C Programming

MISC 2456

INDEX

RULES

braces { }	determine the beginning and end of a function body
called function	In the parenthesis of the function header line (where it appears at the end of the program) each variable is declared with its variable type. The default is integer. i.e. `int max_int(float x, float y)` p209 The passed

variables need not have the same name as their counterparts in the main function. A function header never ends with a semicolon.

case	C is case-sensitive
comments	comments may not be nested
declarations	Declaration statements may go before function main() for global variables, within main() where they apply only to that function, or within called functions.

`char ch = 'a';`	A character variable `ch` is declared and assigned an initial value of `a`. A character variable holds only one character unless it is an array as below.
`char test[5] = "abcd";`	Leave room for the end of string marker `/0`. This array cannot be modified using assignment statements but can be modified using `strcpy()`.. p345
`char *test = "abcd";`	This array CAN be modified using assignments and can hold a greater number of characters than it receives on declaration. `strcpy()` can be used provided that it does not exceed the number of places occupied by the existing string. p345
`FILE *my_file;`	Declares a pointer to a file (which will be opened later). p427
`int distance;` `int distance = 17;`	Declares `distance` as an integer variable. In the second example, the variable is declared as well as initialized with a value. It is a good practice to do this.
`long bignum;`	Declare `bignum` as a long integer.

function	
function name	cannot be a keyword (p.15), conforms to *identifier* rules, always followed by parenthesis `()`, should be mnemonic, traditionally in lowercase but not required.
identifiers	composed of up to 31 letters, digits, and underscores, beginning with a letter or underscore, no blank spaces
main()	each program must have one and only one main function
semicolon ;	follows each statement
strings	are enclosed in double quotes `""`
arithmetic operations	If both operands are integers, the result is an integer. If one or more operands is a floating point or double precision value, the result is a double precision value. This will be on the test. When dividing two integers, the fractional result is dropped, i.e. `9/5 = 1`.
variable	must begin with a letter or underscore, may contain only letters, underscores, or digits, no blanks, commas or special symbols, maximum length 31 characters. Additionally, the instructor prefers they not begin with an underscore and not be more than about 15 characters in length.

KEYWORDS

`auto`	
`break`	instructs the program to exit the current loop
`case`	
`char`	represents the character data type typically using 1 byte of storage for values from -128 to +127, may be used in a declaration statement.
`const`	
`continue`	
`default`	
`do`	
`double`	represents the double precision floating point data type typically using 8 bytes of storage for values up to 1.797693e+308.
`else`	
`enum`	a specifier which creates an enumerated data type, which is a user-defined list of values that is given its own data type name, p.440. The statement consists of the specifier followed by an optional name for the data type and a listing of acceptable values for the data type, i.e. `enum time {am, pm};`
`extern`	
`float`	represents the floating point data type typically using 4 bytes of storage for values up to 3.37e+38, may be used in a declaration statement.
`for`	
`goto`	
`if`	
`int`	represents the integer data type typically using 2 bytes of storage for values up to 32,767, may be used in a declaration statement
`long`	an integer type typically using 4 bytes of storage for values up to 2,147,483,647, may be used in a declaration statement. May also be combined with `unsigned`.
`register`	
`return`	
`short`	an integer type typically using 2 bytes of storage for values up to 32,767, may be used in a declaration statement.
`signed`	
`sizeof()`	an operator that returns the number of bytes of the object or data type included in the parentheses, i.e. `sizeof(num1)  sizeof(long int)`
`static`	
`struct`	
`switch`	
`typedef`	
`union`	

`unsigned` an integer type typically using 2 bytes of storage for positive only integers 0 to 65,535. used in a declaration statement. May also be combined with `long`.

`void` There is no value. When placed before the function name, it means no value will be returned; when placed within the parenthesis it means no value will be given to the function.

`volatile`

`while`

FUNCTIONS

The type of data to be returned by the function is given first, i.e. `int max_int()` . The default type is integer. `void` means no value is returned. Parameters and input data type go inside the parenthesis, i.e. `max_int(float x, float y)` or use `void` if there are no parameters.

`exit()` terminates the program and flushes output file buffers, closes open files, deletes temporary files. The parentheses contain a status value returned to the calling process, a 0 means a normal exit, other numbers indicate that an error occurred.

`fclose()` closes a file. This function breaks the link between the file's external and internal names, releasing the internal file pointer name, which can then be used for another file. p413 Example:

```
fclose(data);
```

The argument should always be a pointer; quotes are not used because `data` is a pointer and not a string.

`fflush(stdin);` clears the input butter. Use this line before reading character data with the scanf() function.

`fgets()` Read n-1 characters from the file and store the characters in the string name. Requires <stdio.h> Example:

```
fgets(stringname, n, filename);
```

`stringname` is the address of a character array. Ordinarily `n` will be the same number that is specified in the variable declaration, which must also take into account the end of string marker `\0`. The function reads characters until stringname is filled or an end of line character \n is encountered. Although this character is not supposed to end up in the string, it seemed to happen to be in program 4. A similar function `fgetc(filename)` reads a single character from a file. p416

`fopen()` opens a file. In the example

```
data = fopen("prog4.dat", "r");
```

`data` is the pointer to the external file, `prog4.dat` is the filename and `r` means to read the file. If the file does not exist, `NULL` is returned. Other arguments are `w` for writing to a new file, `a` for append, `r+` for reading and writing, `w+` for erasing an existing file and opening a blank file for reading and writing, and `a+` for reading, writing, and appending to a file. p408

`fscanf()` reads data from a file. `Example:

```
fscanf(MyFile, "%f", &Var);
```

where `MyFile` is the file to be read from `%f` is the data type and `&Var` is the address of the variable in which it is to be stored. fscanf() stops reading when it encounters whitespace, a newline character, or a data type mismatch. Multiple arguments may be specified. p416 It has a great deal of additional functionality

and can be used to simply move forward in a file. see Nancy's book.

`fseek()` `fseek(filename, 1L, SEEK_CUR);`

Move ahead 1 character. The "L" is a cast conversion of "1" to long integer. p425

`printf()` sends data to the primary display device (the screen). Example:

```
printf("The numbers are %d and %d.\n", int1, int2);
```

Arguments for any format specifiers appear in the same order called at the end of the statement.

`fprintf()` formats data and sends it to the printer. Example:

```
fprintf(FilePtr,"\tHello World\n");
```

`FilePtr` is the pointer to a temporary printer file, `\t` is tab, and `\n` is newline.

`gets()` reads a string entered at the keyboard until encountering a carriage return (new line character), then terminates the string with an end of string `\0` character, discarding the new line `\n` character. p330 Actually in program 4 I had to remove the end of line character from a string obtained using the `fgets()` function. For example:

```
gets(Var);
```

will put the string entered at the keyboard into the character array `Var`.

`if()`

```
if(statement)
    {
    assignment statement or function;
    assignment statement or function;
    }
    else
        {
        do this instead;
        this too;
        }
```

If "statement" is true then the following command(s) will be executed. The braces are only required if there are multiple commands to execute. The `else` commands (which are optional) are only executed if "statement" is false.

`main()` each program must have one and only one main function, tells the compiler where program execution is to begin, calls program modules and determines the sequence of events

`printf()` formats data and sends it to the standard system display device, such as the screen. Example:

```
printf("The total of 6.0 and 15.0 is %4.1f.", 6.0 +
15.0);
```

This results in the display of "The total of 6.0 and 15.0 is 21.0" without the quotes. The statement contains two *arguments* separated by commas. The "`%4.1f`" is a *conversion control sequence* or *format specifier*, more specifically a *control string* or *control specifier*. This tells the computer to insert the result of the next argument here and gives data type and field width information as well. Multiple format specifiers may appear and are associated with multiple arguments in the order presented.

`rewind()` move to the start of the data file. The only argument is the pointer to the data file, i.e. `rewind(in_file);` p425

`scanf()` retrieves data from the keyboard, for example `scanf("%f", &num1);` this statement stops the program and waits for keyboard input. The user types in a number and hits enter. the `scanf()` function retrieves this value and stores it in variable `num1` as a floating point decimal. The & symbol in front of the variable

name `num1` indicates the address of `num1` and is required in the `scanf()` function except when reading a string into a character array. In this case, the array name (without brackets) is the pointer name so no & (ampersand) is required. The function will retrieve characters until it encounters a space or newline. (p330).

ESCAPE SEQUENCES

`\'`	single quote
`\"`	double quote
`\\`	backslash character
`\b`	backspace
`\f`	next page
`\n`	start a new line
`\nnn`	treat nnn as an octal number
`\r`	carriage return
`\t`	move to next tab setting
`\0`	null character marking the end of a string

OPERATORS

OPERATOR	DEFINITION	COMMENTS
`%`	modulus	the remainder after division
`&`	logical bit-by-bit AND	
`|`	logical bit-by-bit OR	
`~`	logical bit-by-bit negation	
`<<`	shift left	
`>>`	shift right	
`++`	increment by one	e.g.: `a ++;` means `a=a+1;`
`--`	decrement by one	e.g.: `a --;` means `a=a-1;`
`+=`	increment by __	e.g.: `a += 2;` means `a=a+2;`
`-=`	decrement by __	e.g.: `a -= 2;` means `a=a-2;`
`*=`	multiply by __	e.g.: `a *= 2;` means `a=a*2;`
`/=`	divide by __	e.g.: `a /= 2;` means `a=a/2;`
`|=`	OR with and update	e.g.: `a |= 2;` means `a=a|2;`
`&=`	AND with and update	e.g.: `a &= 2;` means `a=a&2;`
`<<=`	shift left __ times	e.g.: `a <<= 2;` means `a=a<<2;`
`>>=`	shift right __ times	e.g.: `a >>= 2;` means `a=a>>2;`
`&&`	AND, conditional	
`||`	OR, conditional	

`!`	NOT, conditional
`==`	equal to, conditional
`!=`	not equal to, conditional

OTHER COMMANDS

`/*`	start of a comment, ends with `*/`
`#`	signals an instruction to the preprocessor
`#define`	a preprocessor statement to equate the symbolic constant in the statement with the information or data following it, i.e. `#define SALESTAX 0.05` means give the constant `SALESTAX` the value of 0.05. Wherever `SALESTAX` appears in the program, the value of 0.05 will automatically be substituted. Define statements are not followed by a semicolon and they may be found in include files.
`#include<stdio.h>` `#include<stdlib.h>`	preprocessor statements to include header files. These two are common to most programs.

GLOSSARY

address	the value identifying a memory location or the location of the first byte of memory in a variable. The symbol `&` is the address operator and returns the address of a variable when placed in front of the variable name, i.e. `adr = &num1;`
algorithm	step-by-step description of how to perform a computation
argument	data passed to a function by placing within the parenthesis. Multiple arguments are separated by commas.
assignment operator, p84	`=` assign the value on the right to the variable on the left
	`+=` add the value on the right to the value on the left and store in the variable on the left
	`-=` subtract the value on the right from the value on the left and store in the variable on the left
	`*=` multiply the value on the right by the value on the left and store in the variable on the left
	`/=` divide the value on the left by the value on the right and store in the variable on the left
	`%=` multiply the value on the left by the percentage on the right and store in the variable on the left
	see also increment operator, decrement operator
assignment statement	tells the computer to store a value into a variable, i.e. `num1 = 62;` or `result = num1 + num2;`
atomic data value	a value that is considered a complete entity by itself and is not decomposable into a smaller data type that is supported by the language. For example, although an integer can be decomposed into individual digits, C does not have a numerical digit data type so an integer is an *atomic data type*.
binary operator	requires two operands, i.e. multiplication, division, addition, subtraction, remainder
bit	the smallest storage unit of a computer, storing a 0 or a 1
byte	a group of bits. This usually consists of 8 bits, resulting in 256 possible combinations.
character code	the patterns of 0s and 1s used to represent letters, single digits, and other single characters, i.e. the ASCII code.
character type	letters of the alphabet, digits, and special symbols. A *single character constant* is any one letter, digit, or special symbol enclosed by single quotes like '!' or 'A'.
coding	converting an algorithm into a computer program
compiled language	a programming language in which all commands are translated before any are executed
conversion character	the last character(s) in a *conversion control sequence* or *format specifier*. Conversion characters are: `d` integer, `ld` long integer, `u` unsigned integer, `f` floating point, `lf` double precision, `o` octal, `x` hexadecimal, `e` exponent, `g` exponent or float—whichever is shorter, `c` character, `p` address, `s` string
conversion control	always begins with a % symbol and ends with a conversion character, i.e. `%d`

sequence or format specifier	means its an integer. Additional formatting characters can be placed between the % symbol and the conversion character, i.e. `%7.2f`
counting statement	`COUNT = COUNT + 1` see increment operator
data types	The 4 basic data types are integer, floating point, double precision, and character.
data types, integer	long integer, short integer, and unsigned integer. Long integer allows the value to surpass the maximum 32,767 limit of a 2-byte integer. Short integer may or may not conserve memory space. Unsigned integers are positive integers only which allows values of 0 to 65,536 in a 2-byte memory area. Declaration statements are: `long int var1; short int var2; unsigned int var3`. The word "`int`" may not be required in the statement. The actual size of these integers varies with the computer but may be determined with the `sizeof()` command. Short int may be the same size as int. When the date is converted to an integer number representing the number of days since the turn of the century, a regular integer does not work for dates past 1987.
declaration statement	appears immediately after the opening brace of a function and ends with a semicolon. used to name and define the data type that can be stored in each variable, i.e. `int total; float firstnum; double secnum; char ch;` Multiple variables of the same type can be assigned in one declaration statement, i.e. `char ch1, ch2, ch3;` The space after each comma is not required. A declaration statement can also be used to store an initial value into the variable, i.e. `int num1 = 15; char ch1 = 'a';` A declaration statement for pointer to an integer could look like: `int *g_ptr;`
decrement operator	`--COUNT` means the same as `COUNT = COUNT - 1`
definition statement	a *declaration statement* that defines how much memory is needed for data storage, i.e. `int total; float firstnum; double secnum; char ch;` are all definition statements as well as being declaration statements
double precision	negative or positive numbers having a decimal point. Has greater storage allocation than floating point, to 1.797693e+308 using 8 bytes of memory.
driver function	tells the other functions the sequence in which they are to operate, describes `main()`
escape sequence	in C language, a backslash followed by a character. The backslash is the *escape* which means to escape from the normal interpretation of the character which follows
field width specifier	Defines the width of the field, i.e. `10.3` means that a floating point number will be displayed with a total of 10 digits including spaces and decimal point and will have 3 places following the decimal.
floating point	negative or positive numbers having a decimal point. Has smaller storage allocation than double precision, to 3.383+38 using 4 bytes of memory.
format modifier	may be used in a *conversion control sequence* or *format specifier* immediately after the % symbol, i.e. `%-+10d` means left-justify (-) the display and include a + symbol if the value is positive (+). (By default, the - symbol is displayed for negative numbers anyway.) The # format modifier forces octal and hexadecimal numbers to be printed with a leading `0` and `0x` respectively.
formula	an algorithm written in mathematical equations
function header line	the first line of a function, tells 1) what type of data, if any, is returned from the function, 2) the name of the function, 3) what type of data, if any, is sent into

	the function
global variable	a variable declared before the main function, allowing it to be used in any function in the program.
header file	a file containing information to be placed at the top of a program using the `#include` command
identifier	a combination of letters, digits and underscores used as function names, variables or to name other elements of the C language

increment operator	`++COUNT;` means the same as `COUNT = COUNT + 1;`
	`k = ++n;` increment `n` by `1` and assign the value to `k`
	`k = n++;` assign `n` to `k` and then increment `n` by `1`

indirection operator	The indirection operator `*` when placed in front of a variable name in the declaration statement indicates that it is a pointer variable, see *pointer variable*.
initialize	assign a value to a variable for the first time.
integer value	Also called *integer constant* in C, is any positive or negative number without a decimal point. The maximum size of an integer varies by computer and depends on the storage area allotted. 1 byte: -128 to 127, 2 bytes: -32768 to 32767, 4 bytes: -2147483648 to 2147483647. Use the `sizeof` operator to determine the number of bytes allocated for each integer value.
interpreted language	a programming language in which each statement in the source program is translated individually and executed immediately
interpreter	the program which translates a source program into machine code
literal data	any data in a program that explicitly identifies itself, such as constants 2 and 3.1416.
machine language	consists of 1s and 0s
magic number	a literal value that appears many times in a program
mnemonic	designed as a memory aid
modular program	a program whose structure consists of interrelated segments arranged in a logical and easily understandable order
module	a subprogram within a main program which carries out usually one or two functions
number code	the patterns of 0s and 1s used to represent numbers, i.e. two's complement for example.
object program	the machine language version of a source program
offset	a number or variable referring to the number of addresses beyond the starting address of an array that a particular address is found. For example in the expression `*(g_ptr + 3)` the number 3 is the offset and the expression points to the third address past the initial array element. p307,8
pointer variable	a variable used to store the address of another variable, i.e. `chr_point = &ch` A pointer variable is declared according to the type of the variable to which it points, i.e. `char *chr_point;` The *indirection operator* `*` denotes that `chr_point` is a pointer variable.
preprocessor command	performs some action before the compiler translates the source program into machine code, begins with the # sign
pseudocode	an algorithm written in plain English

random access	Any character can be read without first reading everything before it.
remainder	an arithmetic operation using the operator "%", i.e. `9 % 4 = 1`.
source program	the computer program before compiling
structure	1) the program's overall construction 2) the form used to carry out individual tasks within a program
subscript notation	If `num_ptr` is declared as a pointer variable, the expression `*(num_ptr + I)` can also be written in *subscript notation* as `num_ptr[I]`.
symbolic name, symbolic constant, named constant	an identifier (all caps by convention) that is assigned a permanent value or meaning using a #define statement, i.e. `#define SALESTAX 0.05` No semicolon follows in this example because the preprocessor would substitute `0.05;` wherever `SALESTAX` was found if that were the case.
unary operator	requires only one operand, i.e. make negative "-"
word	one or more bytes grouped, i.e. the IBM computer has two bytes grouped into one 16-bit word having one address. This has advantages of grouping at the expense of cost and complexity.

MATH FUNCTIONS

#include <math.h>

`int abs (int n)` - Get absolute value of an integer.
`double acos(double x)` - Compute arc cosine of x.
`double asin(double x)` - Compute arc sine of x.
`double atan(double x)` - Compute arc tangent of x.
`double atan2(double y, double x)` - Compute arc tangent of y/x.
`double ceil(double x)` - Get smallest integral value that exceeds x.
`double cos(double x)` - Compute cosine of angle in radians.
`double cosh(double x)` - Compute the hyperbolic cosine of x.
`div_t div(int number, int denom)` - Divide one integer by another.
`double exp(double x)` - Compute exponential of x.
`double fabs(double x)` - Compute absolute value of x.
`double floor(double x)` - Get largest integral value less than x.
`double fmod(double x, double y)` - Divide x by y with integral quotient and return remainder.
`double frexp(double x, int *expptr)` - Breaks down x into mantissa and exponent of no.
`labs(long n)` - Find absolute v alue of long integer n.
`double ldexp(double x, int exp)` - Reconstructs x out of mantissa and exponent of two.
`ldiv_t ldiv(long number, long denom)` - Divide one long integer by another.
`double log(double x)` - Compute log(x).
`double log10(double x)` - Compute log to the base 10 of x.
`double modf(double x, double *intptr)` - Breaks x into fractional and integer parts.
`double pow(double x, double y)` - Compute x raised to the power y.
`int rand(void)` - Get a random integer between 0 and 32.
`int random(int max_num)` - Get a random integer between 0 and max_num.
`void randomize(void)` - Set a random seed for the random number generator.
`double sin(double x)` - Compute sine of angle in radians.
`double sinh(double x)` - Compute the hyperbolic sine of x.
`double sqrt(double x)` - Compute the square root of x.
`void srand(unsigned seed)` - Set a new seed for the random number generator (rand).
`double tan(double x)` - Compute tangent of angle in radians.
`double tanh(double x)` - Compute the hyperbolic tangent of x.

SAMPLE CODE

for Statement.

```
int ii;                                   /* a counter           */

for(ii=0;ii<10;ii++)                      /* Do it 10 times      */
{                                         /* { for more than one */
     <statement>;                         /* some statement      */
     <statement>;                         /* some statement      */
}
```

while Statement.

```
int ii = 0;                               /* a counter           */

while(ii<10)                              /* While this is true  */
{                                         /* { for more than one */
     <statement>;                         /* some statement      */
     <statement>;                         /* some statement      */
     ii++;                                /* increment ii        */
}
```

switch Statement.

```
switch(var)                               /* some expression     */
{                                         /*                     */
     case 1:                              /* if var = 1          */
          <statement>;                    /* execute statements  */
          <statement>;                    /* if var = 1          */
          break;                          /* exit switch stmnt   */
     case 2:                              /* if var = 2          */
          <statement>;                    /* execute statements  */
          <statement>;                    /* if var = 2          */
          break;                          /* exit switch stmnt   */
     default:                             /* otherwise           */
          <statement>;                    /* execute statements  */
}
```

The `default` statement is optional. If `break` instructions are not used, program execution will continue into subsequent case areas even if they do not test true.

ifdef Statement.

```
/* Uncomment the appropriate statement to make quick changes in
 * code definitions                                                */

#define method1                                 /* one way          */
//#define method2                               /* another way      */

ifdef method1                                   /* If method1 is    */
extern int SomeThing;                           /* uncommented above*/
#define THIS   10                               /*                  */
#define THAT   20                               /*                  */
#endif

ifdef method2                                   /* If method2 is    */
extern int AnyThing;                            /* uncommented above*/
#define THIS   5                                /*                  */
#define THAT   7                                /*                  */
#endif
```

Basic Printing.

```
FILE *Print;            /* declare the pointer to a file where
                           data will be sent for printing.       */

if((Print=fopen("lpt1","w"))==NULL)              /* These 5 lines    */
     {                                           /* are used in all  */
     printf("\nPRINTER IS NOT READY!!!");        /* programs to open */
     exit(0);                                    /* and check the    */
     }                                           /* printer output.  */

fprintf(Print,"\t     THOMAS PENICK\n");         /* print something  */

fprintf(Print,"\f");     /* advance the printer to the next page    */
fclose(Print);           /* close the print file                    */
```

Access a data file.

```
FILE *Data;             /* declare the pointer to the data file    */

Data = fopen("prog4.dat", "r");          /* open the data file      */
if (Data==NULL)                          /* check the data file     */
     {
     fprintf(Print,"\nThe file prog4.dat cannot be opened.\n");
                                         /* error message           */
     exit(1);           /* stop the program due to an error (1)    */
     }

fclose(Data);                            /* close the data file     */
```

Read and print from a data file.

```
while(fscanf(Data,"%ld",&ID_Num)!=EOF)  /* until the end of the file */
                                        /* also read an integer.     */
                                        /* "Data" points to the file */
     {
     fgets(Name1,25,Data);              /* read string into "Name1"  */
     strcopy(Name2, Name1);             /* call a function to remove
                                        /* the end of line char.     */
     fprintf(Print,"\t%d.%-25s\t%ld\n",Line_Num,Name2,ID_Num);
                                        /* print a line of data      */
     }
```

Programmer-defined called function to remove the end of line character from a string (character array). This is similar to the example on p332.

```
void strcopy(char [], char []);         /* declare the subroutine    */
                                        /* before main function      */

void strcopy(char String2[], char String1[]) /* begin called funct.  */
{
int Cnt = 0
while (String1[Cnt] != '\n')
     {
     String2[Cnt] = String1[Cnt];
     ++Cnt;                             /* increment the counter     */
     }
String2[Cnt] = '\0';                    /* terminate the string      */
return;
}
```

Load a pointer with the address of the first element of an array.

```
int nums[100];                          /* create an array           */
int *nptr;                              /* create a pointer          */
nptr = &nums[0];                        /* load the address          */
nptr = nums;                            /* another way, same as above*/
```

Types of Functions in C Programming

The following examples illustrate various methods of passing values to functions. Except for the function "strcopy()", these are not working functions (code has been omitted).

Subtopics

A FUNCTION WHICH PASSES NO VALUE AND RETURNS NO VALUE

A function may be declared (function prototype) globally or within the calling function:

FUNCTION PROTOTYPE
```
void PrintHead(void);
```
FUNCTION CALL
```
PrintHead();
```
FUNCTION HEADER
```
void PrintHead(void)
{
```
RETURN STATEMENT
```
return;
}
```

A FUNCTION WHICH PASSES TWO FLOATS AND RETURNS A FLOAT

A function can *return* at most one value:

FUNCTION PROTOTYPE
```
float find_max(float, float);
```
FUNCTION CALL
```
maxmum = find_max(firstnum,secnum);
```

The variables used in the function need not and should not have the same names as those passed to the function:

FUNCTION HEADER
```
float find_max(float num1, float num2)
{
```
DECLARE A VARIABLE
```
float Result;
```
RETURN STATEMENT
```
return(Result);
}
```

A FUNCTION WHICH PASSES AN INTEGER ARRAY AND RETURNS AN INTEGER

An alternate method would be to pass by reference using a pointer. In this example the last argument is an integer telling the function how many elements are in the array:

FUNCTION PROTOTYPE
```
int find_max(int vals[], int);
```
FUNCTION CALL
```
biggun = find_max(nums,5);
```

The variables used in the function need not and should not have the same names as those passed to the function:

FUNCTION HEADER
```
int find_max(int nums[], int HowMany)
{
```
DECLARE A VARIABLE
```
int Result;
```
RETURN STATEMENT
```
return(Result);
}
```

A FUNCTION WHICH PASSES VARIABLES BY REFERENCE USING ADDRESSES

FUNCTION PROTOTYPE
```
void sortnum(double*, double*);
```
FUNCTION CALL
```
sortnum(&FirstNum, &SecNum);
```
FUNCTION HEADER
```
void sortnum(double *Num1, double *Num2)
{
```
RETURN STATEMENT
```
return;
}
```

A FUNCTION WHICH PASSES A STRING BY REFERENCE

There is no way that I can find of returning a string from a function. However, if the address of the string is passed, then the function can operate on the string. This example is a working function which takes the string referenced by the second argument, removes the carriage return from the end of it and "returns" it by assignment to the first argument. (This is used for a string which has been retrieved from a text file using the `fgets()` function):

FUNCTION PROTOTYPE

```
void strcopy(char [], char []);
```

The calling function must have declared two appropriate character arrays.

```
char Name1[25];
char Name2{25];
```

FUNCTION CALL

```
strcopy(Name2,Name1);
```

FUNCTION HEADER

```
void strcopy(char Str2[], char Str1[])
{
```

DECLARE A VARIABLE

```
int Cnt = 0;
while (Str1[Cnt] != '\n')
{
     Str2[Cnt] = Str1[Cnt];
     ++Cnt;
}
```

Nothing is returned, but "Str2" is the new version of the original "Name1" and is available in the calling function as "Name2".

RETURN STATEMENT

```
return;
}
```

A FUNCTION WHICH PASSES A STRUCTURE BY NAME

Here "class_list" is a structure type declared globally:

STRUCTURE DECLARATION

```
struct class_list
{
     char Name[31];
     long ID_Num;
     char Class[9]
};
```

The function prototype may be declared globally or within the calling function. Here "class_list" is the type of structure from the structure prototype (declared globally), not the specific structure itself:

FUNCTION PROTOTYPE

```
void PrintReport(struct class_list);
```

A single structure of type "class_list" is created in the calling function (if not globally) and named "load":

STRUCTURE IS CREATED

```
struct class_list load;
```

The structure "load" is passed to the function:

FUNCTION CALL

```
PrintReport(load);
```

The structure prototype name is again used in the function header:

FUNCTION HEADER

```
void PrintReport(struct class_list N)
{
```

REFERENCES TO ELEMENTS

```
N.Name
N.ID_Num
N.Class
```

RETURN STATEMENT

```
return;
}
```

A FUNCTION WHICH PASSES A STRUCTURE BY REFERENCE USING A POINTER

Here "class_list" is a structure type declared globally as before:

STRUCTURE DECLARATION

```
struct class_list
{
     char Name[31];
     long ID_Num;
     char Class[9]
};
```

The function prototype may be declared globally or within the calling function. Here "class_list" is the type of structure from the structure prototype (declared globally), not the specific structure itself. The * indicates that a pointer to the structure will be passed:

FUNCTION PROTOTYPE

```
void PrintReport(struct class_list *);
```

A single structure of type "class_list" is created in the calling function (if not globally) and named "load":

STRUCTURE IS CREATED

```
struct class_list load;
```

The structure is assigned to a pointer.

A POINTER IS DECLARED

```
struct class_list *Ptr;
```

A POINTER IS ASSIGNED

```
Ptr = &load;
```

The pointer to the structure is passed to the function.

FUNCTION CALL

```
PrintReport(Ptr);
```

A corresponding pointer "P" is declared in the function header:

FUNCTION HEADER

```
void PrintReport(struct class_list *P)
{
```

REFERENCES TO ELEMENTS

```
P->Name
P->ID_Num
P->Class
```

RETURN STATEMENT

```
return;
}
```

A FUNCTION WHICH PASSES A STRUCTURE ARRAY

Here "c_list" is a structure type declared globally as before:

STRUCTURE DECLARATION

```
struct c_list
{
     char Name[31];
     long ID_Num;
     char Class[9]
};
```

The function prototype may be declared globally or within the calling function. Here "c_list" is the type of structure from the structure prototype (declared globally), not the specific structure itself. The * indicates that a pointer to the structure will be passed:

FUNCTION PROTOTYPE

```
void PrintReport(struct c_list *);
```

A pointer to a structure of type "c_list" is created.

A POINTER IS DECLARED

```
struct c_list *Ptr;
```

A structure array of type "c_list" is created in the calling function and assigned to pointer "Ptr" and memory is allocated. "Elements" is the number of elements in the array:

STRUCTURE ARRAY IS CREATED

```
Ptr = (struct c_list *) malloc(Elements * sizeof(struct c_list));
```

The pointer to the structure is passed to the function.

FUNCTION CALL

```
PrintReport(Ptr);
```

A corresponding pointer "P" is declared in the function header:

FUNCTION HEADER

```
void PrintReport(struct c_list *P)
{
```

REFERENCES TO ELEMENTS

```
P[i].Name
P[i].ID_Num
P[i].Class
```

RETURN STATEMENT

```
return;
}
```

A FUNCTION WHICH PASSES A FILE NAME

A file pointer is declared in the calling function:

POINTER DECLARATION
FILE NAME ASSIGNMENT

```
FILE *Data;
Data = fopen("class.dat", "r+");
```

The argument is a pointer to a file:

FUNCTION PROTOTYPE
FUNCTION CALL

```
void ReadFile(FILE *)
ReadFile(Data);
```

A new file pointer is declared in the function header:

FUNCTION HEADER
RETURN STATEMENT

```
void ReadFile(FILE *F)
{
return;
}
```

USING POINTERS IN C

A pointer is a variable that contains an address.

How to Interpret & and * Symbols	
Symbol	**Read as…**
&_____	the pointer to the variable _____
***** _____	the value held in the variable pointed to by _____
(_____***)**	cast the pointer that follows into a pointer of type _____
_____ *****	a pointer of type _____ (this is used in a function prototype)

DECLARATION STATEMENT

A pointer variable is declared using the data type of the variable to whose address it points. This is so that the computer will know how many storage locations to access when it uses the variable pointed to. So if we have an integer variable **num** and we want a pointer variable **addr** to store its address, then the declaration statement for the pointer would be:

```
int *addr;
```

This statement means, "I am declaring a pointer called addr of type integer." The address of any integer variable can be stored in the pointer variable addr.

If we add 1 to addr, then it will point to the next integer. In other words, because the pointer has been declared an integer type (16 bits), incrementing the pointer causes the address to shift by two bytes in this case.

ASSIGNMENT STATEMENT

```
addr = &num;
```

The pointer **addr** now contains the address of the variable **num** and ***addr** refers to the value held in the variable **num**. Obtaining a value in this way is known as *indirect addressing* and the symbol ***** is the *indirection operator*.

READING THE VALUE OF THE VARIABLE POINTED TO

```
value = *addr;
```

The variable **value** now contains the value stored at address **addr**.

SCANF()

The **scanf()** function requires the use of addresses of variables.

syntax: `scanf("`*control string(s)*`", &`*variable(s)*`);`
i.e.: `scanf("%d %d", &num1, &num2);`

PASSING ADDRESSES TO FUNCTIONS

To pass addresses to a function (referred to as ***pass by reference***):

```
void sortnum (double *, double *);
                                /* function prototype  */

sortnum(&firstnum, &secnum);  /* the function call   */

sortnum(double *num1, double *num2)
                                /* the function header */
                                /* declaring pointers  */
                                /* to receive passed   */
                                /* addresses           */
```

When the values which are pointed to are used by the function, the indirection operator is used, i.e. **`*num1`** and **`*num2`**. The function may change these values even though they are not global.

```
return;                         /* this function would */
                                /* not "return" a      */
                                /* value               */
```

POINTERS IN ARRAYS

If we have an array, **`grades[]`**, we can store the address of **`grades[0]`** in a pointer:

```
gptr = &grades[0];
```

or

```
gptr = grades;                  /* equivalent to above */
```

Then **`*gptr`** would refer to the value stored in **`grades[0]`**. We can refer to the values stored in other parts of the array by using ***offsets***. **`*(gptr + 1)`** refers to the value stored in **`grades[1]`** and could also be written **`gptr[1]`** even though **`gptr`** was not declared as an array. (page 309) This value could also be referred to by **`*(grades + 1)`** and refers to the second value in the array regardless of the number of storage locations required by the variable type. A distinction between the latter and reference by pointer is that the address stored in a pointer can be changed. **`gptr`** is a pointer and **`grades`** is a pointer constant. Both of these point to the address of **`grades[0]`**.

`gptr` could be made equivalent to **`&grades[1]`** by the statement:

```
gptr++;                         /* increment the address */
                                /* in gptr               */
```

or

```
*gptr++;                        /* this permits first    */
                                /* utilizing the value   */
                                /* in a statement (not   */
                                /* included here).  To   */
                                /* increment before      */
                                /* using you could use   */
                                /* *++gptr;              */
```

or

```
gptr = &grades[1];              /* assignment statement  */
```

POINTERS IN STRUCTURES

Prototype for structure Student Records:

```
struct StudentRecord          /* Structure for holding */
{                             /*   student's record.   */
   char Name[31];             /* Student name          */
   long ID_Num;               /* Student ID number     */
   char Class[9];             /* Student's class       */
};                            /* Yes, a semicolon.     */
```

A structure has been declared above, but no memory has been allocated. This is only a template for a structure. Structures of this type may now be created using the template name, StudentRecord.

```
struct StudentRecord SR1,SR2; /* Two structures        */
```

The above statement creates two structures of the type StudentRecord. This statement could have been combined with the structure declaration as follows:

```
struct StudentRecord          /* Structure for holding */
{                             /*   student's record.   */
   char Name[31];             /* Student name          */
   long ID_Num;               /* Student ID number     */
   char Class[9];             /* Student's class       */
} SR1,SR2;                    /* Two structures.       */
```

In this case, the template name StudentRecord is unnecessary if no additional structures of this type are to be created. It can be omitted.

Arrays of structures may be created:

```
struct StudentRecord StuRec[10]; /* Array of 10 structs */
```

Structure pointer declaration:

```
struct StudentRecord *Recs;
```

The above statement creates a pointer called Recs that can point to a structure of type StudentRecord.

Create a single structure that is pointed to by the pointer Recs:

```
Recs=(struct StudentRecord *)malloc(sizeof(struct SR1));
```

Or create a structure array:

```
Recs=(struct StudentRecord *)malloc(HowMany * sizeof(struct SR1));
```

Single structure members may be addressed in this way:

```
Recs.ID_Num
```

Structure **array** members **must** be addressed using `->`:

```
Recs[i]->ID_Num
```