# A Crash Course in C

## copyright 1995 by Anand Mehta

# Notes

---

## Table of Contents

---

# Introduction

Reasons to use C:

- pointers and structures
- encourages well structured control flow
- modular programming is encouraged
- easy to learn (C has few keywords)
- portability to different machines
- many functions are in libraries, not intrinsic parts of the language
- lexical scoping for variables

References:

- Brian Kernighan and Dennis Ritchie, The C Programming Language (2nd edition) (K\&R)
- Stephen G. Kochan, Programming in C, Programming in ANSI C (Kochan)
- Al Kelley and Ira Pohl, A Book on C
- American National Standard for Information Systems---Programming Language C, X3.159-1989.

Three steps are necessary to use C:

- write the program using an editor
- compile (and link) it; on Unix systems, use

  ```
  % gcc program.c

  % gcc -o .c
  ```

  This automatically runs these three separate processes (preprocessor, compiler, and linker), using the source files .c, to produce the output. If there are no errors, this produces an executable file. The first cc command produces the file a.out, while the second makes the file .
- execute the program

  % name
  % name input.file
  % name output.file
  % name input.file output.file

Different systems have different procedures to use C. See the manual or help files for details.

---

# Chapter 1: Fundamentals

## Example programs

The following are simple examples to get things started. They do not do anything useful, but they illustrate some key characteristics of C.

```
/* this is a comment */
main()                          /* function definition */
{                               /* start of block */
  printf("Hello world\n");      /* output; statement ends with a semicolon */
                                /* use '\n' in printf to get a linefeed */
}                               /* end of block */
```

Things to note in example programs:

- comments cannot be nested.
- main() is the function where the program begins execution, and can only occur once.
- C is case specific.
- all white space (spaces, tabs, blank lines, comments) is equivalent to one space.

```
main()
{
        variable declarations
        statements
}
```

```
main()
{
  int x;                        /* declaration and definition */
  x=5;                          /* assignment */
  printf("%d\n", x);            /* output x as an integer */
}
```

```
main()
{
  int x=17, y=12, z;                            /* declaration, definition, and initialization */
  z = x + y;                                    /* arithmetic expression */
  printf("z = %d + %d = %d\n", x, y, z);        /* each %d prints one integer */
  printf("%d * %d = %d\n", x, y, x*y);
}
```

```

main()
{
  int x;
  scanf("%d", &x);           /*  values input from the keyboard using scanf():  */
  printf("x = %d\n", x);     /*    and a pointer to x  */
}
```

A *common error* when using **`scanf()`** is not giving pointers as the arguments; to input a variable, use

```
int x;
scanf("%d", &x); /* correct */
```

NOT

```
int x;
scanf("%d", x); /* incorrect */
```

## Variables

All variables must be declared and defined before they can be used. Variable names can be composed of characters, digits, and the underscore character ( _), and can usually be up to 32 characters long. Variable names should not begin with an underscore---these names are used by the compiler and the libraries.

Variables have specific types and usage; basic data types:

- **char**: a single byte, used for one character
  can specify signed or unsigned, although signed is preferred for compatibility with int
  - **`int`**: integers
    can modify with short or long
    can specify signed or unsigned
    - **`float`**: single precision floating point (real) number
    - **`double`**: double precision floating point number
    - **`void`**: no type value, for pointers and functions

The specific size of each type depends on the implementation; see `<limits.h>` for details. On Unix systems, `<limits.h>` is usually the file `/usr/include/limits.h`.

The derived data types are

- arrays (`array[10]`)
- pointers (`*pointer`)
- functions (`function()`)
- structures (`structure.member`)
- enumerated (`enumerated_variable`)
- union (`union_name.member`)

All variables must be declared and defined; they can also be initialized and assigned a value.

- *declaration*: specifies the type of the identifier
- *definition*: reserves storage space for the object
- *initialization*: gives an initial value to the object during definition
- *assignment*: gets a new value

```
/***  definition of constants  ***/
main()
{
  char c = 'x';
  char c1 = '0';    /* the character 'zero', with the integer value for ASCII(0) */
  char c2 = '\0';   /* has the "integer" value zero */

  int n = 10;
  int n_oct = 065;  /* octal */
  int n_hex = 0x3d; /* hexadecimal */

  long m = 10L;
  unsigned int k = 304U;
  unsigned long l = 3040UL;

  float x1 = 143.0;
  float x2 = 24.6e-3;
  double y = 34.1L;

}
```

## Input / Output

Basic input/output is obtained using the standard library. The syntax is given below.

```
printf("control string",  variable1,  variable2, ...);
scanf("control string", &pointer1, &pointer2, ...);
```

The control string has place holders for the variables being used (see tables on page 16 - click here) there can also be characters in the control string. For scanf(), the pointers to the variables being inputed must be used. Special characters are listed on table 5 on page 16 (click here).

```
 /**  using printf() and scanf()  **/
main()
{
  int x=10;
  float y;

  printf("(1)  %d\n", x);
  printf("(2)  %d\n", x*5);

  printf("(3)  x = ");
  printf("%d", x);
  printf("\n");                                    /**  same as (4)  **/

  printf("(4)  x = %d\n", x);                      /**  same as (3)  **/

  printf("input x: ");  scanf("%d", &x);       /**  prompts for input  **/
  printf("(5)  x = %d\n", x);

/**  can input several values of different types with one scanf command  **/
  printf("input x, y: ");  scanf("%d %f", &x, &y);
  printf("(6)  x = %d, y = %f\n", x, y);
}
```

Output:

```
(1)  10
(2)  50
(3)  x = 10
(4)  x = 10
input x: 40
(5)  x = 40
input x, y: 20 34.5
(6)  x = 20, y = 34.500000
```

## Keywords and Operators: The C Language

C has a small set of keywords; all are used to either declare data types or control the flow of the program. See table 1 below for a list of the keywords. The C operators are listed in Table 2 at the end of the chapter.

Table 1: Keywords (K & R, p. 192)

| **Data Type Declarations** | | | | | **Control Flow Statements** | | |
|---|---|---|---|---|---|---|---|
| auto | float | long | static | unsigned | break | do | if |
| char | enum | register | struct | void | case | else | return |
| const | extern | short | typedef | volatile | continue | for | switch |
| double | int | signed | union | | default | goto | while |

### Basic Operators

Some basic operators are

- arithmetic operators
  - *, / , %, +, -
  - % is called modulus division (remainder)
- logical operators
  - <,>, <=, >=, ==, !=, &&, ||
  - the logical operators return a 1 (true) or 0 (false)
  - for any conditional test, any non-zero value is true and a 0 is false
  - == is the equality comparision ( not =)
  - != not equal to
  - &&, || logical AND and OR operators
  - A *common error* is using the assignment operator = when the logical operator == is required, e.g. (x = 2) instead of (x == 2); both are valid expressions, so the compiler will not indicate an error.
- assignment operators
  - =, +=, -=, *=, /=, %= !
  - op= assignment, `E1 op=E2 <==> E1=E1 op (E2)`, with E1 evaluated once
  - for example, `x+=2 ==>x=x+2`
  - increment/decrement by 1, as a prefix or postfix
  - ++, --
  - prefix: increments the variable and gives the new value as the result
  - postfix: gives the old value as the result and then increments the variable
- negation
  - !
  - !(0) ==> 1
  - !(any non-zero value) ==> 0
- conditional, compact if-else as an expression instead of a statement
  - ?
- `(type)`
  - casts object into a different type
- , (comma)
  - combines separate expressions into one
  - evaluates them from left to right
  - the value of the whole expression is the value of the right most sub-expression

Examples of Conditional Operators

```
main()
{
  int x=0, y=10, w=20, z, T=1, F=0;

  z = (x == 0);            /***  logical operator; result --> 0 or 1  ***/
  z = (x = 0);             /***  assignment operator; result -->   ***/
  z = (x == 1);
  z = (x = 15);
  z = (x != 2);
  z = (x < 10);
  z = (x <= 50);
  z = ((x=y) < 10);   /***  performs assignment, compares  with 10  ***/
  z = (x==5 && y<15);
  z = (x==5 && y>5 && w==10);
  z = (x==5 || y>5 && w==10);

  z = (T && T && F && x && x);  /***  ==> F; ***/
  z = (F || T || x || x);       /***  ==> T; ***/
  /***  for && and !!, order is specified, stops when result is known,  ***/
  /***  value of x doesn't matter  ***/
}
```

Examples of the Increment and Decrement Operators

```
/**  examples of the increment and decrement operators  **/
```

```
main()
{
  int x,y;

  x=5;
  y = ++x;                                /**  prefix increment  **/
  printf("++x:  x=%d, y=%d\n", x, y);

  x=5;
  y = x++;                                /**  postfix increment  **/
  printf("x++:  x=%d, y=%d\n", x, y);

  x=5;
  y = --x;                                /**  prefix decrement  **/
  printf("--x:  x=%d, y=%d\n", x, y);

  x=5;
  y = x--;                                /**  postfix decrement  **/
  printf("x--:  x=%d, y=%d\n", x, y);
}
```

Output:

```
++x:  x=6, y=6
x++:  x=6, y=5
--x:  x=4, y=4
x--:  x=4, y=5
```

More Operator Examples

```
main()
{
  int x, y, z;

  x = 128;
  y = x / 10;            /**  y = 12, the remainder is dropped  **/
  y = x % 10;            /**  y = 8, which is remainder  **/

  x = 10;
  y = !x;                /**  y = 0  **/
  y = !(!x);             /**  y = 1  **/

  x = 0;
  y = !x;                /**  y = 1  **/

  x = 10;
  x += 2;                /**  x = 12  **/
  x += 2;                /**  x = 14  **/

  x = 10;
  x -= 4;                /**  x = 6  **/

  x = 10;
  x *= 5;                /**  x = 50  **/

  x = 10;
  x /= 2;                /**  x = 5  **/
  x /= 2;                /**  x = 2  **/

  x = 2
  y = (x < 5) ? 5 : 10;              /**  y=5  **/

  x = 8
  y = (x < 5) ? 5 : 10;              /**  y=10  **/

  if (x < 5)          /**  same as the conditional   y = (x < 5) ? 5 : 10;  **/
    y = 5;
  else
```

```
    y = 10;
}
```

### Order of Evaluation

Operands of most operators will be evaluated in an unspecified order; do not assume a particular order and be careful about side effects of evaluations. Also, the order in which function arguments are evaluated is not specified. There are four operators, however, that do have a specified order of operand evaluation: && || , ?:

```
/**  test of order of operations  **/
main()
{
  int x=5, y, i=4;

  y = x * ++x;
  printf("x = %d, y = %d\n", x, y);

  printf("i = %d, ++i = %d\n", i, ++i);
}
```

Depending on the order of evaluation of the function arguments, the output can be

```
x = 6, y = 30
I = 4, ++I = 5
```

or

```
x = 6, y = 36
I = 5, ++I = 5
```

These types of expressions should be avoided.

### Type Conversion

C does some automatic type conversion if the expression types are mixed. If there are no unsigned values, the rules for conversion for two operands are:

- if any long double, convert other to long double
- else if any double, convert other to double
- else if any float, convert other to float
- else
    - convert any char or short to int
    - if any long, convert other to long

If there are unsigned expressions, the conversion rules are in K&R, p. 198.

Any explicit type conversions can be made by casting an expression to another type, using (`double`), (`float`), etc., before the expression to be cast.

Truncation Problem wth Integer Divide

```
/***  truncation problem with integer divide  ***/
main()
{
  int x=8, y=10;
  float z1, z2;

  z1 = x/y;
  z2 = (float) x / (float) y;
  printf("z1 = %6.2f, z2 = %6.2f\n", z1, z2);
}
```

Output:

```
z1 =   0.00, z2 =   0.80
```

## Expressions and Statements

**Expressions** include variable names, function names, array names, constants, function calls, array references, and structure references. Applying an operator to an expression is still an expression, and an expression enclosed within parentheses is also an expression. An *lvalue* is an expression which may be assigned a value, such as variables.
i++ x+y z = x+y

A statement is

- a valid expression followed by a semicolon
- a group of statements combined into a block by enclosing them in braces {}. This is then treated as a single statement.
- a special statement (`break, continue, do, for, goto, if, return, switch, while`,} and the null statement)
  A statement can be labeled, for use with `goto`. Since goto disrupts structured control flow, however, it is not generally recommended.

## Control Flow

Basic control flow is governed by the `if..else, while`,`do...while`, and `for` statements.

### Decision Making

Use the `if...else for` conditional decisions. (*exp* is any valid expression, *statement* is any valid statement)

Syntax:

```
if (exp)
        statement

if (exp)
        statement
else
        statement

if (exp1)
   statement
else if (exp2)
        statement
        .
        .
        .
else
        statement
```

```
 main()        /*** check if a number is odd or even ***/
{
  int i;
  scanf("%d", &i);
  if (i%2 == 0)                /** OR   if (!(i%2))  **/
    printf("i is even\n");
  else
```

```
    printf("i is odd\n");
}
```

Examples of the '`if`' Statement

```
main()
{
  int x=5;

  if (x > 0)
    printf("x = %d\n", x);

  if (x < 10)
    {
      printf("x = %d\n", x);
      x += 10;
    }
  else
    x -= 10;

  if (x==1)
    printf("one\n");
  else if (x==2)
    printf("two\n");
  else if (x==4)
    {
      printf("four\n");
      x /= 2;
    }
  else if (x==5)
    printf("five\n");
  else
    {
      printf("x = %d\n", x);
      x %= 10;
    }

  /**  'else' matches up with last unmatched 'if' in same block level  **/

  if (x > 0)
    if (x % 2)
      printf("odd\n");
    else
      printf("even\n");
  else
    printf("negative\n");

  if (!(x % 2))
    {
      if (!(x % 5))
        x /= 10;
    }
  else
    printf("odd\n");
}
```

### Looping

- while: testing at the beginning of the loop
- do...while: testing at the end of the loop, after executing the loop at least once
- for: almost the same as while, in a compact form

Syntax:

```
while (exp)
        {
```

```
                statement
        }
do
        {
                statement
        } while (exp);

for (exp1-opt ;  exp2-opt ;  exp3-opt)
        {
                statement
        }
```

Here *exp-opt* is an optional expression.

```
main()        /*** print the numbers from 1 to 10 ***/
{
  int i;

  i=1;
  while (i<=10)
    {
      printf("%d\n", i);
      i++;
    }
}
```

```
main()        /*** print the numbers from 1 to 10 ***/
{
  int i;

  i=1;
  do
    {
      printf("%d\n", i++);
    } while (i<=10);
}
```

```
main()        /*** print the numbers from 1 to 10 ***/
{
  int i;
  for (i=1 ; i<=10; i++)
    printf("%d\n", i);
}
```

### Other Control Flow

Other program control flow is governed by the `switch` statement, which allows comparison to a series of constant values, and the `goto, continue, break`, and `return` statements.

Syntax:

```
switch(exp)
 {
        case (const-exp):
                statement-opt
                break;
        case (const-exp):
                statement-opt
                statement-opt
                break;
        .
        .
        .
        default:
                statement-opt
```

```
                break;
 }
```

Here *const-exp* is a constant expression, and *statement-opt* is an optional statement. The `break;` after the default: case is optional.

Syntax:

```
 label:  statement
        goto  label;

        continue;

        break;

        return (exp-opt);
}
```

Here *label* is a statement label.

The `break;` statement is used inside the `while, do...while, for`, and `switch` statements to automatically drop out of the current loop. The `continue` statement is used inside the loops (not `switch`) to jump to the end of the loop. With `while` and `do...while`, this jumps to the next test; with for, it jumps to the increment step, and then the test.

```
/**  Example with 'do...while' and 'switch': waiting for a yes/no answer  **/
main()
{
  char ans, c;
  int answer;

  do
    {
      printf("enter y/n: ");    scanf("%c", &ans);
      switch (ans)
        {
        case 'y': case 'Y':     answer =  1;      break;
        case 'n': case 'N':     answer =  0;      break;
        default:                answer = -1;      break;
        }
    } while (answer == -1);
  printf("answer = %d\n", answer);
}
```

## The C Preprocessor

Two essential preprocessor commands are `#define` and `#include`. Constants and macros are defined with `#define`. The program can be split into separate files and combined with `#include`, and header files can also be inserted. (See sections 2.6 and 2.7)

```
#include <stdio.h>
#include <math.h>
#include "functions.c"

#define MAX 10
#define TRUE 1
#define FALSE 0
```

After inserting the math.h header file, math subroutines can be used properly. To compile using the math library on Unix, use

```
% cc -o .c -lm
```

```
 /**  program to calculate x using the quadratic formula **/
#include
main()
{
  float a, b, c, d, x, x1, x2;
  printf("input a, b, c:  ");
  scanf("%f %f %f", &a, &b, &c);
  d = b*b - 4.0*a*c;
  if (d >= 0)               /**  check if solution will be real or complex  **/
    {
      x1 = (-b+sqrt(d)) / (2.0*a);        /** sqrt() from the math library  **/
      x2 = (-b-sqrt(d)) / (2.0*a);
                        /**  need parentheses for proper order of operations  **/
      printf("x = %f, %f\n",x1,x2);
    }
  else
    printf("x is complex");
}
```

```
/**  an example of formated output  **/
#include
main()
{
  int n=10, i, x;

  for (i=0, x=1; i
```

Output:

```
log(    1) =    0.000
log(    2) =    0.693
log(    4) =    1.386
log(    8) =    2.079
log(   16) =    2.773
log(   32) =    3.466
log(   64) =    4.159
log(  128) =    4.852
log(  256) =    5.545
log(  512) =    6.238
```

The `#if...#endif` preprocessor command can be used to comment out a section of code which may have comments within it.

```
/** using #if for nested comments  **/
#define   TRUE    1
#define   FALSE   0

main()
{
  int x=5;

  printf("x = %d\n", x);

#if FALSE        /**  everything until the matching #endif is commented  **/
  x = 304;
#endif

  printf("x = %d\n", x);
}
```

Output:

```
x = 5
x = 5
```

## Problems

1. Run example programs to become familiar with using C
2. Write a rogram to print all Fahrenheit and Celsius temperatures using the conversion
   C = (F-32)*5/9
   for 20 degree increments from 32 to 212. (See K&R,p. 9 if you are stuck.)
3. Input a number in print all its factors
4. Input a number and decide if it is prime
5. Change the quadradic formula program so that it also prints the complex solutions
6. Input an integer and print the value of each digit in English: `932 => nine three two`
7. Count the number of characters and lines in a file (use '`\n`' to find lines)

---

Table 2: Summary of C operators (K&R, p. 53)

| Operator | Description | Associativity |
|---|---|---|
| () | Function call | **left to right** |
| [] | Array element reference | |
| -> | Pointer to structure member reference | |
| . | Structure member reference | |
| - | Unary minus | **right to left** |
| + | Unary plus | |
| ++ | Increment | |
| -- | Decrement | |
| ! | Logical negation | |
| ~ | Ones complement | |
| * | Pointer reference (indirection) | |
| & | Address | |
| `sizeof` | Size of an object | |
| `(type)` | Type cast (conversion) | |
| * | Multiplication | **left to right** |
| / | Division | |
| % | Modulus | |
| + | Addition | |
| - | Subtraction | |
| << | Left shift | |
| >> | Right shift | |
| < | Less than | |
| <= | Less than or equal to | |
| > | Greater than | |
| >= | Greater than or equal to | |
| == | Equality | |
| != | Inequality | |
| & | Bitwise AND | |
| ^ | Bitwise XOR | |

| | | | | |
|---|---|---|---|---|
| \| | | | Bitwise OR | |
| && | | | Logical AND * | |
| \|\| | | | Logical OR * | |
| ?: | | | Conditional * | **right to left** |
| = | *= | /= | Assignment operators | |
| %= | += | -= | | |
| ^= | &= | \|= | | |
| <<= | >>= | | | |
| , | | | Comma operator * | **left to right** |

* order of operand evaluation is specified

Table 3: Basic printfConversions (K&R, p.244)

| Character | Argument Type; Printed As |
|---|---|
| d, I | **int**; decimal number |
| o | **int**; unsigned octal number (without a leading zero) |
| x, X | **int**; unsigned hexadecimal number (without a leading Ox or OX, using `abcdef` or `ABCDEF` for 10,...,15) |
| u | **int**; unsigned decimal number |
| c | **int**; single character |
| s | **char**; print characters from the string until a '`\0`' or the number of charachters given by the precision |
| f | **double**; [-]*m.dddddd*, where the number of *d*'s is given by the precision (default is 6) |
| e, E | **double**; [-]*m.dddddde ± xx* or [-]*m.ddddddE ± xx* where the number of *d*'s is given by the precision (default is 6) |
| g, G | **double**; use `%e` or `%E` if the exponent is less than -4 or greater than or equal to the precision; otherwise use `%f`; trailing zeros and a trailing decimal point are not printed |
| p | **void ***; pointer (implementation-dependent representation) |
| % | no argument is converted; print a % |

Table 4: Basic scanf Conversions (K&R, p.246)

| Character | Input Data; Argument Type |
|---|---|
| d | decimal integer; int * |
| I | integer; int * ; the integer may be in octal (leading 0) or hexadecimal (leading 0x or 0X) |
| o | octal intger (with or without leading zero); int * |
| u | unsigned decimal integer; unsigned int * |
| x | hexadecimal number (with or without a leading 0x or 0X); int * |
| c | characters; char *. The next input characters (default 1) are placed at the indicated spot. The normal skip over white space is suppressed; to read the next non-white space character, use %1s |
| s | character string (not quoted); char * ; pointing to an array of characters large enough for the string and a terminating `\0' that will be added |
| e, f, g | floating-point number with optional sign, optional decimal point, and optional exponent; float * |
| % | literal %; no assignment is made |

Table 5: Escape Sequences (K&R, p.193)

| | | |
|---|---|---|
| \a bell | \b backspace | \f formfeed |
| \t horizontal tab | \v vertical tab | \\ backslash |
| \' single quote | \" double quote | \ooo octal number |
| \n newline | \? question mark | \xhh hexadecimal number |
| \r carriage return | | |

Table 6: ASCII Character Set (hexadecimal value, 0xNM, base 16)

| **Row (M)** | **Column (N)** | | | | | |
|---|---|---|---|---|---|---|
| | **0x2** | **0x3** | **0x4** | **0x5** | **0x6** | **0x7** |
| 0x0 | SPC | 0 | @ | P | ` | p |
| 0x1 | ! | 1 | A | Q | a | q |
| 0x2 | " | 2 | B | R | b | r |
| 0x3 | # | 3 | C | S | c | s |
| 0x4 | $ | 4 | D | T | d | t |
| 0x5 | % | 5 | E | U | e | f |
| 0x6 | & | 6 | F | V | f | v |
| 0x7 | ' | 7 | G | W | g | w |
| 0x8 | ( | 8 | H | X | h | x |
| 0x9 | ) | 9 | I | Y | i | y |
| 0xA | * | : | J | Z | j | z |
| 0xB | + | ; | K | [ | k | { |
| 0xC | , | < | L | \ | l | \| |
| 0xD | - | = | M | ] | m | } |
| 0xE | . | > | N | ^ | n | ~ |
| 0xF | / | ? | O | _ | o | DEL |

# Chapter 2: Functions

## Reasons for Using Functions

- saves repetition of common routines
- functions can be used by different parts of the program
- modularizes the program, making it easier to use
  - write---can work on pieces independently
  - read---have to understand what the large pieces do, not each statement, hiding
  - unnecessary detail
  - modify---changing one part should have limited effects on other parts
  - debug---can debug the functions independently, then put everything together other people can use the pieces without worrying about details
- gives new variables with each call (automatic variables, lexical scoping)
- allows recursive processes

## Basic Structure

The syntax for declaring a function is

*return-type function-name (argument declarations)*
*{*

*local variable declarations*
*statements*

*}*

The function prototype is a declaration and is needed if the function is defined after its use in the program. The syntax is
*return-type function-name (argument declarations);*

where the argument declarations must include the types of the arguments, but the argument names are optional. If the function is defined before its use, then a prototype is not necessary, since the definition also serves as a declaration.

If the *return-type* is omitted, `int` is assumed. If there are no *argument declarations*, use `void`, not empty parentheses.

Here are four examples of how functions can be used:

- A function that has no arguments and does not return a value:

```
void print_message(void)
{
  printf("hello\n");
}

main()
{
  print_message();
}
```

- A function that takes an argument. The arguments are separated by commas; the order in which they are evaluated is unspecified. The value of each argument is passed to the corresponding parameter of the function.

```
void print_integer(int i)
{
  printf("i = %d\n", i);
}

main()
{
  int n=5;

  print_integer(n);
}
```

- A function that returns a value:

```
int input_integer(void);                /** function prototype declarations **/

main()
{
  int x, y, z;
  x = input_integer();
  y = input_integer();
  printf("the sum is %d\n", z=x+y);
}
```

```
int input_integer(void)
{
  int a;
  do
    {
      printf("input a positive integer: ");
      scanf("%d", &a);
    } while (a <= 0);
  return a;
}
```

- A function that takes an argument m and returns a value:

```
main()
{
  int x, y, z=100;
  int input_integer_le_n(int n);           /** prototype declaration can be **/
                                           /**   inside a function **/
  x = input_integer_le_n(z);
  y = input_integer_le_n(z);
  printf("the sum is %d\n", x+y);
}

int input_integer_le_n(int n)
{
  int a;
  do
    {
      printf("input a positive integer less than %d: ", n);
      scanf("%d", &a);
    } while (a<=0 || a>n);
  return a;
}
```

## Return Statement

A function can return a value of any type, using the `return` statement,

Syntax:
return *exp*;
return (*exp*);
return;

The `return` statement can occur anywhere in the function, and will immediately end that function and return control to the function which called it. If there is no `return` statement, the function will continue execution until the closing of the function definition, and return with an undefined value.

The type of the expression returned should match the type of the function; C will automatically try to convert *exp* to the *return-type*.

## Difference between ANSI-C and "Traditional C"

If the function return type is not int, it must be specified by the function prototype declaration. This is done differently in ANSI C and "Traditional C." For a function returning the maximum of two numbers, the ANSI-C function would be

```
float max(float x, float y);
    /*  OR: float max(float, float); */
```

```
              /**  variable names {x,y} are not necessary, only the types  **/
              /**    are, but using the names makes the code more readable  **/
main()
{
  float x,y,z;
  ...
  z = max(x,y);
  ...
}

float max(float x, float y)     /**  argument types are in the definition  **/
{
  if (x < y)
    return y;
  else
    return x;
}
```

The "Traditional C" declarations would be

```
float max();       /**  argument types are not included in the declaration  **/

main()
{
  float x,y,z;
  ...
  z = max(x,y);    /**  the function call is the same  **/
  ...
}

float max(x,y)
     float x,y;    /**  argument types listed after the definition  **/
{
  if (x < y)
    return y;
  else
    return x;
}
```

## Object Storage Classes and Scope

Functions, as with any compound statement designated by braces {}, have their own scope, and can therefore define any variables without affecting the values of variables with the same name in other functions. To be able to affect the variables, they can be made "global" by defining them externally

Available storage classes for variables are

- **`automatic`**: declared when entering the block, lost upon leaving the block; the declarations must be the first thing after the opening brace of the block
- **`static`**: the variable is kept through the execution of the program, but it can only be accessed by that block
- **`extern`**: available to all functions in the file AFTER the declaration; use extern to make the variable accessible in other files
- **`register`**: automatic variable which is kept in fast memory; actual rules are machine-dependent, and compilers can often be more efficient in choosing which variables to use as registers.

The scope of an object can be local to a function or block, local to a file, or completely global.

- *local to a function/block*: automatic or static variables which can only be used within that function/block. Function parameters (arguments) are local to that function.

- *global (local to a file)*: static variables declared outside all functions, accessible to any functions following the declaration
- *external (accessible in several files)*: declared as extern, accessible to functions in that file following the declaration, even if the variable is defined in another file.

Again, it is important to distinguish between a declaration and a definition.

- *declaration*: specifies the type of the identifier, so that it can subsequently be used.
- *definition*: reserves storage for the object

There can only be one definition of a variable within a given scope.

```
main()
{
  int x;
  int x;     /***  illegal: cannot define x twice  ***/
  x=6;
}
```

Also, external variables can only be defined once, although they can be declared as often as necessary. If an external variable is initialized during the declaration, then that also serves as a definition of the variable.

```
int x;                  /***  definition of global variable  ***/
extern int x;           /***  declaration so other files can use it  ***/
extern int y;           /***  declaration, must be defined elsewhere  ***/
extern int z=0;         /***  declaration, definition, and initialization  ***/
                          /***    can be used by other files  ***/

main()
{
  printf("%d", z);
}
```

```
/**  An example with limited variable scope within a file  **/

main()
{
  int m=2, n=5;
  float x=10.0, y=14.1;
  int count;

  int print_pair_i(int x, int y);
  int print_pair_f(float x, float y);
  int print_pair_d(double x, double y);
  void reset_count(void);

  count = print_pair_f(x, y);    printf("%d\n", count);

  print_pair_i(m, n);
  count = print_pair_i(m, n);    printf("%d\n", count);

  reset_count();
  count = print_pair_f(x, y);    printf("%d\n", count);

  print_pair_d(15.0, 28.0);
  count = print_pair_d(20.0, 30.0);    printf("%d\n", count);
}

int count=0;             /**  all functions following this declaration **/
                         /**    can use this variable **/

int print_pair_i(int x, int y)
{
  printf("(%d, %d)\n", x, y);
```

```
  return ++count;
}

int print_pair_f(float x, float y)
{
  printf("(%f, %f)\n", x, y);
  return ++count;
}

void reset_count(void)        /**  resets the counter that print_pair_i  **/
{
  count=0;                    /**  and print_pair_f use    **/
}

int print_pair_d(double x, double y)
{
  static int count=0;      /** a private copy, supersedes previous variable **/
  printf("(%lf, %lf)\n", x, y);
  return ++count;
}
```

Output:

```
(10.000000,14.100000)
1
(2,5)
(2,5)
3
(10.000000,14.100000)
1
(15.000000,28.000000)
(20.000000,30.000000)
2
```

## Larger Programs

A program can also be made of pieces, with a header file. This uses the `#include` preprocessor command. One way is to compile all the files separately. This can most easily be done with make (see Appendix A).

```
/* FILE: large_prog.c */
#include "large.h"

int max_val;       /***  the ONLY definition of max_val  ***/

main()
{
  int n=1;
  float x=5.0;

  printf("%f, %f, %f\n", A(x), B(), C(n));
  printf("%f, %f, %f\n", A(x), C(n*2), B());
}

float A(float x)
{
  return (x*x*x);
}

/*
**      to compile:
**        % cc -o large large_prog.c large_sub.c
**
**      if large_sub.c will not been changed, can use
**        % cc -c large_sub.c
**      once, followed by
```

```
**        % cc -o large large_prog.c large_sub.o
**      whenever large_prog.c is changed
*/
```

```
/* FILE: large.h */
#include

#define  TRUE   1
#define  FALSE  0

extern int max_val;

extern float A(float x);
extern float B(void);
extern float C(int n);
```

```
/* FILE: large_sub.c */
#include "large.h"

int num;     /***  only functions in this file can access num  ***/

float B(void)
{
  return ((float) num);
}

float C(int n)
{
  num=n;
  return (n*4.0);
}
```

This has the following output, which shows a dependence on the argument evaluation order in **`printf()`**.

Output:

```
125.000000, 0.000000, 4.000000
125.000000, 8.000000, 2.000000
```

Alternatively, the files can be put together by the preprocessor. This is simpler, but it compiles all the files, not just the unchanged ones.

```
/* FILE: large-2.h */
#include

#define  TRUE   1
#define  FALSE  0

int max_val;                  /*** global variable ***/
```

```
/* FILE: large_sub.c */
#include "large.h"

int num;     /***  only functions in this file can access num  ***/

float B(void)
{
  return ((float) num);
}

float C(int n)
{
```

```
  num=n;
  return (n*4.0);
}
```

```
/* FILE: large-2_prog.c */
#include "large-2.h"
#include "large-2_sub.c"

float A(float x);

main()
{
  int n=1;
  float x=5.0;

  printf("%f, %f, %f\n", A(x), B(), C(n));
  printf("%f, %f, %f\n", A(x), C(n*2), B());
}

float A(float x)
{
  return (x*x*x);
}

/***

to compile:
  % cc -o large large-2_prog.c
 ***/
```

## Macros

Small procedures like swap() and max() can also be written as macros using #define

```
#define MAX(x,y)  ((x) > (y) ? (x) : (y))           /**  macro for maximum  **/

float max(float x, float y)                       /**  function for maximum  **/
{
  return (x>y ? x : y);
}
```

There are advantages to each. Since `#define` causes a macro substitution, code is replaced before compilation. It will run faster, since it won't make a function call, but it will evaluate either x or y twice, which may have side effects or take extra computational effort. `MAX` will work on any arithmetic type, while `max()` will only work on floats. The functions `dmax()` and `imax()` are needed for double and integer values.

Macro substitutions do not happen within quoted strings or as parts of other identifiers: `#define NO 0' does not replace `NO' in `x = NOTHING;' or `printf("NO!");.' Also, parentheses are very important:

```
#define RECIPROCAL_1(x)     1/(x)
#define RECIPROCAL_2(x)     1/x

main()
{
  float x=8.0, y;
  y = RECIPROCAL_1(x+10.0);
  printf("1/%3.1f = %8.5f\n", x, y);
  y = RECIPROCAL_2(x+10.0);
  printf("1/%3.1f = %8.5f\n", x, y);
}
```

Output:

```
1/8.0 =  0.05556
1/8.0 = 10.12500
```

To continue a macro definition onto the next line, use a '\' at the end of the current line.

## Problems

8. Write a function to raise a number to an integer power, x_to_int_n(x,n)
9. Write a function to calculate factorial (n)
10. Try to write the factorial function recursively
11. Write a program to input a positive number and print all the prime number less than or equal to that number, using functions like is_prime() and get_positive_int()

---

Table 7: Summary of Storage Classes (Kochan, p. 416)

| If storage class is | And variable is declared | Then it can be referenced | And can be initialized with | Comments |
|---|---|---|---|---|
| static | Outside a function | Anywhere within the file | Constant expressions only | Variables are initialized only once at the start of program execution; values are retained through function calls; default initial value is 0 |
| | Inside a function/ block | Within the function/ block | | |
| extern | Outside a function | Anywhere within the file | Constant expressions only | Variable must be declared in at least one place without the extern keyword or in exactly one place with an initial value |
| | Inside a function/ block | Within the function/ block | | |
| auto | Inside a function/ block | Within the function/ block | Any valid expression; arrays, structs, unions to constant expressions only if {...} list is used | Variable is initialized each time the function/ block is entered; no default value |
| register | Inside a function/ block | Within the function/ block | Any valid expression | Assignment to a register not guaranteed; varying restrictions on types of variables that can be declared; cannot take the address of a register variable; initialized each time function/ block is entered; no default value |
| omitted | Outside a function | Anywhere within the file or by other files that contain appropriate declarations | Constant expressions only | This declaration can appear in only one place; variable is initialized at the start of program execution; default value is 0 |
| | Inside a function/ | (See auto) | (See auto) | Defaults to auto |

| | block | | | |
|---|---|---|---|---|

# Chapter 3: Pointers

## Pointer Definition and Use

A pointer is a variable whose value is the address of some other object. This object can have any valid type: int, float, struct, etc. The pointer declaration syntax is
*type *ptr-name;*
A pointer to an integer is declared as

```
int *p;
```

where `p' is of type `int *', pointer to an integer, and `*p' is of type integer. The type of object the pointer references must be specified before a value can be obtained. The operators used with pointers are the pointer reference/indirection (*) and address (&) operators:

```
main()
{
  int x, *px;                /*  defines the pointer px  */
  px = &x;                   /*  &x ==> address of x  */
  *px = x;                   /*  *px ==> value px points to  */
}
```

where the value of px is the address of x, and *px is equivalent to x.

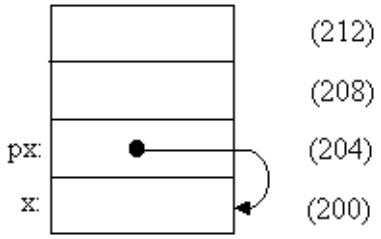



Figure 1: Memory diagram with pointers. The memory address are given by (*n*)

```
/* example program */
main()
{
  int x=5, y=6, *p;
  p = &x;                          /**  pointer needs to point to something  **/
  printf("1.  x=%d, y=%d, *p=%d\n", x, y, *p);
  x = 7;
  printf("2.  x=%d, y=%d, *p=%d\n", x, y, *p);
  *p = 8;
  printf("3.  x=%d, y=%d, *p=%d\n", x, y, *p);
  p = &y;
  printf("4.  x=%d, y=%d, *p=%d\n", x, y, *p);
  *p += 10 * (x * *p);
  printf("5.  x=%d, y=%d, *p=%d\n", x, y, *p);
}
```

Output:

```
1.  x=5, y=6, *p=5
2.  x=7, y=6, *p=7
3.  x=8, y=6, *p=8
```

```
4.  x=8, y=6, *p=6
5.  x=8, y=486, *p=486
```

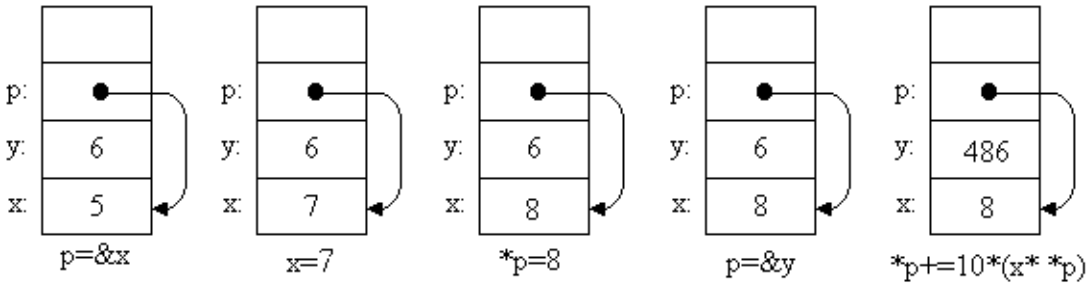



Figure 2: Memory diagram with pointers - the example program

Valid pointer operations:

- assignment to a pointer of the same type
- assigning a pointer to pointer of type (void *) and back
- adding or subtracting a pointer and an integer (including increment and decrement)
- subtracting or comparing two pointers which point to members of the same array
- assigning or comparing to zero

```
/**  Valid Pointer Operations  **/
#define   NULL  0

main()
{
  int x, y;
  int *px=(&x);              /**  can initialize an automatic pointer  **/
  int *py;
  void *pv;

  py = px;                   /**  assign to pointer of same type  **/
  px = (int *) pv;           /**  recast a (void *) pointer  **/
  pv = (void *) px;          /**  recast to type (void *)  **/
  py = px+2;                 /**  for use with arrays  **/
  px++;                      /**  for use with arrays  **/
  if (px == NULL)            /**  compare to null pointer  **/
    py=NULL;                 /**  assign to null pointer  **/
}
```

Invalid pointer operations:

- adding two pointers
- multiply, divide, shift, mask pointers
- add float or double numbers to a pointer
- assign pointers of different types without a cast

```
/**  Illegal Pointer Operations  **/
main()
{
  int x, y;
  int *px, *py, *p;
  float *pf;

  px = &x;              /**  legal assignment  **/
  py = &y;              /**  legal assignment  **/
  p = px + py;          /**  addition is illegal  **/
  p = px * py;          /**  multiplication is illegal  **/
  p = px + 10.0;        /**  addition of float is illegal  **/
```

```
  pf = px;              /**  assignment of different types is illegal  **/
}
```

## Pointers as Function Arguments: "Call by Value"

When a function is called, it gets a copy of the arguments ("call by value"). The function cannot affect the value that was passed to it, only its own copy. If it is necessary to change the original values, the addresses of the arguments can be passed. The function gets a copy of the addresses, but this still gives it access to the value itself. For example, to swap two numbers,

```
main()
{
  float x=5.0, y=6.0;
  void swap_A(float *x, float *y), swap_V(float x, float y);

  printf("x = %6.2f, y = %6.2f\n", x, y);
  swap_V(x, y);
  printf("x = %6.2f, y = %6.2f\n", x, y);
  swap_A(&x, &y);
  printf("x = %6.2f, y = %6.2f\n", x, y);
}

void swap_A(float *x, float *y)      /**  passes addresses of x and y  **/
{
  float tmp = *x;
  *x = *y;
  *y = tmp;
}

void swap_V(float x, float y)        /**  passes values of x and y  **/
{
  float tmp = x;
  x = y;
  y = tmp;
}
```

Output:

```
x =   5.00, y =   6.00
x =   5.00, y =   6.00
x =   6.00, y =   5.00
```

Here, swap_V() does not work, but swap_A() does.

## Arrays

An array is a contiguous space in memory which holds a certain number of objects of one type. The syntax for array declarations is
*type array-name[const-size];*
`static` *type array-name[const-size] = initialization-list;*
`static` *type array-name[] = initialization-list;*

An array of 10 integers is declared as

**int x[10];**
with index values from 0 to 9.

A static array can be initialized:

```
static int x[5] = 7,23,0,45,9;
static int x[] = 7,23,0,45,9;

static int x[10] = 7,23,0,45,9,0,0,0,0,0;
static int x[10] = 7,23,0,45,9;
```

where the remaining five elements of `x[10]` will automatically be 0. Part of the array can be passed as an argument by taking the address of the first element of the required subarray (using &), so `&x[6]` is an array with 4 elements.

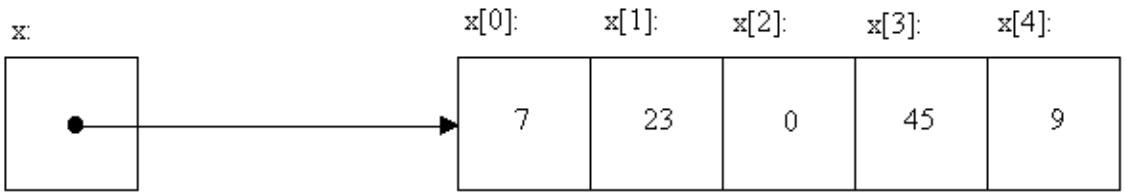



Figure 3: Box-and-pointer diagram of arrays: `static int[5]: = {7, 23, 0, 45, 9};`

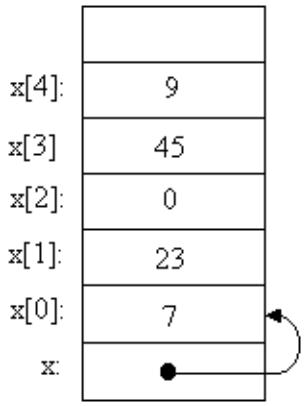



Figure 4: Memory diagram of arrays

```
#define   MAX   10
static int b[MAX] = {1, 5, 645, 43, 4, 65, 5408};

main()
{
  int i;
  int *p, *p_max;
  static int a[] = {1, 5, 645, 43, 4, 65, 5408, 4, 7, 90};

  printf("array elements:  ");
  for (i=0; i *p_max)
      {
        p_max = p;
        printf("new maximum value: %d\n", *p);
      }
    else
      printf("\"distance\" from maximum element: %d\n", (p-p_max));
}
```

Output:

```
array elements:  1  5  645  43  4  65  5408  4  7  90
"distance" from maximum element: 0
new maximum value: 5
new maximum value: 645
"distance" from maximum element: 1
```

```
"distance" from maximum element: 2
"distance" from maximum element: 3
new maximum value: 5408
"distance" from maximum element: 1
"distance" from maximum element: 2
"distance" from maximum element: 3
```

The array and pointer are closely related, in that x[i] and *(x+i) both get the i+1 element in the array, and &x[i] and (x+i) are both pointers to the i+1 element.

There are differences between them, however. An array name is a constant, while a pointer is a variable, so

int x[10], *px;
px = x; px++; /** valid **/
x = px; x++; /** invalid, cannot assign a new value **/

Also, defining the pointer only allocates memory space for the address, not for any array elements, and the pointer does not point to anything. Defining an array (x[10]) gives a pointer to a specific place in memory and allocates enough space to hold the array elements. To allocate space for an array declared as a pointer, use *malloc() or *calloc(), which are declared in stdlib.h, and then free() to deallocate the space after it is used.

```
/***  memory_allocation for arrays  ***/
#include

main()
{
  int n;
  float *a, *b;

  a = (float *) malloc(n * sizeof(float));   /***  not initialized  ***/
  b = (float *) calloc(n, sizeof(float));    /***  initialized to 0  ***/

  if ((a == NULL) || (b == NULL))
    printf("unable to allocate space");

  free(a);
  free(b);
}
```

## Functions Returning Pointers

In addition to passing pointers to functions, a pointer can be returned by a function. Three pointers must be of the same type (or be recast appropriately):

- the function return-type
- the pointer type in the return statement
- the variable the return-value is assigned to

The pointer should not point to an automatic local variable within the function, since the variable will not be defined after the function is exited so the pointer value will be invalid.

```
/***  returns a pointer to the maximum value in an array  ***/

int maximum_val1(int A[], int n);
int maximum_val2(int *a, int n);
int *maximum_ptr1(int *a, int n);
int *maximum_ptr2(int *a, int n);

main()
```

```
{
  int i,n;
  int A[100], *max;

  printf("number of elements: "); scanf("%d",&n);

  printf("input %d values:\n", n);
  for (i=0; i<n; i++)
    scanf("%d", A+i);

  printf("maximum value = %d\n", maximum_val1(A,n));
  printf("maximum value = %d\n", maximum_val2(A,n));

  max = maximum_ptr1(A,n);
  printf("maximum value = %d\n", *max);
}

int maximum_val1(int A[], int n)
{
  int max, i;
  for (i=0, max=0; i max)
      max = A[i];
  return max;
}

int maximum_val2(int *a, int n)
{
  int max=0;
  for ( ; n>0 ; n--, a++)
    if (*a > max)
      max = *a;
  return max;
}

int *maximum_ptr1(int *a, int n)   /***  will work  ***/
{
  int *max = a;
  for ( ; n>0; n--, a++)
    if (*a > *max)
      max = a;
  return max;         /***  max points to an element of the array  ***/
}

int *maximum_ptr2(int *a, int n)   /***  won't work  ***/
{
  int max = *a;
  for ( ; n>0; n--, a++)
    if (*a > max)
      max = *a;
  return &max;     /***  max will not exist after function ends  ***/
}
```

Output:

```
number of elements: 10
input 10 values:
3
4
5
3
6
5
4
7
5
6
maximum value = 7
```

```
maximum value = 7
maximum value = 7
```

## Multidimensional Arrays

A multidimensional array can be defined and initialized with the following syntax:
*type array-name[const-num-rows][const-num-cols];*
`static` *type array-name[const-num-rows][const-num-cols] = init-list;*
`static` *type array-name[][const-num-cols] = initialization-list;*

```
static int x[][3] = {{3, 6, 9}, {4, 8, 12}};   /* static--can be initialized */
static int y[2][3] = {{3},{4}};                 /* OR {{3,0,0},{4,0,0}}  */

main()
{
  int z[2][3];
  printf("%d\n", x[1][2]);                      /**  output: 12  **/
}
```

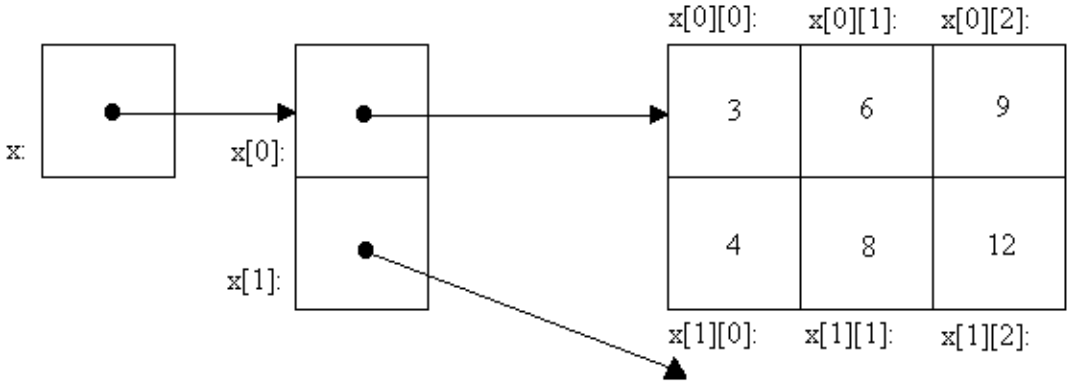
To get a pointer to the ith row of the array, use x[i-1].

Alternatively, a pointer to a pointer can be used instead of a multidimensional array, declared as int **y. Then y points to a one-dimensional array whose elements are pointers, *y points to the first row, and **y is the first value.

When passing a two-dimensional array to a function, it can be be referenced in four ways:

- **y, as a pointer to a pointer;
- *y[], an array of pointers;
- y[][COL], an array of arrays, unspecified number of rows; or
- y[ROW][COL], an array of arrays, with the size fully specified.

Here COL and ROW must be constants.



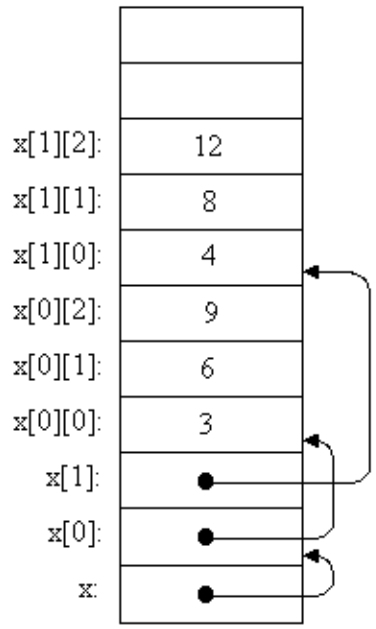




Figure 5: Box-and-pointer & memory diagrams of 2D arrays:

```
static int x[2][3] = {{3, 6, 9},{4, 8, 12}};
```

```
#define  MAX  5
#define  LINEFEED   printf("\n")

int **imatrix(int n, int m);

void print_imatrix1(int A[][MAX], int n, int m);
void print_imatrix2(int *a[], int n, int m);

int sum_imatrix1(int a[][MAX], int n, int m);
int sum_imatrix2(int **a, int n, int m);

void input_imatrix(int **a, int n, int m);


main()
{
  int i, j, n = MAX;
  int **A, **b, C[MAX][MAX];

  A = imatrix(n, n);
  for (i=0; i
```

Output:

```
0       1       2       3       4
1       2       3       4       5
2       3       4       5       6
3       4       5       6       7
4       5       6       7       8

0       0       0       0       0
0       1       2       3       4
0       2       4       6       8
0       3       6       9       12
0       4       8       12      16

100
100

size of array to input: 2
3 4
7 5
    3       4
    7       5
```

## Strings

Strings are simply character arrays, or more accurately, pointers to characters. A string literal is enclosed within quotes,"...", and is an array with those characters and `0' at the end, so "hello" <==>{'h','e','l','l','o','\0'}. The string can be defined as

**`static char *p = "hello"`**

An example illustrating the use of pointers with the string copies one string into another:

```
main()
{
  char *t = "hello", s[100];
  void strcpy(char *, char *);
  strcpy(s,t);
  printf("%s\n", s);                          /**  will print 'hello'  **/
}

/**  strcpy:  copy t to s; pointer version 2  (K&R, p 105)  **/

void strcpy(char *s,char *t)
{
  while (*s++ = *t++)         /**  OR   while ((*s++ = *t++) != '\0')  **/
    ;
}
```

## Command Line Arguments

It is often useful to give a program arguments when it is run. This is done with command line arguments, which are arguments of `main()`. There are two arguments: an integer, `argc`, which is the number of items in the command line (including the program name), and an array of strings, `*argv[]`, which are the actual items. The first string, `argv[0]`, is the name of the function being executed.

If a number is needed, it has to be obtained using `sscanf()`, which is the same as `scanf()` except it takes a string as its first argument. This example prints the square root of a number.

```
#include
```

```
main(int argc, char *argv[])              /**  program to find sqrt(x)  **/
{
  float x;
  if (argc == 2)
    {
      sscanf(argv[1], "%f", &x);
      printf("the square root of %f is %f\n", x, sqrt(x));
    }
  else
    {
      printf("Wrong number of arguments\n");
      printf("usage: %s x \n", *argv);
    }
}
```

## Pointers to Functions

Functions can return pointers, or a pointer can point to a function:

int *f(); /* a function f returning a pointer to an int */
int (*f)(); /* a pointer to a f which returns an int */

Since the precedence of () is higher than *, parentheses are needed around *f to get a pointer to a function.

```
float square(float x);
float cube(float x);
float arith(float x, float (*func)(float));

main()
{
  float x, y, z;
  printf("Enter x: ");    scanf("%f", &x);
  y = arith(x, square);
  z = arith(x, cube);
  printf("x = %f, x^2 = %f, x^3 = %f\n", x, y, z);
}

/**  the arguments for arith() are x and func,
 **    which is a pointer to a function whose argument is one float
 **/

float arith(float x, float (*func)(float))
{
  return (*func)(x);
}

float square(float x)
{
  return x*x;
}

float cube(float x)
{
  return x*x*x;
}
```

## Problems

11. Write a program to input two matrices, add and multiply them, and print the resulting matrices. The matrices can be any size up to a maximum (`#define MAX 5`, for example). Use functions `input_matrix,`

```
print_matrix, multiply_matrix.
```

---

# Chapter 4: Structures

## Syntax and Operations

A structure is a data type which puts a variety of pieces together into one object. The syntax is given below.

```
struct structure-tag-opt {
        member-declarations
        structure-names-opt ; }

        struct structure-tag structure-name;

        structure-name.member ptr-to-structure->member
```

```
struct time
{
        int hur;
        int minute;
        int second;
} now;

main()
{
        struct time later;

now.hour = 10;
now.minute = 30;
now.second = 4;

later = now;
printf("the later time is %d:%2d:%2d\n", later.hour, later.minute, later.second);
}
```

This declares structure tag, struct time, which keeps the members, hour, minute, second, in one piece. The variables now and later are structures of type struct time, declared the same way as integers. The members of the structure can be any type: int, float, double, char, arrays, other structures, and pointers to any of these. To access the members, there are two operators available (. and ->). To access the specified member, use the . operator.

Valid structure operations:

- assigning to or copying it as a unit
- accessing its members
- taking its address with &
- passing it as an argument to functions, returning it by functions
    - With ``Traditional C,'' structures can not be used as arguments or return types of functions. Pointers to structures must be used to interact with functions. If compatibility is needed, pointers to structures should continue to be used exclusively. Unfortunately, this may connect functions together more than necessary.
- initializing it with a list of constant member values, or by assignment for automatic structures.
    - Initialization is not allowed with "Traditional C," so a separate assignment statement should be used.

Invalid structure operations:

- they may not be compared

With ANSI-C, structures are initialized in the same way as arrays,

```
struct time noon = 12, 00, 00;
```

Pointers to structures are very useful, and often necessary when functions are used. For example,

```
struct time now, *t;
t = &now;                        /* identical to x=&y with numerical types */
(*t).hour = 6;                       /* gets the hour */
t->minutes = 30;           /* gets the minutes */
```

The variable t is a pointer to a structure. Since the precedence of . is higher that that of *, the parentheses are needed to get the actual structure from the pointer to it. Since this is used often, the -> operator was made to do the same thing.

Structure members can be other structures. For example,

```
struct time
{
  int hour, minute, second;
} ;
struct date
{
  int month, day, year;
} ;
struct dt
{
  struct date d;
  struct time t;
} ;

main()
{
  struct dt start_class;
  start_class.d.month = 1;
  start_class.d.day = 5;
  start_class.d.year = 93;
}
```

### typedef

`typedef` defines a new type, which can simplify the code. Here is the syntax:

```
typedef data-type TYPE-NAME;
typedef struct structure-tag TYPE-NAME;
typedef struct
{
        member-declarations
} TYPE-NAME ;
```

Using typedef also helps portability, especially with machine dependent data types and sizes. With typedef, the change can be made in one place and is fixed for the whole program. For example,

typedef int Length, Width, Height;
typedef struct time TIME;
TIME now, *t;

will specify the types Length, Width, Height and TIME, and now and t are defined above. typedef is a syntactic simplification to aid reading and modifying the program. It does not actually create new types.

## Array of Structures

An array of structures can also be defined:

```
struct date
{
  int month, day, year;
};
typedef struct date DATE;

main()
{
  int i;
  DATE birthdays[10], *bday;
  bday = birthdays;                  /***  pointer <==> array name  ***/
  for (i=0; i<10; i++, bday++)
    scanf("%d %d %d", &bday->month, &((*bday).day), &birthdays[i].year);

  for (i=0, bday = birthdays; i<10; i++, bday++)
    printf("%2d/%02d/%2d\n", bday->month, bday->day, bday->year);
}
     /***  the %02d pads the field with 0s, not spaces  ***/
```

When bday is defined as a pointer of type DATE (struct date), incrementing will be done properly to get to successive structures in the array.

## Use with Functions

Structures can be used with functions. Just as with other data types, either the structure or a pointer to the structure can be passed to the function. The choice should be based on three things,
1. does the structure need to be changed by the function,
2. is the structure small enough that copying it as a local argument will not affect performance,
3. does compatibility with old compilers require using pointers to the structures.

In addition, the function can return a structure or a pointer to a structure.

```
/***  functions to a increment the time and date  ***/

#define  PRINT_TIME(t)  printf("%2d:%2d:%2d", t.hour, t.minute, t.second)
#define  PRINT_DATE(d)  printf("%2d/%2d/%2d", d.month, d.day, d.year)
#define  LINEFEED       printf("\n")

typedef struct
  {
    int hour, minute, second;
  }  TIME;

typedef struct
  {
    int month, day, year;
  }  DATE;

void time_increment(TIME *t, DATE *d);
void date_increment(DATE *d);
DATE date_increment2(DATE d);

main()
{
  TIME now;
  DATE today;
```

```
  now.hour = 12;  now.minute = 30;  now.second = 15;
  today.month = 1;  today.day = 7;  today.year = 93;

  PRINT_TIME(now);  LINEFEED;
  PRINT_DATE(today);  LINEFEED;

  time_increment(&now, &today);
  PRINT_TIME(now);  LINEFEED;
  PRINT_DATE(today);  LINEFEED;

  date_increment(&today);
  PRINT_DATE(today);  LINEFEED;

  PRINT_DATE(date_increment2(today));  LINEFEED;
    /***  calls date_increment2 three times in macro  ***/
}

/*** time_increment needs to be able to change two values  ***/
void time_increment(TIME *t, DATE *d)
{
  DATE d2;
  if (t->second != 59)      /*** count seconds  0...59  ***/
      ++t->second;
  else if (t->minute != 59)
    {
      t->second = 0;      t->minute++;
    }
  else if (t->hour != 23)
    {
      t->second = 0;      t->minute = 0;      t->hour++;
    }
  else
    {
      t->second = 0;      t->minute = 0;      t->hour = 0;
      date_increment(d);
    }
}

void date_increment(DATE *d)
{
  if (d->day != 31)      /*** assume all months have 31 days  ***/
    d->day++;
  else if (d->month != 12)      /*** count months  1...12  ***/
    {
      d->day = 1;      d->month++;
    }
  else
    {
      d->day = 1;      d->month = 1;      d->year++;
    }
}

/*** an alternative to date_increment, if it only returns one value ***/

DATE date_increment2(DATE d)         /* can also pass date one structure */
{
  if (d.day != 31)      /*** assume all months have 31 days  ***/
    ++d.day;
  else if (d.month != 12)      /*** count months  1...12  ***/
    {
      d.day = 1;
      d.month++;
    }
  else
    {
      d.month = 1;
      d.year++;
    }
```

```
  return d;
}
```

Output:

```
12:30:15
 1/ 7/93
12:30:16
 1/ 7/93
 1/ 8/93
 1/ 9/93
 1/ 8/93
```

For another example, see the complex variable functions.

## Linked Lists

A member of a structure can also be a pointer to the same structure type. This is useful with linked lists and trees.

```
#define  NodeMemory  (NodePtr) malloc(sizeof (struct node))

struct node
{
  int val;
  struct node *r_branch;
  struct node *l_branch;
} ;
typedef struct node * NodePtr;

main()
{
  NodePtr tree, branch;

  tree = (NodePtr) malloc(sizeof (struct node));
  tree->val = 10;
  tree->r_branch = NodeMemory;
  tree->l_branch = NodeMemory;

  tree->r_branch->val = 3;
  tree->l_branch->val = 40;
  printf("%d, %d, %d\n", tree->val, tree->r_branch->val, tree->l_branch->val);
}
```

### union

With union, different types of values can be stored in the same location at different times. Space is allocated to accomodate the largest member data type. They are syntactically identical to structures, Syntax:

```
union union-tag-opt
{
        member-declarations
}  union-names-opt;

union-name.member
ptr-to-union->member
```

```
/**  a simple example with unions  **/

union union_ifd     /** can store either an integer, float, or double value **/
```

```
{
  int ival;
  float fval;
  double dval;
} ;

main()
{
  union union_ifd  u1;

  u1.ival = 10;
  printf("%d\n", u1.ival);
  u1.fval = 10.34;
  printf("%f\n", u1.fval);
  u1.dval = 10.03454834;
  printf("%.10lf\n", u1.dval);
}
```

It is the programmer's reponsibility to know which variable type is being stored at a given time. The code

u1.ival=10;
printf("fn", u1.fval);

will produce an undefined result.

**enum**

The type enum lets one specify a limited set of integer values a variable can have. For example, flags are very common, and they can be either true or false.

Syntax:

```
enum enum-tag-opt {enum-tags} enum-variable-names-opt;

enum-name variable-name
```

The values of the enum variable are integers, but the program can be easier to read when using enum instead of integers. Other common enumerated types are weekdays and months.

The enumerated values can be set by the compiler or set explicitly. If the compiler sets the values, it starts at 0 and continues in order. If any value is set explicitly, then subsequent values are implicitly assigned.

```
enum flag_o_e {EVEN, ODD};
enum flag_o_e test1;
typedef enum flag_o_e FLAGS;

FLAGS if_even(int n);

main()
{
  int x;
  FLAGS test2;

  printf("input an integer: ");  scanf("%d", &x);
  test2 = if_even(x);
  if (test2 == EVEN)
    printf("test succeeded (%d is even)\n", x);
  else
    printf("test failed (%d is odd)\n", x);
}

FLAGS if_even(int n)
{
```

```
  if (n%2)
    return ODD;
  else
    return EVEN;
}
```

This is a more detailed program, showing an array of unions and ways to manipulate them, as well as enum.

```
/***  example with unions  ***/
#define   ASSIGN_U_NONE(x)        {x.utype = NONE;}
#define   ASSIGN_U_INT(x,val)     {x.utype = INT;     x.u.i = val;}
#define   ASSIGN_U_FLOAT(x,val)   {x.utype = FLOAT;   x.u.f = val;}
#define   ASSIGN_U_DOUBLE(x,val)  {x.utype = DOUBLE;  x.u.d = val;}

typedef union
{
  int i;
  float f;
  double d;
}  Arith_U;
typedef enum {NONE, INT, FLOAT, DOUBLE} Arith_E;
typedef struct
{
  Arith_E utype;
  Arith_U u;
}  Var_Storage;

main()
{
  int i;
  Var_Storage a[10];

  a->utype = INT;  a->u.i = 10;             /**  pointer to union operation **/
  a[1].utype = FLOAT;  a[1].u.f = 11.0;
  a[2].utype = DOUBLE;  a[2].u.d = 12.0;
  ASSIGN_U_NONE(a[3]);
  ASSIGN_U_INT(a[4], 20);
  ASSIGN_U_FLOAT(a[5], 21.0);
  ASSIGN_U_DOUBLE(a[6], 22.);

  for (i=0; i<7; i++)
    {
      if (print_Var(a[i]))
        printf("\n");
    }
}

int print_Var(Var_Storage x)
{
  switch (x.utype)
    {
    case INT:        printf("%d",x.u.i);       break;
    case FLOAT:      printf("%f",x.u.f);       break;
    case DOUBLE:     printf("%.8lf",x.u.d);    break;
    default:         return (0);               break;
    }
  return (1);
}
```

## Example: Complex Numbers

A complex number can be represented as

z = a+bi = r * e^(i*theta)
with
a = r * cos(theta)
b = r * sin(theta)
r = (a^2 + b^2)^(1/2)
theta = (tan(b/a))^(-1)

and

z = z1 + z2 = (a1 + a2) + (b1 + b2)i
z = z1 * z2 = (a1*a2 - b1*b2) + (a1*b2 + a2*b1)i
= r1*r2 * e^(theta1 + theta2)i

```
/***  Example using structures to represent complex numbers  ***/
#include
#include "prog4-06.h"

main()
{
  static COMPLEX z1 = {{1.0, 2.0}, {0.0, 0.0}};
  static COMPLEX z[MAX] = { {{1.0, 1.0}, {0.0, 0.0}},
                            {{2.0, -1.0}, {0.0, 0.0}} };

  rect_to_polar(&z1);
  rect_to_polar(z);
  rect_to_polar(z+1);
  complex_print(z1, BOTH);
  complex_print(*z, BOTH);
  complex_print(*(z+1), BOTH);
  z[2] = z1;

  z[3] = complex_add(z[0], z[1]);
  complex_print(z[3], BOTH);
/**  write complex_multiply() as an exercise:  **/
/**  *(z+4) = complex_multiply(*z, *(z+1));    **/
/**  complex_print(*(z+4), BOTH);              **/
}

void complex_print(COMPLEX z, C_FLAG flag)
{
  switch (flag)
    {
    case RECT:
      printf("z = %8.3f + %8.3f i\n", (z.r.a), (z.r.b));
      break;
    case POLAR:
      printf("z = "); PRINT_POLAR(z);
      break;
    case BOTH:
      PRINT_BOTH(z);
      break;
    }
}

void rect_to_polar(COMPLEX *z)
{
  double a = (z->r.a);
  double b = (z->r.b);
  z->p.r = sqrt(a*a + b*b);
  z->p.theta = atan2(b,a);
}

COMPLEX complex_add(COMPLEX z1, COMPLEX z2)
{
```

```
  COMPLEX sum;
  sum.r.a = (z1.r.a) + (z2.r.a);
  sum.r.b = (z1.r.b) + (z2.r.b);
  rect_to_polar(&sum);
  return (sum);
}
```

```
/**  File: prog4-06.h  **/
#define MAX 10

#define  PRINT_RECT(z)  printf("%8.3f + %8.3f i", (z.r.a), (z.r.b))
#define  PRINT_POLAR(z) printf("%8.3f * exp(%8.3f i)", (z.p.r), (z.p.theta))
#define  PRINT_BOTH(z)  { \
           printf("z = "); PRINT_RECT(z); \
           printf(" = "); PRINT_POLAR(z); printf("\n"); }

struct rect
  {
    double a, b;
  };

struct polar
  {
    double r, theta;
  };

struct complex
  {
    struct rect r;
    struct polar p;
  } ;
typedef struct complex COMPLEX;

enum c_flag {RECT, POLAR, BOTH};
typedef enum c_flag C_FLAG;

/*** function prototypes for rect_to_polar, complex_add, complex_print  ***/
void rect_to_polar(COMPLEX *z);
COMPLEX complex_add(COMPLEX z1, COMPLEX z2);
void complex_print(COMPLEX z, C_FLAG flag);

/*** function prototypes for polar_to_rect, complex_multiply, complex_input,
     to be written as exercises  ***/
void polar_to_rect(COMPLEX *z);
COMPLEX complex_multiply(COMPLEX z1, COMPLEX z2);
COMPLEX complex_input(void);
```

Output:

```
z =    1.000 +    2.000 i =    2.236 * exp(   1.107 i)
z =    1.000 +    1.000 i =    1.414 * exp(   0.785 i)
z =    2.000 +   -1.000 i =    2.236 * exp(  -0.464 i)
z =    3.000 +    0.000 i =    3.000 * exp(   0.000 i)
```

### Problems

12. Write the functions polar_to_rect, complex_multiply, and complex_input to be used with the answer to question 11.

---

# Chapter 5: C Libraries

The standard C libraries include functions for

- memory allocation
- math functions
- random variables
- input/output operations
- file manipulations
- string manipulations
- other functions (see K&R, Appendix~B)

## Memory Allocation

To have variable array sizes, dynamic memory allocation can be used.

```
#include

void *malloc(size_t size);
void *calloc(n, size_t size);
void free(void *p);
```

in ANSI C, size_t+ is the size of a character. The (sizeof) operator returns the size of an object in units of size_t. In addition, the type of the pointer returned by malloc and calloc has to be cast as needed. For example,

```
#include

main()
{
  int i, n;
  double *A, *a;

  scanf("%d", &n);
  A = (double *) malloc(n * sizeof (double));

  for (i=0; i
```

## Math Libraries

There are a variety of math functions available, all declared in /usr/include/math.h. Most take arguments of type double and return values of type double. For example,

```
#include

main()
{
  double x,y,z,theta;
  z = sqrt(x);
  z = sin(theta);          /***  theta is in radians  ***/
  z = asin(x);
  z = atan(x);
  z = atan2(y, x);         /***  atan(y/x)  ***/
  z = exp(x);              /***  e^x  ***/
  z = log(x);              /***  ln(x) [natural log]  ***/
  z = pow(x, y);           /***  x^y  ***/
}
```

```
#include

main()
{
  double x, y, theta;
```

```
  scanf("%lf", &x);
  printf("sqrt(%f) = %f\n", x, sqrt(x));
  printf("sin(0.6) = %f\n", sin(0.6));
  printf("atan(10) = %lf\n", atan(10.0));
  printf("atan(10/20) = %lf\n", atan2(10.0, 20.0));
  printf("exp(10) = %lf\n", exp(10.0));
  printf("log(10) = %lf\n", log(10.0));
  printf("log_10(10) = %lf\n", log10(10.0));
  printf("10^1.3 = %lf\n", pow(10.0, 1.3));
}
```

Output:

```
sqrt(10.000000) = 3.162278
sin(0.6) = 0.564642
atan(10) = 1.471128
atan(10/20) = 0.463648
exp(10) = 22026.465795
log(10) = 2.302585
log_10(10) = 1.000000
10^1.3 = 19.952623
```

When compiling, is must be specified that the math libraries are being used. For example, on Unix systems, use

cc -o .c -lm

## Random Variables

Using random variables is system dependent. The ANSI C functions are rand() and srand().

```
int rand(void);
void srand(unsigned int seed);
int RAND_MAX;
```

The function rand() will return a value between 0 and RAND_MAX, where RAND_MAX is defined in and is at least 32,767. The function srand() is used to seed the random number generator. Many systems have better random number generators, and C can usually access them, although this would then not be very portable. A good practice is to write a function or macro which returns a random number, and have this call the system-specific routines.

```
#include

#define  RANDOM_NUMBER_01          ((double) rand() / RAND_MAX)
#define  SEED_RANDOM_GEN(seed)     (srand(seed))

main()
{
  int i, n, seed;
  double *x;

  printf("number of random values: ");  scanf("%d", &n);
  x = (double *) malloc(n * sizeof(double));

  printf("input seed: ");  scanf("%d", &seed);
  SEED_RANDOM_GEN(seed);

  for (i=0; i
```

Output:

```
number of random values: 5
```

```
input seed: 10
0.13864904
0.86102660
0.34318625
0.27069316
0.51536290
```

## Input/Output

The conversions for printf() and scanf() are described in tables 3 and 4. In addition, I/O includes character output, sscanf(), and file manipulation.

```
#include

int i;
char c, s[], file_name[], access_mode[];
FILE *fp;

fp = fopen(file_name, access_mode);
fflush(fp);                          /***  flush the buffers  ***/
fclose(fp);

putchar(c);                               /***  one character  ***/
putc(c, fp);
puts(s);                                  /***  one line  ***/
fputs(s, fp);
printf(format, arg1, ...)                 /***  formatted  ***/
fprintf(fp, format, arg1, ...)
sprintf(s, format, arg1, ...);

c=getchar();
c=getc(fp);
gets(s);
fgets(s,i,fp);           ***  first i characters or till newline  ***/
scanf(format, &arg1, ...)                 /***  formatted  ***/
fscanf(fp, format, &arg1, ...);
sscanf(s, format, &arg1, ...);
```

```
/***  reads in n integers from a file,
      then prints the values to another file as type float  ***/
#include        /***  for file manipulation functions  ***/
#include       /***  for malloc()  ***/

main()
{
  int i, n, *x;
  char file_name[FILENAME_MAX];
  FILE *fp;

  printf("file name for input: "); scanf("%s", file_name);
  fp = fopen(file_name, "r");
  if (fp == NULL)
    {
      printf("error: could not open file %s\n", file_name);
      exit(-1);
    }

  fscanf(fp, "%d", &n);
  x = (int *) malloc(n * sizeof(int));

  for (i=0; i
```

## Strings

A variety of string functions are available.

```
#include

int i;
size_t n;
char *s, *s1, *s2, *to, *from;;
s = strcat(s1, s2);
s = strchr(s, char c);
i = strcmp(s1, s2);              /***  s1=s2 ? 0 : (s1>s2 ? 1 : -1)  ***/
s = strcpy(to, from);            /***  returns *to  ***/
n = strlen(s);
s = strstr(s1, s2);              /***  is s2 in s1?  ***/
s = strncat(s1, s2, int n);      /*** only use first n characters  ***/
i = strncmp(s1, s2, int n);
s = strncpy(to, from, int n);
```

```
#include
#include

main()
{
  char *s, *ct = "hello";
  s = (char *) malloc(100 * sizeof(char));

  strcpy(s, ct);
  printf("%s\n", s);

  if (strcmp(s, ct) == 0)
    printf("strings equal\n");
  else
    printf("\"%s\" and \"%s\" differ\n", s, ct);
}
```

```
#include
#include

main()
{
  int i, j;
  float x;
  char *s1 = "10 20 15.5", *s2;
  s2 = (char *) malloc(100 * sizeof(char));

  sscanf(s1, "%d %d %f", &i, &j, &x);
  printf("i = %d, j = %d, x = %f\n", i, j, x);

  sprintf(s2, "i = %d, j = %d, x = %f", i, j, x);
  printf("the string is \"%s\"\n", s2);
}
```

Output:

```
i = 10, j = 20, x = 15.500000
the string is "i = 10, j = 20, x = 15.500000"
```

# Appendix A: Make Program

The `make` function is used to simplify the complition process. It has three main features:

- target specifications with dependencies
- command lines to be executed
- assignment of strings to variables

The target is the file to be produced, either an executable file or an object file. The dependency list specifies the files on which the target depends, so if any of the dependency files has been modified since the target file was created, `make` will create a new target file. The command lines specify how the target is supposed to be make. Although this is primarily using the `cc` command, other commands can also be executed. The syntax is given below.

```
# comments
var = string value
$(var)          # uses variable value

target: dependencies
TAB     command-lines
```

The following commands run make:

```
% make -k
% make -k
```

When calling `make`, the -k argutment indicates that all specified files shluld be compiled, even if there is an error compiling one file. Otherwise, `make` will stop when there is a compile error in any file.

```
# a makefile for any simple, one file program, with no dependencies

CC = /bin/cc            # specifies compiler to be used

PROGS=p         # gets file name from shell

$(PROGS) : $(PROGS).c
            $(CC) -o $(PROGS) $(PROGS).c
```

Output:

```
% make program
/bin/cc -O program.c -o program
```

```
# a makefile for program large, Section 2

CC = /bin/cc                            # specifies compiler to be used

OBJS = large_prog.o large_sub.o         # object files to be used

large:  $(OBJS)                         # makes the executable file "large"
        $(CC) -o large $(OBJS)

$(OBJS): large.h                                # makes any needed / specified object files

clean:                                  #cleans up - removes object files
        rm $(OBJS)
```

Output:

```
% make -k large
/bin/cc -O -c large_prog.c
/bin/cc -O -c large_sub.c
```

```
/bin/cc -o large large_prog.o large_sub.o

% make -k
/bin/cc -O -c large_prog.c
/bin/cc -O -c large_sub.c
/bin/cc -o large large_prog.o large_sub.o

% make -k large_prog.o
/bin/cc -O -c large_prog.c

% make clean
rm large_prog.o large_sub.o
```

---

# Appendix B: C Style Guide

When writing C code, especially when other people will use or modify the code, it is useful to adhere to a consistent set of coding guidelines. The following is not the only acceptable style, but rather an example of a fairly conventional style guide.

## General Style

The first question is the placement of statements and braces. Consistency makes it easier to follow the flow of the program, since one can get used to looking for matching and optional braces in specific places.

```
/*---   avoid putting more than one statement on the same line   ---*/
  statement1; statement2;           /* BAD        */

  statement1;                       /* OK         */
  statement2;
```

```
int func(int arg)
{                           /* no indentation on the first set of braces     */
                            /*   around a procedure                          */

  statement;                /* each step of indentation is two spaces        */

  if (...)                  /* braces are indented and the matching pairs    */
    {                       /*   are lined up in the same column             */
      statement;
    }

  do                        /* 'do' braces should be lined up at the         */
    {                       /*   same column, with the 'while' on the same   */
      statement;            /*   line as the close parenthesis, to avoid     */
    } while (...);          /*   confusion with the 'while' statement        */

label:                      /* labels are lined up with the enclosing braces */
  statement;

  switch (...)
    {
    case xxx:               /* case labels are indented clearly mark the     */
      statement;            /*  places where a 'break' statement is          */
      /*fall thru*/         /*  expected but missing                         */
    case yyy:
      statement;
      break;
    default:
      statement;
```

```
      break;                /* should have a break statement even at the end */
    }                       /*  of the default case                          */

}                           /* all end braces line up with open braces       */
```

```
/*---   the statement following an 'if', 'while', 'for' or 'do'
        should always be on a separate line; this is especially true for the
        null statement   ---*/

  if (...) statement;               /* BAD        */
  if (...)                          /* OK         */
    statement;

  for (...) statement;              /* BAD        */
  for (...)                         /* OK         */
    statement;

  for (...);                        /* VERY BAD   */
  for (...)                         /* OK         */
    ;

  while (...) statement;            /* BAD        */
  while (...)                       /* OK         */
    statement;

  while (...);                      /* VERY BAD   */
  while (...)                       /* OK         */
    ;

  do statement; while (...);        /* VERY BAD   */
  do                                /* OK         */
    statement;
  while (...);
```

```
/*---   arrange nested 'if...else' statements in the way that is easiest to
        read and understand   ---*/

  if (...)                      /* OK, but confusing   */
    statement;
  else
    {
      if (...)
        statement;
      else
        {
          if (...)
            statement;
        }
    }

  if (...)                      /* BETTER              */
    statement;
  else if (...)
    statement;
  else if (...)
    statement;
```

```
/*---   the above rules can be violated if the overall legibility is
        improved as a result   ---*/

  switch (...)                          /* OK        */
    {
      case xxx:
        statement;
```

```
      break;
    case yyy:
      statement;
      break;
  }

switch (...)                              /* BETTER (by lining up the         */
  {                                       /*  statements, one can contrast    */
    case xxx: statement; break;           /*  the difference between the      */
    case yyy: statement; break;           /*  branches more easily)           */
  }


if (...)                                  /* OK        */
  {
    statement1;
    statement2;
  }
else
  {
    statement1;
    statement2;
  }

if (...) { statement1; statement2; }      /* BETTER (do this only for very    */
else     { statement1; statement2; }      /*  short statements!)              */


if (...)                                  /* OK        */
  statement;
else if (...)
  statement;
else if (...)
  statement;

if      (...) statement;                  /* BETTER    */
else if (...) statement;
else if (...) statement;
```

## Layout

- Avoid using the TAB character in a source file. Use spaces instead. This is because there is no standard governing how TAB's are expanded. Thus, the same file may look very differently from one editor to another.
- Avoid having any text past column 78.

## Coding Practice

- Always use ANSI-C style protocols for procedures
- Avoid using `char', `float' or any structures or unions as arguments to functions or values returned by functions if compatibility with ``Traditional C'' is needed
- Avoid relying on whether `char' is signed or unsigned; use `signed' or `unsigned' explicitly if necessary
- Avoid modifying literals. For example,

```
void proc(void)
{
  char *p = "a";
  ...
  *p = 'b';            /* VERY BAD */
}
```

- Whenever possible, use a single assignment statement to modify an entire structure or union. However, do not use this to initialize structures or unions. For example,

```
typedef struct
{
  int x, y;
} COORD;

void proc1(COORD *p)
{
  COORD c;
  ...
  c.x = p->x;                  /* OK, but verbose   */
  c.y = p->y;
  memcpy(&c, p, sizeof(c));    /* OK, but slow      */
  c = *p;                      /* BEST              */
}

void proc2(COORD *p)
{
  COORD c = *p;          /* BAD, since not all compilers support initializion */
  ...                    /*  of structures (i.e., not portable)               */
}
```

### Naming Conventions

These are some general naming conventions:

- Avoid using '_' as the first character of a name; these names are generally reserved for use by the compiler and libraries.
- Pre-processor symbols (#define's) should almost always be in upper case, with '_' as an optional separator (e.g., DELETE_CHAR).
- Type names can be either
  - all upper case, with '_' (e.g., RING_BUFFER), or
  - upper case followed by mixed case, sans '_' (e.g., RingBuffer ).
- Variables can be either
  - all lower case, with or without '_' (e.g.,old_value, oldvalue)
  - lower case followed by mixed case, sans '_' (e.g., oldValue).
- Procedure names can be either
  - all lower case, with or without '_' (e.g., get_first, getfirst)
  - upper case followed by mixed case, sans '_' (e.g., GetFirst)
- One can have each global name preceded by a one to three character prefix that indicates where the name is defined. E.g., RbRemoveElement}, RB_remove_element (indicating that these procedures are defined in the RB module). Some older compilers do not allow global names with '_' or more than 8 characters, but this restriction usually no longer applies.
- A local procedure/variable should have a name that is as descriptive as possible. For example,

```
static int MDinsls1(...)           /* too cryptic */
static int insert_list_head(...)   /* much better */
```

---

# Appendix C: Answers to Problems

```
/**  "A Crash Course in C", day 1, problem 2: print fahrenheit
 **  and celcius temperatures for 32-212 F in steps of 20 F **/

#define  FREEZE  32
#define  BOIL   212
```

```
#define  STEP    20

main()
{
  int f;
  for (f=FREEZE; f<=BOIL; f+=STEP)
    printf("F = %3d, C = %5.1f\n",f,(f-32.0)*5.0/9.0);
}
```

```
/**  "A Crash Course in C," problem 3:
 **    input a number and print all its factors  **/
#include

main()
{
  int i,n;

  printf("input a number: ");  scanf("%d",&n);
  for (i=2; i<=n; i++)
    if (!(n%i))
      printf("%6d is a factor\n", i);
}
```

```
/**  "A Crash Course in C," problem 4
 **    input a number and decide if it is prime  **/
#include
#include
#define TRUE   1
#define FALSE  0

main()
{
  int i,n, nmax, prime_flag=TRUE;

  printf("input a number: ");  scanf("%d",&n);
  nmax = (int) sqrt((double) n);

  for (i=2; i<=nmax; i++)
    if (!(n%i))
      prime_flag=FALSE;

  if (prime_flag)
    printf("%6d is prime\n", n);
  else
    printf("%6d is not prime\n", n);
}
```

```
/**  "A Crash Course in C," problem 5
 **    program to calculate x using the quadratic formula
 **/
#include
main()
{
  float a, b, c, d, x1, x2;
  printf("input a,b,c: ");
  scanf("%f %f %f",&a, &b, &c);
  d = b*b - 4.0*a*c;
  if (d >= 0)   /***  check if solution will be real or complex  ***/
    {
      x1 = (-b+sqrt(d)) / (2.0*a);
                    /***  need parentheses for proper order  ***/
      x2 = (-b-sqrt(d)) / (2.0*a);
                    /***  use sqrt() from the math library  ***/
      printf("x = %f, %f\n",x1,x2);
    }
  else
```

```
    {
      x1 = -b/(2.0*a);
      x2 = sqrt(-d)/(2.0*a);
      printf("x = %f + %fi, %f - %fi\n", x1, x2, x1, x2);
    }
}
```

```
/**  "A Crash Course in C," problem 6
 **    input an integer and print it in English  **/

main()
{
  int n, pow_ten;
  printf("input an integer: "); scanf("%d",&n);
  do {
    pow_ten=1;

    while (n / pow_ten)
      pow_ten *= 10;

    for (pow_ten/=10; pow_ten!=0; n%=pow_ten, pow_ten/=10)
      switch (n/pow_ten)
        {
        case 0:  printf("zero ");   break;
        case 1:  printf("one ");    break;
        case 2:  printf("two ");    break;
        case 3:  printf("three ");  break;
        case 4:  printf("four ");   break;
        case 5:  printf("five ");   break;
        case 6:  printf("six ");    break;
        case 7:  printf("seven ");  break;
        case 8:  printf("eight ");  break;
        case 9:  printf("nine ");   break;
        }
    printf("\ninput an integer: "); scanf("%d",&n);
  } while (n>0);
}
```

```
/**  "A Crash Course in C," problem 7
 **    count the number of characters and lines   **/
#include

main()
{
  char c;
  int characters, lines;

  while ((c=getchar()) != EOF) {
    characters++;
    if (c == '\n')
      lines++;
  }
  printf("%d characters, %d lines\n",characters, lines);
}
```

```
/**  "A Crash Course in C," problem 8
 **    float x_to_int_n(float x, int n) raise a number to an integer power  **/

float x_to_int_n(float x, int n);

main()
{
  int n;
  float x;
  printf("input x, n: ");
  scanf("%f %d", &x, &n);
```

```
  printf("%f^%2d = %f\n", x, n, x_to_int_n(x,n));
}

float x_to_int_n(float x, int n)
{
  float y=1.0;
  for ( ; n>0; n--)
    y *= x;
  return y;
}
```

```
/**  "A Crash Course in C," problems 9, 10
 **    int factorial(int n);        calculate factorial(n)
 **    int factorial_r(int n);      calculate factorial(n) recursively  **/

int factorial(int n);
int factorial_r(int n);

main()
{
  int n;
  printf("input n: ");
  scanf("%d", &n);
  printf("factorial(%d) = %d\n", n, factorial(n));
  printf("factorial(%d) = %d\n", n, factorial_r(n));
}

int factorial(int n)
{
  int fact=1;
  for ( ; n>1; n--)
    fact *= n;
  return fact;
}

int factorial_r(int n)
{
  if (n>1)
    return n*factorial_r(n-1);
  else
    return 1;
}
```

```
/** "A Crash Course in C," problems 11
 **     input a number and find all primes less than it
 **/

#include

#define TRUE 1                  /**   define flag values  **/
#define FALSE 0

int is_prime(int n);            /**   function prototype declarations  **/
int get_positive_int(void);

main()
{
    int n, i;
    while (n=get_positive_int())
        for (i=2; i<=n; i++)
        if (is_prime(i))
             printf("%d is prime\n", i);
}

int is_prime(int n)
{
```

```
    int i, max;
    max = (int) sqrt((double> n);

    for (i=2; i<=max; i++)
    if (!(n%i))
         return FALSE;
    return TRUE;
}

int get_positive_int(void)
{
    int n;
    do
         {
        printf("input a positive number, 0 to quit: ");
        scanf("%d", &n);
         }   while (n < 0);
     return n;
}
```

```
/**  "A Crash Course in C," problems 11
 **    matrix functions:
 **      input_matrix(), print_matrix, add_matrix, multiply_matrix  **/

#define MAX   5

void print_matrix(float A[][MAX], int n);
void input_matrix(float A[MAX][MAX], int n);
void add_matrix(float A[][MAX], float B[][MAX], float C[][MAX], int n)
void multiply_matrix(float A[][MAX], float B[][MAX], float C[][MAX], int n)

main()
{
  int n;
  float A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];

  printf("input size of matrix: "); scanf("%d",&n);
  if (n<MAX) {
    input_matrix(A,n);
    input_matrix(B,n);
    print_matrix(A,n);
  }
  else
    printf("size of matrix is too large\n");
}

void print_matrix(float A[][MAX], int n)
{
  int i,j;
  for (i=0; i<n; i++) {
    for (j=0; j<n; j++)
      printf("%f\t",A[i][j]);
    printf("\n");
  }
}

void input_matrix(float A[MAX][MAX], int n)
{
  int i,j;
  float *a;
  for (i=0; i<n; i++) {
    for (j=0, a=A[i]; j<n; j++)
      scanf("%f",a++);
  }
}

void add_matrix(float A[][MAX], float B[][MAX], float C[][MAX], int n)
```

```
{
  int i,j;
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      C[i][j] = A[i][j] + B[i][j];
}

void multiply_matrix(float A[][MAX], float B[][MAX], float C[][MAX], int n)
{
  int i, j, k;

  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      for (k=0, C[i][j]=0.0; k<n; k++)
        C[i][j] += A[i][k] * B[k][j];
}
```

```
/**  "A Crash Course in C," problems 13
 **    functions complex_input(), polar_to_rect(), complex_multiply()
 **/

COMPLEX complex_input(void)
{
  COMPLEX z;
  int ans;
  double x,y;
  printf("input (0,a,b) or (1,r,theta): ");
  scanf("%d %lf %lf",&ans, &x, &y);
  if (ans == 1) {
    z.p.r = x;  z.p.theta = y;
    polar_to_rect(&z);
  }
  else if (ans == 0) {
    z.r.a = x;  z.r.b = y;
    rect_to_polar(&z);
  }
  else {
    printf("invalid coordinate system\n");
    z.r.a = 0.0; z.r.b = 0.0; z.p.r = 0.0; z.p.theta = 0.0;
  }
  return z;
}

void polar_to_rect(COMPLEX *z)
{
  double r = (z->p.r);
  double theta = (z->p.theta);
  z->r.a = r * cos(theta);
  z->r.b = r * sin(theta);
}

COMPLEX complex_multiply(COMPLEX z1, COMPLEX z2)
{
  COMPLEX prod;
  prod.p.r = (z1.p.r) * (z2.p.r);
  prod.p.theta = (z1.p.theta) + (z2.p.theta);
  polar_to_rect(&prod);
  return (prod);
}
```