UNIT 9 FUNCTIONS -II

Structure **Page No.**

9.1 INTRODUCTION

The C language imparts to the function concept a reach and power. 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 set 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 maintains addresses and parameter values is again empty. In Sec. 9.2 of this Unit we will look at some simple examples of recursive functions.

In C it is also possible to **#define** 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 function" 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. In Sec. 9.3 of this Unit, we will discuss macros and in Sec. 9.5 of this unit, we will discuss macros with parameters.

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 PC as well as a 64-bit PC. The macro conditionals will enable you to build in that portability inside your program. We will disucss conditional compilation of programs in Sec. 9.4 of this unit.

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 connection 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: `print()` 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` and 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 function 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. With help of memory allocation functions we can allocate memory while it is executing.

Objectives

After studying this unit, you should be able to

- define and use recursive functions;
- use macros as functions;
- write programs that can take command-line arguments;
- write functions with variable-length argument lists;
- create and understand complicated Declarations; and
- do 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 computation**. Therefore, a recursive function is one which uses itself in its own definition. If I 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 therefore 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:

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

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. See Fig. 1 on the following page. 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. A recursive algorithm for the tower of Hanoi Puzzle with **n** 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 */
    Hanoi (( n - 1), SOURCE, TARGET, TEMP);
```

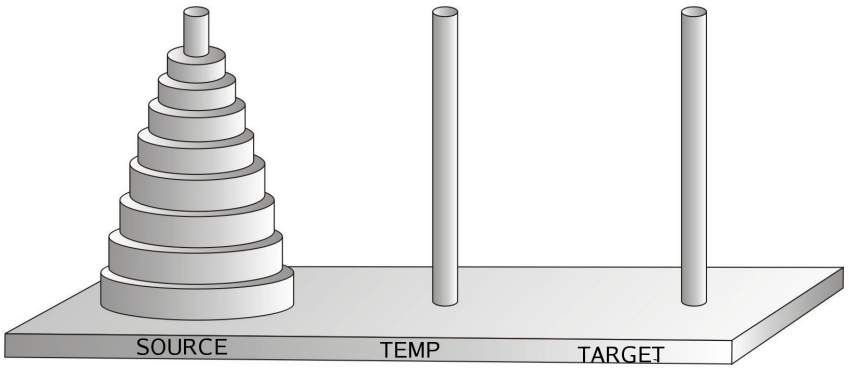

Fig. 1: Tower of Hanoi puzzle.

```
    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; File name unit9-prog1.c */
#include <stdio.h>
void hanoi(int num_disks, char *SRC, char *TMP, char * TRGT);
int main(void)
{
    int discs;
    char SOURCE[10], TEMP[10], TARGET[10];
    printf("How many discs? \n");
    scanf("%d", &discs);
    printf("Source peg is? \n");
    scanf("%s", SOURCE);
    printf("Target peg is? \n");
    scanf("%s", TARGET);
    printf("Intermediate peg is? \n");
    scanf("%s", TEMP);
    printf("\n");
    hanoi(discs, SOURCE, TEMP, TARGET);
    return (0);
}
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–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(also known as GCD) 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:

```
hcf (hcf (a, b), c);
```

The HCF of four numbers is given by the call:

```
hcf (hcf(hcf (a, b), c), d);
```

or equivalently and more simply by

```
hcf (hcf (a, b),hcf(c, d));
```

The inner calls to `hcf()` are not recursive: `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; File name: unit9-prog2.c */
#include <stdio.h>
int hcf(int a, int b);
int main(void)
{
    int x, y, z, w, answer;
```

```
    printf("Enter four positive integers: ");
    scanf("%d %d %d %d", &x, &y, &z, &w);
        answer = hcf(hcf(x, y), hcf(z, w));
    printf("%d\n", answer);
    answer = hcf(hcf(hcf(x, y), z), w);
    printf("%d\n", answer);
    return (0);
}
int hcf(int p, int q)
{
    int divisor, dividend, remainder;
    dividend = (divisor = p < q ? p : q) == p ? q : p;
    while (remainder = (dividend % divisor)) {
        dividend = divisor;
        divisor = remainder;
    }
    return divisor;
}
```

Here are some exercises for you.

E1) `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);
```

Rewrite Program 9.2 using this recursive definition of `hcf()`.

E2) Create and execute Program 9.3 below:

```
1  /* Program 9.3; File name: unit9-prog3.c */
2  #include <stdio.h>
3  char lifo(char c);
4  int main(void)
5  {
6      char anykey;
7      lifo(anykey = getchar());
8      putchar(anykey);
9      return (0);
10 }
11 char lifo(char C)
12 {
13     if (C == '\n')
14     return '\0';
15     C = getchar();
16     lifo(C);
17     putchar(C);
18     return (0);
19 }
```

Can you explain its output?

E3) The Fibonacci numbers are most easily evaluated iteratively. 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!

E4) The C language has no operator for exponentiation. Write a recursive function `power (x, n)` that returns x to the nth power.

E5) Write a recursive function `fact (n)` to compute the factorial of a small positive integer.

E6) Give the output of the following program:

```
1  /* Program 9.4; File name: unit9-prog4.c */
2  #include<stdio.h>
3  int x, y;
4  void func(void);
5  void main(void)
6  {
7      int z = 1;
8      printf("Entering main().\n");
9      while (x++ <= 3) {
10         printf("x = %d, y = %d, z = %d.\n",x, y, z);
11         func();
12         y++;
13         z++;
14         main();
15         printf("x = %d, y = %d, z = %d.\n", x, y, z);
16     }
17     printf("Finally...:\n");
18     y--;
19     z++;
20     printf("x  = %d, y = %d, z = %d.\n", x, y, z);
21 }
22 void func(void)
23 {
24     static int a = 5;
25     int b = 5;
26     printf("Entering func().\n");
27     printf("a = %d, b = %d,\n", a, b);
28     x++;
29     y++;
30     a--;
31     b--;
32 }
```

We close this section here. In the next section, we will discuss how to create C macros, what are their advantages etc.

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 **#define**d 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 **#define**d 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 **#define**d 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 e 2

#define n 3
```

Then, no replacements 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 **#define** statement:

```
#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_THIS_VALUE_OF_LONG_PRODUCT AND  NO_MORE
```

At first glance Program 9.5 doesn't look 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(''\%f\n'',
#define LONG_PRODUCT 1.23 * 2.34 * 3.45 * 4.56 * 5.67 * 6.78
#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 the 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 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 `consts.h`, and **#include** this file wherever the values are needed.

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

Suppose we are working on program that will run on a variety of machines with different word sizes. How can we ensure that the program compiles correctly on all the machines? This can be achieved using macros that facilitate conditonal compilation, as we will see in the next section.

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 the 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 **#include**d) */

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

All program lines up to the **#endif** are processed.

Similarly the conditional **#ifdef** checks to see if something has already been **#define**d:

```
#ifdef PC_32BIT
#define WORD_SIZE 32
#else
#ifdef PC_64BIT
#define WORD_SIZE 64
#endif
```

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

```
#define PC_32BIT
```

or

```
#define PC_64BIT
```

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 compiling 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 up to the next **#elif** or **#endif** are processed, else they are ignored.

In Unit 2, we briefly discussed macro definitions. We also saw how a function can be defined using a macro. In the next section, we will discuss macros with parameters that enable us to define functions as macros.

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 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 order 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 an 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 the 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 .

E7) Write a macro to swap to values of two **int**s X and Y. Under what conditions may your macro fail?

E8) 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 - Y
4)	C2F(X)	to convert Celsius temperatures to Fahrenheit
5)	F2C(X)	to convert Fahrenheit temperatures to Celsius
6)	M2K(x)	to convert miles to kilometres
7)	K2M(X)	to convert kilometres to miles
8)	LB2KG(X)	to convert pounds to kilogrammes
9)	KG2LB(X)	to convert kilogrammes to pounds
10)	L2G(X)	to convert litres to gallons
11)	G2L(X)	to convert gallons to litres

E9) Write the following macros: (R stands for radius or base radius, H for the height)

1)	VOL_SPHERE(R)	to compute the volume of a sphere
2)	AREA_SPHERE(R)	to compute the are of a sphere
3)	VOL_CYL(R, H)	to compute the volume of a cylinder
4)	AREA_CYL(R, H)	to compute the area f 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

Before graphical user interfaces were created, all the programs have to run from the command line. Even now, many programs are run from the command-line. In the next

section, we will see how to create programs that can take arguments from the command line.

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 <CR>
```

Promptly the machine responds: 319.59. In this instance the number of parameters (excluding the program name itself, `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 are 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 force 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 `argv`, which is a value of type **int**; and `argc`, which is an array of pointers to **char**. `main()` is declared as follows:

```
void main (int argc, char * argv [])
```

`argc` keeps a count of the number of parameters passed, including the program name itself; so `argc` 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 `argc`. The first parameter is "`add`", while the eight is the quantity "78.90". Each parameter is stored ad 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 strings, it could equivalently be declared:

```
char ** argv;
```

Further, it is guaranteed that `argv [argc]` is the null pointer. Thus `argv [8]` has the value 0, and points nowhere.

One complication that we must deal with straightaway before writing the code for add.c is that the value 12.34, etc., (the amounts of your bills) are stored as alphanumeric strings in `argv`[]. 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 `<stdlib.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 `-3.14e-53`. Luckily for us, `atof()` can take even such strings in its stride. `add.c` is Program 9.6 below:

```
/* Program 9.6; File name: unit9-prog6.c */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    int i;
    double result = 0;
    for (i = 1; i < argc; i++)
        result += atof(argv[i]);
    printf("%.2f\n", result);
    return (0);
}
```

Besides `atof()`, the header `<stdlib.h>` contains declarations for several functions for number conversion, memory allocation, random number generation and related utilities. `atoi()` 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; File name: unit9-prog7.c */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    int i;
    for (i = argc - 1; i > 0; i --)
        printf("%s", argv[i]);
    printf("\n");
    return (0);
}
```

You have to input each letter separated by space. If you want to enter a space precede it by `\␣`, i.e. a backslash followed by a space. For example, if you want to enter the string "A man", you will have to enter it as `A␣\␣␣man`.

Here are some exercises for you to try.

E10) Write a program that prints the maximum, minimum and average of the values typed to it in the command line.

E11) Modify Program 9.7 so that it reverses every word in the string input to it. Therefore for a palindromic input like:

```
able was i ere i saw elba
```

the output should be identical to the input.
(**Hint:** Use `strlen()` function discussed in Unit 8.)

Earlier, we had mentioned that `scanf()` is a C library function. You would have noticed that in all the functions we have come across, with the exception of functions

like scanf() and printf(), we knew before hand the number of arguments of the functions. How can we create functions like scanf() and printf() which can have any number of arguments. We will discuss this in the next section.

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(). In this section we shall write a function addem() 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 **#include**d; 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 **#define**d in <stdarg.h>. It initialises vals_ptr to point to the first unnamed argument; therefore va_start must be called before vals_ptr can be used. va_start **must** have at least one named argument. We've called this argument **int** * how_many; though we have no 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(s) 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 that may be required before control is sent back to main().

```
/* Program 9.8; File name: unit-prog8.c */
#include <stdio.h>
#include <stdarg.h>
int addem(int *how_many, ...);
int 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);
    return (0);
}
int addem(int *how_many, ...)
```

```
{
    int current_value, total = 0;
    va_list vals_ptr;
    va_start(vals_ptr, how_many);
    for (; current_value = va_arg(vals_ptr, int);)
        total += current_value;
    va_end(vals_ptr);
    return total;
}
```

Here are some exercises for you.

E12) Modify Program 9.8 above so that 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.)

E13) 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.

So far, we have seen functions that have variables of some type of other, **char**, **int** etc, as arguments. Suppose we want to write a function that takes another function as an argument, how can we do it? For example, if we want to write a function that integrates functions or differentiates functions ? How can we we declare such a function? We will discuss these issues in the next section.

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.
2) The closer an operator is to the identifier, the higher is its priority.

Consider the two declarations:

```
char * func_1 ();
char (* func_2) ();
```

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() 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 World\n"));
```

Program 9.9 verifies these assertions.

```
/* Program 9.9; file name: unit9-prog9.c */
#include <stdio.h>
char seventh_char(char *string);
int main(void)
{
    char (*func_2) (); /* ptr to fn () returning char */
    static char first_string[] = "Hello World";
    static char second_string[] = "How about you?";
    static char third_string[] = "Well, well!";
    static char exclam[] = "!!!!!!!!";
    /* assigns address of 7th_char to func_2 */
    func_2 = seventh_char;
    putchar((*func_2) (first_string));  /*invokes 7th_char */
    putchar((*func_2) (second_string));
    putchar((*func_2) (third_string));
    putchar((*func_2) (exclam));
    return (0);
}
char seventh_char(char *String)
{
    return (String[6]);
}
```

```
/* Program 9.9:  Output */
Wow!
```

Why do we need the parentheses around *func_2:

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

If they were absent, because of lower priority of the dereferencing operator as compared to the parentheses operator, the expression would be equivalent to de-referencing the result returned by the "function call":

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

The result is not an address!

To recapitulate, the assignment:

```
func_2=seventh_char;/*no parentheses*/
```

assigns the address of the seventh_char to func_2. If parentheses are included in the assignment:

```
func_2 = seventh_char ();/*Wrong*/
```

the compiler will flag an error because the assignment attempts to associate the value returned by the function seventh_char – a char – with a pointer to a function returning char(the definition of func_2): a type mismatch error.

Similarly, the assignment

```
func_2 = & seventh_char ();/*Wrong*/
```

is invalid. `seventh_char` represents a function call(because the priority of the parentheses is higher than that of the & operator), and the assignment attempts to extract the address of the returned value, an rvalue.

Likewise, the assignment

```
func_2 = & seventh_char;
```

is also suspect. The function name `seventh_char` is already an address, so the expression attempts to extract and assign the address of an address; in other words, `func_2` is forced the erroneous interpretation of a pointer to a pointer to a function. Pre-ANSI compilers may report and error, but ANSI compliant compilers allow this usage: they ignore the address-of operator in this context.

> **Note**
>
> Dereferencing a function pointer is equivalent to making a function call. Thus
>
> ```
> putchar ((*func_2) (first_string));
> ```
>
> is equivalent to
>
> ```
> putchar (seventh_char (first_string));
> ```
>
> According to ANSI standard, the usage
>
> ```
> putchar (func_2 (seventh_char));
> ```
>
> is also acceptable, but the first form in which the pointer is explicitly dereferenced is preferable because older compilers may not support this syntax.

Include the following instruction in your program and compile it with any ANSI compiler.

```
putchar (func_2(first_string));/*in ANSI C*/
```

You will find that it is equivalent to

```
putchar (seventh_char (first_string));
```

The reason is that while `func_2` is a pointer to the function `seventh_char()`, `*func_2) ()` is a function call itself. In ANSI C we can treat `func_2` as if it were a function – as in `func_2(first_string)` – or we can explicitly dereference the pointer as in `(*func_2)(first_string)`.

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 `f()`, can be executed through `g()`. So `g()` can be 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 the two declarations:

```
int * x [5];
```

and

```
int (* y) [5];
```

The first declares `x` to be an array of pointers to **int**, by rule 1, since the subscript operator has a higher priority than the indirection operator; the second declares `y` to be a pointer to an array of **int**s. Similarly, the declaration:

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

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

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

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

This means:

(*(* y()) []) () 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 returning ints;
y () is a pointer to an array of pointers to functions returning ints;
y is a function returning a pointer to an array of pointers to functions returning ints.

E14) Declare a function f() to return a pointer to an array of pointers to **float**.

In next section, we will see how to allocate memory dynamically as and when needed by the program.

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 an array of dimension 1000 to store all primes (up to ten million) of the type $k^2 + 1$. We had no way of estimating 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 storage will be required as the program proceeds. Then it makes 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. The 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, ....(up to or including N)
```

How many will they be? One way to tackle the problem is to declare an array of **long int**s 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 [50000];
```

does this for us. It sets aside 50000 words of memory. But there may be times when this may be too much 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 designated area, 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. There's a lovely theorem (named the Hadamard - de la valle theorem after its two discovers) which tells us that for a given N, the approximate (asymptotic) number of primes less than N is the integral:

$$\int_2^N \frac{1}{\ln x} \quad /* \text{ approx number of primes} \leq N \; */$$

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 uninitialised bytes of core, or NULL if it was unable to honour the request. The `calloc (num_items, sizeof item)` function has two arguments: the number of locations to be reserved, and the size of each in bytes. It returns a pointer to `num_items * sizeof item` consecutive byte 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()` or `malloc()` must be cast to the appropriate types, as in:

```
primes_ptr = (int *) calloc (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 `malloc()`, where `ptr` is the pointer returned by the allocator. It is an error to release memory by a call to `free()`, that had not previously been allocated by `calloc()` or `malloc()`. Blocks may be freed in any order, independently of the order in which they were requested.

In Program 9.10 the quantity `num_primes` is computed by the function `how_many_primes()`. This functions uses the trapezoidal rule to integrate the expressions `1 / ln (x)` from 2 through N. Given that a definite integral of a function cab be interpreted as the are 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 b, then the width of any strip is `h = (b - a) / n`. The ith 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 trapezoidal rule. See Fig. 2 on the next page.

The formula is:

$$\text{Integral} = (0.5f(a) + f(a+h) + f(a+2h) + \cdots + (a+(n-1)h) + 0.5f(a+nh))h$$

This formula is used in `how_many_primes (upper_limit, lower_limit)`. The number of primes returned by that function–`num_primes`–is passed to `calloc()`. `calloc()` computes the amount of memory required `(num_primes * sizeof int)`, and returns a pointer to it if it is available, else the NULL pointer.

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

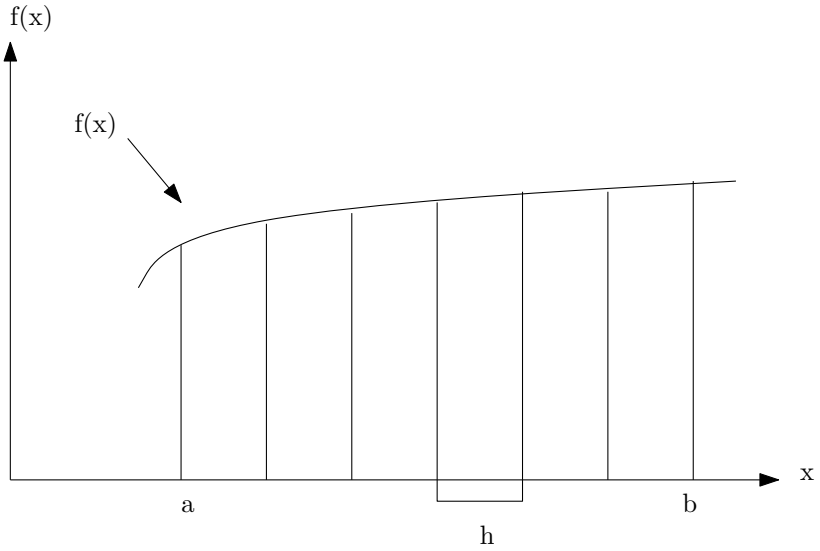

Fig. 2: Area under a curve.

```
/* Program 9.10; file name:unit9-prog10.c */
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
int how_many_primes(int upper_limit, int lower_limit);
int 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. Enter N: ");
    scanf("%d", &upper_limit);
    num_primes = how_many_primes(upper_limit, lower_limit);
    printf("Hadamard - de la Valle 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);
        return (0);
    }
    top_ptr = primes_ptr;
    *primes_ptr = first_prime;
    *++primes_ptr = second_prime;
    /* Find all the primes less than N and
       store them in primes_ptr. */
    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;
            }
        }
    /* Make bottom_ptr point to the largest prime
       found */
    bottom_ptr = primes_ptr;
```

```
    for (primes_ptr = top_ptr; primes_ptr <= bottom_ptr; primes_ptr++)
        printf("%d\n", *primes_ptr);
    return (0);
}
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 / log((double) lower_limit + i * width_of_strip))
            * width_of_strip;
    return (area + end_areas);
}
```

Listing 1: A Program to illustrate the use of `calloc()`

The program uses two other **int** pointers, `top_ptr` and `bottom_ptr`. `Primes_ptr` as returned by `calloc()` returned in `top_ptr`. As primes are generated, each new one that is found is stored:

```
*(++primes_ptr) = next_try;
```

Finally, when all primes up to N have been found, the current value of `prime_ptr` is saved in `bottom_ptr`; the **for**`()` loop

```
for (primes_ptr = top_ptr; primes_ptr <= bottom_ptr; primes_ptr++)
    printf("%d\n", *primes_ptr);
```

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

Program 9.10 in Listing 1 on the facing page is a simple example illustrating how `calloc()` is called. A place where `malloc()` would be useful is in a program to balance a bank account's pass book, whose requirements 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 `cheque_issue()` below is a "bare bones" illustration of **malloc**() in such situations; it has no error checking, and its external variables would be external to `main()`.

```
/* Program cheque issue */
char cheque_for_who_ever[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_who_ever);
    printf("Enter amount being issued: ");
    scanf("%f", &cheque_amt);
    ptr = (char *) malloc(strlen(cheque_for_who_ever));
```

```
    strcpy(ptr, cheque_for_who_ever);
    printf("%s %.2f", ptr, cheque_amt);
}
```

We complete this unit by recapitulating our discussion in the next section.

9.10 SUMMARY

The C language 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 bye its **#define**d 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 `argc`, 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.h>`.

9.11 SOLUTIONS/ANSWERS

E1) See Listing 2.

```
#include<stdio.h>
int hcf(int p, int q);
int main()
{
    int p,q;
    printf("Enter two numbers separated by space:\n");
    scanf("%d %d", &p, &q);
    printf("%d",hcf(p,q));
    return (0);
}
int hcf(int p, int q)
{
    if(p < q)
        return(hcf(q,p));
    else if(p >= q && p % q == 0)
            return (q);
```

```
    else{
        return(hcf(q,p % q));
    }
}
```

Listing 2: Solution to exercise 1.

E2) Suppose you type 'abc' and press <CR>. This is what the program will do:

1) Save the character in `anykey` and calls `lifo()` with the argument `'a'`. The control reaches line no. 12 of the program. Since this test for the carriage return character fails, the line `C = getchar()` is executed and `C` gets the value `'b'`. The function `lifo()` is called for the second time with the value `'b'`.

2) In the second call, the control reaches line no. 12. Again the the test for carriage fails. The function `lifo()` is called for the third time with the value `'c'`.

   The third time, since we typed the carriage return after `'c'`, the test in line 12 is passed and the control returns to the second call of the function at line 16, from where the function was called for the second time. The rest of the function is executed. Remember that, in the second call, the value of `C` is `'c'` and this is printed by line number 17. After this, the control returns to the first call of `lifo()` and in this call, the value of `C` is `'b'` and the instruction in line 17, prints this value. Finally, control returs to the main program at line 8. The value of `anykey`, which is `'a'` is printed.

So, the program prints the characters we have typed in reverse order!

E3) The program is in Listing 3.

```
#include<stdio.h>
int no_of_calls =0;
int fib(int n);
int main ()
{
    printf("fib(10) is %d\n",fib(10));
    printf("No of calls for fib(10) is %d",no_of_calls);
    return (0);
}
int fib(int n)
{
    no_of_calls++;
    if (n == 1 || n == 2)
        return (1);
    else
        return(fib(n-1)+fib(n-2));
}
```

Listing 3: Solution to exercise 3.

E4) See Listing 4.

```
/*Answer to exercise 4. File name:unit9-ans-ex-4.c*/
#include<stdio.h>
int power(int x,int n);
int main()
{
    int x, n;
    printf("Enter integers x and n.\n");
    printf("n should be positive.\n");
    scanf("%d %d", &x, &n);
    printf("%d",power(x,n));
    return (0);
}
```

```
int power (x,n)
{
    if (n == 1)
        return(x);
    else
        return(x*power(x,n-1));
}
```

Listing 4: Solution to exercise 4.

E6) The output, with line numbers added, is given below:

```
/*Output from program 9.4*/
1.  Entering main().
2.  x = 1, y = 0, z = 1.
3.  Entering func().
4.  a = 5, b = 5,
5.  Entering main().
6.  x = 3, y = 2, z = 1.
7.  Entering func().
8.  a = 4, b = 5,
9.  Entering main().
10.  Finally...:
11.  x = 5, y = 3, z = 2.
12.  x = 5, y = 3, z = 2.
13.  Finally...:
14.  x = 6, y = 2, z = 3.
15.  x = 6, y = 2, z = 2.
16.  Finally...:
17.  x = 7, y = 1, z = 3.
```

1) The variables x and y which are external to `main()` are automatically initialised to 0. The program starts execution from `main()`. The value of the automatic variable z is initialised to 0. The first `printf()` statement prints the line "`Entering main().`". The control enters the **while**`()` loop since x is 0. After the loop condition is tested, x is post incremented to 1. The values x=1, y=0 and z=1 are printed in the **second** line of the output..

2) The control enters `func()` with `x = 1`, and `y = 0`. The `printf()` statement in line 26, prints the line "`Entering func().`" as the **third** line of the output. The `printf()` statement in line 27 prints the values of a, b as the **fourth** line; both the values are 5. Inside `func()` the values of x and y are incremented to 2 and 1 respectively. The values of a and b are decremented to 4.

3) The control now returns to line 12 in the **first** call to `main()`. The values of y and z are incremented to 2 and 2, respectively.

4) With these values, `main()` is called again in line 14 for the **second** time. x and y retain their values but the value of z becomes 1 again. The line "`Entering Main:`" is printed again in the **fifth** line. The **while**`()` loop condition is true and the value of x is incremented to 3. The **sixth** line in the output, x=3,y=2,z=1 is printed.

5) The control now reaches `func()`. Since a is declared as a **static** variable, it retains the value 4 it had earlier. But, the value of b reverts back to 5. The `printf()` statement in 26 is printed in the **seventh** line of the output. Then, `printf()` statement in line 27 prints the values of a and b as 4 and 5 in the

eighth line of the output. The values of x and y are now incremented to 4 and 3, respectively and the values of a and b are decremented.

6) The control returns to line 12 in the **second** call to `main()`. Again, the values of y and z are incremented to 4 and 2, respectively.

7) The function `main()` is called for the **third** time in line 14. The value of z reverts to 1. The `printf()` statement in line 17 prints the statement `"Entering main()."` as the **ninth** line of output. This time, the control doesn't enter loop since `x` is 4, but its value is incremented to 5. The `printf()` in line 17 prints the output `"Finally ...:"` as the **tenth** line of the output. In lines 18 and 19, the values of `y` and `z` are incremented to 3 and 2 respectively. The `printf()` statement in line 20 prints the values of `x`, `y` and `z` as 5, 3 and 2 in the **eleventh** line of output.

8) The control now returns to line 14 in the **second** call of `main()`. The value of `z` before the third call was 2 and this value and the values of `x` and `y` are printed by line 15 as **tweleveth** line of output. Once again, the loop condition fails, but the value of `x` becomes 6. Then, the `printf()` statement in line 17 prints `"Finally ...:"` as the **thirteenth** line of output. The value `y` is decremented to 2 and the value of `z` is incremented to 1. These values are printed by line 20 as the **fourteenth** line of output.

9) Now, the control returns to line 14 in the **first** call of `main()`. The value `z=2` before the second call of `main()` is picked up now. The values of `x`, `y` and `z` are now printed by line 15 as the **fifteenth** line of output. The loop condition is once again and `x` is incremented to 7. The `printf()` statement in line 17 is prints `"Finally...''` as **sixteenth** line of output. The value of y is decremented to 1 in line 18 and the value of z is incremented to 3 in line 19. The `printf()` statement in line 20, prints the values of `x`, `y` and `z` as the **seventeenth** line of output.

E7) `#define SWAP(X,Y) X = X - Y; Y = Y + X; X = Y - X;` This will fail if we write `SWAP(X++.Y++)`.(Where will the problem arise?)

E11) Here is the program.

```
/* Program 9.3; File name: unit9-prog3.c */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
char reverse(char *C);
int main(int argc, char *argv[])
{
    int i,n;
    for (i = argc - 1; i > 0; i --){
        n = strlen(argv[i])-1;
        for(;n >= 0;n --)
            printf("%c",argv[i][n]);
        printf(" ");
    }
    printf("\n");
    return (0);
}
```

Listing 5: Answer to exercise 11.

E12) See Listing 6.

```
#include <stdio.h>
#include <stdarg.h>
int addem(int how_many, ...);
int main(void)
```

```
{
    int sum, num_args;
    sum = addem(6, 1, 2, 3, 4, 5, 6);
    printf("Sum of arguments was: %d\n", sum);
    sum = addem(8, 10, 20, 30, 40, 50, 60, 70, 10);
    printf("Sum of arguments was: %d\n", sum);
    return (0);
}
int addem(int how_many, ...)
{
    int current_value, total = 0,i;
    va_list vals_ptr;
    va_start(vals_ptr, how_many);
    for (i = 1;i <= how_many ;i++){
        current_value = va_arg(vals_ptr, int);
        total += current_value;
    }
    va_end(vals_ptr);
    return total;
}
```

Listing 6: Solution to exercise 12.

E13) See Listing 7.

```
#include <stdio.h>
#include <stdarg.h>
int max(int how_many, ...);
int main()
{
    int maximum;
    maximum = max(6, 1, 2, 3, 4, 5, 6);
    printf("Max of arguments was: %d\n", maximum);
    maximum = max(8, -11, -20, -30, -40, -50, -60, -70, -2);
    printf("Max of arguments was: %d\n", maximum);
    maximum=max(1,-5);
    printf("Max of arguments was %d.",maximum);
    return (0);
}
int max(int how_many, ...)
{
    int current_value, maximum = 0,i;
    va_list vals_ptr;
    va_start(vals_ptr, how_many);
    maximum=va_arg(vals_ptr, int);
    if(how_many == 1)
        return (maximum);
    for (i = 2;i <= how_many ;i++){
        current_value = va_arg(vals_ptr, int);
        if(current_value > maximum)
            maximum = current_value;
    }
    va_end(vals_ptr);
    return (maximum);
}
```

Listing 7: Solution to exercise 13.

E14) **float** `*(*f())[]`

`(*f())` is an array of pointers to **float**.

`f()` is a pointer to an array of pointers to **float**.

`f` is a function returning a pointer to an array of pointers to **float**.