UNIT 8: FUNCTIONS

Structure

8.0 Introduction
8.1 Objectives
8.2 Function Prototypes and Declarations
8.3 Functions and Scope
8.4 Pointers as Function Arguments
8.5 Unidimensional Arrays as Function Arguments __ String Functions
8.6 Multi-Dimensional Arrays as Function Arguments
8.7 Summary

8.0 INTRODUCTION

Large programs for large "real world" applications require quite different techniques for their creation than the ones we have so far studied. As programs grow in size and complexity, it becomes more and more difficult to keep track of the logic; so programs are divided into separate modules called functions, with each function reflecting a well-under-stood activity, and of manageable proportions. The idea is to divide and conquer. For example, imagine a program to manage your personal savings bank account, a program that would monitor are several activity on a computer that is recorded in your pass book. Now there are several things to be considered while designing such a program: for one, you may deposit cash. The program should return your updated balance; or you may want to withdraw some, in which case the program should report if the amount to be withdrawn exceeds your current balance; if not, it should return the new balance.

Second, when a cheque is deposited, the program should ask for its details and store them for future reference: by whom issued, in which connexion, of what amount, cheque number, name of bank, whether the cheque was local or outstation (because a collection charge must be deducted from such cheques) the date of issue, etc.; and similarly when you write a cheque for someone, with the program updating the balance at the end of each transaction. When you enter the date of a cheque, or of a withdrawal/deposit, the program must verify if that was a valid date.

And finally, the program should compute the interest that fell to your share, at the end of the current accounting period, at the rate specified by the bank, and add it to your balance. As you can see, there are several "black box" activities, none hard to program in themselves, but unrelated to each other. One single program encompassing every activity would be quite cumbersome; its logic would quickly become obscure. What can be more natural than to divide the program into functions, a separate function accounting for each task?

But there is another reason why functions are a very useful invention, and why all programming languages have them. Quite often the same activity must be repeated several times in an executing program. Instead of repeating the lines of code representing the activity at various places in the program, it would make sense to give that piece of code a name, and cause it to be executed by invoking it by its name whenever required.

C functions have names that follow the same rules as variable names. with an extra feature: parentheses after the name. The parentheses may contain a list of arguments. separated by commas. A function may have no arguments: on the other hand, it may have several. On invocation, the calling program "injects" the values of these arguments into the function. The arguments passed to the function are processed by it in accordance with its instructions. If required, the function may send back the result of its computation to the calling program, when it returns control to it. That result has of course a type, given by the return type of the function. A function that has the void return type can return no value to the calling program, and an error message will result if you attempt to ask for its value. If the return type of a function is omitted in its declaration, it is assumed to be int.

Sometimes a function written for one problem can be used in different problems; it can serve as a "building block" to help build other programs. A function to validate a date would be equally useful in a program for railway reservations and in one to handle savings accounts. C carries this "pick up and reuse" design philosophy to a greater extent than other languages. Even input and output are implemented via functions available through "standard libraries" provided with your compiler.

The user-defined functions which build a C program may exist in different source files, as Long as a function is not split over more than one file; each of these files may be separately compiled and linked, and loaded together at execution time.

Basically then a function is lines of C code grouped into a named unit, written to perform an identifiable activity. Such a module can be invoked by a "calling" program. When a function is called, control passes from the calling program to the function. But just before this happens the current value of the Program Counter which, remember, is the address of the next instruction to be executed, is saved. When the function has been executed, control returns to the calling program, to the address saved from the PC.

Execution begins where it had stopped prior to the call. The return address is stored in a data structure known as a stack, which has the property that the last item pushed in is the first to be popped out. Therefore if, while a called function is executing, it invokes another function, the current PC contents are stored on top of the stack, i.e. on top of the last return address. The stack is "pushed" in. When the second function ends its execution. it returns control back to the first function where it had left off. The stack is "popped". The return address that's now on top is is the one to which control will return when this function finishes its execution.

ANSI C and Classic C differ considerably in the way in which they declare and define functions, though the Classic C style is accepted by current ANSI C compilers. We shall point the differences as we go along, but it is well to learn the ANSI C way of working with functions, even if you have a Classic C compiler for one very good reason: C++ follows the ANSI C standard.

8.1 OBJECTIVES

After working through this Unit you will be able to

- declare function prototypes
- create functions to perform specific tasks
- call functions by value
- call functions by reference
- understand how to limit a variable's scope across functions
- pass arrays as arguments to functions

8.2 FUNCTION PROTOTYPES AND DECLARATIONS

For our first example of functions, let's turn to Program 8.1. It determines the area of a triangle whose sides are given. It consists of three functions, other than main (), viz. begin (), form _ triangle () and area-triangle (). These functions are declared at the beginning of the program, through function prototypes:

void begin (void),

int form _ triangle (double a, double b, double c);

double area _triangle (double x, double y, double z).

The prototype declarations tell the compiler three important things about the functions:

1. to expect to encounter them further down the program;

2. the return type of the function, e.g. void, int or double;

3. the number and kind of arguments the function is to handle: none whatever (represented by the keyword void in the parentheses, as in begin()), or of specified type, as in the declarations of form-triangle () and area triangle()

The parameter names e.g. **double a, double b, double c,** etc., listed in a function's prototype need not be the same as in the function's declaration. Indeed, names in a prototype are optional. one could write:

double area -triangle (double, double, double);

but you will agree that this is a poor way to say what you really mean: that a triangle has three sides whose lengths are doubles.

The first of our functions, begin (), has no arguments. Indeed, a function need have none. begin () tells a user what the program does. Here it is:

```
void begin (void)

    {
      printf ("This program determines whether three
      numbers\n");
      printf ("that you enter can form the three sides of a\n");
      printf ("triangle. If they can, the program prints\n");
      printf ("its area... Enter values for the three sides:");
    }
```

Upon its invocation by main (), which is done simply by naming it in main (), thus:

```
            begin ();
```

control passes to begin (), where its printfs () are executed. Control re-enters main () automatically after the terminating brace of begin () is encountered, to resume from its next statement.

The first line of any function is its declaration:

```
void begin (void)
int form _triangle (double alpha, double beta, double gamma)
double area _triangle (double lambda, double mu, double nu)
```

Apart from declaring its return type, the declaration contains a comma separated list of the function's formal parameters, if it has any. If not, the function's parentheses include the keyword void.

Note that the function's declarator is not terminated by a semicolon; it must agree with the function's prototype, the declaration which is made before main (), in all particulars: return type, name, and parameters and their types. Parameter names in a function's prototype and its declarator may be different.

The function's declarator is followed by its body. enclosed in braces. Further variables may be declared inside the body of the function, if need be: their scope is limited to the periphery of the function. We will see an example of this shortly, in the function area-triangle ().

The second function, named form -triangle () in Program 8.1 determines if the three numbers input are such as to form the sides of a triangle. (We would look a little silly trying to find the area of a triangle that couldn't exist!) Recall that in a triangle the sum of any two of its sides must exceed the length of the third. That is how our function decides if the lengths passed to it can form a triangle. The values of the three lengths must be "handed over" to the function, which should then determine if a triangle is possible, returning 1 if true, () if false. Here is the function form triangle ():

```
int form - triangle (double alpha, double beta, double gamma)
                {
                        if      (       (alpha + beta > gamma) &&
                                        (beta + gamma > alpha) &&
                                        (gamma + alpha > beta))
                                return 1;
                        else
                                return 0;
                }
```

Via its optional return statement a function can return at most a single value to the calling program. which is converted to the return type of the function if it happens to be different.

In Program 8.1, after control returns to main () from begin (), it invokes this function in the following way:

```
                if (form -triangle (side- 1, side-2, side-3))
```

The call is executed from within the if (); the arguments side_1, side_2 and side_3 are "injected" in to the three parameters of form _triangle (): side _1 is passed to alpha, side-2 to beta and side-3 to gamma. The instructions within the function itself are written in terms of its formal parameters, but the values used in the function's computation are the values that came from main (). You might be wondering about the difference in meaning between arguments and parameters: the function call from main () uses arguments; the function definition itself is in terms of parameters. side-1, side-2 and side-3 are arguments; alpha, beta and gamma are parameters It should be obvious that the types of formal parameters in a function's definition should match the type of the arguments used in the call, each to each.

The keyword return provides a way for the function to return a value to main (): the syntax of its usage is

```
                                return (expression);
```

where expression is converted to the return type of the function, if need be. Parentheses around the expression returned are optional: they make for clarity. It is not required that an expression follow return: the statement:

```
                                return;
```

is syntactically correct. In this case control returns to the caller but without an accompanying value. On the other hand, a function need have no return statement: control exits from the function, back whence it came, when its right brace is met. This happens in begin (), which has no return. Realise therefore that a function can return at most one value to the calling program via its return statement.

The function form _triangle (returns 1 or 0 to main (). If it returned 1 . implying that the three numbers entered could form a triangle, area _triangle () is invoked from within main ()'s printf():

```
    if (form-triangle (side-1, side-2, side-3))
       printf ("Area of triangle is: %f\n". area-triangle (side-1,
    side_2, side-3));
```

The function area-triangle () illustrates a further optional feature of functions: local variables:

```
  double area-triangle (double a, double b, double c)

        {
           double s, area;
           s = (a + b + c) / 2;
           area = sqrt (s * (s - a) * (s - b) * (s - c));
           return (area);
        }
```

In additional to its formal parameters a, b and c, area _triangle (has two local automatic variable's and area. They are created only when control enters the function; they are accessible only from within the function. Their values are lost when control departs from the function. See Program 8.3.

area _triangle () uses Brahmagupta's formula. The area of a triangle with sides a, b and c, and semi-perimeter s is given by:

```
          area = sqrt (s * (s - a) * (s - b) * (s - c))
```

area _triangle () returns this value to main (), via its local variable, area. Here is the program in its entirety:

```
/* Program 8.1*/
#include
#include
void begin (void); /* function prototypes*/
int form -triangle (double a. double b. double c);
double area-triangle (double x, double y, double z).,
void main (void)
    {
      double side- 1, side- 2. side-3
      begin ();
      scanf ("%If %If %1r", &side-1, &side-2, &side-3);
      if (form -triangle (side- 1, Side- 2,side- 3))
         printf ("Area of triangle is: %f\n", area-triangle (side 1,
    side-2, side 3));
      else
      printf ("The lengths that you entered cannot form a triangle.
    \n");
    }
void begin (void)
    {
      printf ("This program determines whether three numbers\n");
      printf ("that you enter can form the three sides of a\n");
      printf ("triangle. If they can, the program print\n");
      printf ("its area ... Enter values for the three sides:");
    }
```

```
int form_ triangle (double alpha, double beta, double gamma)
        {
          if (alpha + beta > gamma) &&
            (beta + gamma > alpha) &&
            (gamma + alpha > beta))
              return 1;
          else
              return 0;
        }
double area triangle (double a, double b, double c)
        {
          double s, area;.
          S = (a + b + c) / 2;
          area = sqrt (s * (s - a) (s - b) * (s - c));
          return (area);
        }
```

It is in the subject of functions that there are major changes between Classic C and ANSI C. Classic C did without prototyping. the function form _triangle () would simply have been declared before main() thus;

```
                    int form triangle 0;
```

No Parameter list was admissible; moreover. since the default return type of functions is int, even the declaration was not required! And was as a consequence, often skipped, making for opaque code. On the other hand, since area triangle () returns a double result, the declaration:

```
                  double area triangle ();
```

was in Classic C, and remains in ANSI C,. a requirement; but with the following permitted relaxation of this rule.

Now it is not necessary to place functions after main () in your program, as we have done above. Functions may certainly be written before main (); indeed, if the function definition appears before the first call to it, in Classic C even its (prototype) declaration is not required!

One probably does not gain very much by this saving where there are large programs involving several functions, because care must then be taken to arrange the functions in such a way that each function appears in the code before it is called by any other function. It is easier to declare all functions prototypes before main (), and list the functions themselves after main (), a practice to which we shall invariably adhere.

The second major difference between Classic C and ANSI C is in the function definition it-self. In the former, the parameters listed in the function's header are not typed; instead, their types are declared in a separate statement preceding the function's body, thus:

```
    double area -triangle (a, b, c)
        double a, b, c; /* A Classic C parameter type
                    declaration */
            {
                double s, area,
                s = (a + b + c)/ 2;
                area = sqrt (s * (s - a) * (s - b) * (s - c));
                return (area);
            }
```

If you have a Classic C compiler, remember to declare all functions before main (), without a parenthesized parameter list. The function's header should contain only its formal parameters, separated by commas; their types must be declared in a separate statement sandwiched between the function's header and its body.

Check Your Progress 1

Question 1: Write a c function that determine whether an integer passed to it is a prime number; it it is not get main () to print its smallest factor.

Question 2: Write a c function that determines whether an integer passed to it is a perfect squre; if it is, get main() to print the squre root.

Question 3: Write a C function to return the largest Fibonacci number lets than a positive integer number passed to it.

Question 4: Write a C function to return the number of primes less than a positive integer passed to it.

Question 5: Write a c function to determine the number of days between two dates passed to it. (Hint: Find the number of days, say m, between the first date and January 1,1900. Your program should return the absolute value of the difference n-m.)

8.3 FUNCTIONS AND SCOPE

When one function is invoked by another. the arguments used by the "caller" are transmitted, as we have seen, into the formal parameters of the "called" function. However, the called function is sent only copies of the caller's variables; it can use them in its computation, but it cannot change them, except locally, within its own boundary. Such changes are not reflected in the values of the caller's variables. The called function has no power to change them. The reason for this is that the function's variables are created in the invocation at addresses different from those of the caller's variables. The called function has no access to the caller's address space, which remains insulated from it. This is illustrated in Program 8.2 in which the function float inflation () is sent the prices of milk, potatoes and bread of a few years ago. It inflates them, prints them, and returns control back to main However, the corresponding variables in main 0 remain unaltered.

```
/* Program 8.2 */
#include
void inflation (float m, float p, float b);
main ()
      {         float milk-price = 2.50, potatoes-price = 0.50, bread-price
      2.85;
        printf ("Unit prices of milk, potatoes and bread in 1984\n");
        printf ("\n were %.2f, %.2f and %.2f respectively. n\n",
                              milk __price, potatoes- price, bread-price);
        printf ("Let \'s see what inflation did to them ...\n\n");
        printf ("Transferring these prices to inflation
                              inflation (milk _price, potatoes-price, bread price);
        printf ("\n Now control is back in main ()... \n\n");
        printf ("Print those prices once more....\n\n");
        printf ("milk: %.2f, potatoes: %.2f and bread: %.2f\,n\n",
                              milk -price, potatoes-price, bread-Price);
        printf ("What ??? They haven't changed at all in main ()!!! \n");
      }
   void inflation (float milk, float potatoes, float bread)
         {
```

```
        float percent_ rise, annual -price- ratio;
        int i, num _years;
        printf ("We welcome you to inflation 0 ... \n\n");
        printf ("Enter a % % rise for inflation (range 0.. 100):");
        sccinf ("%f", &percent- rise);
        printf ("\nEnter number of years inflation occurred:");
        scarif ("%d", &num_years);
        annual-price-ratio = (100 + percent -rise) / 100;
        for (i = 1; i <= num-years; i ++)
          {
            milk *= annual-price_ratio;
            Potatoes *= annual-price-ratio;
            bread *= annual_price_ratio;
          }
        printf ("\nAs a result of %.of%% inflation, prices %d  years later are...\n",
        percent-rise, num_years);
        printf ("\nmilk %.2f, potatoes %.2f, bread %.2f\n\n",  milk, potatoes, bread);
        printf ("Thank you for bearing with inflation() ... \n\n");
        printf ("Now return to main () ... \n\n");
      }

          /* Program 8.2: Output */
```

```
Unit prices of milk, potatoes and bread in 1984

were 2.50, 0.50,ind 2.85 respectively.

Let's see what inflation did to them...

Transferring these prices to inflation ( )...

We welcome you to inflation () ...

Enter a % rise for inflation (range 0..100): 10

Enter number of years inflation occurred: 10

As a result of 10% inflation, prices 10 years later are...

milk 6.48, potatoes 1.30, bread 7.39

Thank you for bearing with inflation ( )...

Now return to main()...

Now control is back in main()...

Print those prices once more...

milk: 2.50, potatoes: 0.50 and bread: 2.85

What ??? They haven't changed at all in main 0 !!!
```

The automatic variables declared within a function's body are local to it. They are created when control passes to the function, are in existence until control remains within the functions, and are annihilated when it departs there from. In Program 8.3 the function fl () is called twice in succession by main (). In its first invocation fl ()'s automatic variable x is assigned a value which it returns to main (). But conditions are such that in its second invocation x cannot be assigned a value; one might suppose then that the last value of x, the value it had when the function was first invoked, may still be available: not so. That variable was destroyed and its value was lost when control returned to main 0. Ale second call to fl returns an unpredictable value for x. In fact, by a little modification of Program 8.3 you can easily show dial each call to fl (creates x at a different address.

```
/* Program 8.3*/
#include
int fl (int a, int b);
main ()
         {
           int alpha, beta, gamma;
           alpha = 10;
           beta = 20;
           gamma = fl (alpha, beta);
           printf ("alpha = %d, beta = %d, gamma = %d\n", alpha, beta,
           gamma);
           alpha = 20;
           beta = 30;
           gamma = fl (alpha, beta);
           printf ("alpha = %d, beta = %d, gamma = %d\n", alpha, beta,
          gamma);
         }

    int f1 (ini p, int q)
         {
            int x;
            if (p * q < 300)
            x = p * q;
            return (x);
         }
                  /*Program 8.3: Output */

alpha = 10, beta = 20, gamma = 200
alpha = 20, beta = 30, gamma = 30
```

However, if the variable x in f1 () is declared static:

```
                    static int x;
```

it does retain its value between invocations. It doesn't melt away into the wilderness when the function is not in use. And is initialized to the value 0, unless a different initial value is specified. That initialization is made in the first call to the function. Subsequent calls to it ignore the initialization but use the fast value that the static variable was left with, as you will verify by executing Program 8.4. But before proceeding to do so it may be useful to review Section 6.5 and Programs 6.17 - 6.19 once more.

```
/* Program 8.4 */
#include
int f1 (int a, int b).,
main ( )
         {
          int alpha, beta, gamma;
          alpha = 10;
          for (beta = alpha; beta > alpha / 3; beta -)
             {
```

```
            printf ("Proceeding to f1 0..\n");
            gamma = f1(alpha, beta);
            printf ("alpha = %d, belt = %d, gamma = %d\n", alpha, beta, gamma);
          }
        /* contd. */

         }
int f1 (int p, int q)
        {
          static int x. count = 0; /* initialization unnecessary, count is by
          default 0*/
          printf ("Entered f1 () ... \n");
          printf ("Current number of calls to function: %d\n", ++ count);
          printf ("On entry x was found to have the value: %ft", x);
          if (p * q > 70)
            x = p * q;
          printf ("...Returning to main ()... \n");
          return (x);
        }
```

Using an argument list is not the only way by which the variables of one function can be passed to another. We recall from Unit 6 (Programs 6.18 and 6.19) that variables declared externally to main () are accessible inside it; they are also accessible inside any functions that follow main (). Program 8.5 has a date validation function with which it communicates via the global variables Day, Month and Year.

```
/* Program 8.5 */
#include
#define LEAP-YEAR (Year%4==0&&Year%100!=0)\
                        || Year % 400 == 0
int Day, Month, Year;
int valid date ()
main ()
        {
          printf ("\n This program finds the day corresponding to a given date:\n");
          printf ("\nEnter date, month, year .... format is dd-mm- yyyy.\n");
          printf ("\nEnter a 1 or 2-digit number for day, followed by a\n");
          printf ("\ndash, followed by a 1 or 2-digit number for month,\1i");
          ptintf ("\nfoliowed by a dash, followed by a 2 or 4-digit number\n");
          printf ("\nfor the year. Valid year range is 1582 - 4902, inclusive. \n");
          printf ("\n(A 2-digit number for the year will imply 20th century \n");
          printf ("\nyears.)\n\n\n Enter dd- mm-yyyy: ");
          while (scanf ("%d-%d-%d", &Day, &Month, &Year)!= 3)
                    {
                      printf ("\n Invalid number of arguments or hyphens  mismatched \n");
                       printf ("\n Re-enter:
                    }
          if (valid_date
            printf ("Date entered is OK. \n");
          else printf ("Date entered is not OK. \n")
        }
    int valid-date ()
            {
              if (Year < 0)
                return 0;
              if (Year < 100)
                Year += 1900;
              if (Year < 1582 11 Year > 4902)
                return 0;
              if (!(LEAP YEAR) && (Month == 2)&& (Day > 28))
                return 0;
              if ((LEAP-YEAR) && (Month == 2)&& (Day > 29))
                return 0;
              if (Month < 1 || Month > 12)
                return 0;
              if (Day < 1 || Day > 3 1)
                return 0;
              if ((Day > 30) && (Month == 4 || Month == 6 ||
                 Month == 9 || Month
                return 0;
              return 1..
            }
```

This example nicely illustrates the major advantage of functions. valid-date 0 may be used with any program that processes dates (not of course the kind that we cat!) Most of it has been adapted from Program 5. 16.

For another example of the use of global variables, let's look at Program 8.6, which solves a problem asked in Unit 6: determine ine how many zeroes there are at the end of the number 1000!.. Now, each zero in that expansion signifies the presence of a factor of 2 and a factor of 5 in the product for 1000! If we had a function that could determine how many times 2 and 5 were factors in each of the multiplicands, and update two variables named two occurs and five_ occurs each time 2 or 5 was found as a factor, then we would be in business. The smaller of the values two occurs and five occurs would be the number of zeroes in 1000!

The function is2or5factor () updates the two global variables int two occurs and five_ occurs declared before main (). They are not passed to the function by main 0. they're already visible in it. The ternary operator in main () yields the number of zeroes in 1000!

```
/* Program 8.6 */
#include
void is2or5factor (int n);
int two-occurs, five-occurs;
void main ()
      {
        int number;
        for (number = 1000; number >= 2; number-}
          is2or5factor (number);
        printf ("Number of zeroes in 1000! is: %d\n",
        two-occurs > five-occurs ? five-occurs : two-occurs);
      }

void is2or5factor (int number)
          {
            int quotient;
            quotient = number;
            while (quotient % 2 == 0)
              {
                two-occurs ++;
                quotient /= 2;
              }
          while (quotient % 5 == 0)
              {
                five-occurs ++;
                 quotient /= 5;
            }
        }
```

Convenient as they are, the indiscriminate use of global variables is not recommended. For one thing, functions have the power to change them. That makes those variables vulnerable; suppose you're using a function "off the shelf", so to speak, that as a side effect, changes the values of its variables: lo, that change, desirable or not, will be transmitted to your program. A function should be given capabilities on an "as needed" basis; if a date value sent down by main () must be protected, global variables will not be suitable. But with a list of formal parameters for communication, a function becomes a module that may be "plugged in" to any "driver" program with a matching set of arguments. There will be no fear that values of she driver program's variables may be changed by the function.

One important feature of global variables that should be kept in mind is that if a function has a local variable of the same name as a global variable, the local variable has precedence over the global. Analogously, a variable declared within a block has precedence over an external variable of die same name. Recall the behaviour of Programs 6.16 - 6.19. The local is "in scope" when control enters the function. The external definition is ignored.

In order to be able to use an externally declared variable inside a function, such a variable may be explicitly redeclared inside the function with the extern qualifier:

```
                    extern int x;
```

The external keyword tells the C compiler and linker that the variable thus qualified is actually the one of the same name defined externally; this is the one that will be used in the function. New storage is riot created for it, because it refers to a variable that already exists. This brings us to the important distinction between the terms definition and declaration: a definition creates a variable and allocates storage. A declaration describes a type but does not reserve space. Precisely for this reason an initial value cannot be specified with an extern declaration: initialization means that a value is assigned at the time that storage is allocated; because storage is not allocated by the extern declaration, it cannot include an initialization Therefore, inside a function the following statement is unacceptable:

```
extern int x = 5, 1* WRONG */
```

Now you might wonder: of what use can extern declarations be, when external variables are by default available within the functions that follow them? True: but what if a variable is required inside a function, yet is itself defined after die function" An externally defined variable is in die scope only of tile functions that follow it, not of those that precede it. Then, you will admit, the extern declaration can serve a useful purpose. It tells the C system: "Look this extern variable is one that you haven't seen before: but don't panic, have patience, it's defined externally to a function that's defined lower down, or possibly in another file, and will soon coming within your ken!" Again, C functions can be stored ii separate files. If a variable is externally declared in one file, and is needed inside the functions stored in the other files, the extern declaration in the functions will make it accessible to them.

There is one last scope rule relating to externally defined static variables that must be stated here for completeness (we hope that you will christen your variables carefully, and may never need to use it): suppose there is a situation in a multi-file program in which it is required to use a variable within a file, and within it only; suppose also that a variable of the same name is defined globally in another file. To restrict the scope of a variable to the functions contained in a single file, and to prevent name conflicts across other files, the variable is defined static externally in the file to which it must be confined. For example, suppose that your program and the functions that it uses are stored in three files, file-1, file-2 and file-3:

contents of	contents of	contents of
file-1	file-2	file-3

int x;	int y;	static int x, y;
Main ()	Func-2 ()	func- 4 ()
{}	{}	{}
extern int y;	extern int x;	
func_1 ()	func-3 ( )	Func - 5 ()
{}	(int y;}	{}

Then int x defined in file-1 is available in main (), in func_1 () because it is external to it in the same file, and in func_3 (), because of the extern qualifier before its definition in file- 2. int y, defined in file-2 at the top of func_2 () is available also in func_1(); it's redefined inside func_3 (), where the automatic variable has precedence over its externally defined namesake; the external static int variables x and y in file are accessible only inside that file; they have nothing to do with other variables of the same names in file-1 and file_2. Precisely how multiple programs are compiled and linked is a compiler-operating system feature that differs from compiler to compiler. We can only refer you to your system's manuals.

To understand these rules let's look at Program 8.6, which consists of three functions other than main (), func_1 (), func_2 () and func_3 () each of type int. global_var is a global variable and therefore accessible by every function except func_3 (), wherein it's redefined:

int global_var = 2;

The int variable a is defined externally to func_2 0 and func_3 (). Naturally it is accessible in these functions; but it is also available in main () and in func-1 0, because it is redeclared in these functions:

extern int a;

When main () calls func_1 () it sends in the values 5, 7 and 5 to L, M and N respectively. The function changes global-var, M and a: the change in M is riot reflected back in main (). M is only a local variable, a country cousin, so to speak. And the function returns a value to main ()'s int variable, c.

Next the current values of a, b and c are passed to func_2 (), to P, Q and R respectively. These local variables are made fresh assignments within-func-2 (); as before the corresponding main () variables are unaffected. But it's an altogether different story when func_2 () changes global_var and a. These latter changes are visible to every function that has access to them. Finally runc_3 () is invoked. Here global_var is the name of a local chieftain, whom func_3 () honours more than the outsider of the same name. And a is not redeclared in func-3 0. It's available there by default.

```
/* Program 8.7 */
#include
int func_1 (int 1, int m, int n);
int func_2 (int p, int q, int r);
int func_3 (int x, int y, int z);
int global_var = 5;
void main ()
        {
          extern int a;
          int b = 5, c;
          a ++;
          printf ("main 0 changes externa to %d\n", a);
          c = func_1 (global_var, a, b);
          printf ("\non return from func_1 ()\n");
          printf ("a = %d\n", a);
          printf ("b = %d\n", b);
          printf ("c = %d\n", c);
          printf ("global_var = %d\n", global-var);
          c = func_2 (a, b, c);
          printf ("\non return from func-2 ()\n");
          printf ("a = %d\n", a);
          prinif ("b = %d\n", b);
          printf ("c = %d\n", c);
          printf ("global-var = %d\n", global-var);
          c = func-3 (a, b, c);
          printf ("\non return from func_3 ()\n");
          printf ("a = %d\n", a);
          printf ("b = %d\n", b);
          printf ("c = %d\n", c);
          printf ("global_v = %d\n", global-var);
        }

int func-1 (int L, int M, int N)
        {
          extem int a;
          printf ("\nIn func-1 ()...\n");
          printf ("L gets main 0\'s value of global_var %d\n", L);
          global_var ++;
         --a;
          M += a;
          printf ("global_var, a, M in func_1 (: %d, %d, %d\n",
         global_var, a, M);
          printf ("Returning ... \n");
          return (L * global_var + M * global-var - N * a);
        }

int a = 6,
int func_2 (int P, int Q. int R)
        {
          printf ("\nIn func-2 () ... \n");
          printf ("P, Q, R gets main ()\'s a. b, c: %d, %d, %d\n", P, Q, R);
          P - = 2;
          Q-= 3;
          R-=4;
          a.= 5;
          global~Var -;
          printf ("Returning...\n");
          return (a * a - P * Q + global-var * R).,
        }

int func-3 (int X int Y int Z)
        {
          int global-var = 2;
          printf ("\nIn func_3 0 ... \a");
          printf ("X gets main 0\'s value of a: %d\n". X);
          Z -= 2 - - a;
          Y += a - Z;
          printf ("global-var. a in func-3 0: %dg %d\n". global-var. a);
          printf ("Returning ... \n");
          return (X + Y + Z);
        }
```

Check Your Progress 2

Question 1: Predict the out put of Program 8.4 and 8.6. Verify results be creating, compiling and executing and executing these program.

Question 2: What does the following program print?

```
/*Program 8.8*/
#include
int alpha(int a1, int a2, int a3);
int x = 2, y, z;
void main ()
        {
            {
              auto int y=4, z=6;
              printf ("%d\n",(x = alpha (x,y,z)));
              printf ("%d\n",(y = beta (x,y,z)));
            }
            {
              printf ("%d\n",(x= alpha (x,y,z)));
              printf ("%d\n",(y= beta (x,y,z)));
            }
        }

    int alpha (int A1, int A2, int A3)
        {
          exterm int w;
          return (y = w= A1+ A2+ A3);
        }

    int w = 5;
    int beta (int B1, int B2, int B3)
        {
          x = 7;
          return (x = x+ y+ w+ B1+ B2+ B3);
        }
```

Question 3: What does the following program, spread over three files, print? The contents of file_1 are:

```
/*Program 8.9 */
#include
int x = 5;
void func_1 (void);
void func_2 (void);
void func_3 (void);
void func_4 (void);
void func_5 (void);
void main (void);
        {
          printf("Inside main ()\n");
          func_3 ();
          printf ("After calling func_3 () x is: %d\n", x);
          x = 10);
         func_1 ();
         func_5 ();
        }
```

```
extern int y;
void func_1 (void)
        {
         printf ("Inside func_1 ()\n");
         printf ("x in func-1 () is: %d\n",x);
         printf ("y in func-1 () is: %d\n",y);
         func-2 ();
         printf ("After a call to func-2 () y is; %d\n", y);
         func-4 ();
        }
```

The contents of file-2 are:

```
Int y = 10;
Void func-2 (void)
        {
          y = 5;
        }

extern int x;
void func-3 (void)
        {
         int y = 15;
         x = 20;
        }
```

And the contents of files-3 are:

```
Static int x,y;
Void func-4 (void)
        {
         x = -5;
         y = -10;
        }

void func-5 (void)
        {
          printf ("in func_5 (), x=%d, y=%d\n", x,y);
        }
```

8.4 POINTERS AS FUNCTION ARGUMENTS

One apparent drawback of functions is that they can return at most one value back to the calling program. We say apparent with some justification, because the reality is that a function may be made to send back as many different values as may be required (though not via the return statement, whose load carrying capacity is limited to but one expression). A C function has in fact all the power of a FORTRAN function and subroutine combined. One mechanism by which a function may circumvent the limitation imposed by the return statement lies in C's use of global variables. This corresponds to FORTRAN's COMMON statement. As we have seen in Program 8.8. a function can assign the results of its computations to global variables available to main (). In that program the function is2orSfactor () sets the values of two variables for main 0 to use: two -occurs and rive-occurs. Another mechanism for sending back more than one value is provided by pointers: the correspondence here is with Pascal's var parameters.

We learnt from Program 8.2 that a function cannot change the values of local variables written in the argument list in its invocation; the function is sent only "copies" of the caller's variables. The calling program's variables remain unaffected by whatever the function does to its copies. This mode of invoking a function is called "call by value". However, when the arguments listed in a function call are pointers, it is the memory addresses of the caller's variables that are transmitted to the function. The function can change tile contents stored at those addresses, and these changes will be visible in main (). A function with pointer variables for parameters can return several values to main (), in the objects that are pointed to by its parameters. This type of function invocation is called "call by reference." Consider Program 8.10 and its output:

```
/* Program 8. 10*/
#include
void cant-change (int a, int b, int c);
void can-change (int *1, int *m, int*n);
void main (void)
        {
          int x = 4, y = 5, z = 6;
          printf ("In main (): x = %d, y = %d, z = %d.\n", x, y, z);
          cant-change (x, y, z);
          printf ("Back in main (): x = %d, y = %d, z = %d.\n", x, y, z);
          printf ("cant-change () couldn\'t change x, y and z.\n"),
          can-change (&x, &y, &z);
          printf ("Back in main (): x = %d, y = %d, z = %d\n", x, y, z);
          printf ("can-change () could change x, y and z.\n");
        }

void cant-change (int A, int B, int c)
           {
             printf ("Entered cant-change ()...\n");
             A = A * A;
             B = A + B * B;
             C = B + C * C;
             Prientf ("Parameters reset in cant-change ().\n");
             Prientf ("A= %d, B= %d, C= %d.\n",A,B,C);
           }

void can-change (int *A, int *B, int *C)
           {
             printf ("Entered can-change ()...\n");
             *A= *A* *A;
             *B *A+*B**B;
             *C *B+*C**C;
             printf ("Parameters reset in can-change ().\n");
             printf ("*A= %d, *B = %d, *C = %d.\,n", *A, *B,
           *C);
           }
```

```
                /* Program 8.10: Output */

In main (): x = 4, y = 5, z = 6.
Entered cant-change ()...
Parameters reset in cant-change ().
A = 16, B = 41, C = 77.
Back in main 0: x = 4, y = 5, z = 6.

cant_change () couldn't change x, y ind z.
Entered can_change
Parameter reset in can_change
*A = 16, "B = 41, *C = 77.
```

Back in main (): x = 16, y = 41, z = 7 7.
can change () could change x, y and z.

The function can - change (does a subroutine's work: it sends back three different values into three different variables in main (). This program should solve, one mystery that may have, plagued you: why scanf () uses pointers and printf () doesn't. The reason is that scanf () must, write new values into existing memory locations. If it used identifiers instead of pointers, it wouldn't have been able to alter the contents of those locations.

Program 8.11 determines the length, slope, land y-intereept of a line whose end-points are given. The function line ()'s argument list includes the co-ordinates of the end-points, as well .is pointers to the variables that must be computed. Note that the function has a return type of int; it returns a value to main (), the value of error_trap, which indicates the success or failure of the computation. The function performs its error checking before attempting to execute an operation that could be illegal, e.g. divide by zero: this would occur if the co- ordinates of the end-points were the same, or the line were parallel to the y-axis.

```
/* Program 8.11 */
int line (float x1, float y1, float x2, float y2,
float * length, float * slope, float * y_intercept);
#include < stdio.h >
#include < math.h >
void main (void)
        {
           float x1, y1, x2, y2, length, slope, y_intercept.,
          int error-trap;
           printf ("Enter co-ordinates in the formi x, y for");
           printf (" point where line begins:")
           scanf ("%f,%f", &x1, &y1);
           printf ("Co-ordinates at end of line:")
           scanf ("%f,%f", &,x2, &y2);
           error_trap = line (x1, y1, x2, y2, &length, &slope, &y_intercept);
           if (error_trap = = -2)
              printf ("No line possible between (%f,%f) and (%f,%f)\,n", x1, y1, x2, y2);
           else
              if (error_trap = =-1)
                printf ("Line has infinite slope. Length is: %f\n", length);
              else
                printf ("Length = %f, Slope = %f, Y-intercept = %f: ",
                           length, slope, y_intercept);
        }

    int line (float x1, float y1, float x2, float y2, float * length, float * slope, float * y_intercept)
        {
           if ((x 1 = = x2) &r & (y 1 = = y2))
              return -2;
           else
              if (x 1 = = x2)
            {
               * length = y2 > y 1 ? (y2-- y1) : (y1 - y2);
               return -1;
            }
             else
            {
               * slope = (y2 - y1) / (x2 - x1);
               * length = sqrt ((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
               * y-in tereept = y1 - *slope * x1;
```

```
                return ();
            }
        }
```

Check Your Progress 3

Question 1: Rewrite Program 8.8 using pointers instead of external variables.

Question 2: Write a C program that uses pointers and a function to compute both the area and the perimeter of a triangle whose sides are given.

Question 3: Write a C program that uses pointers and a single function to compute the coordinates of the ex-centre of a triangle, given the co-ordinates of its verticles. Your program should return the radius of the ex-circle.

Question 4: Repeat question 3 for the in-circle of the triangle. (In question 2, 3 and 4 be sure to check that you are not dealing with non-existent triangles!)

Question 5: Write a C program that uses pointers and a single function to compute the two roots of a quadratic equation, given its coefficients. Your function must be able to cope with the four cases: non-existent roots (a = 0); real and different roots; equal roots; and complex roots.

Question 6: Write a C program that uses pointers and a single function to implement the elementary arithmetic operations of complex numbers. Depending on the operand entered (+, - , *, /) your function should be able to return the result of the desired operation. (In Questions 5 and 6 pay special attention to the conditions in which the function may fail, and how it should then report the error status back to main ().)

Question 7: Write a function that returns the date a given number of days after a given date. That is, your program should be able to answer a question such as: "What date will it be 73 days after 30 Dec., 1991?"

8.5 UNDIMENSIONAL ARRAYS AS FUNCTION ARGUMENTS: STRING FUNCTIONS

Since the name of an array is a pointer, when an array is passed to a function, the function receives a pointer, the address of its zeroth element. This saves storage and enhances program efficiency: the need for copying the entire array into the function's address space is obviated. But the mechanism also imparts to the function the blower to change the contents of the array. Let's look at Program 8.12, which uses a function called uppercase () to convert the lowercase elements of a string array to uppercase.

```
/* Program 8.12 */
#include
void uppercase (char * pointer);
void main (void)
        {
          char string_holder [100];
          puts ("Type a few good words, and press return.");
          gets (string-holder);
```

```
            uppercase (string_holder);
            puts (string_holder);
          }

void uppercase (char * ptr)
              {
                int i;
                for (i = 0; * ptr; ptr ++)
                   if ((ptr [i] >= 'a') && (ptr [i] <= 'z'))
                  ptr [i] -= 32;
              }
```

C permits the following alternative declaration inside the function header:

```
void uppercase (char ptr [])
```

These declarations create pointers: arrays are not copied into functions.

Given the fact that C strings are terminated by the null character, it is a simple matter to write functions to manipulate strings in various ways. The function int howlong (), which has a string pointer for its formal parameter, returns the number of non-null characters in the string:

```
int howlong (char string [])
          {
            int i = ();
             while (* string++)
              ++ i;
            return i.,
          }
```

Here, string is a pointer that holds the address of a string array sent down from the calling function. string is incremented until the null byte in the array is sensed. Correspondingly a counter i is also incremented. On exit from the loop i holds the number of visible chars in the string array. For a second example consider the function joinem () below, which has two string pointers for parameters: it appends the second string to the first: .

```
void joinem (char * string_1, char * string_2)
          {
             for (; *string_1; string_1 ++ )
                 ;
             for (; *string_1 ++ = *string_2 ++; )
                 ;
          }
```

The first for (;;) loop seeks the end of the first string, from where the second loop begins to copy in the chars of the second string. Again, the strings' null characters tell the loops where to stop. Needless to say, the calling function must store the first string in an array big enough to store the second with it. Will the.,second for (;;) loop of joinem (copy in the null byte of string_2?

The function void copyover () has two string pointers for its parameters. It copies the second string over the first:

```
void copyover (char * string_1, char * string_2)
          {
             for (; *string_1 ++ = *string__2 ++; )
          }
```

These functions are available in the standard library where they are known by other names: howlong () is called strlen (), joiners () is called strcat (from string concatenation), and copyover (is called strepy (). ANSI C compliant compilers come with a header < string.h > which contains declarations for these and several other functions for string manipulation. Older compilers do not have a < string.h >; the functions are directly available. Some of the functions defined in < strings > are listed below (your compiler may provide several others):

Table 8.1
Some < string.h > functions

char * strepy (s1, s2)	copies s2 to s1, returns s1
char *strnepy (s1, s2, n)	copies at most n characters of s2 to sl, returns s1
char *streat (s1, s2)	appends s2 to s1, returns s1
char *strneat. (s1, s2, n)	appends at most n characters of s2 to s1, returns s1
int stremp (s1, s2)	compares s1 to s2, returns a negative, zero or positive value depending on s1 being less than, equal to or greater than s2
int strnemp (s1, s2, n)	compares at most n characters of si to s2, returns a negative, zero or positive value depending on s1 being less than, equal to or greater than s2
char *strchr (s1, c)	returns pointer to first occurrence of char c in s1, or NULL of c is not in s1
char *strrchr (s1, c)	returns pointer to last occurrence of char c in s1, or NULL of c is not in s1
char * strpbrk(s1, s2)	returns pointer to first occurrence in string s1 of any character in string s2, NULL if none is present
char * strstr (s1, s2)	returns pointer to first occurrence of string s2 in s1, NULL if not present
size_t strlen (s) produced by the	returns length of s as an unsigned int (size_t is the type of result sizeof operator, an unsigned int.)

Though string arrays are by far the most commonly used arrays in C programs, it is frequently required to process lists of numbers such as students' marks, a savings account's transactions, etc. Program 8.13 illustrates how an array of ints is passed to a function; the function returns the maximum and minimum elements of the array and the meanof all its elements. The interesting thing about Program 8.13 is that it can work for arrays of any size; one doesn't have to keep track of the number of elements. The sizeof operator applied to an array yields the number of bytes in it Dividing this number by sizeof (int) gives the number of elements in the array:

```
        num_el = size of array / sizeof (int);
```

```
/* Program 8.13 */
#include < stdio.h >
float minmax (int arrayofints [], int num_el, int * max_el, int * mines);
void main (void)
    {
      int * min. * max;
      float avg;
      static int array [ ] = {
                              - 1,   - 2,     - 3,     - 4,     - 5,     - 6,
                                7,     8,       9,      10,      11,      12,
                             - 13,   -14,     -15,     -16,     -17,     -18,
                          19,     20,       21,       22,      23,       24,
                        };
     avg = minmax (array, sizeof array / sizeof (int), min, max);
     printf ("Max is %d, Min is %d, Avg of elements is %.2f\n", *
    max, * min, avg);
    }

float minmax (int intarray [], int num_elements, int *m, int *n)
    {
      int i;
      float total = ();
      *m = *n = *intarray;
      for (i 0; i < num_elements; i ++)
       {
          if (*intarray + i) > * m)
            *m = * (intarray + i);
          if (* (intarray + i) < * n)
            *n = * (intarray + i);
          total += * (intarray + i);
       }
      return (total / i);
    }
```

Check Your Progress 3

Question 1: Write a function that returns the day which fell on a given date passed to it. Make sure that your function returns a message "Invalid date", if an illegal date is passed to it.

Question 2: Write a function that accepts two one-dimensional arrays A [] and B [] and returns their scalar product:

A[0] * B[0] +A [1] * B [1] + A[2] * B[2] +...+ A[n] *B[n]

Question 3: Write a function that accepts two one-dimensional arrays A [] and B [] and returns their sum in a third array C []

Question 4: Write a function that accepts a string and determines the number of times a given character appears in it. For example, if the string is "Malayalam", and the character is 'a', your program should print the value 4.

Question 5: Modify Program 8.13 so that it returns to main () the subscripts of the maximum and minimum elements in it.

Question 6: A set of marks in the range 0-100 is given. Write a C function to determine how many of the scores fell in the ranges 0-10, 11-20, 21-30, ...91-100.

Question 7: You are given a list of numbers, some of which may occur more than once in the list. Write a C function to remove duplicate numbers from the list.

Question 8: Write a C function to sort an array of numbers using the interchange sort. Your function should be able to handle any "reasonably sized" array of ints.

8.6 MULTI-DIMENSIONAL ARRAYS AS FUNCTION ARGUMENTS

If a two-dimensional array, say int numbers [41[6] is to be passed to a function addrows (), the parameter declaration must specify rather more than merely:

int numbers []; or,
int numbers [] [];

The first form of the declaration does not specify the two-dimensional nature of the array; the second does not go far enough in describing the shape of the array: is numbers a 2 x 12 matrix, or 3 x 8, or 4 x 6 or 6 x 4, etc. Since there are in fact 6 elements in each row, the number of rows is immaterial; the number of columns fixes the dimensions of the
array. Either of the following header declarations will do:

```
addrows (int numbers [4][6])
    {}
      or
addrows (int numbers [][6])
    {}
      or
addrows (int (*numbers) [6])
    {}
```

The, first declaration is explicit; the second tells that numbers [][6] has 6 columns; the third informs the compiler that the parameter is a pointer to a row of six integers. The parentheses around *ii urn hers are necessary since the subscript operator has a higher priority than the in direction operator. Without these parentheses the declaration would have implied an array of six painters to integers. Generally the first subscript limit of a multidimensional array need not be specified in its declaration in functions; all other dimensions must be specified. The third form of the declaration should give you a hint that numbers [0] is precisely the same as &numbers [0][0], numbers [1] holds the address of numbers [1][0], and numbers [3] points to numbers [3][0]. For C a two-dimensional array is an array of one- dimensional arrays!

Program 8.14 illustrates how the two-dimensional array matrix [4][6] is passed to a function addrows (). which returns the sums of elements in each of its four rows.

```
/* program 8.14 */
#include
void addrows (int mat [][6], int s []);
void main (void)
      {
        static int sums [4];
        static int matrix [4][6] = {
                                          { 1, 2, 3, 4, 5, 6},
                                          {7, 8, 9, 10, 11, 12},
                                          {13, 14, 15, 16, 17, 18},
                                          {19, 20, 21, 22, 23, 24},
                                       };
       addrows (matrix, sums);
        printf ("Row sums are: %d, %d, %d and %d\n",
        sums [0], sums [1], sums [2], sums [3]);
      }

void addrows (int mat [][6], int *sl)
```

```
{
  int i, j;
  for (i=0;i < 4; i ++)
     for ( j=0;j < 6; j ++)
    sl [i] = sl [i] + mat [i] [j];
}
```

Check Your Progress 5

Question 1: Write a C function to interchange any two rows of a two-dimensional array passed to it.

Question 2: Write a function to determine if a two-dimensional array passed to it is symmetric, that is, if

A [i][j] == A [j][i]

for any valid i and j.

Question 3: Write a C function to transpose, a matrix, that is, your function should make its rows into columns, and vice-versa.

Question 4: Write a C function to multiply two conformable matrices. Use your function to find the product.

P= M x M x M

for the matrix M [3][3] given by

```
{  {4, 9, 2},
   {3, 5, 7},
   {8, 1, 6}
}
```

Question 5: Write a C function to determine if a given matrix input to it is a magic square, that is, if all its rows. columns and diagonals suit to the same number, Use your function to prove that both M and P in Question 4 above are magic squares.

Question 6: Consider the 3-dimensional arrays

```
static int cube [3] [3] [3] =
                    {
                     9, 13, 2, 11, 27, 4, 22, 2, 18,
                     23, 3, 16, 7, 14, 21, 12, 25, 5,
                     10, 26, 6, 24, 1, 17, 8, 15, 19
                    }
```

Question 7: Write a program that prints a calendar for any year between 1900 and 2100 AD.

8.7 SUMMARY

This Unit introduced the concept of functions, the building blocks of all C programs. Functions help in making modular code: each logical activity of a program can be coded as a separate function, of manageable size and complexity. Thus large programming projects can be decomposed into separate activities, each encoded via a "stand alone" function. C functions can be stored in separate files, that are independently compilable; they may later be linked to a program and executed with it. So they serve a most important purpose: reusability. The same function may be used over and over again in a variety of different situations. ANSI C requires all function to be prototyped. The prototype declaration contains information about the type of value a function returns, and about the number and types of parameters it uses.

In this unit we saw how data are passed to functions, and how functions return the results of their computations back to the calling program. Functions cannot alter the values of program variables that are transmitted to them. However, if a pointer is passed as an argument to a function, in effect the function has been informed where in memory the variable is located. The function can then alter its value. Functions can manipulate and change the contents of their pointer arguments: these are in fact the only arguments that they can modify.

The scope of a variable determines its visibility within and across functions and files. Externally defined variables are visible in all the functions that follow their definition within a file. An automatic variable of the same name as an externally defined variable has precedence in the function in which it is defined. The extern qualifier helps resolve forward references, and references in multiple programs. Static variables within a function retain their values between function invocations, in contrast to automatic variables which are reborn each time that their defining function is invoked.

Because the name of an array is a pointer to its zeroth element, when an array is passed to a function, it is modifiable by the function. This unit introduced string functions, and multi-dimensional arrays as arguments to functions. Such arrays can be regarded as arrays of arrays.