序在s2前面；返回值等于0，则说明两个字符串一样；返回值大于0，则说明s1按字典顺序在s2后面。

int stricmp(char * s1, char * s2)

比较两个字符串，大小写无关。其他和strcmp同。

int strlen(const char * string)

计算字符串的长度

void * memcpy(void * s1, void * s2, int n)

将内存地址s2处的n字节内容拷贝到内存地址s1

void * memset(void * s, int c, int n)

将内存地址s开始的n个字节全部置为c

字符串转换函数

标准库函数

有几个函数，可以完成将字符串转换为整数，或将整数转换成字符串等这类功能。

它们定义在 stdlib.h中:

int atoi(char *s)

将字符串s里的内容转换成一个整型数返回。比如，如果字符串s的内容是"1234",那么函数返回值就是1234

double atof(char *s)

将字符串s中的内容转换成浮点数。

字符串转换函数

标准库函数

char *itoa(int value, char *string, int radix);

将整型值value以radix进制表示法写入string。

比如：

char szValue[20];

itoa(32, szValue, 10);

则使得szValue的内容变为 “32”

itoa(32, szValue, 16);

则使得szValue的内容变为 “20”

动态内存分配

- C语言的数据结构，例如数组，通常是固定大小的
  - 因为在写程序时强制选择了大小，所以固定大小的数据结构可能会有问题
- C语言支持动态内存分配(dynamic storage allocation)：在程序执行期间分配内存的能力
  - 利用动态内存分配，可以设计根据需要扩大（和缩小）的数据结构

动态内存分配

- 动态内存分配主要用于字符串、数组和结构体
  - 动态分配的结构体可以链接形成链表、树或其他数据结构
- 为了动态分配内存，需要调用内存分配函数

内存分配函数

动态内存分配

- <stdlib.h> 头文件声明了三个内存分配函数：

  malloc—分配内存块，但不进行初始化

  calloc—分配内存块，并对内存块进行清零

  realloc—调整先前分配的内存块大小

- 这些函数返回void *（通用指针）类型的值

空指针

- 如果内存分配函数无法分配所需大小的内存块，函数会返回空指针
  - 空指针是能区别于所有有效指针的特殊值
- 在把函数返回值存储到指针变量后，需要检查该指针变量是否为空指针

空指针

- 测试malloc函数返回值的例子：

```
p = malloc(10000);
if (p == NULL) {
  /* allocation failed; take appropriate
action */
}
```

- NULL是一个宏，表示空指针
- 一些程序员把malloc函数的调用和NULL的测试组合在一起：

```
if ((p = malloc(10000)) == NULL) {
  /* allocation failed; take appropriate
action */
}
```

空指针

- 指针测试真假的方法和数的测试一样：所有非空指针都为真，而只有空指针为假
- 语句

```
if (p == NULL) …
```

可以写成

```
if (!p) …
```

- 而语句

```
if (p != NULL) …
```

则可以写成

```
if (p) …
```

动态分配字符串

动态内存分配

- 动态内存分配对字符串操作非常有用
- 字符串存储在字符数组中，而且可能很难预测这些数组需要的长度
- 通过动态分配字符串，可以推迟到程序运行时才作决定

用malloc为字符串分配内存

动态内存分配

- malloc函数具有如下原型：

```
void *malloc(size_t size);
```

- malloc函数分配size个字节的内存块，并返回指向该内存块的指针
- size_t是在C库中定义的无符号整数类型

用malloc为字符串分配内存

动态内存分配

- 为给n个字符的字符串分配内存空间，可以写成：

```
p = malloc(n + 1);
```

这里，p是一个char *类型变量

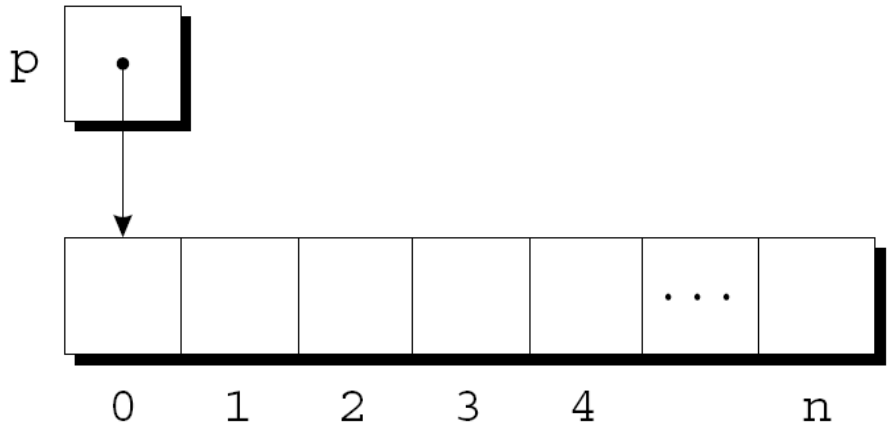
- 每个字符需要一个字节的内存，n+1给空字符留了空间
- 一些程序员喜欢对malloc函数的返回值进行强制类型转换，但这个转换不是必需的：

```
p = (char *) malloc(n + 1);
```

用malloc为字符串分配内存

动态内存分配

- 使用malloc函数分配的内存不会清零，所以p指向有n+1个字符的未初始化数组：



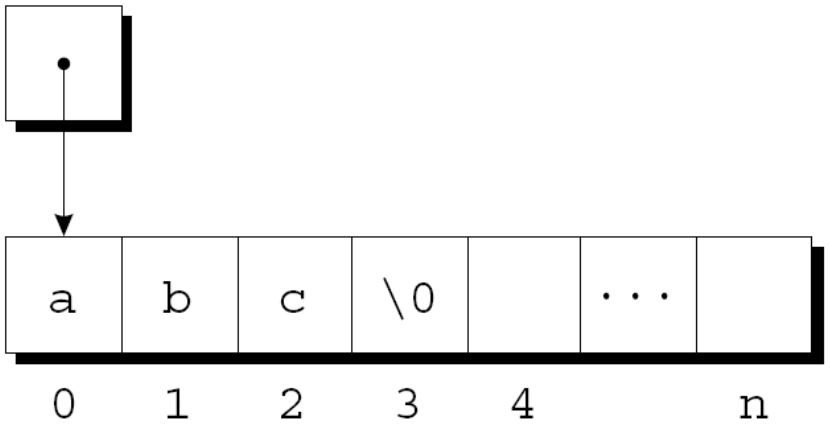
用malloc为字符串内存分配

动态内存分配

- 对该数组进行初始化的一种方法是调用strcpy函数：

```
strcpy(p, "abc");
```

- 数组中的前4个字符分别是a, b, c, 和\0:



在字符串函数中使用动态内存分配

动态内存分配

- 动态内存分配使编写返回指向“新”字符串的指针的函数成为可能
- 考虑编写函数把两个字符串连接起来而不改变其中任何一个字符串
  - 度量待连接的两个字符串的长度，然后调用malloc函数为结果分配适当大小的内存空间
  - 把第一个字符串复制到新的内存空间中
  - 调用strcat函数来拼接第二个字符串

在字符串函数中使用动态内存分配

动态内存分配

```
char *concat(const char *s1, const char
  *s2)
{
  char *result;

  result = malloc(strlen(s1) + strlen(s2) +
  1);
  if (result == NULL) {
    printf("Error: malloc failed in
  concat\n");
    exit(EXIT_FAILURE);
  }
  strcpy(result, s1);
  strcat(result, s2);
  return result;
}
```

在字符串函数中使用动态内存分配

动态内存分配

- concat函数可能的调用方式：

```
p = concat("abc", "def");
```

- 调用后，p将指向字符串“abcdef”，此字符串存储在动态分配数组中
- 像concat这样的动态分配内存的函数必须小心使用
  - 当不在需要concat函数返回的字符串是，需要调用free函数来释放它占用的空间
  - 如果不这样做，程序最终会耗光内存

动态分配数组

动态内存分配

- 动态分配数组有和动态分配字符串相同的好处
- 数组和指针间的紧密联系是的动态分配数组用起来像普通数组一样简单
- 尽管malloc函数可以为数组分配内存空间，但有时会用calloc函数代替，因为calloc函数会初始化分配的内存
- realloc函数允许根据需要对数组进行扩展或缩减

用malloc为数组分配内存

动态内存分配

- 假定程序需要一个由n个整数构成的数组，这里n可以在程序运行时计算出来
- 首先声明指针变量：

```
int *a;
```

- 一旦n的值已知了，程序调用malloc函数为数组分配存储空间：

```
a = malloc(n * sizeof(int));
```

  - 始终使用sizeof运算符来计算每个数组元素所需的空间大小

用malloc为数组分配内存

动态内存分配

- 可以忽略a是指针的事实，把它当作数组名来使用
- 例如，可以使用下列循环对a指向的数组进行初始化：

```
for (i = 0; i < n; i++)
  a[i] = 0;
```

- 当然用指针算术运算代替下标操作来访问数组元素也是可行的

calloc函数

动态内存分配

- calloc函数也可以用来为数组分配内存
- calloc函数的原型为：

```
void *calloc(size_t nmemb, size_t size);
```

- calloc函数的性质：
  - 为nmemb个元素的数组分配内存，每个元素有size个字节长
  - 如果请求的空间不足，返回空指针
  - 通过把所有位设置为0的方式进行初始化

calloc函数

- 下列calloc函数调用为n个整数的数组分配存储空间：

```
a = calloc(n, sizeof(int));
```

- 通过调用以1为第一个实参的calloc函数，可以为任何类型的数据项来分配空间：

```
struct point { int x, y; } *p;

p = calloc(1, sizeof(struct point));
```

realloc函数

- realloc函数可以调整动态分配数组的大小
- realloc函数的原型为：

```
void *realloc(void *ptr, size_t size);
```

  - ptr必须指向先前通过调用malloc、calloc或realloc函数获得的内存块
  - size表示内存块的新尺寸，新尺寸可能大于或小于原有尺寸

realloc函数

- realloc函数的性质：
  - 当扩展内存块时，realloc函数不会对添加进内存块的字节进行初始化
  - 如果recalloc函数不能按要求扩大内存块，那么返回空指针，且原有内存块中数据不变
  - 如果realloc函数被调用时，以空指针作为第一个实参，则行为类似于malloc函数
  - 如果realloc函数被调用时，以0作为第二个实参，那么它会释放掉内存块

realloc函数

动态内存分配

- 希望realloc函数非常高效：
  - 在要求减小内存块大小时，realloc函数应该在原先的内存块上就地进行缩减
  - 同理，扩大内存块时也不应该对其进行移动
- 如果无法就地扩大内存块，realloc函数会在别处分配新的内存块，然后把旧块中内容复制到新块中
- 一旦realloc函数返回，一定要对指向内存块的所有指针进行更新，因为内存块可能被移动了

释放内存

动态内存分配

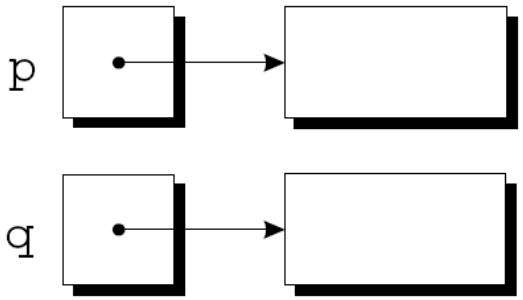
- malloc和其他内存分配函数所获得的内存块来自一个称为堆(heap)的存储池
- 过于频繁地调用这些函数（或让这些函数申请大内存块）可能会耗尽堆，导致这些函数返回空指针
- 更糟的是，程序可能分配了内存块，然后又丢失了对这些块的记录，从而浪费了空间

释放内存

- 例如：

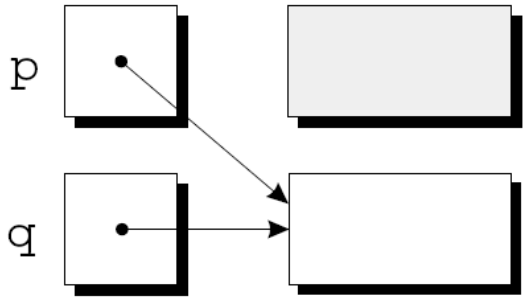
```
p = malloc(…);
q = malloc(…);
p = q;
```

- 在执行完前两条语句后，p和q分别指向一个内存块：



释放内存

- 在把q赋值给p后，两个指针都指向了第二个内存块：



- 没有指针指向第一个内存块，因此再也不能使用此内存块了

释放内存

动态内存分配

- 对程序而言，不可再访问到的内存块被称为垃圾(garbage)
- 留有垃圾的程序存在内存泄漏(memory leak)
- 一些语言提供垃圾收集器，用于垃圾的自动定位和回收，但C语言不提供
- 相反，每个C程序负责回收各自的垃圾，方法是调用free函数来释放不需要的内存

free函数

- free函数在<stdlib.h>中有如下原型：

```
void free(void *ptr);
```

- 把指向不再需要的内存块的指针传递给free函数：

```
p = malloc(…);
q = malloc(…);
free(p);
p = q;
```

- 调用free函数会释放p所指向的内存块

悬空指针问题

- 使用free函数会导致一个新问题：悬空指针(dangling pointer)
- free(p)释放p指向的内存块，但不会改变p自身
- 如果忘记了p不再指向有效内存块，可能导致问题：

```
char *p = malloc(4);
…
free(p);
…
strcpy(p, "abc");    /*** WRONG ***/
```

- 修改p指向的内存可能导致严重的错误

悬空指针问题

- 一种解决方案：free后，把指针置为空

```
free(p);
p = NULL;
```

- 悬空指针有时很难被发现，因为几个指针可能指向相同的内存块
  - 在释放内存块后，所有的指针都悬空了

编程题

- 3718：给定两个unsigned short类型整数，判断一个是否能由另一个经过循环左移若干位得到
- 3719：将输入的学生信息按姓名排序后输出
- 2677：肿瘤检测