Unit 9: FUNCTIONS 11

Structure

9.0 INTRODUCTION

The C language imparts to the function concept a reach and power greatly exceeding the capabilities of functions in other popular programming languages. For one thing, a C function can call itself. Functions that call themselves are said to be recursive. This is a powerful idea; it enables elegant solutions where the iterative approach of loop structures can provide only cumbersome alternatives. In such a call the values of the current parameters as well as the return address are preserved, and control reenters the function from its beginning with a new sot of parameters. In this new computation again the function can call itself, and again in the computation arising from this last call, and then once more, and so endlessly. Each call to a recursive function can generate a new call to it. There is the very real danger therefore of an infinite number of calls never punctuated by a return; the stack, which must hold not merely each return address but also the values of the parameters created in each call, can soon become full, causing a carelessly written recursive program to be gracelessly terminated. For a recursion to be successful, at some point such conditions must be created that no further calls to the function will be required. Then control can return to the address it last came from; then back again to where that call came from, and further back. telescoping inwards in this way until the stack which maintained addresses and parameter values is again empty. In this Unit we will look at some simple examples of recursive functions.

In C it is also possible to #derine macros as "functions" with parameters. Such macros look like functions because they have parentheses. The parentheses enclose a list of parameters. The macro #definition (i.e. expansion) is written in terms of these parameters. Inside a program the parameters can be replaced by constants, variables or expressions. When encountered, the macro is expanded in situ in accordance with its definition. The value of the expression generated appears in place of the macro "function" in the program, with the dummy parameters being replaced by the arguments listed. Such "macro functions" are not really functions, however, and their parameters do not work precisely like function arguments; macro functions are riddled with pitfalls and must be used with care.

The macro conditionals of C provide a facility whereby programs can be compiled subject to certain predeclared conditions. Suppose you have a program whose behaviour depends upon the word size, of the host machine; yet you would want it to be runnable without change on both a 32-bit VAX as well as a 16- bit IBM PC. The macro conditionals will enable you to build in that portability inside your program.

C functions have other powerful features: for example, it is possible to extract the address of a function and store it in a pointer of the same type as the return value of the function. Dereferencing the pointer results in a call to the function! This concept is frequently used in advanced level programming; it is the key to developing abstract data types (ADTS) and polymorphic or type-independent data structures. One of the problems beginners face in this connexion is in unravelling the meaning of complicated declarations, which occur frequently in advanced-level programming. In this Unit you will also learn how to understand complicated declarations and how to create them if needed.

Then again, functions can be created to cope with a variable number of arguments: printf.() and scanf () are two examples. But you may create such functions yourself. We'll see how this is done in a function called addem () , which returns the sum of its arguments. In one call you may send in 15 values to it; in the next, 20.

Also in this unit we will learn how to supply arguments to a program when it begins executing. Such arguments are called "command-line arguments", because the arguments are supplied in the same line as the command itself. How can this feature be used? Well, computer users apply it everyday in such simple commands as:

```
copy file_1 file_2 < CR >
```

This command makes a copy of file_1 ard names the copy file_2. The copy program is . designed to accept file names at run time. The file names are parameters, but copy does not know their names in advance. In all the functions we've used so far, the parameters came from within the program, where they had been "wired in". Here parameters are supplied when the program begins to execute!

Finally in this Unit we will look at two memory allocating functions, calloc () and malloc (). These functions are required when memory must be allocated to a program while it is executing, i.e. dynamically. As we know, the one drawback of arrays is that their sizes must be predeclared. They are therefore of limited practical use in situations where one doesn't know beforehand how much storage the data which will be input to the program, or which it will generate, can require.

9.1 OBJECTIVES

After reading this Unit you should be able to understand and write programs with

- Recursive Functions
- Macros as Functions
- Command-line Arguments
- Variable-length Argument Lists
- Complicated Declarations
- Dynamic Memory Allocation

9.2 RECURSIVE FUNCTIONS

Recursive, functions are functions that call themselves, so **a recursion is the invocation of a computation inside a currently executing identical cornputation**. Therefore, a recursive function is one which uses itself in its own definition. If 1 ask, "What is the meaning of meaning?", and you begin to answer that question with the words, "The meaning of meaning is......", you will be in an infinite recursion. You will be using the word in giving its definition. Each time that you use it I can insist that you go back to define it! Each time that you attempt to define the word, you must use it. We will therfore both be trapped in an infinite recursion. Fortunately the recursive procedures of computer science are free (still) of a metaphysical component, and can be used to create elegant algorithms for problems that would be difficult to solve iteratively.

For an illustrative example, consider a function int sum (int n) that returns the sum of the first n integers. If n were 100, it would evaluate the expression:

$$1 + 2 + 3 + \ldots + 98 + 99 + 100.$$

One way to define such a function might be:

sum (n) = n + sum (n - 1);

This call states a self-evident truth: "The sum of the first n integers is n plus the value of a function which sums the first (n - 1) integers." If we knew how to compute sum (n - 1), it would be an easy matter to find sum (n). We would just add n to find the answer. Again,

sum (n - 1) = (n - 1) + sum (n - 2);

To find sum (n - 1) we would want to call sum (n - 2); once we had the result from that call, we would add n - 1, and that would be the value of sum (n - 1)! The call to sum (n - 2) would require a call to sum (n - 3), and so on. Finally we would want sum (1), which is simple: it happens to be 1. Here is the function:

```
int sum (int n)
    {
       if (n = = 1)
        return 1;
       else
       return (n + sum (n - 1));
    }
```

The method works because whenever the function invokes itself it uses an argument smaller than the one in its last call. The value of the function in its final call is defined without self- reference. Recursion provides another example of the divide and conquer technique: to solve a large problem, divide it into successively smaller problems; and solve those to solve the large problem.

Probably the most famous example of recursion is the towers of Hanoi puzzle, in which 64 disks of different radii are piled on a peg. Called SOURCE, in order of increasing radius from top to bottom. Two other pegs are given, namely TEMP and TARGET and the puzzle is to transfer all the disks to TARGET, with two conditions:

i) only one disk can be moved at a time

ii) a disk cannot be placed upon a smaller one.

TEMP is used as a temporary location necessitated by proviso (ii)above while effecting the transfer.

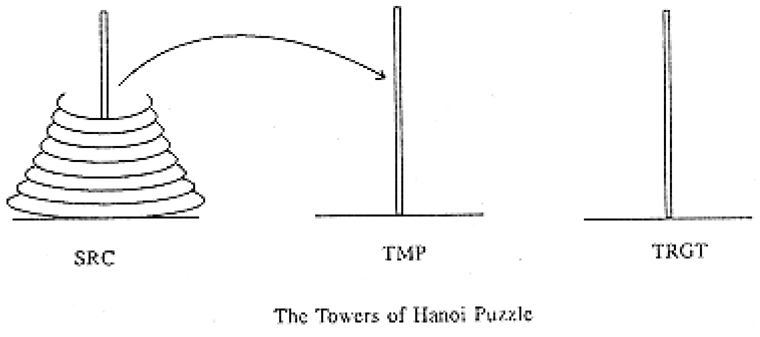


Fig. 1

A recursive algorithm for the tower of Hanoi puzzle with ii disks is easy to state:

Hanoi (n, SOURCE, TEMP, TARGET)

```
        {
                if (n = = 1)
                move disk from SOURCE to TARGET,
          if (n 1)
                {
                /* move n - 1 disks from SOURCE to TEMP, using TARGET */
                Hannoi ((n - 1), SOURCE, TARGET, TEMP);
                move remaining disk from SOURCE to TARGET,
                /* move n - 1 disks from TEMP to TARGET, using SOURCE */
                Hanoi ((n - 1), TEMP, SOURCE, TARGET);
                }
            }
```

The algorithm is implemented in Program 9.1.

```
/* Program 9.1*/
#include <stdio.h
void hanoi (int num-disks, char * SRC, char * TMP, char * TRGT);
void main (void)
        {
                int discs;
            char SOURCE [10], TEMP [10], TARGET [10];
                printf ("How many discs?");
                scanf ("%d", &discs);
                printf ("Source peg is?");
                scanf(" %s", SOURCE);
                printf ("Target peg is? ");
                scanf("%s", TARGET);
                printf ("lntermediate peg is?");
                scanf("%s",TEMP); printf("\n");
            hanoi (discs, SOURCE, TEMP, TARGET),
        }

void hanoi (int num_discs, char * A, char * B, char * C)

        {
                if (num_discs = = 1)
                printf ("Move disc from %s to %s\n", A, C);
          else
                {
                hanoi (num-discs - 1, A, C, B);
                printf ("Move disc from %s to %s\n", A, C);
                hanoi (num_discs - 1, B, A, C);
                }
            }
```

```
    /* Program 9.1: Output */

How many discs? 4
Source peg is? Source
Target peg is? 'Target
Intermediate peg is? Temporary

Move disc from Source to Temporary
Move disc from Source to Target
Move disc from Temporary to Target
Move disc from Source to Temporary
Move disc from Target to Source
Move disc from Target to Temporary
Move disc from Source to Temporary
Move disc from Source to Target
Move disc from Temporary to Target
```

Move disc from Temporary to Source
Move disc from Target to Source
Move disc from Temporary to Target
Move disc from Source to Temporary
Move disc from Source to Target
move disc from Temporary to Target

Recursion is important for computer science because many data structures __ a data structure is an organisation of data with linkages and ordering relationships to achieve a certain objective via a particular procedure __are defined recursively; for example, a tree is a set of nodes that:

i. is either empty; or,
ii. has a designated node called the root from which descend zero or more subtrees.

Realise that the disk subdirectories in PC-DOS or Unix are tree-structured. Naturally the best algorithms to manipulate recursive data structures must themselves be recursive in nature. Iterative procedures for such data structures are often difficult or inconvenient. However, every recursive algorithm can be replaced by an iterative one. Even the Tower of Hanoi puzzle can be solved by iterative methods, and some elegant ones were discovered in the early eighties by P. Buneman and L. S. Levy, and by T. R. Walsh.

Quite different from recursion is the case of chained calls to a function. Suppose we have a function that returns the HCF of two integers a and b. Then, if we wish to use it to find the HCF of three numbers a, b and c, we may invoke it twice in the following way:

```
        hef (hef (a, b). c);
```
The HCF of four numbers is given by the call:

```
        hef (hef (hef (a, b). c)7 d);
```
or equivalently and more simply by
```
        hef (hef (a, b), hef (c, d))
```

The inner calls to hcf () are not recursisic: hcf () does not call itself. It's the programmer who does.so.

Program 9.2 uses Euclid's Algorithm to compute the HCF of two numbers:

```
/* Program 9.2 */
# include < stdio.h
int hef (int a, int b);
void main (void)
        {
         int x, y, z, w, answer;
         printf ("Enter four positive integers:");
         scanf ("%d %d %d %d", &x, &y, &z, &w);
         answer = hef (hef (x, y), hef (z, w));
         printf ("%d\n", answer);
         answer = hef (hef (hef (x, y), z), w);
         printf ("%d\n", answer);
        }

int hef (int p, int q)
        {
         int divisor, dividend, remainder;
         dividend = (divisor = p
         while (remainder = dividend % divisor)
              {
               dividend = divisor;
               divisor = remainder;
              }
                                return divisor.,
          }
```

Check Your Progress 1

Question 1: **int hcf (int p, int q)** is an iterative function. However its definition suggests the following recursive implementation:

```
if (p < q)
            hcf ( p, q )= hcf (q, p);
        else if (p q & & p % q = = 0 )
                    return q;
                else
                    return hcf (q, p% q)
```

Rewrute Origran 9.2 using this recursive definition of **hcf ( )**.

Question 2: Create and execute Program 9.3 below:

```
/* Program 9.3*/
#include < stdio.h
char lifo (char c);
void main (void)
            {   char anykey;
             lifo (anykey = getchar ( ) )
             putchar (anykey);
            }
char lifo       (char C)
            {
             if (C = = '\n')
             return'\0';
             C = getchar ()
             lifo ( C )
             putchar ( C )
            }
```

Can you explain its output?

Question 3: The Fibonacci numbers are most easily evaluated interactively. However, their definition:

fib (n) = fib (n-1) + fib (n-2)

invites a recursive computation, in which unfortunately there are great overheads because of the several superfluous calls. Include the function below in a program to compute **fib (10)**.

```
Int fib    (n)
        {
        if (n = = 1 | | n = = 2 )
                return 1;
        else return (fib (n-1) + fib (n -2));
        }
```

Modify your program to print how many calls are made to the function. The number might surprise you!

Question 4: The C language has no operator for exponentiation. Write a recursive function **power (x,n)** that returns x to the nth power.

Question 5: Write a recursive function **fact (n)** to compute the factorial of a small positive integer.

Question 6: Give the output of the following program:

```
/*                                    Program                                  9.4*/
int                                       x,                                      y;
void func        (void);
void main        (void)
            {
             int                                                                z=1;
             while (x ++ < =9)
                    {
                     printf   ("x   =%d,   y   =%d,   z   =   %d.\n",x,   y,   z,);
                     func                                                        ();
                     y                                                          ++;
                     z                                                          ++;
                     main                                                        ();
                     printf   ("x   =%d,   y   =%d,   z   =   %d.\n",x,   y,   z,);
                    }
                 y--;
                 z                                                              ++;
                 printf   ("x   =%d,   y   =%d,   z   =   %d.\n",x,   y,   z,);
                }

void func        (void)

            {
             static              int              a              =               5;
             int                     b                     =                       5;
             printf      ("a      =%d,      b      =      %d.\n",a,      b,       );
             x                                                                  ++;
             y                                                                  ++;
             a                                                                  --;
             b                                                                  --;
            }
```

9.3 MACROS

We have frequently made use of macro definitions in our programs. In this section we shall study their properties in somewhat greater detail. Typically, a macro definition has the form:

```
#define      name   replacement-string
```

A replacement string in a #define may include other, previously #defined identifiers. In such cases, after the replacement has been made at the appropriate position in the program text. the substitution is scanned again to determine if it contains other #defined identifiers, which are then replaced by their current tokens. A long replacement string of a #define may be continued into the next line by placing a backslash at the end of the part of the string in the current line. The scope of a macro definition is limited to the lines of code which follow below in the same file. Only a #defined token is substituted, but not when it occurs inside a quoted string, nor if it happens to be included in the name of another identifier. For example, consider the definitions:

```
#define h 1

#define c 2

#define n 3
```

Then no replacement will be made in the statements below:

```
int                          hen;

int                          eggs;
```

```
char * fact = "Many hens lay many eggs.";

eggs= hen;
```

The #undef preprocessor directive cancel a previous #definition:

```
#undef h
```

When there are several definitions in a program, it is convenient to gather them together in a "header file" which is #included before main (), as in Program 9.5 below:

```
/* Program 9.5 */
#'include "defs.h"
#include <stdio.h
CONSIDER_THIS_PROGRAM
DO_THE_FOLLOWING

        WRITE_THIS_STRING "Much can be done by #defines!" AND
            WRITE_THIS_STRING "Consider this number:" AND
                WRITE_THE_VALUE_OF LONG_PRODUCT AND
NO_MORE
```

At first glance Program 9.5 doesn't took very much like a C Language program. But the #definitions in defs.h, processed before the other program statements, ensure that the compiler gets to see a compilable program. Here are the contents of the file defs.h:

```
#define DO_THE_FOLLOWING BEGIN
#define BEGIN {
#define WRITE_THIS_STRING printf("%s\n",
#define WRITE_THE_VALUE_OF printf("%U",
#define LONG_PRODUCT 1.23 * 2.34 * 3.45 * 4.56 * 5.67\
                    *6.78 * 7.89 * 8.90 * 9.01

#define AND);
#define NO_MORE END
#define END }
#define CONSIDER_THIS_PROGRAM main ()
```

As we already know, when the preprocessor encounters a #include directive, it replaces the directive itself by the lines in the named file. The contents of the file become part of the contents of' he program itself.

Observe that the file defs.h has been enclosed by double quotes in the #include, rather than by angle brackets as in the case of stdio.h. The double quotes tell the preprocessor to search for the named file first in the directory which contains the source file. If the file is not found there, the system directory is searched. The angle brackets <> instruct the preprocessor to seek the specified file only in the system directory.

#inclusion of files is a very powerful feature of C. It's used typically in situations like the following: suppose a physicist uses the values of constants such as the velocity of light, the charge of the electron, Planck's constant. etc. in her computations. Then, instead of assigning values to them in each of her source programs, she can collect them in one place, say in a file called const.h, and #include this file wherever the values are needed.

#included files may contain more than just #defines: variable declarations or definitions, function prototype declarations and functions may occur in them, too. More than one file may be #included in a program; however if a #included file_A makes reference to quantities defined in another #included file, say file_B, then the #inclusion of file_B must precede that of file_A. More, a #included file may itself contain other #Include directives.

9.4 CONDITIONAL COMPILATION

Some C macros provide the possibility of conditional compilation; because macros are evaluated during preprocessing, with the use of conditionals it is possible to control the compilation process itself. Suppose the physicist programmer wants to avoid multiple #inclusion of her file consts.h. She might write:

```
#ifndef
ELECTRON_CHARGE      /* i.e. if consts.h not #included */
#include "consts.h"


#endif
```

This conditional means:

if (**ELECTRON_CHARGE** has not been #**defined**)

/* (i.e. consts.h hasn't so far been #Included)*/

```
        #include "consts.h"
```

All program lines upto the #endif are processed.

Similarly the conditional #ifdef checks to see if something has already been #defined:

```
#ifdef                                                    VAX
#define                     WORD_SIZE                      32
#else
#ifdef                                                  IBMPC
#define                     WORD_SIZE                      16
#endif
```

These statements assume that there was a previous #definition, one or other of the following:

```
#define VAX
```

or

```
#define IBMPC
```

If you had a program that was dependent on the word size of the host machine, you would want to include one or other of these #definitions before coinpiling it for the target machine. Note that the apparently incomplete statement:

```
#define VAX
```

suffices; it imparts to VAX a nonzero value, though for clarity you might wish to write:

```
#define VAX 1
```

The #if and #elif (else if) preprocessor statements provide more extensive control over compilation. The #if checks to see if its argument, which must be a constant expression, evaluates to a non-zero value: then subsequent lines upto the next #elif or #endif are processed, else they are ignored.

9.5 MACROS WITH PARAMETERS

A macro definition may include parenthetical parameters; the expanded definition_the replacement string_is then written in terms of the parameters:

```
#define                    PRIMES(N)                    N*N-79*N+1601
#define          VOL_CUBOID(A.          B,          C)          A*B*C
#define BIGGER(A. B) A B ? A: B
```

Each occurrence of a formal parameter will be replaced by the actual argument in the program. Thus, with the statements:

```
x = 3,y = 4,z = 5;
volume = VOL_CUBOID(x. y. z);
```

the value assigned to volume will be 60. Such "inline-functions" are convenient to use, of course. But they have disadvantages, too: for example, if you need to find the volumes of cuboids at several different places in your program, each "call" to the macro at those places will translate into a line of code, which will be inserted into your program's text in substitution of the macro. Your executable program could become much longer than if you had used a function, and called it instead each time that you needed to compute a volume. On the other hand, calling a function entails the overhead of passing values, saving return addresses and the like. Using macros may consume space; using functions may consume time. Moreover in using functions you must worry about argument types; will you write different functions for different data types?

Quite apart from considerations of space, time and the labour of writing. two notes of caution are in the use of macros with parameters. First suppose that the sides of our cuboid are increased by 2 units each. The statement:

```
volume = VOL_CUBOID(x + 2, y + 2, z + 2);
```

will not yield the volume of a cuboid of sides 5, 6 and 7: instead, the macro will expand to:

```
volume = x + 2*y + 2*z,+ 2.,
```

giving a result which is quite wrong. You might think that the correct answer will follow if parentheses are used in the macro's #definition:

```
#define VOL-CUBOID(A,B,C) (A)*(B)*(C)
```

This would take care of the present difficulty:

```
volume = (x + 2)*(y + 2)*(z + 2);
```

but it wouldn't work in every situation: if you had an expression:

```
q = 600.0/ VOL-CUBOID (3,4,5);
```

it would translate to:

```
600.0 / (3)*(4)*(5);
```

yielding in answer off by a factor of 20! Correct placement of parentheses is therefore extremely important. in the present instance the proper way to #define VOL_CUBOID would be:

```
#define VOL-CUBOID(A,B,C) ((A)*(B)*(C))
```

Second, consider what might happen if you had a macro to determine the volume of a cube, and you wanted to use it to compute the volumes of two cubes, one of side 17 and the other of side 18 units. In your program you might write:

```
#define VOL-CUBE(X) ((X)*(X)*(X))

int                    side,                    volume;
side                    =                    17;
volume = VOL-CUBE(side);
```

No doubt this would give you tile correct volume for a cube of side 17. But if, for the second cube you wrote:

```
volume = VOL_CUBE(++ side);
```

the Macro processor would in all innocence translate it:

```
volume = ((++ side)*(++ side)*(++ side));
```

side gets incremented thrice. and you will be returned the volume of a cuboid of sides 18, 19 and 20! That wouldn't have happened if you'd used a function instead. Moral: unpleasant side-effects can result if you use the incrementation or decrementation operators inside macros with parameters.

Check Your Progress 2

Question 1: Write a macro to swap the value of two ints X and Y. Under what condition may your macro fail?

Question 2: Write macros:

1)	MIN(X,Y)	to return the lesser of its two arguments
2)	MAX(X,Y)	to return the larger of its two arguments
3)	ABS(X,Y)	to return the absolute value of X - v
4)	C2F(X)	to convert Celsius temperatures to Fahrenheit
5)	F2C(X)	to convert Fahrenheit temperatures to Fahrenheit
6)	M2K(x)	to convert miles to kilometers
7)	K2M(X)	to convert kilometers to mail
8)	LB2KG(X)	to convert pounds to kilogrammes
9)	KG2LB(X)	to convert kilogrammes pounds
10)	L2G(X)	to convert liters to gallons
11)	G2L(X)	to convert gallons to liters

Question 3: Write the following macros: (r stands for redius or base redius, H for the height)

1)	VOL-SPHERE(R)	to compute the volume 0f a sphere
2)	AREA_SPHERE(R)	to Compute the area of a sphere
3)	VOL-CYL(R.H)	to compute the volume of a cylinder
4)	AREA-CYL(R.H)	to compute the area of a cylinder
5)	VOL-CONE(R.H)	to compute the volume of a cone
6)	AREA_CONE(R.H)	to compute the area of a cone

9.6 COMMAND-LINE ARGUMENTS

One of the most useful features of C is that it is possible to pass parameters to a program when it begins executing; moreover, the number of parameters thus passed need not be fixed. Programs can be written to cope with a variable number of parameters supplied at run time.

For a straightforward example, suppose that you've gone grocery shopping and have made purchases at several shops. On returning home you wish to add up the amount you've spent. Fortunately you have a program called add.c, with which all you need do is type the command add and the amounts of your bills at the operating system prompt:

```
add 12.34 23.45 34.56 45.67 56.78 67.89 78.90
```

Promptly the machine responds: 319.59. In this instance the number of parameters (excluding the program name it-self, add) was seven. On another occasion that number may be 17. No matter: C can handle them! The implication is that main () itself is now a function to which values arc passed by you. The driving program is you yourself, and main () the function that you invoke. dynamically passing arguments when doing so. Convenient as they are, command line arguments the regrettable interpretation that the user is herself a program, that passes values to a function!

To write programs with command line arguments, one must provide main() itself with a parameter list, since main () is now a function driven by the user. Two parameters suffice. These are customarily called arguments which is a value of type int; and argy, which is an array of pointers to char. main () is declared as follows:

```
void main (int arge. char * argv [])
```

arge keeps a count of the number of parameters passed, including the program name itself, so arge is at least 1. In the present example, the number of parameters (including the program's name string, add), is eight; that is the value of arge. The first parameter is "add", while the eighth is the quantity "78.90". Each parameter is stored as a separate string in the array of pointers to char, argv []. Thus:.

```
argv                    [0]                    ==                    "add",
argv               [1]               ==               "               12.34",
...........
argv [7] == "78.90"
```

because argv [ ] is an array of string, it could equivalently be declared:

```
char ** argv;
```

Further, it is guaranteed that argv [argc] is the null pointer. Thus argv [8]has the value ()and points nowhere.

One complication that we must deal with straightaway before writing the code for add. e is that the values 12.34. etc., (the amounts of your bills) are stored as alphanumeric strings in argy []. How can one do arithmetic with them? In order to be able to do so. naturally we must first extract their numeric values. The library function atof () (for ascii to float) comes to our aid. This function, declared in stdiib.h , converts its string argument to a double Precision floating point number. Such a function is not terribly difficult to write: it must First look for an optional sign in the numeric string, and then extract each char digit, one by one. To begin with, suppose it finds a digit 'd'. Its ordinal value will be ('d' - '0'). If the next digit encountered is 'e', the number built so far will have the value 10 * ('d' - '0') + ('e' - '0'). proceeding in this way, the number is constructed from the string, digit by digit; for every digit encountered, the number built so far is multiplied by 10, and the last digit found is added in until a decimal point is encountered. Digits after the decimal point are weighted by negative powers of 10. Of course the programming becomes tricky if your function must be able to cope with inputs like -3.14c-53. Luckily for us, atof () can take even such strings in its string. add.e is Program 9.6 below:

```
/*                    Program                    9.6                    */
#include                                                            <stdio.h
#include                                                            <stdib.h
void main (int arge, char * argv [])
          {
                                        int                              i;
                         double          result          =          0;
            for (i = j; i < arge; i ++)
                 result += atof (argv [i]);
                              printf          ("%.2f\n",          result);
             }
```

Besides atof (), the header <stdib.h contains declarations for several functions for number conversion, memory allocation, random number generation and related utilities. atol () converts its string argument to an int value, and atol () returns a long result from a digit string.

For another example of command line arguments consider Program 9.7 below which echoes its string input in reverse order.

```
/*Program                         9.7                         */
#include                                                 <stdio.h
```

```
#include                                                                    <stdib.h
void main (int arge, char * argv [])
            {
             int                                                              i;
             for (i = argc - 1; i 0; i -)
                    printf ("%s ", argv [i]);
                }
```

Check Your Progress 3

Question 1: Write a program that prints the maximum and average of the values typed to it in the command line.

Question 2: Write a program that perform computations incolving addition, subtraction and multiplecation of several numbers typed to it in the command line, in the format;

Compute a op b op c o0p d ....

Where op may be the +, -, or * operators for arithmetic. (A more challenging program is to allow op to include/, the operator for division.)

Question 3: Modify Program 9.7 so that it reverses ever world in the string input to it. therefore for a palidromic input like:

able was i ere i saw elba

the output should be identical to the input.

9.7 VARIABLE-LENGTH ARGUMENT LISTS

It is frequently necessary to deal with situations where the number of arguments in a function's call may be arbitrary. Two common examples are printf () and scanf 0. In this section we shall write a function addem 0 that returns the total of its int arguments, whose number is not defined. The only proviso - one we have chosen for our convenience - is, that the last argument is assigned the value 0; this lets us control the for (;;) loop in which the arguments are added: a zero value for an argument tells the loop where to stop. In Program 9.8 consider first the prototype declaration of the function addem ():

```
int addem (int * how -many, ...);
```

The declaration means that the number and types of arguments may vary; the "three dots" ... declaration can appear only at the end of an argument list. The header <stdarg.h must be #included; it declares a new type va_list; va_list is used to declare a pointer which can be made to point to each of the unnamed arguments subsumed in the declaration ....In addem ( ) we call this pointer vals_ptr.

va start is a macro #defined in <stdarg.h. It initializes vals_ptr to point to the first unnamed argument; therefore va__start must be called before vals_ptr can be must have at least one named argument. We've called this argument int * how many; though we have do use for it in this program, one can put it to a variety of good causes: in printf () and scanf () this argument is the control string. Scanning the control string, char by char, can help locate the format conversion characters, and deduce the number and type(.%) of arguments present after the control string. Each call of va_arg (returns one unnamed argument. and stores it in a local variable called current value:

```
current-value = va-arg (vals-ptr, int);
```

The second argument of va -arg () is a type name. Va-arg ()uses it to determine what type to. return. and how big a step to take to get to the next argument. The for (;;) loop below works alright because we've included a terminating 0 in the list passed to addem()from main();

```
        For(; current-value = va-arg(vals_ptr, int); )
        total += current-value;
```

The final call to va end () from addem () does any house- keeping dust may be required before control is sent back to main ().

```
/* Program 9.8 */
#include <stdio.h
#include <stdarg.h
int addem (int * how-many, ...);
void main (void)
        {
         int sum. * num -args;
         sum = addem (num_args, 1, 2, 3,4,5,0);
         printf ("Sum of arguments was: %d\n", sum);
         sum = addem (num_args, 10, 20,30,40, 50,60, 70, 0);
             printf ("Sum of arguments was: %d\n', sum).,
        }
int addcm (int * how-many. ...)
        {
                int current_value, total = 0;
          va_list vals_ptr;
     /* contd.*/

         va_start (vals_ptr, how_many);
         for (; current_value = va_arg(vals_ptr, int); )
         total += current_value;
         va_end (vals_ptr);
         return total;
        }
```

Check Your Progress

Question 1: Modify Program 9.8 above so that the int pointer how_many conveys the number of arguments passed to addem ( ). (The for (;;) loop of addem () should be processed * how_many times; a zero as the argument list terminator is then not required.)

Question 2: Write a C language program that calls a function max () with a variable length argument list to return the maximum value of the arguments passed to it.

9.8 COMPLICATED DECLARATIONS

How would you declare a function returning a pointer to an array of pointers to float? Beginners often find C declarations a source of confusion, but the process of constructing the correct declaration to suit a particular purpose, and conversely, the correct interpretation of a given declaration is algorithmic, and therefore mechanical. In any declaration, for '*" read "pointer to"; "()" means "function returning"; and "[]' implies "array of. There are two operative rules: (1) the parentheses and subscript operators have a higher priority than the indirection operator, and (2) the closer an operator is to the identifier, the higher is its priority. Consider the two declarations:

```
     char * func-1 0;
     char (* func-2) 0;
```

In the first declaration, the priority of the parentheses operator over the indirection provides the proper interpretation: func_1 () is a function that returns a pointer to char. In the second declaration the parentheses around * func_2 have the highest priority, by rule 2: so func_2 is a pointer to a function returning char. What does this mean? Suppose there is a function of type char called seventh_char () that returns the seventh character of a string. Then the assignment:

```
func_2 = seventh_char,
```

makes func_2 point to seventh_char. The name seventh_char (i.e., the function's name, without parentheses) is precisely the memory address of the function seventh_char ()! In other words, seventh_char is a pointer to the function seventh_char (). Just as the name of an array is a pointer to itself, so also the name of a function is the memory address where the function code begins. Func_2 is declared to be a pointer to any function returning char. It is therefore perfectly legal to assign seventh_char to func_2. Dereferencing func_2 is exactly the same as invoking seventh-char () directly! So the statement:

```
putchar ((* func_2) ("Hello World\n"));
```

will call seventh-char 0 with the argument "Hello World". which will deposit the letter W at the current position of the cursor, as indeed would the statement:

```
putchar (seventh_char ("Hello Wols\n"));
```

Program 9.9 verifies these assertions.

```
/* Program 9.9 */
#include <stdio.h
char seventh-char (char * string);
void main (void)
    {
        char (* func~2)(); /* ptr to fn 0 returning char */
         static char first-String [.1 = "Hcllo World".,
         static char second-String = "How about you?";
           static char third-string "Well, well!";
          static char exclam [1 = "!!!!!!! ",
       func-2 = seventh-char., 1* assigns address of 7th~phar to func-2
          putchar func-2) (first-string)); invokes 7th-char
             putchar func- 2) (second-string));
               putchar func_2) (third_string));
                putctiar func-2) (exclam));
    }

char seventh-char (char * String)

    {
                return (String [6]);
    }
            /* Program 9.9: Output */
                 Wow!
```

When a pointer to a function f(is used as an argument to another function g(, say, then f (), and any other function that returns a pointer like r (, can be executed through g (. So g () can ly.% used as the "envelope" to execute several different functions, if need be, each returning pointers of the same type. This is often a great convenience in advanced level programming.

For another example of a somewhat complicated declaration, let's illustrate the difference between tile two declarations:

```
                int * x [51.

and             int (* y) [51;
```

The first deci#cires x to be an array of pointers to int, by rule 1, since.the subscript operator has a higher priority than the indirection operator; th scond declares Y to be a pointer to an array of ints. Similarly, the declaration:

```
int * ptr [5][6];
```

makes ptr an arrœly of 5 elements because the leftmost subscript operator has the highest priority, by rules 1 and 2. Those clernenlo, must be pointers, because the * operator has bi, rule 2 a hi glier priority than the rightmost subscript operator. So ptr is an array of 5 pointers to a six element array of ints.

For a last example let's look at a more complex declaration:

```
int (* (* y 0) []) 0;
```

This means:

(* (* y ()) []) 0 is an int.,
* (* y () [] is a function returning an int;
(* y () [] is a pointer to a function returning an int;
* y () is an array of pointers to functions retuniing ints;
y () is a pointer to an array of pointers to functions returning int:s;
y is a function returning a pointer to an array of pointers to functions returning ints.

Check Your Progress 4

Question 1: Declare a function of () to return a pointer to an affay of pointers to float.

9.9 DYNAMIC MEMORY ALLOCATION

One of the most useful features of C is that it is possible to allocate memory to a program while it is executing. One does not need to declare in advance an array to store the data that the program will generate, or which will be input to it. Quite often the size of that data is impossible to estimate beforehand, and one must experiment with the most suitable size of array to declare. If you'll leaf back to glance at Program 6.14 for a moment, you will note that we had there declared wi array of dimension 1000 to store all primes (upto ten million) of the type k * k + 1. We had no way of estimaiting the numbers of that type of prime, and we trusted on our luck that a thousand storage locations would suffice. But sometimes one is able to compute quite reliably how much st(jrage will be required as the program proceeds. Then it iriakes sense to obtain blocks of memory on an "as needed" basis. For a concrete illustration we'll go back again to the problem about determining prime numbers. But this time we'll pose the question differently: store all the prime numbers less than a limit N, unknown to the program, which must be scanned from the keyboard when the program begins executing. Let n represent the number of primes less than N. Typically, the user types in some "large" value for N, and the program must generate and hold in memory all prime numbers less than it. 'i'he problem is then to allocate sufficient memory and to address each location of it to store the primes as they are generated:

2, 3, 5, 7, 11, 13 ...... (upto or including N)

How many will they be? One Way to tackle the problem is to declare an array of long ints of some anticipated size, say 50,000, optimistically hoping that so large an array will suffice every time the program is run. The declaration:

```
long primes [500001;
```

does this for us. It sets aside 50000 words of memory. But there may be times when this may be too intich memory (when there are too few primes to store), and times when this may be too little, when n is large. In the first case memory is wasted; in the second, the primes generated will overflow their designatedtrea, and may be written over potentially important code or data !

It so happens, however, that it is possible to estimate the number of primes that will be generated in each run of the program. llere's a lovely theorem (named the Ha(tamard - de la V,illee theorem after its two discoverers) which tells us that for a given N, the approximate (asymptotic) number of primes less than N is the integral:

$$\int_{2}^{N} dx / \ln (x)$$

/* approx number of primes <= N */

Since N is known only at run time, n, the number of primes to be stored, can be found by computing the integral. The number of bytes of storage required will be n * sizeof prime.

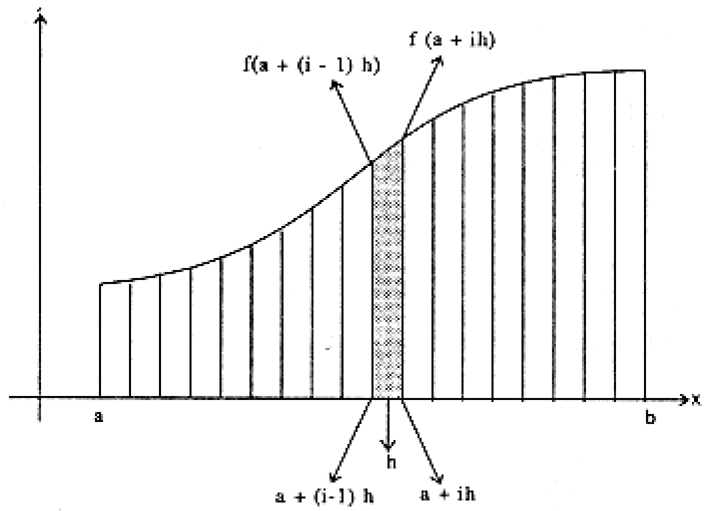
The malloc (and calloc (functions of C are used to dynamically allocate memory. The malloc (n) function has a single argument: the number of bytes to allocate. It returns a pointer to a block of n uninifialised bytes of core, or NULL if it was unable to honour the re- quest. The calloc (num_items, sizeof item) function has two arguments: the number of loca- tions to be reserved, and the size of each in bytes. It returns a pointer to num.items * sizeof item consecutive bytes of memory, which are initialised to zero. If it was unable to honour the request, calloc (returns the NULL pointer.

The pointer returned by calloc 0 or malloc (must be cast to the appropriate type. as in:

```
    primes_ptr = (int *) canoe (num-primes', sizeof (int));
```

(int *) casts the value returned by calloc (to a pointer to int. A call to the function free (ptr) frees the space allocated by a previous call to calloc (or nialloc (, where ptr is the pointer returned by the allocator. It is an error to release memory by a call to free (, that had not pie- viously been allocated by calloe (or mallow (. Blocks may be freed in any order, inde- pendently of the order in which they were requested.

In Program 9.10 the quantity numprimes is computed by the function how - many_primes (. This function uses the trapezoidal rule to integrate the expression 1 1 In (x) from 2 through N. Given that a definite integral of a function can be interpreted as the area underneath the curve between the limits specified, the only thing we have to do is to approximate that area. One way of doing so is to regard the area as composed of a series of strips parallel to the y- axis. Each strip has a curved upper boundary, but we may nevertheless regard each as a trapezium, provided the width of the strips is not too large, and still expect to home in to a reasonable answer by adding the areas of each of these trapezia. If there are n strips, and the limits of the integration are a and h, then the width of any strip is h (h - a) 1 n. The iLh strip has x-values a + (i - 1) * h and a + i * h, and thus its area is 0.5 * h(f (a + (i - 1) * h) + f (a + i * h)). Adding all these strips together gives the formula for the Lrapezoidal rule. The for- mula is: Integral = (0.5 * f (a) + f (a + h) + f (a + 2 * h) +... f (a + (n - 1h) + 0.5 * f (a + it * h)) * h



The trapezoidal rule for the evaluation of $\int_a^b f(x)dx$

Fig. II

This formula is used in how_many_prime (upper_limit, lower limit). The number of primes returned by that function _ num_primes - is passed to calloc (). canoe (com- putes the amount of memory required (numprimes * sizeof int), and returns a pointer to it if it is available, else the NULL pointer:

```
if ((primes,-pLr = (int *) calloc (num_primes, sizeof (int))) == NULL)
```

The program uses two other int pointers, top_ptr and bottom_ptr. primes_ptr as returned by canoe 0 is saved in top_ptr. As primes are generated, each new one that is found is stored at the location currently pointed by priinesptr:

```
* ++ primcs_ptr = new-try;
```

Finally, when all primes upto N have been found, the current value of primes_ptr is saved in bottorn.ptr; the for (;;) loop:

```
for (primes_ptr = top_plr; primes-ptr <= bottom-ptr; primes-ptr ++)
printf ("%d\n", * primes-ptr);
```

lets you look at all these primes as dicy scroll past you on the screen. Note that the com- parison of two pointers is a valid pointer operation.

```
/* Program 9. 10 */
#include <stdio.h
#include <inath.h
#include <stdlib.h
int how_many_primes (int upper_limit, int lower_limit);
void main ()
        {
         int upper_limit, lower_limit = 2, num_primes;
         int * primes_ptr, * top_ptr, * bottom_ptr;
         int first_prime = 2, second_prime = 3, next_try, divisor;
            printf ("This program uses calloc () to\n");
             printf ("create storage for all primes\,n").,
             printf("less than a limit N.EnterN: ");
                  scanf ("%d", &upper_limit);
           num_primes = how_many_primes (upper_limit, lower_limit);
          printf ("Hadamard - de la Valice estimate: %d primes.\n". num_primes);
         if ((primes_ptr = (int *) calloc (num__primes, sizeof (int))) = = NULL)
        {
          printf ("Not enough memory to store %d primes ... \n", num_primes);
                                exit                    (EXIT_FAILURE);
        }
    top_ptr                     =                       primes_ptr;
    *           primes_ptr              =               first_prime;
    *       ++          primes_ptr          =           second_prime;
    for (next_try = second_prime + 2; next_try < = upper_limit; next_try += 2)
        for (divisor = 3; ; divisor +=2)
        {
         if       (next_try       %       divisor       =       =       0)
                                                                  break;
         if (divisor * divisor next_try)
                {
                    *       (++       primes_ptr)       =       next_try;
                                                                  break;
                }
            }
     bottom_ptr                    =                        primes_ptr;
     for  (primes_ptr  =  top_ptr,  primes_ptr  <=  bottom_ptr,  primes_ptr  ++)
                printf          ("%cf\n",           *           primes-ptr);
    }

int how-many-primes (int N, int lower-limit)
            {
                    int                 num_intervals,                  i;
            double        width_of_strip.        end_areas,        area        0.0;
                   if               (N                <=                1000)
                    num_intervals                =                     100;
                                                                       else
                   if               (N                <=               10000)
                                num_intervals           =               800;
                                                                       else
                                if          (N          <=            20000)
                                        num_intervals                  1800;
                                                                       else
                                        num_intervals                  3000;
    width_of_Strip    =    ((double)    (N    -    lower_limit))/    num_intervals;
   end_areas (1.0 / log ((double) lower_limit) +
          1.0 /log ((double) N)) * 0.5 * width_of_strip;
           for      (i      1;      i      <      num_intervals;      i      ++)
          area += 1.0 1 log ((double) lower_limit + i * width_of_strip)
          * width_of_strip;
             return        ((int)        (area        +        end        areas));
    }
```

Program 9.10 is a simple example illustrating how canoe (is called. A place where malloc (would he useful is in a program to balance. a bank account's pass book, whose requirement we briefly described in the last Unit. As each new record is added, we have to set aside more memory for the new data items. The function clieque_issue (below is a "bare t)ones" illustration of matioe (in such situations; it has no error checking, and its external variables would be external to main ().

```
char                        cheque-for-whom                        [80];
float                                                          cheque_amt,
#include                                                        <string.h
#include                                                         <stdio.h
#include                                                        <stdlib.h
void cheque-issue (void)
            {
                    char                    *                       ptr;
            printf         ("Cheque         is         in         name         of:
                gets                (cheque-for                  -whom);
```

```
        printf ("Enter amount being issued:
            scanf ("%f", &cheque-amt);
    ptr = (char *) malloc (strlen (cheque-for-whom));
            strcpy (ptr, cheque-for-whom);
        printf ("%s %.2f", ptr, cheque-amt);
}
```

9.10 SUMMARY

The C langage allows functions to call themselves. If such a recursive function is called, then the copy being executed will call itself, and so on ad infinitum, unless there is a mechanism (usually an if (with a control variable) to terminate the series of calls. Recursive algorithms can often provide elegant solutions where iterative procedures would be cumbersome.

C macros can be written with parameters. Such a "macro function" can be placed anywhere inside a program, where the macro processor replaces it by its #defined expansion in terms of the "arguments" in the call. In contrast to functions, while such macros save the overheads of passing arguments. the fact that they are expanded in situ makes the executable program to become longer than if a function was used instead. The #if, #ifdef, #ifndef, #else and #endif preprocessor statements enable programs to be compiled subject to prescribed conditions.

Functions may be written to accept a variable number of arguments. Moreover, main () too may have parameters, one of which, customarily called arge, is an int; the other, char * argv [] is an array of an arbitrary number of pointers to char. main (is supplied with parameters in programs that must process arguments from the command line. Each argument typed in is stored as a string pointer in argv [], and can be manipulated from there.

For beginners it is sometimes difficult to understand or create complicated declarations. The process of making or analysing a complicated declaration is purely rule-based however, and one can be accomplished in a series of small steps. The calloc () and malloc () functions can be used to dynamically allocate memory; memory obtained in this way can be freed by free (. These functions are accessible by including the header <stdlib.li.