## Block 2

### PROGRAMMING IN C

<table>
<thead>
<tr>
<th>UNIT</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Control Structures-II</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>Pointers and Arrays</td>
<td>37</td>
</tr>
<tr>
<td>8</td>
<td>Functions</td>
<td>63</td>
</tr>
<tr>
<td>9</td>
<td>Functions-II</td>
<td>93</td>
</tr>
<tr>
<td>10</td>
<td>Files and Structs, Unions and Bit-fields</td>
<td>119</td>
</tr>
</tbody>
</table>
Curriculum Design Committee

Dr. B.D. Acharya                Prof. O.P. Gupta             Prof. C. Musili
Dept. of Science & Technology  Dept. of Financial Studies  Dept. of Mathematics and Statistics
New Delhi                      University of Delhi          University of Hyderabad

Prof. Adimurthi                Prof. S.D. Joshi             Prof. Sankar Pal
School of Mathematics           Dept. of Electrical Engineering  ISI, Kolkata
TIFR, Bangalore

Prof. Archana Aggarwal          Dr. R. K. Khanna            Prof. A.P. Singh
CESP, School of Social Sciences  Scientific Analysis Group  PG Dept. of Mathematics
JNU, New Delhi

Prof. R. B. Bapat               Prof. Susheel Kumar         Prof. Parvin Sinclair
Indian Statistical Institute, New Delhi  Dept. of Management Studies  Prof. Sujatha Varma
 Prof. M.C. Bhandari             IIT, Delhi                  Dr. S. Venkataraman
Dept. of Mathematics            Prof. Veni Madhavan     
IIT, Kanpur                      Scientific Analysis Group  
 Prof. R. Bhatia                 DRDO, Delhi
Indian Statistical Institute, New Delhi  Prof. J.C. Mishra
 Prof. A. D. Dharmadhikari      Dept. of Mathematics       
Dept. of Statistics             IIT, Kharagpur
University of Pune

Course Design Committee

Prof. C.A. Murthy                Faculty Members
ISI, Kolkata                     School of Sciences, IGNOU
Prof. S.B. Pal                    Dr. Deepika
IIT, Kharagpur                   Prof. Poornima Mital
Dr. B.S. Panda                    Dr. Atul Razdan
IIT, Delhi                       Prof. Parvin Sinclair
Prof. C.E. Veni Madhavan         Dr. S. Venkataraman
IISc, Bangalore

Block Preparation Team

Prof. Sachin B. Patkar (Editor)  Dr. S. Venkataraman
Dept. of Electrical Engineering  School of Sciences
IIT, Mumbai                      IGNOU

This Block is a modified version of Block 3 of CS-04.

December 2007

©Indira Gandhi National Open University, 2007


All rights reserved. No part of this work may be reproduced in any form, by mimeograph or any other means without written permission from
the Indira Gandhi National Open University.

Further information on the Indira Gandhi National Open University courses may be obtained from the University’s office at Maidan Garhi,
New Delhi-110 068.

Printed and Published on behalf of Indira Gandhi National Open University, New Delhi, by Director, School of Sciences.
PROGRAMMING IN C

In this Block, in continuation with the first block, we study some more control structures. As you know, control structures control the program flow. In the first Unit, we will discuss two more control structures, the *for(;;)* and the binary *if-then-else* structure.

We will discuss pointers and arrays in the second Unit of this block. Pointers are closely related to arrays. So, it is not surprising that they are useful in working with arrays. How pointers are not only useful in manipulation of arrays; we will use them in the next block where we discuss data structures like linked lists, stacks and queues.

Functions in C allow us to design our programs in a modular fashion. We will devote two units in this block for the discussion of functions. In these two units, we will see how to declare and use functions. We will also discuss recursive functions, a feature that enables us to develop elegant programs for many problems that are recursive in nature. We will also discuss the scope of variables in functions.

We will also discuss arrays in greater detail in the third and fourth units. We will see how they are convenient for storing homogeneous data like marks of students, etc. In mathematics, we use arrays for storing matrices, vectors, coefficients of polynomials, etc.

Real world problems can lead to large programs. Since it is difficult to maintain such programs in a large single file, the program is usually split up into many files. To be able to do so, we should be able to control the scope of variables in different files; we have to decide, for example, which variables in file A should be accessible in file B and which should not be accessible and vice-versa. This is discussed in third unit of this block. We will also see how to allocate memory dynamically when the program is running; this will avoid problems arising from allocating too much or too little memory for data.

In scientific computing, we often generate data for analysis. But, we will be able to analyse data only if we can save the data in a file. In the last Unit of the block, we will discuss some functions of C that enable us to work with files. We will also discuss *struct* and *union*, two data structures that allow us to work with heterogeneous, but related data in a single variable. We will also discuss bit manipulation in this Unit. Such techniques are especially useful in graphics programming.
### 6.1 INTRODUCTION

There are two ways in which the linear flow of control may be altered in computer programs: one involves alternate paths provided by if ()-else or switch statements; the other is through the repetitive execution of a set of statements. The first mechanism is called a branch, the second a loop. We’ve seen examples of both in the last unit. In this unit we will tour through the most common applications of loops: iterations in mathematical calculations, and the processing of groups of related data items called arrays. We will introduce you to the for loop in Sec. 6.2. In Sec. 6.3, we will then discuss the simplest kind of arrays, the unidimensional arrays. Unidimensional arrays are similar to vectors. Indeed, we will use them to represent vectors. If we have a simple variable, we saw that we can declare and initialise it in one statement. However, initialising arrays is not that simple. In Sec. 6.4, we will see how to initialise arrays.

We have seen that C program variables fall into four types; in addition, they have an associated attribute called storage class, which for any variable determines two things:

1) When the variable comes into existence in an executing program and when it ceases to exit;
2) The blocks (and, where relevant, files) of code in which the variable may be referenced.

This “domain of visibility” is called the scope of the variable. This unit ends with examples of the application of scope rules.

### Objectives

After studying this unit, you should be able to

- write programs using the for (; ;) loop;
- create and make use of one dimensional arrays;
- use the sizeof operator; and
- apply the register, auto and static storage class rules.

### 6.2 THE for (; ;) LOOP

In this section, we introduce you to the for (; ;) loop. As we mentioned in the introduction, one of the most common applications of the for (; ;) loop is for processing arrays. What is an array? Vectors and Matrices are examples of arrays. The
array concept enables a group of data to share a common name. Each datum in the
group—called an array **element**—is distinguished from other data by the assignment of a
unique number, called the array **subscript**, to each datum. The compiler assigns a block
of memory to store (at consecutive locations) the elements of the array. To access a
particular element, we can use its subscript. To illustrate how this could be useful, let us
take the following situation: A company has 8000 employees and it has to pay a bonus
of 10% to each of its employees. We can declare an array of 8000 elements called
employees, with each element of the array holding the wage of an employee. The wage
of each employee can be accessed using the subscript of the data. The task of finding
10% of the wages of employees is a repetitive task. We have to repeat the same set of
statements to find 10% of a number on each of the 8000 elements. So, we use a loop,
with the subscript of the array elements as loop index. In each iteration of the loop, we
access the wage of one employee and process it to find the bonus using the same set of
statements.

For processing an array, we will see that it is convenient to use a **for** loop although we
can manage quite often with a **while** loop that we discussed in Unit 5.

The **for** (; ;) loop is at once the most versatile and the most popular of the three loop
structures of C, because it contains information about loop entry, control and re-entry
all in one place. The **for** (; ;) statement evaluates three expressions, which are
separated by two semicolons in the parentheses following the keyword **for**. The
semicolons are a syntactical requirement for the **for** loop; the three expressions are not;
any or all of them may be absent. The general **for** (; ;) loop has therefore the
following appearance:

```c
for (first_expr; second_expr; third_expr)
    the loop body, a simple or compound statement;
```

The statements within the loop's body are executed repetitively when the loop is
entered.

Each component expression of the **for** (; ;) statement has a particular (but not
essential) purpose. Most often, **first_expr** is an assignment expression that is used
to initialise the loop index: **first_expr** is evaluated before the loop is entered.
Though it is customary to initialise the loop index via **first_expr** it is not a
requirement: as in the **while**() loop, the loop index may be initialised by a preceding
statement; in any case, **first_expr**, if it is present, is evaluated first, before the loop
is entered.

Entry into the loop is determined by the truth or falsity of **second_expr**, which is a
Boolean. The loop is entered only if the Boolean is **true**, else it's skipped. In either
case, whether the loop is entered or not, **first_expr** is computed. (If
**second_expr** is absent, it's regarded as **true**, and the loop body is executed
anyway.)

After a loop execution, control is transferred to evaluate **third_expr**, which is used
to re-initialise the loop index. Then **second_expr** is re-examined. The loop is
re-entered if the Boolean is still **true**.

For example:
```c
for (i = 3; ;)
    printf("%d\n", i);
```
sets i = 3 in the loop body. But
```c
i = 3;
```
printf("%d\n", i);

does this just as well. In each case, the printf() statement is executed forever, because an absent Boolean(second_expression) is regarded as true. You would have noticed that the third_expression is also missing. C syntax does not require that third_expr should be present. Note well that the for (;;) loop repeatedly executes the single (simple or compound) statement which follows it; on occasion that statement may be the null statement; your logic may be such as to require it. But, more often, placing a semicolon immediately after a for (;;) or a while () may be plain carelessness. The outcome of this may not be what you want. You have been warned! Be on your guard!

The second of the expressions in a for(;;) statement is a Boolean; the loop is entered if the Boolean evaluates to true; not otherwise. If first_expr assigns such a value to the loop index that second_expr is false, the loop is not entered.

The printf () below can never be executed:

```c
for (i = 3; i == 5;)
    printf("%d\n", i);
```

Since i is given the value 3 before the loop is entered, and since the condition for entry is i = 5, which is false, the loop is skipped, and i is not printed. However, the printf () in the next loop will be printed infinitely often:

```c
for (i = 3; i = 1;)
    printf("%d\n", i);
```

The third expression within a for (;;) is evaluated after each execution of the loop’s body. If the loop condition, namely second_expr is still found to be true after its last iteration, the loop’s statements are executed again; third_expr re-initialises the loop index after each iteration; generally it increments the variable initialised by first_expr; but it may change (increase or decrease) the index by any amount. second_expr determines whether this is new value of the loop index is such as to allow a further iteration of the loop statement.

The printf () in the following loop will be executed 2 times:

```c
for (i = 3; i < 5; i++)
    printf("%d\n", i);
```

Initially, i is 3. The Boolean i < 5 is true; the loop is entered, and the current value of i is printed. Then the loop index i is re-initialised, so that it gets the value 4. The Boolean is found to be still true(i < 5); the loop body is executed once more. Next the third_expr in the for (;;) statement sets i to 5; the Boolean becomes false; the loop is not entered again.

Since first_expr is evaluated only once, it’s often used to print an introductory remark before loop entry. See the program in Listing 4 on page 9. In the program in Listing 1 on the next page we use the for (;;) loop to sum the integers from 1 to 100; initially, i is assigned the value 1, and sum is 0; the Boolean i < 101 is true to begin with, so i is added into sum. Then i is post-incremented in the re-initialisation part of the for (;;) loop; the Boolean is tested again and found to be true, and the loop statement is therefore executed once more. The process is repeated until i = 101, at which time the Boolean i < 101 becomes false.

However, almost every program that we write using for(;;) can be written using while(). You will find this out if you try exercise 7 of this Unit. If we want to use a
Programming in C

/∗ Program 6.1; File name: unit6-prog1.c ∗/
#include <stdio.h>
int main()
{
    int i, sum = 0;
    for (i = 1; i < 101; i++)
        sum += i;
    printf("The sum of the numbers from 1 to %d is %d\n", i - 1, sum);
    return (0);
}

Listing 1: Example of a for(;;) loop.

/∗ Program 6.2; File name: unit6-prog2.c ∗/
#include <stdio.h>
int main()
{
    int i, sum = 0;
    for (i = 1; i < 101; sum += i++)
        printf("The sum of the numbers from 1 to %d is %d\n", i - 1, sum);
    return (0);
}

Listing 2: for(;;) loop with empty body.

while() instead of for(;;) we can do so by putting the Boolean condition in the round brackets after while and carry out incrementation within the body of the while loop. We can put the initialisation statements before the body of the while loop.

In the program in Listing 2 below, we use the for (;;) property that the loop re-initialisation is performed after each execution of the loop body, which in this case consists of the null statement: Program in Listing 3 depends on the for (;;) property that the Boolean is evaluated each time before the loop is entered.

/∗ Program 6.3; File name: unit6-prog3.c ∗/
#include <stdio.h>
int main()
{
    int i, sum = 0;
    for (i = 1; i < 101 && (sum += i++));
        printf("The sum of the numbers from 1 to %d is %d\n", i - 1, sum);
    return (0);
}

Listing 3: for(;;) loop with empty third_expression.

Here are some exercises to check your understanding.

E1) Consider the following fragment we saw earlier. What will it print?
    for (i = 3; i = 1;)
        printf("%d\n", i);

E2) State the output of the programs in Listing 4, Listing 5, and Listing 6 on the facing page.

E3) Replace the for (;;) statement in the program in Listing 1 above by:
    for (i = 1, sum = 0; i < 101; i++)
/* Program 6.4; File name:unit6-prog4.c */
#include <stdio.h>
int main()
{
    int j = 5;
    for (printf("Will this loop be entered?\n"); j < 2; j = 1)
        printf("Print this, if the loop is entered...\n");
    printf("Was the loop entered?\n");
    return (0);
}

Listing 4: Exercise 2.

/* Program 6.5; File name:unit6-prog5.c */
#include <stdio.h>
int main()
{
    int j = 5;
    for (printf("Will this loop be entered?\n"); j < 6; j = 7)
        printf("Print this, if the loop is entered...\n");
    printf("Is the Boolean j < 6 still true ?\n");
    j < 6 ? "Yes." : "No.");
    printf("because j is now %d\n", j);
    return (0);
}

Listing 5: Exercise 2.

/* Program 6.6; File name: unit6-prog6.c */
#include <stdio.h>
int main()
{
    int j;
    for (j = 0; j < 2; j++) {
        printf("The for (;;) loop is really easy to use.\n");
        printf("I've said this %s.\n", j == 0 ? "once" : "twice");
    }
    printf("What I say %d times must be true!\n", j);
    return (0);
}

Listing 6: Exercise 2.

/* Program 6.7; File name: unit6-prog7.c */
#include <stdio.h>
int main()
{
    int i, sum_odd = 0, sum_even = 0;
    for (i = 1; i <= 100; i++)
        i % 2 ? (sum_odd += i) : (sum_even += i);
    printf("The sum of the even numbers and odd numbers from 1 to 100 is \nsum_even, sum_odd); 
    return (0);
}

Listing 7: Sum of odd and even integers from 0 to 100.
Realise that the comma operator can be used with the expressions controlling a for (;;) statement. That in fact is its most common usage.

Observe that the value of the loop index is retained on exit from the loop.

E4) In the program in Listing 3 on page 8, can the && operator be replaced by the || operator? Can it be replaced by the comma operator? Are the parentheses around sum += i ++ required?

E5) The program in Listing 7 on the previous page uses the if - then - else operator to sum the odd and even integers from 1 through 100 separately. Why are the parentheses in the if - then - else operator required?

E6) Create, compile and execute programs in Listing 1—Listing 7 above, replacing for (;;) statements by while() statements.

E7) Rewrite the programs for the Russian Peasant Multiplication Algorithm and the Collatz problem using for (;;) loops.

Let’s work through the program Listing 8 below to quickly recapitulate what we’ve learnt about the for (;;) loop.

```c
/* Program 6.8; File name: unit6-prog8.c */
#include <stdio.h>
int main()
{
    int a, b, c;
    for (b = 0; b++ < 2; c = b++)
        a = b;
    printf("a = %d, b = %d, c = %d\n", a, b, c);
    for (a = b = 0, c = 6561; a += b, c /= 3; b++, c /= 3)
        b++;
    printf("a = %d, b = %d, c = %d\n", a, b, c);
    for (a = b = c = 0; a = (b++ < 5) && c++ <= 6; a = b++, c++)
        printf("a = %d, b = %d, c = %d\n", a, b, c);
    for (a = 0, b = 2; ++b < 20 - (c = b / 2 > 5 ? -a : ++a);
        b += a;
        printf("a = %d, b = %d, c = %d\n", a, b, c);
    return 0;
}
```

Listing 8: A quick recap.

In the first for (;;) loop b is initialised to 0. The Boolean b ++ < 2 is true, the loop statement is executed so that a gets the current value of b, 1. In the re-initialisation, b is post-incremented to 2, so c gets the value 1. In the second round of execution, the evaluation of the Boolean post-increments b to 3. The Boolean itself is now false, so the loop body is not again executed. a, b and c are assigned the values 1, 3 and 1 respectively.

In the next for (;;) loop, note that a and b are each 0 to begin with, while c is 3^8. In both the testing as well as the re-initialisation phase the value of c is decreased by a factor of 3; and the loop is exited when c becomes 0. Before the loop is first entered, c is reduced to 3^7; b is post-incremented to 1 in this iteration of the loop; then in the re-initialisation, it’s again post-incremented, while c is reduced to 3^6. Before the loop is re-entered, a gets the value 2, and c becomes 3^5. The process continues, until b has the value 8, and a the value 20.

In the third for (;;) loop, the Boolean post-increments b and c each to 1, and is itself true, so a gets the value 1, and the loop is entered. The printf() prints the current values of a, b and c. Before the Boolean is tested for the next round of execution, b
and c are each post-incremented to 2, while a remains at 1. The Boolean is still true, but b and c are post-incremented to 3 and a is reassigned the value 1. In the next reinitialisation, a gets the value 3, while b and c are each post-incremented to 4. When the Boolean is tested with these values of b and c, a is reset at 1, while b and c are post incremented to 5. These values are printed. The re-initialisation of b and c makes the Boolean false for the next execution of the loop’s body.

It is left as an exercise for you to show that on execution of the last loop a = 3, b = 17 and c = 3.

In a for (;;) loop, the continue statement re-directs control to immediately re-evaluate third_expr (followed, in the usual way, by a re-evaluation of second_expr). Loop execution continues if second_expr remains true. The break statement forces control out of the for (;;) loop.

In the next few examples we shall apply the loop and other control structures to digging out some interesting properties about prime numbers. How can you tell whether a number is prime or not? Pretty simple: divide it by 2, 3, ... and if it’s divisible by any, it’s composite (that’s what mathematicians call numbers that are not prime). Of course, after you’ve tested it for divisibility by 2, you need use only odd numbers for further divisors. (Why?) And what’s the largest divisor that you need try before you can conclude that the given number was a prime? The logic of Program 5.11 teaches us that if a number is composite, it has at least one divisor no greater than its square root (why?): so, after having verified that the given number is not even (because if it’s even it’s certainly composite, unless it’s 2), we need merely use the series of divisors

\[ 3, 5, 7, \cdots \left(\text{int}\sqrt{n}\right) \]

to determine if it is a prime. The program below does just that:

```c
/* Program 6.9; File name: unit6-prog9.c */
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int number, factor;
  printf("Enter a number, I'll tell you if it's a prime:");
  scanf("%d", &number);
  if (number == 2 || number == 3) {
    printf("%d is a prime ...\n", number);
    exit(0);
  }
  else if (number % 2 == 0) {
    printf("%d is composite...\n", number);
    exit(0);
  }
  for (factor = 3;; factor += 2)
    if (number % factor == 0) {
      printf("%d is composite...\n", number);
      break;
    } else if (factor * factor > number) {
      printf("%d is a prime...\n", number);
      break;
    }
  return (0);
}
```

Listing 9: Program to check if a number is a prime.

All prime numbers after 5 must end in one of the four digits 1, 3, 7 and 9. Are such types of prime number equally common? Let’s find out for the primes less than the limit of unsigned int. Listing 10 on the next page is a modified version of Listing 9;
Programming in C

the `switch` statement traps each kind of prime.

```
#include <stdio.h>
int main()
{
    unsigned limit, number, factor, endigit1 = 0;
    unsigned endigit3 = 0, endigit7 = 0, endigit9 = 0;
    printf("Enter number upto which test will run:\n");
    do {
        printf("Value entered must be greater \n than 7, not greater than 65533:");
        scanf("%u", &limit);
    } while ((limit < 7) || (limit > 65533)); /*end do */
    for (number = 7; number <= limit; number += 2)
        for (factor = 3;; factor += 2)
            if (number % factor == 0)
                break;
            else if (factor * factor > number) {
                switch (number % 10) {
                    case 1:
                        endigit1++;
                        break;
                    case 3:
                        endigit3++;
                        break;
                    case 7:
                        endigit7++;
                        break;
                    case 9:
                        endigit9++;
                        break;
                }
                break;
            }
    printf("Primes upto %u ending with 1 = %u\n", limit, endigit1);
    printf("Primes upto %u ending with 3 = %u\n", limit, endigit3);
    printf("Primes upto %u ending with 7 = %u\n", limit, endigit7);
    printf("Primes upto %u ending with 9 = %u\n", limit, endigit9);
    return (0);
}
```

Listing 10: Program to find the number of primes ending 3, 5 and 7.

Here is the output of the program for all primes less than or equal to 65533:

```
Enter number up to which test will run
value entered must be greater 7, less than 65533:
65533
Primes up to 65533 ending with 1 = 1625
Primes up to 65533 ending with 3 = 1645
Primes up to 65533 ending with 7 = 1647
Primes up to 65533 ending with 9 = 1622
```

A problem related to that of determining if a given number is prime is that of listing its prime factors. Program 6.11 reads an `int` value from the keyboard, and factorises it into
primes. Let’s recollect some common sense about decomposing a number into its factors before delving into its logic:

A number is a factor of another if it divides the latter without remainder, that is:

\[
\text{if } (\text{number} \mod \text{factor} == 0) \quad \text{then search has been successful;}
\]

The same factor may be repeated several times within a number. If a factor is found, it should be tested over and over again to see if it divides the last quotient found:

\[
\text{while } (\text{search for a factor has been successful}) \\
\text{test if factor divides last quotient;}
\]

The only even prime factor that a number may have can be 2; after casting out all factors of 2, an odd quotient must be the result. This quotient can have only odd factors, if any. Therefore, beginning with 3, other factors, if any, can only be generated by:

\[
\text{factor += 2} \\
\text{/* if last factor was odd */}
\]

With these preliminaries out of the way let’s look at Listing 11:

```c
/* Program 6.11; File name: unit6-prog11.c */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int number, quotient, factor, no_pfactors = 1;
    printf("Enter a number, I'll print its prime factors:");
    scanf("%d", &number);
    quotient = number;
    if (number == 2) {
        printf("2 is a prime. The factors are 1, 2.");
        exit(0);
    }
    printf("Prime factors with multiplicity are: 1");
    /* cast out factors of 2, if any */
    while (quotient % 2 == 0) {
        no_pfactors++;
        printf("", %d", 2);
        quotient /= 2;
    }
    /* only odd factors can remain, if any */
    for (factor = 3; factor <= quotient; factor += 2)
    {
        while (quotient % factor == 0) {
            no_pfactors++;
            printf("", %d", factor);
            quotient /= factor;
        }
        if (no_pfactors == 2)
        {
            printf("\nThe number is prime.");
            printf("\n");
            return (0);
        }
    }
    printf("\nThe number is prime.");
    printf("\n");
    return (0);
}
```

Listing 11: Program to find all the prime factors of a number.

Let us now discuss the program in detail. The lines 10 to 13 dispose the case of 2. Lines 16 to 20 check if the number is divisible by 2. If it is, it removes 2 from the factors, keeping track of the prime factors removed via the statement no_pfactors++. Then, lines 22 to 27 remove the odd prime factors if any, again keeping track of number factors using the statement no_pfactors++. Lines 28 and 29 check if the number of prime factors is 2 and prints the output ‘The number is a prime.’ if it is.

Try the following exercises now to check your understanding of our discussion.
E8) Execute Program 6.11 for various values of input.

E9) There are two shortcomings in the program in Listing 11 on the preceding page. One is, if the number is prime, the program will divide by all the numbers up to the number. Another is, if the power of the largest prime factor \( q \) dividing the number is one, the program will check all the factors up to \( q \) while it should be enough to check all the factors up to \( \sqrt{q} \). For example, if the number is 142 = \( 2 \times 71 \), the loop will check all the factors up to 71 while it is enough to check till 9. Fix these two shortcomings.

One of the most important uses of loops in computer programs is in what is known as iteration, which is the repeated execution of a set of statements aimed at generating successively “better” values of the quantity that is sought.

For a simple example of the method, let’s solve the quadratic equation:

\[
x^2 + 6.3x - 2.7 = 0
\]

We can write this equation as

\[
x(x + 6.3) = 2.7
\]

or

\[
x = \frac{2.7}{x + 6.3}
\]

Writing the unknown \( x \) in terms of itself as we have done may not seem like a very clever thing to do, but you will agree that the method can become interesting if we write

\[
x_{n+1} = \frac{2.7}{x_n + 6.3}
\]

Setting a first guess for \( x_0 \), we can determine \( x_1 \); having found \( x_1 \), we can substitute it back in the formula to find \( x_2 \), and the process may be carried on until the absolute value of the difference between successive values of \( x \) has become as small as we want; but the process need not always converge; what if the equation had complex roots? Therefore it’s a good idea to set a limit on the number of iterations to be performed, so that you don’t get “caught in loop”, going round and round in circles looking for a root that isn’t there! The program in Listing 12 on the next page below uses the mathematics library `<math.h>` function `fabs()`, which returns the absolute value of a floating point expression. Control breaks out of the for (;;) loop if the difference between two successive values found for \( x \) is less than \( TOL \), or if the number of iterations has exceeded 10. The file `<math.h>` comes with the C compiler, `#include` it whenever you need to use any of the functions listed in Table 6.1. (Your compiler may provide a few more). Here is the output of Program 6.12

One root of eqn. is: 0.40282.

Convergence achieved after 4 iterations.

Convergence to the desired root may not always quite so rapid as in the last example, and in some cases you may be misled into believing, after having performed a large number of iterations that a real root does not exist, when in fact there may be one.

There is a way of checking whether an iterative process to solve

\[ x = F(x) \]
/* Program 6.12; File name:unit6-prog12.c */
#include <stdio.h>
#include <math.h>
#define TOL .00001
int main()
{
  int i;
  double x_zero = 1.0, x_one;
  for (i = 0;; i++) {
    x_one = 2.7 / (x_zero + 6.3);
    if (fabs(x_one - x_zero) <= TOL) {
      printf("One root of eqn. is: %10.5f\n", x_one);
      printf("Convergence achieved after %d \n iterations.\n", i);
      break;
    } else if (i > 9) {
      printf("No convergence after 10 iterations.\n");
      break;
    }
    x_zero = x_one;
  }
  return (0);
}

Listing 12: An iterative method to find the roots of an equation.

will converge: it will, provided the absolute value of the first derivative of F(x) in the vicinity of the desired root is less than 1.

Here are some exercises. Try them to see if you have understood the use of loops.

E10) Modify the program in Listing 9 on page 11 to verify the claim that the numbers generated by the formula:

\[ \text{number} = n^2 - n + 41 \]

for all \( n \) ranging from 1 through 40 are primes.

Repeat for:

\[ \text{number} = n^2 - 79n + 1601 \quad [1 \leq n \leq 79] \]

E11) Execute the program in Listing 11 on page 13 for various inputs and verify that it works satisfactorily. Now modify it to print all the factors of a number input to it, instead of only its prime factors. For example, with 28 as input, it should list the factor set as 1, 2, 4, 7, and 14. 28 is an example of a perfect number, one in which the sum of all the factors of the number equals the number itself. A number is called abundant if the sum of its factors exceeds the number, deficient if this sum is less than the number. Modify your program to determine if a number is abundant, perfect or deficient.

E12) Write a C program to determine how many zeros there are at the end of the number \( 1000! = 1 \times 2 \times 3 \times \cdots \times 1000 \)

E13) Verify that nearly 1300 iterations are required to find one root of the equation:

\[ x^2 - 6.0x + 8.99999 = 0 \]

correct to six places of decimals.
Find one root of 

\[ 3x - \sqrt{1 + \sin(x)} = 0 \]

in the neighbourhood of 0.4.

All arguments \( x, \ y \) in the following are assumed to be \textbf{double}; \( n \) is an \textbf{int}. All functions return \textbf{double}. The expressions in Table 1, \texttt{int * exp}, \texttt{double * ip}

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sin(x) )</td>
<td>sine of ( x ), ( x ) in radians</td>
</tr>
<tr>
<td>( \cos(x) )</td>
<td>cosine of ( x ), ( x ) in radians</td>
</tr>
<tr>
<td>( \tan(x) )</td>
<td>tangent of ( x ), ( x ) is radians</td>
</tr>
<tr>
<td>( \text{asin}(x) )</td>
<td>inverse sine of ( x ), (-1.0 \leq x \leq 1.0)</td>
</tr>
<tr>
<td>( \text{acos}(x) )</td>
<td>inverse cosine of ( x ), (-1.0 \leq x \leq .0)</td>
</tr>
<tr>
<td>( \text{atan}(x) )</td>
<td>inverse tangent of ( y/x ), value in ([-\pi/2, \pi/2])</td>
</tr>
<tr>
<td>( \text{atan2}(y/x) )</td>
<td>inverse tangent of ( y/x ), value in ([-\pi, \pi])</td>
</tr>
<tr>
<td>( \sinh(x) )</td>
<td>hyperbolic sine of ( x )</td>
</tr>
<tr>
<td>( \cosh(x) )</td>
<td>hyperbolic cosine of ( x )</td>
</tr>
<tr>
<td>( \tanh(x) )</td>
<td>hyperbolic tangent of ( x )</td>
</tr>
<tr>
<td>( \exp(x) )</td>
<td>exponential function</td>
</tr>
<tr>
<td>( \log(x) )</td>
<td>natural logarithm of ( x, x &gt; 0 )</td>
</tr>
<tr>
<td>( \log10(x) )</td>
<td>base 10 logarithm of ( x )</td>
</tr>
<tr>
<td>( \text{pow}(x, y) )</td>
<td>( x ) raised to the power ( y )</td>
</tr>
<tr>
<td>( \sqrt{x} )</td>
<td>square root of ( x )</td>
</tr>
<tr>
<td>( \text{ceil}(x) )</td>
<td>smallest integer not less than ( x )</td>
</tr>
<tr>
<td>( \text{floor}(x) )</td>
<td>largest integer not greater than ( x )</td>
</tr>
<tr>
<td>( \text{fabs}(x) )</td>
<td>absolute value of ( x )</td>
</tr>
<tr>
<td>( \text{idexp}(x, n) )</td>
<td>( x \times 2^n )</td>
</tr>
<tr>
<td>( \text{frexp}(x, \text{int * exp}) )</td>
<td>returns the mantissa of a double value and stores the characteristic as a power of 2 in ( \text{exp} )</td>
</tr>
<tr>
<td>( \text{modf}(x, \text{double * ip}) )</td>
<td>returns the positive fractional part of its first argument. Stores integral part in ( \text{ip} )</td>
</tr>
<tr>
<td>( \text{fmod}(x, y) )</td>
<td>floating point remainder of ( x/y ), with the same sign as ( x ).</td>
</tr>
</tbody>
</table>

are pointers, discussed in a later Unit.

In the next section we will introduce you to arrays.

### 6.3 UNIDIMENSIONAL ARRAYS

It is frequently necessary in computer programs to define several related variables of the same type. For example, suppose that the marks of forty students of a class must be sorted in descending order, and printed. You can easily see that to declare 40 \texttt{int} variables—\texttt{mrks_1, mrks_2, mrks_3, ..., mrks_40}—to hold each student’s marks, and to assign values to them, and then compare each against the other to do the sorting, will surely make for a most cumbersome program. It will be easier to sort manually than to write a program with forty variables to do so! And what if you had not 40 but 400,000 students appearing in an examination (not unusual in our country) and had to sort their marks? Then manual procedures wouldn’t be feasible, either. Might it not be possible to declare 40 or (400,000) \texttt{ints} together, in one go, by a single statement, e.g.

```c
int marks [40];
```

that would reserve 40 consecutive storage locations en bloc, each location containing one student’s marks? Indeed it is: such a data structure is called an \textbf{array}, and the
individual values in the consecutive locations which comprise it are called elements of the array. The number of elements in an array is called its dimension. Each of the set of forty marks can be assigned to array elements, one student’s marks per element, so that there’s a one-one correspondence between array elements and marks.

The marks of the \textit{ith} student will then be accessible in the \texttt{int marks [i]}, where the index (or subscript) \texttt{i}—an integer—begins at 0 for the first location, and has consecutive values for consecutive array elements. In C, in contrast to FORTRAN, array indices begin at 0. Incrementing the index \texttt{i} changes the object of reference from the current element to the next. By ranging over all values in the interval \([0 - 39]\), the index \texttt{i} can enable access to any of the 40 elements of the array. So that in a program for the sort, instead of comparing two independent, unrelated \texttt{int} variables, say \texttt{mrks_7} against \texttt{mrks_8}, one could compare the value of \texttt{marks [i]} against \texttt{marks [j]}, for any \texttt{i} and any \texttt{j}.

As you will have realised, the array data structure is a powerful concept, useful wherever a collection of similar data items are to be stored or manipulated.

The C declaration
\begin{verbatim}
int marks [40];
\end{verbatim}
reserves 40 consecutive storage locations to hold 40 variables of type \texttt{int}. See the illustration below. Though the addresses of the array elements are only illustrative, note that they increase in steps of four bytes, the width of \texttt{int} in all the 32-bit machines.

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Array Element</th>
<th>Contents of array element</th>
</tr>
</thead>
<tbody>
<tr>
<td>6700</td>
<td>marks [0]</td>
<td>77</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>6728</td>
<td>marks [7]</td>
<td>53</td>
</tr>
<tr>
<td>6732</td>
<td>marks [8]</td>
<td>32</td>
</tr>
<tr>
<td>6736</td>
<td>marks [9]</td>
<td>81</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>6856</td>
<td>marks [39]</td>
<td>64</td>
</tr>
</tbody>
</table>

\begin{footnotesize}
\textbf{Table 2: The array \texttt{marks [40]}, consisting of 40 consecutive \texttt{ints}}
\end{footnotesize}

Program in Listing 13 on the next page illustrates how values are read in into the array \texttt{marks [ ]}. These marks are then averaged (\texttt{avg}), and the numbers of students who received marks greater than or equal to the average mark (\texttt{abv_avg}) is printed, as well as the number of those who obtained below average marks (\texttt{bel_avg}). Finally the array is sorted to arrange the entries in descending order. The declaratory statement of Program 6.13
\begin{verbatim}
int marks [40];
\end{verbatim}
declares \texttt{marks[ ]}; to be an \texttt{int} array of dimension 40. The \texttt{for (;;)} loop which follows immediately is used to read in a value into each of these elements. The variable \texttt{i} begins at 0 (since array subscripts start from 0) and runs up to 39 to cover all the forty student’s marks. But because we are more used to counting from 1 onwards, the \texttt{printf()} is written to print student’s numbers from 1 through 40. Note in the \texttt{scanf()} that the ampersand is prefixed to the array element name just as it is in the case of scalar variables. In the next unit we shall exploit the connexion between arrays and pointers to write this expression more elegantly.

The \textbf{interchange sort} is the simplest of all sorting algorithms. It uses two \texttt{for (;;)} loops. The first picks out, turn by turn, each of the forty elements of \texttt{marks [ ]}. In the
/* Program 6.13; File name:unit6-prog13.c */
#include <stdio.h>

int main()
{
    int marks[40];
    int i, j, temp;
    float total = 0.0, avg;
    int abv_avg = 0, bel_avg = 0;
    for (i = 0; i < 40; i++) {
        printf("Enter marks of student No. %d : ", i + 1);
        scanf("%d", &marks[i]);
    }
    for (i = 0; i < 40; i++)
        total += marks[i];
    avg = total / 40;
    for (i = 0; i < 40; i++)
        marks[i] >= avg ? abv_avg++ : bel_avg++;
    for (i = 0; i < 40; i++)
        for (j = i+1; j < 40; j++)
            if (marks[i] >= marks[j])
                continue;
            else {
                temp = marks[i];
                marks[i] = marks[j];
                marks[j] = temp;
            }
    printf("In descending order marks are:\n\n");
    for (i = 0; i < 40; i++)
        printf("%d\n", marks[i]);
    printf("The average mark is %f\n", avg);
    printf("%d students obtained above average\n\n", abv_avg);
    printf("%d students obtained below average\n\n", bel_avg);
    return (0);
}

Listing 13: An example using array.

second loop, the element picked out is compared with each of the elements that follow it in the array. If a larger element is found, their values are interchanged.

As has been remarked before, a good design principle to bear in mind is to use variable names such as num_students to hold quantities likely to change in different runs of the program (instead of constants such as 40 as we have introduced in Program 6.13). That way, if the number of students is different from 40, as it may be next year, the change will have to be made in only one place: where the value of the variable is assigned. Your programs will then also be more readable: your readers will not wonder where the 40 came from! In Program 6.13 the number 40 appears in 8 different places! So you might think it good policy to declare:

int num_students = 40;
int marks [num_students]; /* WRONG */

But the second declaration will not work: the compiler will revolt. On the other hand, a macro definition of num_students, which is processed before the compiler begins to allocate storage:

#define NUM_STUDENTS 40

followed by

int marks [NUM_STUDENTS];
Consider again our example of generating primes up to hundred million of the form \(k^2 + 1\) given in Listing 14. As these primes are generated, we’d want to store them in consecutive long words of RAM. So in this case it would be nice to have a set of wider predeclared storage locations than we had before, an array of type \texttt{long int} to pile the primes in:

\begin{verbatim}
long primes [1000]; /* assuming there are 1000 such primes */
\end{verbatim}

After the \texttt{i}th prime has been generated and assigned to the array element \texttt{primes[i]}, the next prime generated must naturally be placed in the next longword, \texttt{primes[i+1]}. So this time we’d want that incrementing the index \texttt{i} should lead us from the current longword to the next longword. C provides us this convenience: we don’t have to worry about whether array elements are \texttt{ints} or \texttt{longs}: incrementation or decrementation of the index directly leads us to the next or the previous element of the array, no matter that the elements are \texttt{chars} or \texttt{ints} or \texttt{floats} or \texttt{doubles}, or of a user defined data type.

This is a powerful feature of arrays: incrementing the index in the array \texttt{marks[]}, where each student’s mark occupies an \texttt{int}, enables one to move from one item of type \texttt{int} to the next. But incrementing the index in the array \texttt{primes[]}, where each object is four bytes wide, still lets you get to the next element! Once the type of an array is known to C, it lets you navigate to any element of the array, no matter how large (or small) the size of individual items in it may be, merely by setting the value of the subscript \texttt{i} as desired. Arrays bestow the power of random access to their elements, through the subscript \texttt{i}.

\begin{verbatim}
/* Program 6.14; File name: unit6-progl4.c */
#define TRUE 1
#define FALSE 0
#include <stdio.h>
int main()
{
    long int primes[1000], number;
    int i = 0, factor, k, isprime;
    primes[0] = 2L;
    for (k = 2; k <= 10000; k += 2) {
        number = k * k + 1;
        isprime = TRUE;
        for (factor = 3; factor * factor <= number; factor += 2)
            if (number % factor == 0) {
                isprime = FALSE;
                break;
            }
        if (isprime) {
            primes[++i] = number;
            printf("k = %d,%ld\n", k, primes[i]);
        }
    }
    return (0);
}
\end{verbatim}

\textbf{Listing 14: A program to list primes of the form \(k^2 + 1\).}

Note that in the last example we dimensioned the array \texttt{primes[]} at 1000, fondly hoping that there’ll be fewer than 1000 primes to store. But if there were more? Well, they would overflow the area reserved for them, and C wouldn’t whisper a word of warning, as other languages considerately do, it would let you overwrite potentially valuable instructions or data, and you would find your program gracelessly terminated!

On the other hand if there happened to be fewer than 1000 primes, (the actual number is
/* Program 6.15; File name: unit6-prog15.c */
#include <stdio.h>
int main()
{
    char result[1001];
    int numerator, denominator, integer_part, next_num, i;
    printf("This program evaluates fractions to a thousand decimal places.\n");
    printf("Enter rational fraction in the form a/b: ");
    scanf("%d/%d", &numerator, &denominator);
    integer_part = numerator / denominator;
    next_num = numerator % denominator;
    result[0] = ' ';
    for (i = 1; i < 1001; i++) {
        next_num *= 10;
        result[i] = next_num / denominator + '0';
        next_num %= denominator;
    }
    printf("%d", integer_part);
    for (i = 0; i < 1001; i++)
        putchar(result[i]);
    return (0);
}

Listing 15: Decimal expansion of fractions.

840, and the largest is 99800101; Check this.) we’d be wasting valuable memory by our declaration. This example highlights the disadvantage of reserving memory by fixed-sized arrays: you lose by declaring too little, you lose by declaring too much; C offers the alternative of dynamic memory allocation by which a program can request for memory during execution, as and when required. Memory allocating functions are described in a later unit.

For an interesting application of char arrays, consider the problem of storing and manipulating long sequences of digits, such as would arise in the evaluation of a rational fraction, such as 97/113; naturally such sequences cannot be stored as numbers with arithmetical significance; the best that we can do is to store them as a char array. In the program in Listing 15, as each digit is generated, it is converted to char by noting that in the ASCII code the collating order of the digits is their natural order: '0', '1', '2', etc. (so e.g. '2' is obtained by adding 2 to '0').

Think now of a ticket-issuing program for seats on a train with a seating capacity for a thousand passengers. Minimally the program should be able to book seats as long as they are available, as well as cancel (and make available for other travellers) reservations previously made. Now the status of a seat can be either “available” (A) or “booked” (B). It is clear that this time we must set up a correspondence of each seat on the train by a single byte of RAM. If a seat is available, the contents of the corresponding byte are set to the char ‘A’; but when that seat is booked, the value in the byte is change to ‘B’; and changed back to ‘A’ again if the reservation is later cancelled. Now, if each seat in the train is numbered from 1 upwards, then its status will be mirrored by the contents of the byte which corresponds to it, an ‘A’ or a ‘B’ (except for the minor detail that the physical seat number will be one greater than the subscript of the array element corresponding to it). This time we make the declaration:

char seats[1000];

It sets aside 1000 bytes to store the seats of the train. Each element in the array is accessible by a different value of the subscript. As before, the index begins at 0, and runs to one less than the number of seats in the train. Booking seat #7 would imply an
assignment of the form:

```c
seats[6] = 'B';
```

Freeing the 23rd seat after a cancellation would mean:

```c
seats[22] = 'A';
```

It is in these sorts of situation that the array concept is most useful.

Here is an exercise for you to try.

E15) Write a program for finding the mean, median and standard deviation of data. You program should prompt for the number of datum, scan them in one by one and then print the answer.

In the next section we will discuss arrays and the `sizeof` operator. We will see that `sizeof` operator is useful for dynamic memory allocation and in creating and using data structures like linked lists and queues.

### 6.4 THE INITIALISATION OF ARRAYS, AND THE `sizeof` OPERATOR

The commonest type of `char` array is a string:

"This string array has 35 elements."

If you were to count the `chars` in the string above, you would find only 34, including the spaces and the full-stop (\'.\'), and you might feel cheated. But the fact is that C inserts the null character (`\'\0\') at the end of every string, as an aid to knowing when the end of the string has been reached. So the zeroth element of this array is `T`, and its element with index value 34 (that is element number 35) is `\0`.

To assign this string to `char` array called `example[]`, the appropriate C statement is:

```c
static char example[] = "This string array has 35 elements.";
```

In most pre-ANSI Cs the keyword `static` is necessary, but we'll wait a bit before we go on to learn its significance. (As we've seen, C variables have a type. They are also distinguished by `storage classes`--one of which is `static`--which determine the duration of time a variable exists, and the blocks, functions or files from which it can be accessed. Storage classes are discussed in the next section.) For now, note the very important fact that while assigning the string above to `example[]` we did not explicitly specify the `dimension of example[]`, i.e. how many elements to set aside for it. There's nothing written inside the square braces: C "dimensions" such arrays appropriately by itself, relieving you of the drudgery of having to count each character in the string, and adding an extra to the total for the null character. As remarked above `example[0]` has the value `T`, `example[1]` is `h`, `example[33]` is `.`, and `example[34]` is `\0`.

**Note:** Because in the initialisation of arrays the `static` declaration is not required by compilers conforming to ANSI C, the declaration `char example[] = "This string array has 35 elements.";` works for ANSI C compliant compilers.

Any `static` array can be initialised by listing its individual elements:
static long primes[] = { /* of the form k * k + 1 */
2, 5, 17, 37, 101}

Here primes[] too is dimensioned by default: its dimension is 5. primes[1]
has the value 5, primes[3] is 37.

Consider now the declaration:
static long primes[10] = {
2, 5, 17, 37, 101};

This array has 10 elements of which none but the first five have been specified. As
before, primes[0] is 2, primes[1] is 5, and primes[4] is 101. The
remaining elements: primes[5], primes[6] etc. are each initialised to 0. Static
array elements (and static scalar—i.e. single, not aggregate—program variables) get the
value 0 if you do not assign any other value to them. In Classic C too this holds true for
arrays or variables that have been declared static.

We have seen that if a static array is to be initialised to non-zero values, each element of
it must be explicitly define. This can sometimes be a lot of work. Going back to the
example of booking train seats, initially, before any reservations are done the status of
each seat must be “available”. The corresponding array elements must each be set to
‘A’:

static char seats[1000] = {
'A', 'A', 'A', ...
/* 1000 values */
... 'A', 'A', 'A'};

Unfortunately there isn’t a “repeat factor” in C for initialising array elements, as there is
in FORTRAN; but one can escape the chore of writing in a thousand ‘A’s by assigning
the value ‘A’ to seats[i] for all values of i, in a loop.

We have seen that it is not necessary to dimension an array that is initialised. One can
determine the size in bytes of an undimensioned array, and of a variety of other objects,
using the sizeof operator. This is a unary operator that associates from right to left and
which returns the size (as an int—unsigned int in ANSI C—number of bytes), of the
datum type, or variable (which may be an aggregate variable such as a struct (Unit 10)
or an array), or expression, named on its right, for example:
sizeof (int)

It is often important for a program to know the sizes of the fundamental data types on
the machine on which it is being executed. For example, suppose that memory must be
allocated to store a number n of ints, as they are generated during an executing
program; and suppose also that the number n is not known in advance, but is computed
at run time. Typically n could be the number of goldbachian decompositions of an even
integer, and storage may be required to hold the n pairs of primes adding up to the
integer.

Note

Goldbach conjecture (1742) says that any even number greater than 4 can be
decomposed into the sum of two odd primes, thus:
14 = 3 + 11 = 7 + 7
16 = 3 + 13 = 5 + 11
30 = 7 + 23 = 11 + 19 = 13 + 17 etc.

Though no counter example has been found so far, a proof is still awaited. Nor
do we know precisely how many such decompositions are possible for an
arbitrary even integer. Interestingly enough, numbers divisible by 15 seem to
have significantly more partitions into two primes than other, approximately
equal, numbers.

The number of bytes to be set aside will then be 2 x 2 x n on a machine on which the
size of \texttt{int} is two bytes (two bytes per prime), but $2 \times 4 \times n$ on one in which the size of \texttt{int} is 4 bytes (a VAX or Macintosh). (Of course, most of the modern PCs are 32 bit machines which have \texttt{ints} of size of 4 bytes. Now a days 64 bit PCs which have an \texttt{int} size of 64 are becoming more and more common.) Neither of these expressions is portable. But $2 \times \texttt{sizeof (int)} \times n$ can be used without modification on any system.

The program in Listing 16 illustrates the syntax of the \texttt{sizeof} operator. Note carefully the parentheses around the basic types. Observe that the program yields the expected sizes of dimensioned well as undimensioned arrays.

/* Program 6.16; File name:unit6-prog16.c */
\#define STRING "What isthe number of bytes in this string?"
\#include <stdio.h>

\texttt{int} main()
{
    \texttt{char} a = 'b';
    \texttt{static char} vowels[] = { 'a', 'e', 'i', 'o', 'u' };
    \texttt{static char} vowel_string[] = "aeiou";
    \texttt{static int} five_primes[] = { 2, 3, 5, 7, 11 };
    \texttt{static double} tenbignums[10] = { 3816.54729, 1654.72938 };
    printf("The size of char on this system \ \ is: \ %d\n", \texttt{sizeof (char)});
    printf("Therefore the size of char a is: \ %d\n", \texttt{sizeof a});
    printf("However, the size of \textquoteleft b\textquoteright\ is: \ %d\n", \texttt{sizeof \textquoteleft b\textquoteright});
    printf("The size of short on this system \ \ is: \ %d\n", \texttt{sizeof (short)});
    printf("The size of int on this system \ \ is: \ %d\n", \texttt{sizeof (int)});
    printf("The size of long on this system \ \ is: \ %d\n", \texttt{sizeof (long)});
    printf("The size of float on this system \ \ is: \ %d\n", \texttt{sizeof (float)});
    printf("The size of double on this system \ \ is: \ %d\n", \texttt{sizeof (double)});
    printf("The number of bytes in vowels [] \ \ is: \ %d\n", \texttt{sizeof vowels});
    printf("The number of bytes in vowel_string [] \ \ is: \ %d\n", \texttt{sizeof vowel_string});
    printf("The number of bytes in five_primes [] \ \ is: \ %d\n", \texttt{sizeof five_primes});
    printf("The number of bytes in tenbignums [] \ \ is: \ %d\n", \texttt{sizeof tenbignums});
    printf("The number of bytes in STRING is: \ %d\n", \texttt{sizeof STRING});
    return (0);
}

Listing 16: The \texttt{sizeof} operator.

The output of the program on 32 bit PC running linux is appended below:

/*Program 6.16: Output*/
The size of char on this system is: 1
Therefore the size of char a is: 1
However, the size of \textquoteleft b\textquoteright\ is: 4
The size of short on this system is: 2
The size of int on this system is: 4
The size of long on this system is: 4
The size of float on this system is: 4
The size of double on this system is: 8
The number of bytes in vowels [] is: 5
The number of bytes in vowel_string [] is: 6
The number of bytes in five_primes [] is: 20
The number of bytes in tenbignums [] is: 80
The number of bytes in STRING is: 43

carefully study the output and try the related exercises given below.

E16) Explain the third line in the output above.
E17) Count carefully the number of bytes in vowel_string[]. Why is there a discrepancy between the number of bytes you counted, and the number in the tenth line of output?
E18) Modify the above program to prove that the null string is precisely one byte long.
E19) Execute Program 6.16 on different compilers, and architecturally different computers if possible. Do you get the same results each time? Or the same results as above?
E20) Examine the following declaration:
static char hello[] = {
    'h', 'e', 'l', 'l', 'o'
};
Why is hello[] not a string?
E21) Is the 6-element array hello[] declared below a C string?
static char hello[6] = {
    'h', 'e', 'l', 'l', 'o'
};
What is the value of hello[5]?
E22) What is the difference between an assignment statement and an initialisation?

we have already different elementary data types into which we can classify variables in C. There is another way of classifying variables. This is based on storage classes. We discuss various storage classes of variables and the scope of the variable (i.e. the portion of the program where the values of the variable are available) in the next section.

6.5 STORAGE CLASSES AND SCOPE

Consider a program to find a few values of n such that the sum of the first n primes is itself a prime number. One design for the program logically divides it into two parts: in the first part the program creates an array of say a thousand primes; in the second, it adds the elements of the array to a variable called sum_n, and tests each value of sum_n as it is generated for primeness. The program in Listing 17 on the next page is based on such a logical division. It consists of two blocks, the first of which contains code to generate the array primes[1000]; while the second tests every other value of sum_n (because half the values of sum_n are even, anyway, and therefore composite) and lists those that are prime. (In the next unit, where functions are introduced, we shall learn to encode each separate activity as a function, as a more appropriate division of labour.)
The second block of the program in Listing 17 declares and defines two new variables, \( n \) and \( \text{sum}_n \); these variables, and the variables \( \text{number}, \text{factor}, \text{primes}[1000] \) and \( i \) from the previous block, in fact all program variables that we’ve encountered thus far are examples of \textbf{automatic} variables.

```c
/* Program 6.17; File name: unit6-prog17.c */
#include <stdio.h>
int main()
{
 /* Outermost block */
    int number, factor, primes[1000], i;
 /* This section generates a thousand primes and stores them in primes [] */
    primes[0] = 2;
    primes[1] = 3;
    i = 2;
    for (number = 5;; number += 2) {
        for (factor = 3;; factor += 2)
            if (number % factor == 0)
                break;
            else if (factor * factor > number) {
                primes[i++] = number;
                break;
            }
        if (i == 1000)
            break;
    }
 /* This section finds values of \( n \) such that the sum of the first \( n \) primes is itself a prime. */
{
 /* Nested block */
 /* The variables of the enveloping block: number, factor, primes [1000] and i are available here. */
    int n;
    unsigned sum_n = 2;
    for (n = 1; (sum_n += primes[n]) < 32767; n++) {
        if (n % 2 == 0)
            continue;
        for (factor = 3;; factor += 2)
            if (sum_n % factor == 2)
                break;
            else if (factor * factor > sum_n) {
                printf("The sum of the first \%d primes \n is \%u and is itself a prime.\n", n + 1, sum_n);
                break;
            }
    }
 /* The variables of the nested block are not available here. */
    return (0);
}
Listing 17: An example to illustrate scope of variables.
```

Automatic variables are so called because they are automatically created when control enters the block (or function) in which they are defined, and destroyed when control leaves that function. Thus, when a program begins executing at \texttt{main()} its variables spring to life, their values are known and can be modified and displayed, but pouf! they’re blown out like a candle so soon as the program finishes execution! Similarly, when control departs from a function, we shall find in the next unit that the automatic variables “local” to the function cease to exits. Their values are lost forever. You can no
Programming in C

further refer to them in the program. And if control reenters their declaring function, no
memory is retained of their former values.

However, variables defined inside a nested block don’t quite die away when control
exits their defining block: they merely become dormant, hibernating until control
should perchance reenter their block: when they are rejuvenated to their former values.

Variables local to a block cannot be referred to outside that block; thus in the program
in Listing 17 on the previous page \( n \) and \( \text{sum}_n \) will not be available beyond the
confines of their defining block; but a variable defined previously, in an enclosing
block, is available in the enclosed block: the elements of \( \text{primes}[\] \) can be added to
\( \text{sum}_n \). Variables defined externally to a block are accessible inside it, and inside any
nested blocks. (That is why our \#defines have been written outside of \text{main}() in the
preceding programs: the tokens are available in all the code that follows.) In fact, you
may declare program variables, if you wish, before \text{main}(): then they will be accessible
in all the code that follows. Changes made to them in one place will be known in
every other place where they are referenced. Such externally defined variables are by
default initialised to zero. See Program 6.19.

But there is one important distinction between a macro token and an externally defined
variable: a variable inside a nested block may be christened with the same name as a
variable outside it: if such be the case, the local value has precedence over the global.

Programmers who use duplicated names for automatic variables in functions or blocks
often use the optional keyword \text{auto} in their declarations, thus:

\begin{verbatim}
auto int i;
\end{verbatim}

to remind their readers (and themselves) that the local value of the variable is to be used
in the present instance. \text{auto} has been the default class for all the scalar variables we
have used thus far. Study the program in Listing 18 on the facing page and its output to
become familiar with the properties of automatic variables.

/* Program 6.18: Output */

Control is presently in outer block, where
\( i = 5, j = 6, k = 7 \)
control has now entered the inner block...
Local value of \( i \) has precedence over global value...
Therefore its value is: 1
\( j \) has not been defined in inner block.
what value does it have here? \( j = 6 \)
\( k \) is defined externally to this block...
It can be accessed here: \( k = 7 \)
Now modify \( i, j \) and \( k \) inside the inner block...
\( i = 10, j = 11, k = 8 \)
Control now reenters the outer block...
Are the inner block values of \( i \) and \( j \) retained?
No, because... \( i = 5, j = 6 \)
However, the last value of \( k \) is still with us...
\( k = 8 \)
Control has now entered the inner block...
/* Program 6.18; File name: unit6-prog18.c */

#include <stdio.h>

int k = 7;

int main()
{
    int i = 5, j = 6;
    printf("Control is presently in outer block, where\n");
    printf("i = %d, j = %d, k = %d\n", i, j, k);
    redefineij:
    {
        int i = 1, j;
    innerblock:
    printf("Control has now entered the inner block...\n")
    if (k == 8) {
        printf("i and j were not initialised \n when control\n");
    printf("entered the inner block for \n the second time...\n");
    printf("Are their former values retained?\n");
    printf("Yes, because...now i = %d, j = %d\n", i, j);
    k++;
    goto endprogram;
    }
    if (k == 9) {
        printf("for the third time...\n");
        printf("This time i, j are redefined...\n");
        printf("Now i = %d, but j = %d\n", i, j);
        k++;
        goto endprogram;
    }
    printf(" Local value of i has precedence over \n global value ... \n ");
    printf(" Therefore its value is:%d \n ", i);
    printf("j has not been defined in inner block. \n ");
    printf(" What value does it have here ? j = %d \n ", j);
    printf(" k is defined externally to this block... \n");
    printf(" It can be accessed here : k = %d \n ", k);
    printf(" Now modify i, j and k inside the \n inner block... \n ");
    i = 10;
    j = 11;
    k++;
    printf(" i = %d, j = %d, k = %d \n ", i, j, k);
    }
    printf(" Control now reenters the outer block... \n ");
    printf(" Are the inner block values of i and j \n retained ? \n ");
    printf(" No, because...i = %d, j = %d \n ", i, j);
    printf(" However, the last value of k is still \n with us...k = %d \n ", k);
    goto innerblock;
    endprogram:
if (k == 9)
    goto redefineij;
    return (0);
}

Listing 18: Program demonstrating automatic variables.
i and j were not initialised when control entered the inner block for the second time...
Are their former values retained?
Yes, because... now i = 10, j = 11
Control has now entered the inner block...
for the third time...
This time i, j are redefined...
Now i = 1, but j = 11

Akin to the auto storage class is the register class: when the register storage specification (register, too is a C keyword) is prefixed to a variable’s declaration, thus:

register int i;

then the compiler may try to place the variable i in a machine register, if a register is available; but it is by no means bound to oblige you. Programmers may want to load frequently accessed variables such as loop indices inside registers for greater efficiency; but most compilers take care of such things automatically anyway. Nor can one assign large or aggregate data types, such as doubles or arrays to registers. Finally, the \& operator cannot be applied to register variables.

Another c storage declarator is static. The keyword static (which has nothing to do with electricity!) is used to declare variables which must not lose their storage locations or their values when control leaves the functions or blocks wherein they are defined. In C jargon, static variables retain their values between invocations. The initial value assigned to a static variable must be a constant, or an expression involving constants. But static variables are by default initialised to 0, in contrast to auto.

Suppose that the variables i and j in the inner block of Program 6.18 had been declared thus:

static int i = 1, j;

Then necessarily j would have been set at 0 when control entered the inner block the first time; and i would have retained its last value, 10, when control had entered the inner block for the third time.

Classic C requires the static declaration for arrays which must be initialised.

E23) Change the declarations in the inner block of Program 6.18, execute it and verify that static variables behave as expected.

E24) State the output of the program in Listing 19:

/* Program 6.19; File name: unit6-prog19.c */
#include <stdio.h>
int alpha = 5;
it beta;
it main()
{
	auto int alpha = 10;
	printf("%d, %d\n", alpha, beta);
beta = 2;
innerblock:
{
	int alpha = 15;
	static int gamma = 20;
We now end this Unit by summarising our discussion in the next section.

### 6.6 SUMMARY

In this unit, we have studied the following:

1. The properties of the \texttt{for} (;;) loop, and its application to a variety of situations. The \texttt{for} (;;) statement contains three expressions, each playing a particular though not indispensable role: the first expression initialise the loop variable; the second determines whether the loop be entered; and the third re-initialises the loop variable after a round of execution to present it again to the second expression for its scrutiny. These expressions are separated by semicolons, which must be present whether or not the expressions are. If the second expression is missing, it is regarded as \texttt{true}.

2. We studied arrays of one dimension, which are used to hold a group of variables of the same type. Each element of the array is identified by its subscript or position in the array. Array subscripts begin at 0 for the first element. The array concept makes possible random access to any element. In Classic C arrays which must be initialised must be declared as \texttt{static}.

3. The \texttt{sizeof} operator is a unary operator which, associating from right to left, yields as an \texttt{unsigned int} the number of bytes in its operand. This may be a basic type, a variable or an array.

4. C has four storage classes, three of which: \texttt{auto}, \texttt{register} and \texttt{static} have been studied in this Unit. The storage class of a variable determines the longevity and visibility of the variable. Auto and register variable may take arbitrary values if they are not initialised: externally defined variables or static variables are by default initialised to zero. A variable external to a block is accessible inside the block.

### 6.7 SOLUTIONS/ANSWERS

E1) The \texttt{first_expression} sets the value of \texttt{i} to 1. Next, the boolean expression is checked. Recall that the expression \texttt{i = 1} has value 1, i.e. non-zero value. So, the loop will be entered and it will print 1, the present value of \texttt{i}. Since the \texttt{third_expression} is missing, it will evaluate the second expression, whose value is \texttt{true} as we saw before. The program will print 1 again and again.
E2) **Program 6.4:** The variable \( j \) is declared and initialised to 5 in line 5. The `printf()` in the `first_expression` in line 6 is executed. Since the loop condition is not satisfied, `for(;;)` loop is exited. So, the `third_expression` is not executed. The next `printf()` statement in line 8 is executed.

**Will this loop be entered?**
**Was the loop entered?**

**Program 6.5:** The variable \( j \) is declared and initialised to 5 in line 5. Next, the `printf()` in line 6 is executed. Then, the loop is entered because the condition \( j < 6 \) is true(\( j = 5 \)). The `printf()` statement in lines 7 and 8 in the body of the `for(;;)` is executed. Then, the statement \( j = 7 \) in `third_expression` of the `for(;;)` loop is executed. Since the Boolean condition in `second_expression` is no longer satisfied, loop is exited and the `printf()` statement in line 9 is executed. The `if-then-else` in line 10, returns the value “No,” since the value of \( j \) is 7. The `printf()` statement in line 11 is printed.

**Will this loop be entered?**
**Print this, if the loop is entered...\( j = 5 \)**
**Is the Boolean \( j < 6 \) still true?**
**No, because \( j \) is now 7.**

**Program 6.6:** Try this one yourself. The output is given below:

*The for (;;) loop is really quite easy to use.*
*I've said this once.*
*The for (;;) loop is really quite easy to use.*
*I've said this twice.*
*What I say 2 times must be true!* 

E4) If the `&&` operator is replaced by the `||` operator, since the second expression `(sum += i ++)` has a positive value, the expression `i < 101 && (sum += i ++)` will always have a positive value. So, the loop will not terminate at all even after \( i \) crosses the value 1.

It cannot be replaced by the comma operator either, because in this case the value of the expression is the value of the second expression `(sum += i ++)` and again the loop will not terminate and the `printf()` after the `for(;;)` loop will not be printed.

The parentheses around `(sum += i ++)` is necessary. Since `&&` has higher priority than `+=`, this expression is interpreted as `(i < 101 && sum) += i++` 

Since the LHS of the above expression is not a lvalue and `+=` can be applied only to an lvalue, the compiler displays an error message.

E5) The first set of brackets is not necessary. We can write `i % 2 ? sum_odd += i : (sum_even += i)`; because till the `:` is encountered, it the first expression is not complete. However, we cannot write `i % 2 ? sum_odd += i : sum_even += i;`. Since the incrementation operator has lower priority, this is interpreted as `(i % 2 ? sum_odd += i : sum_even) += i;` and as before, the LHS of the increment operator is not an lvalue.

E6) A modified version of the program in Listing 1 on page 8 is given in Listing 20 on the next page. You can similarly modify the programs in Listing 2 and Listing 3 on page 8.

The modified version of Listing 4 on page 9 is in Listing 21 on the facing page.
/* Program 6.1 using while();
File name: unit6-ans-ex6-1.c */
#include <stdio.h>
int main()
{
    int i = 1, sum = 0;
    while (i < 101)
    {
        sum += i;
        i++;
    }
    printf("The sum of the numbers from 1 to %d is %d\n", i - 1, sum);
    return (0);
}

Listing 20: Modified version of the program in Listing 1 on page 8.

/* Program 6.4; File name:unit6-ans-ex6-2.c */
#include <stdio.h>
int main()
{
    int j = 5;
    printf("Will this loop be entered?\n");
    while (j < 2) {
        printf("Print this, if the loop is entered...");
        j = 1;
    }
    printf("Was the loop entered?\n");
    return (0);
}

Listing 21: Modified version of the program in Listing 4 on page 9

E7) See Listing 22 for Russian peasant algorithm using for(;;) and Listing 23 for the program for checking the Collatz conjecture.

/* File name:unit6-ans-ex-7-1.c */
/* Russian peasant multiplication using for()*/
#include <stdio.h>
int main()
{
    int val_1, val_2, lesser, greater, result = 0;
    printf("Russian Peasant Multiplication Algorithm\n");
    printf("Enter multiplier: ");
    scanf("%d", &val_1);
    printf("Enter multiplicand: ");
    scanf("%d", &val_2);
    greater = (lesser = val_1 < val_2 ? val_1 : val_2)
        == val_1 ? val_2 : val_1;
    for (; lesser; lesser /= 2, greater *= 2) {
        result += lesser % 2 ? greater : 0;
    }
    printf("%d\n", result);
    return (0);
}

Listing 22: Russian Peasant Algorithm using for(;;)

/* File name:unit6-ans-ex-7-2.c */
/* Collatz problem using for() loop*/
#include <stdio.h>
int main()
{
    int input, cycle_length = 0;
    do {
        /* code */
    } while (/* condition */);
}
Programming in C

/* Enter a positive trial number: */
scanf("%d", &input);
}
while (input <= 0); /* End of do */
for (; input - 1; cycle_length++) {
    if (input % 2 == 1) /* input was odd */
        input = input * 3 + 1;
    else
        input /= 2;
} /* End for */
printf("1 appeared as the terminating digit after %d iteration(s)", cycle_length);
return (0);
}
Listing 23: Collatz problem using for(;;).

E9) The modified program is given in Listing 24.

/* Answer to exercise 9; File name: unit6-ans-ex-9.c */
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int number, quotient, factor, no_pfactors = 1;
    printf("Enter a number, I'll print it\'s prime factors:");
    scanf("%d", &number);
    quotient = number;
    if (number == 2) {
        printf("2 is a prime. The factors are 1, 2.");
        exit(0);
    }
    printf("Prime factors with multiplicity are: 1"); /* cast out factors of 2, if any */
    while (quotient % 2 == 0) {
        no_pfactors++;
        printf("", %d", 2);
        quotient /= 2;
    } /* only odd factors can remain, if any */
    for (factor = 3; factor <= quotient; factor += 2) {
        if (factor * factor > quotient) {
            printf("", %d", quotient);
            no_pfactors++;
            break;
        }
        while (quotient % factor == 0) {
            no_pfactors++;
            printf("", %d", factor);
            quotient /= factor;
        } /*End for */
        if (no_pfactors == 2)
            printf("\n The number is prime.");
    }
    printf("\n");
    return (0);
}
Listing 24: Solution to exercise exercise. 9

E10) See the program for n² - n + 41 in Listing 25.

/* Solution to exercise 10; File name: unit6-prog9.c */
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int number, factor, i;
    for (i = 1; i <= 40; i++) {
        number = i * i - i + 41;
        if (number == 2 || number == 3) {
            printf("%d is a prime ...
", number);
            exit(0);
        } else if (number % 2 == 0) {
            printf("%d is composite...
", number);
            exit(0);
        }
        for (factor = 3;; factor += 2)
            if (number % factor == 0) {
                printf("For i = %d, f(i) = %d is composite...
", i, number);
                break;
            } else if (factor * factor > number) {
                printf("For i = %d, f(i) = %d is a prime...
", i, number);
                break;
            }
    }
    /*End For */
    return (0);
}

Listing 25: Solution to exercise 10.

We leave the next polynomial as an exercise to you.


/* Answer to exercise 11. File name: unit6-ans-ex-11.c */
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int number, i, quotient, factor, no_2factors = 0;
    int so_factors = 0, temp = 1;
    printf("Enter a number, I'll print it\'s factors:");
    scanf("%d", &number);
    quotient = number;
    if (number == 2) {
        printf("2 is a prime. The factors are 1, 2.");
        exit(0);
    }
    printf("Factors are: ");
    /* Find the factors that are powers of 2, if any */
    while (quotient % 2 == 0) {
        no_2factors++;
        quotient /= 2;
    }
    /* only odd factors can remain, if any */
    for (factor = 1; factor <= quotient; factor += 2)
        if (quotient % factor == 0) {
            printf("%d, ", factor);
            temp = factor;
            (temp < number) ? so_factors += temp : 0;
        } /*For each odd factor of the quotient, print
    all the corresponding even factors of the number.*
    for (i = 1; i <= no_2factors; i++) {
        printf("%d, ", temp *= 2);
        (temp < number) ? so_factors += temp : 0;
    };
}
printf("\n");
temp = 1;
if (so_factors < number)
    printf("The number is deficient.");
else if (so_factors > number)
    printf("The number is abundant.");
else
    printf("The number is perfect.");
return (0);

Listing 26: Solution to exercise 11.

E12) We proceed as follows: We loop through 1 to 1000. In each iteration of the loop, we find the power of 2 and the power of 5 dividing the loop variable. Since we do not want the loop variable to be modified, we use a dummy variable. The program is in Listing 27.

E13) The program is in Listing 28.
Control Structures-II

Listing 28: Solution to exercise 13

E14) See Listing 29.

/*Solution to exercise 14, unit 6. File name:unit6-ans-ex-14.c*/
#include <stdio.h>
#include <math.h>
define TOL .000001
int main()
{
    int i, no_iterations = 2500;
double x_zero = 0.4, x_one;
    for (i = 0;; i++) {
        x_one = sqrt(1 + sin(x_zero)) / 3.0;
        if (fabs(x_one - x_zero) <= TOL) {
            printf("One root of eqn. is: %10.5f\n", x_one);
            printf("convergence achieved after %d \n", i);
            break;
        } else if (i > no_iterations) {
            printf("No convergence after %d iterations.\n", no_iterations);
            break;
        }
        x_zero = x_one;
    }
    return (0);
}

Listing 29: Solution to exercise 14

E15) The program is in Listing 30. We read in the data using a for(;;) loop. We then find the mean and standard deviation using for(;;) loops. Note the use of pow() for squaring. We sort the data using insertion sort and find the median for the data.

/*Program to find the mean, median and mode.
File name; unit6-ans-ex-15.c*/
#include <stdio.h>
#include <math.h>
int main()
{
    float mean, median, temp, sum, data[40];
double sd = 0;
    int i, j, n;
    printf("Enter the number of data elements:\n");
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        printf("Enter data \%d:\n", i + 1);
        scanf("%f", &data[i]);
    }
    for (i = 0; i < n; i++)
        sum += data[i];
    mean = sum / n;
    for (i = 0; i < n; i++)
        sum += data[i];
    mean = sum / n;
sd += pow((data[i] - mean), 2);
sd = sqrt(sd / n);
printf("The mean is %f\n", mean);
printf("The standard deviation is %+6.3f\n", sd);

/* We sort the data using interchange sort. */
for (i = 0; i < n; i++)
  for (j = i + 1; j < n; j++)
    if (data[i] <= data[j])
      continue;
    else {
      temp = data[i];
      data[i] = data[j];
      data[j] = temp;
    }

/* We find the median. */
if (n % 2 == 0) {
  median = (data[(n - 2) / 2] + data[n / 2]) / 2;
} else {
  median = data[(n - 1) / 2];
}
printf("Median is %f", median);
return (0);

Listing 30: Solution to exercise 15.

E16) While a is a character variable, and is stored in 1 byte, ‘b’ is equivalent to its numerical value and is considered as int.

E17) This is because the vowel_string contains the terminating null as the sixth character.

E18) Change the definition of STRING to "", compile and run again.

E20) It is not a string because it does not have the terminating null.

E21) Yes, it is a string. Because of the static declaration, hello[5] is assigned the value \0. The numerical value of hello[5] is zero.

E22) An assignment can be made anywhere in a program, but initialisation is done only at the time of the declaration of the variable.
7.1 INTRODUCTION

This unit introduces “hard core” C: pointers and their manipulation. Pointer are conceptually quite simple: they’re variables that hold the memory addresses of other variables.

To concretise concepts, think of an array the elements of which, as you know, are placed in consecutive locations of storage, at regularly increasing addresses. It is apparent that the address of any element is immediately computable relative to that of the zeroth: for, if the address of the zeroth element of the array be \(x\), say, and if each element of it be \(w\) bytes wide, then, relatively to its beginning, the address of the \(k\)th element must be \(x + k \times w\), \(k = 1, 2, \ldots\), etc. You can now see that pointers, which are variables to store memory addresses, should be very closely allied to arrays. In an array, the address of any element is one “unit” greater than that of the preceding element. Different types of “units”-- `chars`, `floats` or `doubles`, or user-defined types--would of course have differing numbers of bytes inside them. so going from one array element to the next (by incrementing its subscript) is equivalent to “incrementing” the pointer to the current element; incrementation here meaning increasing the value of the pointer by the width, in bytes, of the object it was pointing at. So pointers can provide an elegant, and often faster, alternative to array manipulation.

At this point there is one question that might occur to you: why are pointer variables necessary at all? Many languages such as FORTRAN and COBOL get by quite happily without them.

Quite apart from providing a faster alternative to array manipulation, pointers are used in C to dynamically allocate memory, i.e. while a program is executing. This aspect, as we have hinted before, is important in situations where it is impossible to know in advance the amount of storage required to store the data be generated, or which will be input. In Program 6.14 we did not know in advance how much space to set aside for primes up to hundred million of the type \(k^2 + 1\). In a sense we gambled: we chose, without any evidence, the array dimension to be 1000. Declaring too large an array may waste memory; but too small an array might cause the program to run haywire or be aborted. Dynamical allocation gets around this problem by enabling as much memory to be created as required, when required. If memory can’t be found, the allocating function can inform the program accordingly.

Finally, C uses pointers to represent and manipulate complex data structures.

In Sec. 7.2, we discuss and the operations that can be carried out using pointers. In Sec. 7.3, we discuss how pointers can be used to manipulate arrays.
scanf() is extremely versatile function that C uses for keyboard input. In Sec. 7.4 of this unit, we will explore some of its properties further. And finally, in Sec 7.5, we shall look at multi-dimensional arrays where more than one subscript is required to locate an element; matrices and magic squares are common examples of two-dimensional arrays.

Objectives
After studying this unit, you should be able to

- explain the purpose of pointers and use them;
- explain the underlying unity of pointers and arrays;
- use the subscript operator;
- exploit the versatility of scanf();
- use the string I/O functions puts() and gets(); and
- handle multidimensional arrays.

7.2 POINTER VARIABLES AND POINTER ARITHMETIC

One of the most powerful features of C, and one that makes it quite close to assembly language, is its ability to refer to the addresses of program variables. In C one can in fact declare variables to be pointers, that is variables which will hold the memory addresses of other variables.

The declaration

```
char *x;
```

declares x to be a pointer; x can now be assigned the address of a char-like variable. Similarly one can declare pointer variables to point to (i.e. hold the addresses of) ints, or longs, or floats or doubles, in short, of variables of any type, including user-defined types.

The statements

```
int *x, *y, z = 10;
long *p, q;
float *s;
double *t;
```

declare

i) x and y to be pointer variables of type int. That is to say, x and y can hold the addresses of int like variables, such as z;
ii) p is a pointer to longs. It can store the address of a long int, for instance of q;
iii) s can store the addresses of float variables;
iv) t is a pointer to double.

The rvalues of pointer variables are memory addresses.

How does one extract the memory address of a program variable? And, conversely, given a memory address, how does one determine the contents at that address? To answer these question, let’s first recall our picture of memory: a sequence of boxes, each identified by a unique positive number or address, with the addresses written to the left of the boxes. A C variable has precisely these two parts associated with it: its lvalue or address, fixed once and for all after compilation and linkage, and its rvalue, the
Pointers and Arrays

contents of the box, which may vary as the program executes. Given a variable, C allows its address to be extracted; conversely, given an address, it is possible to obtain the contents at that address.

Let’s look again at the first of the declarations above:

```c
int *x, *y, z = 10;
```

The C “address of” operator is the ampersand, &. It’s a unary operator, that, applied to a variable (strictly, to an lvalue), yields its memory address:

```
address of z == &z
```

&z is the address at which the computer stores the int variable z, and this remains fixed throughout program execution. Attempting to change it in any way is an error.

This address may be assigned to x, because x has been declared to be a pointer to variables of type int:

```c
x = &z;
```

x now holds the address of z, which you may actually print:

```c
printf("The address of z is:	%u 
", x);
```

As remarked above, the “address of” operator can only be applied to lvalues. Like all unary operators it has a priority just below the parentheses operator, and groups from right to left. The “address of” operator is not unfamiliar; the scanf() function has an argument list of pointers to memory locations.

Reciprocally C has a “contents at address”, or “dereferencing” operator, the asterisk, *.

This unary operator yields the rvalue resident at a memory address. Applied to a pointer variable which has been initialised to an address, it finds the contents at that address.

In the example above, the assignment:

```c
x = &z;
```

marks x holds the address of int z. *x gives the contents at the address that’s held in x; the output from the printf() below will therefore be the rvalue of z, 10:

```c
printf("The contents of the address held in x are:	%d
", *x);
```

See the program in Listing 1.

```c
/* Program 7.1; File name: unit7-prog1.c */
#include <stdio.h>
int main ()
{
  int *x, z = 10;
  x = &z;
  printf("The address of z is:\t\%u \n", x);
  printf("The contents of the address held in x \ 
are:\t\%d\n", *x);
  *x = 20;
  printf("The new value of z is:\t\%d\n", z);
  return (0);
}
```

Listing 1: Example to illustrate ‘address of’ operator and dereferencing operator.

To clarify these concepts further, let’s assume that the address of the int variable z is 38790, and that the contents at this address are 10. See Fig.1 below. In the assignment:

```c
x = &z;
```
x gets the value “address of z”, i.e. 38790. So, if the address of x itself is 56780, the contents at 56780 will be 38790.

<table>
<thead>
<tr>
<th>address</th>
<th>contents of address</th>
</tr>
</thead>
<tbody>
<tr>
<td>z, located at 38790</td>
<td>10</td>
</tr>
<tr>
<td>other program variables</td>
<td>&quot;</td>
</tr>
<tr>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td>x, located at 56780</td>
<td>38790</td>
</tr>
</tbody>
</table>

The contents at x is the address of z
The contents at z is the value 10

Figure I

The subsequent assignment:

*x = 20;

changes the value of the int object whose address is held in x, namely z, to 20. Here’s the picture:

<table>
<thead>
<tr>
<th>address</th>
<th>contents of address</th>
</tr>
</thead>
<tbody>
<tr>
<td>z, located at 38790</td>
<td>20</td>
</tr>
<tr>
<td>other program variables</td>
<td>&quot;</td>
</tr>
<tr>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td>x, located at 56780</td>
<td>38790</td>
</tr>
</tbody>
</table>

Figure II - The value of z modified indirectly

Let’s now look at the program in Listing 2 on the next page. c1, c2, c3, ..., c9 are char variables that have been assigned the initial values ‘A’, ‘b’, ‘h’, etc; pointer is a pointer to char. The first printf() outputs the current values of c1, c2, c3, ..., c9. Then pointer is assigned the address of c2.

pointer = & c2;

In the next assignment the contents currently resident at pointer are altered to ‘p’:

* pointer = ‘p’;

c2 therefore now has the value ‘p’. One could of course replace the two assignments:

pointer = & c2, * pointer = ‘p’;

by the single assignment:

c2 = ‘p’;

but that doesn’t show the pointers at the back of it. Pointers do the same, albeit indirectly - manipulating values, so to speak, from a distance, the way puppeteer dances her puppets. In the program in Listing 2 on the facing page, pointer is made to point successively at c2, c3, ..., c9 and to make new assignments to them.

In the program in Listing 3 on the next page the pointers to int p, q and r are assigned the address of the int x, y and z, respectively. Then the contents of the addresses in p and q are set to 5 and 3, which become the respective values of x and y. Post-incrementation of x and post-decrementation of y change them to 6 and 2; these are the new values of *p and *q. Therefore the expression:

* r = *p * *q + * p / * q

makes the contents at the address held in r to be 15. This is the value that z gets.
/ * Program 7.2; File name:unit7-prog2.c */
#include <stdio.h>
int main ()
{
    char c1 = 'A', c2 = 'b', c3 = 'h', c4 = 'i',
        c5 = 's', c6 = 'h', c7 = 'e', c8 = 'k', c9 = ' ';
    char *pointer;
    printf("%c%c%c%c%c%c%c%c", c1, c2, c3, c4, c5, c6, c7, c8, c9);
    pointer = &c2;
    *pointer = 'p';
    pointer = &c3;
    *pointer = 'a';
    pointer = &c4;
    *pointer = 'r';
    pointer = &c5;
    *pointer = 'a';
    pointer = &c6;
    *pointer = 'j';
    pointer = &c7;
    *pointer = 'i';
    pointer = &c8;
    *pointer = 't';
    pointer = &c9;
    *pointer = 'a';
    printf("%c%c%c%c%c%c%c%c", c1, c2, c3, c4, c5, c6, c7, c8, c9);
    return (0);
}

Listing 2: Pointer operation.

/* Program 7.3; File name:unit7-prog3.c */
#include <stdio.h>
int main ()
{
    int x , y , z;
    int *p, *q, *r;
    p = &x;
    q = &y;
    r = &z;
    *p = 5;
    *q = 3;
    x++;
    y--;
    *r = *p * *q + *p / *q;
    printf("The contents at the address in p are %d\n", *p);
    printf("The contents at the address in q are %d\n", *q);
    printf("The contents at the address in r are %d\n", *r);
    return (0);
}

Listing 3: Incrementing and decrementing pointers.

If you think for a moment about how pointer variables are declared, with separate declarations for each type, a question that may occur to you is the following: addresses, whether they be of char, int, float or double variables, must each be very like one another. An address is an address is an address. Can the address of a char be qualitatively different from that of an int, or of a double from that of a long? Then why must pointer declarations distinguish between addresses of variables of differing data
Programming in C

In other words, why couldn’t the inventor of C do with just one declaration for a pointer variable, such as for example:

```c
pointer c, x, y, p, s, t;
```

rather than separate declarations for each type:

```c
char *c;
int *x, *y, z = 10;
long *p, q;
float *s;
double *t;
```

The fact is that there is a subtle difference between pointers which refer to variables of different types. By informing the compiler about the type of object being referenced by the pointer, the pointer can be “scaled” appropriately to point to the next object of the same type. Recall that in the IBM PC for example, a `char` value sits in a single byte of memory, an `int` in two bytes, a `float` in four and a `double` in eight. So when a `char` pointer is “dereferenced”, it should give the contents of the single byte whose address it holds. But when an `int` pointer is dereference, it must give the contents of two consecutive bytes (regarded as an integral unit, a word), because an `int` is two bytes big. Similarly, when a pointer to `float` is dereferenced, it must yield the contents of the set of four bytes (two words) which hold that `float` variable. Thus, by associating a type with a pointer variable it helps to dereference the value of the referenced variable properly in accordance with its size.

There is another reason why it makes sense to distinguish between pointers to different data types. We have seen that there is a close relationship between pointers and arrays in C. Stepping from one array element to the next directly translates to incrementing a pointer which holds the address of the first element of the array, as shown in Figure II below:

```
<table>
<thead>
<tr>
<th>element [i]</th>
<th>element [i + 1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>pointer_value</td>
<td>pointer_value + 1</td>
</tr>
</tbody>
</table>
```

Incrementation changes pointer by width of element it points at.

- Incrementing a pointer to `char` increases it by 1.
- Incrementing a pointer to `double` increases it by 8.

**Figure II**

Now, a string array is a collection of individual bytes. The address of any one byte is one greater than the address of the preceding. So in going from one character of the string to the next, you increment the pointer to that byte. But if you have an array of `doubles`, then, in going from one element of the array to the next, again incrementing the pointer that stores the address of that element, you would actually increase the pointer’s value by eight, because `doubles` are eight bytes wide! Adding one to a pointer produces different results if the pointer is a `char` or `int` or `float` or of any other type. So pointers are not all alike. Neither are they integers in the sense of `int`.

Incrementing an `int` increases its value by one. Incrementing a pointer increases its value by the width of the object it points to. For a 32 bit PC, incrementing a pointer to `int` increases its value by 4! See Fig.II., and the output of Program 7.4, which you should study carefully. Because pointers are not ints, C does not allow many of the operations on pointers, that are possible with `ints`: for example, you can’t multiply two pointers together; nor should you want to: pointers are addresses, much like the house numbers on a street. what could one possibly want to multiply house numbers for? The only valid operations on pointers are:

(i) assignment of pointers of similar type, using a cast operator if need be;
Points and Arrays

(ii) adding an integer to, or subtracting an integer from, a pointer;

(iii) subtracting or comparing two pointers that point to elements of the same array; and

(iv) assigning or comparing a pointer to null.

A null pointer points nowhere.

In the program in Listing 4 four pointers are assigned the respective addresses of four variables of the basic types. The output shows that adding 1 to each pointer increased it by the width of the type it pointed at.

/* Program 7.4; File name:unit7-prog4.c */
#include <stdio.h>

int main ()
{
    char *char_pointer, char_variable;
    int *int_pointer, int_variable;
    float *float_pointer, float_variable;
    double *double_pointer, double_variable;

    char_pointer = &char_variable;
    int_pointer = &int_variable;
    float_pointer = &float_variable;
    double_pointer = &double_variable;

    printf("Address of char_variable: \%u, Address + 1: \%u\n", char_pointer, char_pointer + 1);
    printf("Address of int_variable: \%u, Address + 1: \%u\n", int_pointer, int_pointer + 1);
    printf("Address of float_variable: \%u, Address + 1: \%u\n", float_pointer, float_pointer + 1);
    printf("Address of double_variable: \%u, Address + 1: \%u\n", double_pointer, double_pointer + 1);

    printf("\nMoral: Adding 1 to a pointer does not necessarily increase it by 1. Pointers are not ints.\n");
    return (0);
}

Listing 4: Effect on incrementation of pointers of different types.

Here is the output of Program 7.4 above on my computer. Very probably it will be different on yours.

/* Program 7.4: Output */
Address of char_variable: 3220247843, Address + 1: 3220247844
Address of int_variable: 3220247832, Address + 1: 3220247836
Address of float_variable: 3220247824, Address + 1: 3220247828
Address of double_variable: 3220247808, Address + 1: 3220247816

Moral: Adding 1 to a pointer does not necessarily increase it by 1. Pointers are not ints.

Here are some exercises for you to try.

E1) Create, compile and execute Program 7.1, 7.2, 7.3 and 7.4.
E2) Pointers hold the addresses of program variables. In turn, they have addresses too. Modify Program 7.1 to determine the address at which the pointer x is stored.

In the introduction, we mentioned the close relationship between the pointers and arrays. In the next section, we will discuss this relationship in detail.

### 7.3 POINTERS, ARRAYS AND THE SUBSCRIPT OPERATOR

We have already remarked upon the close relationship between pointers and arrays. That relationship is crystallised in the following truth:

**The name of an array is the pointer to its zeroth element.**

Let primes[1000] be an array of ints. Then

```c
primes == & primes[0]
```

The name primes is itself a pointer! Its value is the address of primes[0]. primes + 1 is the address of the next element of the array, primes + i is address of its (i + 1)th element. Once can use this property in the following way: suppose you wish to scan values into the elements of an array int marks[40]. Then the following for(); loop will do:

```c
for (i = 0; i < 40; i++)
    scanf("%d", (marks + i));
```

This statement works because the arguments of scanf() must be pointers, and marks + i is a pointer to the i-th element after the zeroth of marks. Compare this statement with the corresponding one in Program 6.13:

```c
for (i = 0; i < 40; i++)
    scanf("%d", & marks[i])
```

The program in Listing 5 uses the pointer notation to store the first 1000 primes in an array, int primes[1000].

```c
/* Program 7.5; File name: unit7-prog5.c */
#define TRUE 1
#define FALSE 0
#include <stdio.h>

int main()
{
    int primes[1000], number;
    int i = 0, factor, isprime;
    *primes = 2;
    for (number = 3;; number += 2) {
        isprime = TRUE;
        for (factor = 3; factor * factor <= number; factor += 2)
            if (number % factor == 0)
                isprime = FALSE;
        if (isprime) {
            *(primes + ++i) = number;
            if (i == 1000)
                break;
        }
    }
    return (0);
}
```

Listing 5: Computing first 1000 primes.
The declaration:

```c
int primes [1000];
```

creates space for 1000 primes beginning at the memory address `primes`; `primes` is a
pointer, as is `(primes + ++ i)`. But the alternative declaration:

```c
int * primes;
```

in this program wouldn’t have worked; because it merely defines a pointer to `int`; it
does not create storage to pile the 1000 primes in, as they are generated. If you do
attempt to store them heedlessly, using a statement like:

```c
* (primes + ++i) = number;
```

you would be writing them in places where the compiler does not expect them. In turn,
it will give you unexpected results!

The commonest types of `char` arrays in C are strings. Strings are very convenient to
manipulate for an especial reason: A C sting is a pointer to itself! In plain words, the
string:

"I am a pointer."

is a pointer: it points to the byte which contains the first character of the string, namely
the letter ‘I’. As in all strings, the last byte here is of course the null character. To prove
that this string is in fact a pointer pointing to the byte containing ‘I’, execute the
following simple program:

```c
/* Program 7.6; File name:unit7-prog6c. */
#include <stdio.h>
int main()
{
    int i = 0;
    for (; *("I am a pointer." + i) != '\0'; i++)
        putchar(*("I am a pointer." + i));
    return (0);
}
```

Listing 6: Program showing relationship between pointers and strings.

The operation:

```c
* ("I am a pointer." + i)
```

extracts the ith character of the string. For some value of i this will become the null
caracter. At that time the conditional expression of the for (;;) loop will have become
false, and the loop will be exited.

Precisely because a string is a pointer to itself, and is therefore a memory address, it is
possible to print the addresses of the bytes in which the string is stored. The program
in Listing 7 does just this. We use the %u format conversion character, since memory
addresses are unsigned quantities.

```c
/* Program 7.7; File name:unit7-prog7.c */
#include <stdio.h>
int main ()
{
    char * fact = "We are the best.";
    for (; * fact != '\0'; fact ++)
        printf("%c %u\n", * fact, fact);
    return (0);
}
```

Listing 7: Addresses of the characters in strings.
Given the fact that a string is a pointer to itself, the definition:

```c
char * fact = "We are still the best."
```

makes sense. `fact` now points to the first byte of the string. But a subsequent assignment to `fact`:

```c
fact = "We will always be the best."
```

will cause it to point to a different address of memory, the address where this new string begins. It is important for you to realise that the declaration of `fact` allocates enough memory to hold not merely the pointer variable `fact` but also the entire length of the string. The output of the program in Listing 8 should convince you of this truth. It prints the address of `fact`, and the addresses of each byte of the string to which it points.

```c
/* Program 7.8; File name:unit7-prog8.c */
#include <stdio.h>

int main ()
{
    char * fact = "We will always be the best.";
    printf("Bytes allocated for fact: %d\n", sizeof fact);
    printf("Memory address of fact: %u\n", & fact);
    for (printf("String contents:\n");
        * fact != '\0'; fact++)
        printf("\t\t%c is at %u\n", * fact, fact);
    return (0);
}
```

Listing 8: Example to illustrate memory allocation for strings.

Contrast this with a string array, defined in classic C thus:

```c
static char string_array[] = "An array of chars"
```

Here the address `string_array` is fixed when the program has been compiled and linked. The size of `string_array[]` is just big enough to hold each byte in the string, including its invisible terminator. This means that though each element of `string_array[]` may be altered, the pointer `string_array` cannot be assigned any other value. So the incrementation:

```c
string_array++; /* {WRONG}
because string_array is a fixed address */
```

is taboo; compare this with:

```c
fact++; /* {ACCEPTABLE},
because fact is a pointer */
```

in Listing 5 on page 44 and Listing 6 on the previous page in which `fact` was declared as a pointer to a string; but it could be assigned the address of any `char`. `string_array` is a pointer to the array of that name, sitting at a fixed address.

To recapitulate the definition:

```c
char * x = "We will always be the best."
```

defines a pointer variable `x`, and allocates memory to store the string, as well as the pointer to it. `x` is a variable. It may be given other values as the program executes, the address of any other `char`. On the other hand, the definition:

```c
static char x[] = "We will always be the best"
```

defines an array at a fixed memory location. `x` is a `constant` pointer: it points to the zeroth element of `x[]`. A new assignment cannot be made to `x`. In as much as pointers can be used interchangeably with arrays (because, for any array `a[]`, `a + i == & a [i]`), the converse is also true. For any pointer `p`, the quantity
p [i] has a meaning. It is the value of the ith object from the one that p points to, of the same type. Regarded in this way the symbols [] constitute an operator called the subscript operator. This operator associates from left to right and has a priority as high as the parentheses operator. Now it is not quite necessary that the ith object from the one referenced by p be “ahead of” or “before it” in memory: in other words, the subscript i may be a positive or negative number! It is in this sense only that C allows negative subscripts. Program 7.9 has an example of the subscript operator.

/* Program 7.9; File name: unit7-prog9.c */
#include <stdio.h>

int main()
{
    int i;
    char *fact = "We are the best."
    for (i = 0; fact[i] != \0; i++)
        putchar(fact[i]);
    return (0);
}

Listing 9: Use of subscript operator.

/* Program 7.9: Output */
We are the best.

Given that strings are pointers, it must now be clear that wherever printf() is deployed to print a string, its string argument may be replaced by a pointer to the string. Conversely because strings are arrays, whenever scanf() reads a string, a character array must be in place to hold the characters entered. Consider Program 7.10:

/* Program 7.10; File name: unit7-prog10.c */
#include <stdio.h>

int main()
{
    char tree_1[20], tree_2[20], tree_3[20], tree_4[20]
    printf("Name your favourite tree:");
    scanf("%s", tree_1);
    printf("Name another tree that you like:");
    scanf("%s", tree_2);
    printf("Name a third:");
    scanf("%s", tree_3);
    printf("Name a fourth tree that you appreciate:");
    scanf("%s", tree_4);
    printf("These are your favourite trees: %s, %s, %s and %s\n", tree_1, tree_2, tree_3, tree_4);
    printf("You\'re a good person!!\n");
    return (0);
}

Here are some exercises for you.

E3) Create, compile and execute Program 7.5 – 7.10.

E4) Give the output of the programs in Listing 10 and Listing 11 on the next page.

When scanf() (with the % format conversion character) examines the stream of characters arriving from the input, it ignores all leading white space character–blanks, tabs, newlines or formfeeds; so in Program 7.10 if you responded with:

/* Program 7.11; File name: unit7-prog11.c */
#include <stdio.h>
Programming in C

```c
int main()
{
    int i, *j;
    static int primes[] = {2, 3, 5, 7, 11};
    j = primes;
    for (i = 0; i < size_of primes / size_of(int); i++)
        printf("%d\n", *j++);
    for (j = &primes[4]; j >= &primes[0];)
        printf("%d\n", --*j --);
    return (0);
}
```

Listing 10: Program for exercise 4 on the previous page

```c
/* Program 7.12; File name:unit7-prog12.c */
#include <stdio.h>
int main()
{
    int i, *j;
    static int primes[] = {2, 3, 5, 7, 11};
    j = primes + 4;
    for (; j >= primes + 1; j--)
        printf("%d\n", j[0]);
    for (j = primes, i = 0; i < 4;)
        printf("%d\n", j[++i]);
    j = primes + 4;
    for (i = 0; i <= 4; i++)
        printf("%d\n", j[-i]);
    return (0);
}
```

Listing 11: Program for exercise 4 on the previous page

the array tree_1 [] would still contain the seven characters 'B', 'a', 'n', 'y', 'a', 'n' and \0 only. When a string of characters is read in via %s, then not only does scanf() ignore any initial white space characters, it also stops reading further as soon as a white space character is encountered; this you may verify by entering:

```
Ficus Religiosa
```

for the name of your favourite tree in Program 7.10. The “Ficus”, you will note, is stored in tree_1 [], “Religiosa” in tree_[2]! But if a number is sandwiched between the % and the s, for example %6s, then the specified number of characters is read, unless a white space character is encountered first. So, how can we read in a string with embedded space in C? We will discuss this when we discuss the scanf() function detail in the next section.

7.4 A DIGRESSION ON scanf()

Executing Program 7.10 teaches us that leading white space characters in the input to scanf() with %s are ignored, while embedded white space characters cause it to stop reading further input; but there are two noteworthy cases, which we’ll examine in turn. The first arises when %c is used as the format conversion specifier. Then every single character—including any white space characters—in the input is read. Consider:
Pointers and Arrays

scanf("%d%c%d", &x, &y, &z);

and suppose the input to this statement was:

5m6

the x would get the value 5, y the value ‘m’ and z the value 6 respectively. But if

instead the input was:

<SPACE>7<SPACE>m<SPACE>8

the value assigned to x would be 7, y would be assigned the char value ‘m’ (SPACE), and

z would retain its former value of 6. The reason for this behaviour is as follows: in

reading an int with %d, scanf() ignores all leading white space character; when it

receives the input 7 it assigns it to x. With the %c format conversion character, each

keystroke is significant. IF <SPACE> is typed next, that is the value that y gets. Then

m is received; but the corresponding format specifier (%d) expects an int value. Now

scanf() stops reading as soon as it encounters a character that is not valid for the type

being read. The attempt to assign the value m to z fails. This is illustrated in Program

7.13, appended below, and a sample output:

/* Program 7.13; File name: unit-7prog13.c */
#include <stdio.h>

int main()
{
    int x, z;
    char y;
    printf("Type in the values 5m6 for x, y and z without spaces:");
    scanf("%d%c%d", &x, &y, &z);
    printf("x = %d, y = %c, z = %d\n", x, y, z);
    printf("Type in the values 7<SPACE>m<SPACE>8 for x, y and z: ");
    scanf("%d%c<SPACE>%d", &x, &y, &z);
    return (0);
}

/* Program 7.13: Output */
Type in the values 5m6 for x, y and z without spaces:
    5m6
    x = 5, y = m, z = 6
    Type in the values 7<SPACE>m<SPACE>8 for x, y and z:
    7 m 8
    Now x = 7, y = , z = 6

For clarity, we have underlined the user inputs. Programs 7.9 and 7.10 would have
behaved somewhat differently if the <SPACE> character had been included inside the
control string, separating %c from the second %d:

scanf("%d%c %d", &x, &y, &z);

Then an arbitrary number of blanks are permitted in the input to separate the value
entered for y from that for z. This is illustrated by the two calls to scanf() in Program
7.14. Note the difference in their control strings. Then note the difference in the
assignments they made (or could not make) to x, y and z.

/* Program 7.14; File name: unit7-prog14.c */
#include <stdio.h>

int main() {
    int x, z;
    printf("Type in the values 5m6 for x, y and z without spaces:");
    scanf("%d%c%d", &x, &y, &z);
    printf("x = %d, y = %c, z = %d\n", x, y, z);
    printf("Type in the values 7<SPACE>m<SPACE>8 for x, y and z:");
    scanf("%d%c<SPACE>%d", &x, &y, &z);
    return (0);
}

/* Program 7.14: Output */
Type in the values 5m6 for x, y and z without spaces:
    5m6
    x = 5, y = m, z = 6
    Type in the values 7<SPACE>m<SPACE>8 for x, y and z:
    7 m 8
    Now x = 7, y = , z = 6

For clarity, we have underlined the user inputs. Programs 7.9 and 7.10 would have
behaved somewhat differently if the <SPACE> character had been included inside the
control string, separating %c from the second %d:

scanf("%d%c %d", &x, &y, &z);

Then an arbitrary number of blanks are permitted in the input to separate the value
entered for y from that for z. This is illustrated by the two calls to scanf() in Program
7.14. Note the difference in their control strings. Then note the difference in the
assignments they made (or could not make) to x, y and z.
Program 7.14: Output

Enter values for x, y and z, using spaces for separators

Remember to enter a non-white space character for y: 34 b 45

x = 34, y = b, z = 45

Enter values for x, y and z, but this time no spaces between values for x and y: 45 b 56

/* We typed spaces! */

x = 45, y = , z = 45

The last line of the output above shows that scanf() terminates reading the input stream as soon as it encounters therein a character invalid for the type listed (it was given a 'b' when it expected a numeric value for z, and did not read the 56 meant for it). For decimal ints valid characters are an optional sign, followed by the digits 0 - 9; for octals the digits 0 - 7; and for hex the digits 0 - 9, a - f or A - F. A filed width (e.g. %4d, %6c, %3o, %2x, etc.) may be specified: reading stops as soon as the specified number of characters has been read in, or, as we noted in the last example, a charcter is entered that does not match the type being read. When a field width is supplied with %c, it is assumed that the corresponding argument is a pointer to a character array, at least as big as the field width specified. See Program 7.20.

The second case in which white spaces in the input to a scanf() can be made significant is through a bracketed string read, explained below.

We learnt from Program 7.10 that the %s specification reads a sequence of characters of which the first must be a non-white space character; and reading terminates as soon as a white space character is met with. The corresponding argument must be a character array, at least one byte bigger than the sequence of non-white space characters entered, to hold the terminating null character: scanf() automatically appends it when it’s done reading. If a field width is specified, then no more characters than that width can be read in. Given this propensity of scanf() to stop reading a string as soon as it encounters a white space character inside it, how can string such as:

'I am a string.'

be stored via scanf()? For this purpose one uses a bracketed string read, % [...]

where the square brackets [] are used to enclose all characters which are permissible in the input. If any character other than those listed within the brackets occurs in the input string, further reading is terminated. Reciprocally, one may specify within the brackets those characters which, if found in the input, will cause further reading of the string to
Pointers and Arrays

be terminated. Such input terminators must be preceded by the caret character (\^) . For example, if the tilde symbol (~) is used to end a string, the following scanf() will do:

```
char string[80];
scanf("%[^~]", string);
```

Then, if the input for string consists of embedded spaces, no matter: they will all be accepted by scanf(); and reading will stop when a tilde (~) is entered. This is illustrated in Program 7.15 and its output:

```
/* Program 7.15; File name: unit7-prog15 */
#include <stdio.h>
int main ()
{
    char string[80];
    printf("Enter a string, terminate with a tilde (~)...\n");
    scanf("%[^~]", string);
    printf("%s", string);
    return (0);
}

/* Program 7.15: Output */
Enter a string, terminate with a tilde (~)...I am a string.
I am a string.
```

Though the terminating tilde is not itself included as an element of the string read, it stays in the “read buffer”–the area of memory designated to store the input–and will be picked up by th next call to scanf(), even though you may not want it! This is illustrated by Program 7.16 and its output. There, the second call to scanf() is executed automatically, and the “dangling” tilde is assigned to the char x. The call to putchar() prints the value of x.

```
/* Program 7.16; File name: unit7-pog16.c */
#include <stdio.h>
int main()
{
    char string[80];
    char x;
    printf("Enter a string, terminate with a tilde (~)...\n");
    scanf("%[^~]", string);
    scanf("%c", &x);
    /* The leftover from the last scanf() is read here. This scanf() doesn't wait for you to enter another char. */
    printf("%s", string);
    putchar(x);
    return (0);
}

/* Program 7.16: Output */
Enter a string, terminate with a tilde (~)...I am a string.
I am a string.
```

Compile and execute Program 7.16. You will find that the machine executes the second scanf() without so much as a by-your-leave! Such dangling characters must be “assorted away” by a subsequent call to scanf() with %c, or to getchar() else they may interfere in unexpected ways with subsequent calls to scanf() or getchar().
If there are non-format characters within the control string of a `scanf()`, they must be matched in the input. For example, consider the `scanf()` we’d used in Program 5.16:

```c
scanf("%d-%d-%d", &Day, &Month, &Year);
```

The hyphens in the control string are non-format character. Their presence implied that three `ints` were to read in, with their values to be separated by a single intervening hyphen. Using any other separators would have led to error. To execute Program 7.17 below correctly, you would have to admit that you like C!

```
/* Program 7.17; File name: unit7-prog17.c */
#include <stdio.h>
int main ()
{
    int x, y;
    printf("Type \"I like C!, then values for x and y...\”");
    if (scanf("I like C! %d %d", &x, &y) == 2)
        printf("Thank you! The sum of x and y \n is: %d\n", x + y);
    else
        printf("I will not add those numbers for you!\n");
    return (0);
}
```

Program 7.17 uses another interesting property of `scanf()`, one we’ve seen before (refer to Program 5.16): that it returns the number of items that it has successfully read. If this does not equal the number expected by the program, an error message can be output.

An asterisk(*) in the control string of a `scanf()`, placed between the % sign and the format conversion character which follows it, causes the corresponding input to be ignored. Consider the `scanf()`:

```c
scanf(\"%f %*f %f\", &x, &z);
```

Suppose the input was:

```
12.34 45.46 78.90
```

Then x and z would get the values 12.34 and 78.90 respectively.

The valid characters for an input of `floats` are an optionally signed sequence of decimal digits, followed by an optional decimal point and another sequence of decimal digits, followed if need be by the letter e and E and an exponent, which may be signed. The modifier l with the d or f specifiers is used to read in and assign `longs` and `doubles` respectively. The modifier h with the d specifier reads `short` `ints`. These rules are summarised below:

- `%d` reads `ints` in decimal notation: argument must be a pointer to `int`;
- `%ld` reads `long int`s: argument must be a pointer to `long`;
- `%hd` reads `short` `ints`: argument must be a pointer to `short`;
- `%u` reads `unsigned` `ints`: argument must be a pointer to `unsigned`;
- `%o` reads numbers in octal notation;
- `%lo` reads `long` octal;
- `%ho` reads `short` octals;
- `%x` reads hex `ints`;
- `%lx` reads `long` hex `ints`;
- `%hx` reads short hex `ints`;
- `%e`, `%f` and `%g` read numbers expressed in floating point notation;
- `%le`, `%lf`, `%lg` require arguments to point to `double`;
- `%c` reads single characters; argument is a pointer to `char`;

---

**Programming in C**
Akin to putchar() and getchar(). C provides in addition to printf() and scanf() other functions for greater convenience of string I/O. These are puts() and gets().

puts() outputs its string argument on the terminal’s screen:

puts ("This is an easy function to use, isn’t it?");

Again, the argument of puts() may be the name of a string array. Consider the following program:

/* Program 7.18; File name: unit7-prog18.c */
#include <stdio.h>
int main ()
{
    static char string_1[] = "I hope that you enjoy C.";
    static char string_2[] = "It’s great fun!"
    puts (string_1);
    puts (string_2);
    return (0);
}

Neither of the arrays string_1[] or string_2[] ends with a newline; yet the output of the program is printed in two lines, with the cursor stationed on the third, as you can see by executing the program. The puts function replaces the terminating null character of its string argument by a newline.

gets() gets the characters that you type in at the keyboard, until <CR> is pressed. This bunch of characters is stored as a string array, for which memory must be allocated before gets() is invoked. In Program 7.19 below that array of 55 chars is get_string[]. The <CR> indicating end of keyboard input is not stored in the array; gets() replaces it by a null character, marking the end of the input string.

/* Program 7.19; File name: unit7-prog19.c */
#include <stdio.h>
int main ()
{
    char get_string[55];
    puts ("Tell me how you are");
    puts ("in 55 keystrokes or less.");
    puts ("That’s inclusive of the <CR>,");
    puts ("which we expect you to type presently!");
    printf("%s", "Begin here: ");
    gets (get_string);
    puts ("Thanks for that detailed description.");
    return (0);
}

gets() does more than merely getting a string from the keyboard. If it was able to read the input string successfully, it returns the address of the array to which the string is assigned; if not, or if no characters are read, it returns the null pointer, which by convention has the value 0, and points nowhere.

E5) Describe what happens if you input more than 55 characters to Program 7.19.

E6) Give the output of the following program, in which a field width is specified with %c:

/* Program 7.20; File name: unit7-prog20.c */
#include <stdio.h>
int main ()
{
    char array[7]; int i;
    
}
Programming in C

```c
printf("Type in the letters ABCDEFG: ");
scanf("%7c", array);
for (i = sizeof array - 1; i >= 0; i --)
    putchar (array [i]);
return (0);
}
```

E7) Is the array [] of Program 7.20 a string?

We close this section here. We will discuss multidimensional arrays in the next section.

## 7.5 MULTIDIMENSIONAL ARRAYS

The simplest example of a two-dimensional array is a matrix, defined as a rectangular array of numbers. The picture is quite similar to that of rows of chairs in a classroom. Each element is accessible by two subscripts, the first referring to the row in which the element lies, the second to the column number within that row. Such a two-dimensional array is declared quite simply:

```c
float matrix [5][6];
```

The declaration for `matrix [][]` reserves 120 bytes of storage; the first subscript, running over the rows, would vary from 0 through 4; the second from 0 through 5. The element of a two-dimensional array are stored in “row major” form. This means that they are so arranged in memory that, for minimal access time, it is the rightmost subscript that should be made to vary the faster. To process every element of the array `matrix [][]`, you would need two `for (;;)` loops. The outer loop would run over the rows, the second over the elements within a row:

```c
for (i = 0; i < 5; i++)
    for (j = 0; j < 6; j++)
        matrix [i][j] = 0.0;
```

Note that this arrangement of loops makes the rightmost subscript vary the fastest. You may find the phrase “minimal access time” a bit strange, if you pause to consider that the CPU can retrieve any RAM location in the same amount of times; but if the array was loaded from disk (where also it is stored in row major form), it is possible that not all if it may have been “read in” in a given disk access. For large arrays such may indeed be the case in “virtual memory” machines, where disk storage may be thought of as an extension of RAM. If an array element is required that is not currently in RAM, a (very much slower) disk access would be needed to fetch it. If this happens frequently, with the CPU being forced to pause while the disk is searched, (as, for example, in processing a large two-dimensional array in column major order) the machine will spend more time in performing disk I/O than in computing. It is then said to be “thrashing”, very much like a novice swimmer splashing the water vigorously, yet unable to make any forward progress. Designers of operating systems are interested in these considerations.

The declaration:

```c
static int fifty_seven [5][5] =
{
    19, 8, 11, 25, 7, 12, 1, 4, 18, 0,
    16, 5, 8, 22, 4, 21, 10, 13, 27, 9,
    14, 3, 6, 20, 2
};
```

initialises a $5 \times 5$ array of `ints`; `fifty_seven [0][0]` has the value 19, `fifty_seven [1][0]` is 12 and `fifty_seven [2][0]` is 16.
Alternatively, you may declare such an array by initialising each row:

```c
static int fifty_seven[5][5] =
{
{19, 8, 11, 25, 7},
{12, 1, 4, 18, 0},
{16, 5, 8, 22, 4},
{21, 10, 13, 27, 9},
{14, 3, 6, 20, 2}
};
```

This notation teaches us that a two dimensional array is actually a one dimensional array, each element of which is itself an array. Realise, therefore, that such an array can be thought of as an array of pointers. This idea is explored more fully below.

Program 7.21 reads in two matrices conformable to multiplication, and computes their product. In this program mat_1, a 4 × 5 matrix multiplies mat_2, a 5 × 6 matrix, and stores the result in mat_3, a 4 × 6 matrix. The [i][j]th element of mat_3 is obtained by multiplying the ith row of mat_1 into the jth column of mat_2, each element of the row to the corresponding element of the column, and then summing the several partial products.

```c
/* Program 7.21; File name: unit7-prog21.c */
#include <stdio.h>
int main()
{
    int i, j, k, mat_1[4][5], mat_2[5][6], mat_3[4][6];
    printf("This program multiplies two matrixes\n");
    printf("of dimensions 4 x 5 and 5 x 6.\n\n");
    printf("Enter 20 elements for matrix # 1: ");
    for (i = 0; i < 4; i++)
        for (j = 0; j < 5; j++)
            scanf("%d", &mat_1[i][j]);
    printf("Enter 30 elements for matrix # 2: ");
    for (i = 0; i < 5; i++)
        for (j = 0; j < 6; j++)
            scanf("%d", &mat_2[i][j]);
    for (i = 0; i < 4; i++)
        for (j = 0; j < 6; j++)
            mat_3[i][j] = 0;
    for (k = 0; k < 5; k++)
        for (i = 0; i < 4; i++)
            for (j = 0; j < 6; j++)
                mat_3[i][j] = mat_3[i][j] + mat_1[i][k] * mat_2[k][j];
    printf("\nThe given matrixes are:\n\n");
    for (i = 0; i < 4; i++)
        for (j = 0; j < 5; j++)
            printf("%3d", mat_1[i][j]);
    printf("\n");
    printf("\nand:\n\n");
    for (i = 0; i < 5; i++)
        for (j = 0; j < 6; j++)
            printf("%3d", mat_2[i][j]);
    printf("\n");
    printf("\nTheir product is:\n\n");
    for (i = 0; i < 4; i++)
        for (j = 0; j < 6; j++)
            printf("%3d", mat_3[i][j]);
    printf("\n");
}
```
return (0);
}

A string is a one-dimensional array: it has elements along a line. Any element can be specified by a single subscript, which gives its position in the string. If you had a bunch of strings, of equal length, each in a separate line, you would need two subscripts to specify a particular character: the first would refer to the row, e.g. the zeroth, first, second, ... string in which the char lies, the second to its position in that string, the column number.

This is a two-dimensional array of chars. It has 25 columns and 3 small rows.

The picture we have so far presented is for string arrays unnecessarily restrictive, because strings by their nature will be of arbitrary length. However, since strings are pointers to char, it is possible to think of a bunch of several strings as a one dimensional array of pointers. Consider the declaration:

```c
static char * pntr_array [] =
{
    "This is the first string.",
    "This is the second string, a little longer.",
    "This is the third string.",
    "And this is the fourth and final string."
};
```

First, what does this declaration mean? Is pntr_array an array of pointers to char, or is it a pointer to an array of chars? This may appear to be a confusing question but is easily answered if you reason as follows: since the subscript operator, [], has a higher precedence than the dereferencing operator, * (pntr_array []), is a char, pntr_array [] is a pointer to char, and pntr_array is an array of pointers to char. Each element of it is a pointer to a string, because a string is a pointer to itself. The declaration of pntr_array[] reflects this:

```c
static char * pntr_array [] =
{
    pntr_1, pntr_2, pntr_3, pntr_4
};
```

where pntr_1, pntr_2, etc, are pointers to the first, second, etc. strings respectively. pntr_array [] is an array of four pointers. The program Listing 12 on the facing page prints these strings: The outer loop of Program 7.22 runs over the four pointers; the inner loop applies the subscript operator with j as subscript to extract the jth element of the ith string. (Realise that as a loop condition the expression:

```c
pntr_array [i][j]
```

is equivalent to:

```c
pntr_array [i][j] ! = '\0'
```

The inner loop is exited when the string terminator is sensed.)

Program 7.20 creates a ragged array, in which only as much space is allocated to store the strings as is required for each separately (And for their pointers). If we had used a two dimensional array to store these strings, the row dimension would have been at least as great as to accommodate the largest string. That would have wasted valuable memory when short strings had to be stored.
/ Program 7.22: File name: unit7-prog22.c */
#include <stdio.h>
int main()
{
    int i, j;
    static char *pntr_aray[] =
    {'This is the first string."
        "This is the second string, a little longer.",
        "This is the third string."
        "And this is the fourth and final string."
    };
    for (i = 0; i < 4; i++) {
        for (j = 0; pntr_aray[i][j]; ++)
            putchar((pntr_aray)[i][j]);
        putchar("\n");
    }
    return (0);
}

Listing 12: Array of strings as two dimensional arrays

char ** pntr_aray;

Clearly, *pntr_aray is a pointer to char; when *pntr_aray is dereferenced, via
**pntr_aray, it yields a char value. Then, if x is a pointer to char, *pntr_aray
can be assigned the value of x. The implication is that pntr_aray itself can be thought
of as the address of x, a pointer. So pntr_aray declared as above can be thought as a
pointer to a pointer. The contents of pntr_aray are an address. That address tells
where x is stored. In x itself is another address. That address is the address of a char
value. ** marks for double indirection.

Now suppose that x, y, z, w, are a number of a string pointers, i.e. assume that
they are defined thus:
char * x = "I am string.";
char * y = "I am one, too.";
char * z = "I am a third string.";
char * w = "And I am the fourth and final here.";

Then it is possible to make the assignment:
* pntr_aray = x;

Using the versatile subscript operator, another way of saying the same thing as:
pntr_aray [0] = x;

The one may continue:
pntr_aray [1] = y;
pntr_aray [2] = z;
pntr_aray [3] = w;

Or, equally,
pntr_aray [0] = "I am a string.";
pntr_aray [1] = "I am one, too.";
pntr_aray [2] = "I am a third string.";
pntr_aray [3] = "And I am the fourth and final here.";

Thus, the declaration
char ** pntr_aray;
implies an equivalent (though undefined) two-dimensional ragged array! An example of this usage may be seen in Program 7.23.

E8) Rewrite Program 7.21 to multiply two matrices using the pointer notation.

E9) Write a C program to test if a string entered from the keyboard is a palindrome, that is, that it reads the same backwards and forwards, e.g. “Able was I ere I saw Elba.”

E10) Give the output of the following program:

```c
/* Program 7.23; File name: unit7-prog23.c */
#include <stdio.h>
int main ()
{
    char ** pntr_aray;
    pntr_aray [0] ="I am a string."
    pntr_aray [1] ="I am one, too."
    pntr_aray [2] ="I am a third string."
    pntr_aray [3] ="And I am the fourth and final here."
    putchar (pntr_aray [0][0]);
    putchar (pntr_aray [0][1]);
    putchar (pntr_aray [2][11]);
    putchar (pntr_aray [1][5]);
    putchar (pntr_aray [0][1]);
    putchar (pntr_aray [1][6]);
    putchar (pntr_aray [1][5]);
    putchar (pntr_aray [0][8]);
    putchar (pntr_aray [0][1]);
    putchar (pntr_aray [3][13]);
    putchar (pntr_aray [0][10]);
    putchar (pntr_aray [0][11]);
    putchar (pntr_aray [2][11]);
    putchar (pntr_aray [0][1]);
    putchar (pntr_aray [2][7]);
    putchar (pntr_aray [2][8]);
    putchar (pntr_aray [2][9]);
    putchar (pntr_aray [0][7]);
    putchar (pntr_aray [0][1]);
    putchar (pntr_aray [0][8]);
    putchar (pntr_aray [3][14]);
    putchar (pntr_aray [3][15]);
    putchar (pntr_aray [2][18]);
    putchar (pntr_aray [2][8]);
    putchar (pntr_aray [0][13]);
    return (0);
}
```

We have now completed the unit. You may like go through the summary given in the next section to recapitulate what we have studied so far.

## 7.6 SUMMARY

In this unit we have studied the following:

1. Pointers are `int` like variables that hold the addresses of other variables.

2. The rules of pointer arithmetic: The incrementation of pointers is different from the incrementation of other variables. For example, depending on whether it points to `int`, `char`, on incrementation it points to 8 bytes or one byte ahead.
3. Three operators are used in working with pointers, the “address of” operator &,
the “contents of” operator * and the subscript operator [ ].

4. The name of a array is a pointer to its zeroth element. So, all the rules of pointer
arithmetic apply to array name.

5. The format conversion rules followed by scanf().

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>Reads ints in decimal notation. The argument must be a pointer to int.</td>
</tr>
</tbody>
</table>
| %i     | Reads an integer with a prefix or suffix. Prefixes allowed are a sign (+ or -) or 0
to denote an octal constant, or one of 0x, 0X to denote a hex constant. Suffixes allowed are u or U to denote an unsigned int and l or L to denote a long int. |
| %ld    | Reads long ints; the argument should be a pointer to long. |
| %hd    | Reads short ints; argument must be a pointer to unsigned. |
| %u     | Reads unsigned ints; argument must a pointer to unsigned. |
| %o     | Reads numbers octal notation. |
| %lo    | Reads long octals. |
| %ho    | Reads short octals. |
| %x, %X | Reads hex ints. |
| %lx, %lX | Reads long hex ints. |
| %hx, %hX | Reads short hex ints. |
| %e, %E, %f, %f and %G | is for reading numbers expressed in floating point notation. |
| %le, %lE, %lf, %lf and %lG | is for reading double. The arguments should be pointers to double. |
| %Le, %LE, %Lf, %Lf and %LG | The arguments must be pointers to long double. |
| %c     | Reads single characters. White space characters are significant. Therefore, to
read the next non-space character, use %ls. Argument corresponding to %c is a pointer to a char. |
| %s     | Used for reading a character string. Characters are read until a white space
character is encountered. The corresponding argument is an array of
chars, which must be big enough to hold the characters entered, plus the null
character which scanf() appends. |
| %p     | Reads a pointer. The argument expected is a pointer to void. |
| %n     | records the number of characters read so far in this call to scanf(); while no
characters are read from the input stream against \%n, its corresponding data
argument is a pointer to int. |
% [...]

Reads a string until a character that is not found in the square braces is encountered. If the first character in the braces is ^, then the subsequent characters listed there serve as terminators. The input stream is read until one of the characters listed after the caret is encountered.

%%

Matches a single percent sign % in the input stream, without any assignment being made.

6. We discussed multidimensional arrays and ragged arrays in which different rows are allowed to have different length.

7.7 SOLUTIONS/ANSWERS

E2) Add the line

```c
printf("The address of x is %u", &x);
```

before the `return (0);` statement.

E5) The program will crash. This is because there is no memory available and the program will try to write to a memory location not allocated to the program.

E8) The program is given in Listing 13. Note the second * in the statements

```c
*(mat_3+i)+j= 0;
```

for accessing the value mat_3[i][j].

```c
/* Program 7.21; File name: unit7-prog21.c */
#include <stdio.h>

int main()
{
    int i, j, k, mat_1[4][5], mat_2[5][6], mat_3[4][6];
    printf("This program multiplies two matrixes\n");
    printf("of dimensions 4 x 5 and 5 x 6.\n\n"لة
    printf("Enter 20 elements for matrix # 1: ");
    for (i = 0; i < 4; i++)
        for (j = 0; j < 5; j++)
            scanf("%d", (*(mat_1+i)+j));
    printf("Enter 30 elements for matrix # 2: ");
    for (i = 0; i < 5; i++)
        for (j = 0; j < 6; j++)
            scanf("%d", (*(mat_2+i)+j));
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 6; j++) {
            *(mat_3+i)+j)= 0;
            for (k = 0; k < 5; k++)
                *(mat_3+i)+j) = *(mat_3+i)+j) + 
                     *(mat_3+i)+k) * *(mat_2+k)+j);
        }
    printf("The given matrixes are:\n\n");
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 5; j++)
            printf("%3d", mat_1[i][j]);
        printf("\n");
    }
    printf("\n\n\n");
```
for (i = 0; i < 5; i++) {
    for (j = 0; j < 6; j++)
        printf("%3d", mat_2[i][j]);
    printf("\n");
}
printf("\nTheir product is:\n\n");
for (i = 0; i < 4; i++) {
    for (j = 0; j < 6; j++)
        printf("%3d", mat_3[i][j]);
    printf("\n");
}
return (0);

Listing 13: Solution to exercise 8.

E9) See Listing 14 for the program:

#include <stdio.h>
#define NUM_CHAR 40
int main()
{
    int i = -1,k;
    char c, input[NUM_CHAR];
    printf("Enter a string of length atmost %d\n", NUM_CHAR);
    printf("Terminate your string with <CR>:\n");
    printf("All your characters must be in lower case.\n");
    while ((c = getchar()) != '
') {
        i++;
        input[i]=c;
    }
    for (k = 0; k <= i/2; k++)
        if (input[i-k] != input[k]){
            break;
        }
    if (k < i/2)
        printf("The string is not a palindrome.");
    else
        printf("\nThe string is a palindrome.");
    return (0);
}

Listing 14: Solution to exercise 9.
UNIT 8 FUNCTIONS

8.1 INTRODUCTION

Large programs for large “real world” applications require quite different techniques for their creation than the ones we have so far studied. As programs grow in size and complexity, it becomes more and more difficult to keep track of the logic; so programs are divided into separate modules called functions, with each function reflecting a well-understood activity, and of manageable proportions.

The idea is to divide and conquer. For example, imagine a program to manage your personal savings bank account, a program that would monitor every activity that is recorded in your pass book on a computer. Now there are several things to be considered while designing such a program: for one, you may deposit cash. The program should return your updated balance; or you may want to withdraw some, in which case the program should report if the amount to be withdrawn exceeds your current balance; if not, it should return the new balance.

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

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

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

Sometimes a function written for one problem can be used in different problems; it can serve as a “building block” to help build other programs. A function to validate a date...
Programming in C

would be equally useful in a program for railway reservations and in one to handle savings accounts. C carries this “pick up and reuse” design philosophy to a greater extent than other languages. Even input and output are implemented via functions available through “standard libraries” provided with your compiler.

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

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

Objectives

After studying this unit, you should be able to

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

8.2 FUNCTION PROTOTYPES AND DECLARATIONS

As we saw in the introduction, basically, a function is lines of C code grouped into a named unit, written to perform an identifiable activity. Such a module can be invoked by a “calling” program. When a function is called, control passes from the calling program to the function. But just before this happens the current value of the Program Counter which is the address of the next instruction to be executed, is saved. When the function has been executed, control returns to the calling program, to the address saved from the PC. Execution begins where it had stopped prior to the call. The return address is stored in a data structure known as a stack, which has the property that the last item pushed in is the first to be popped out. Therefore if, while a called function is executing, it invokes another function, the current PC contents are stored on top of the stack, i.e. on top of the last return address. The stack is “pushed” in. When the second function ends its execution, it returns control back to the first function where it had left off. The stack is “popped”. The return address that’s now on top is the one to which control will return when this function finishes its execution. We will discuss more about this when we discuss stacks in the next block.

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

For our first example of functions, let’s turn to the program in Listing 1 on page 67. It determines the area of a triangle whose sides are given. It consists of three functions, other than `main()`, viz. `begin()`, `form_triangle()` and `area_triangle()`. These functions are declared at the beginning of the program, through **functions prototypes**:

```c
void begin (void):
int form_triangle (double a, double b, double c);
double area_triangle (double x, double y, double z);
```

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

1) to expect to encounter them further down the program;
2) the return types of the function, e.g. `void`, `int` or `double`;
3) the number and kind of arguments the function is to handle: none whatever (represented by the keyword `void` in the parentheses, as in `begin()`), or of specified type as in the declarations of `form_triangle()` and `area_triangle()`.

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

```c
double area_triangle (double, double, double);
```

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

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

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

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

```c
begin ()
```

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

The first line of any function is its **declarator**:

```c
void begin (void)
int form_triangle (double alpha, double beta, double gamma)
double area_triangle (double lambda, double mu, double nu)
```

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

Note that the function’s declarator is not terminated by a semicolon; it must agree with the function’s **prototype**, the declaration which is made before `main()`, in all particulars: return type, name, and parameters and their types. Parameter names in a function’s prototype and its declarator may be different.
The function’s declarator is followed by its body, enclosed in braces. Further variables may be declared inside the body of the function, if need be: their scope is limited to the periphery of the function. We will see an example of this shortly, in the function `area_triangle()`.

The second function, named `form_triangle()` in the program in Listing 1 on the next page determines if the three numbers input are such as to form the sides of a triangle. (We would look a little silly trying to find the area of a triangle that couldn’t exits!) Recall that in a triangle the sum of any two of its side must exceed the length of the third. That is how our function decides if the lengths passed to it can form a triangle. The values of the three lengths must be “handed over” to the function, which should then determine if a triangle is possible, returning 1 if true, 0 if false. Here is the function `form_triangle()`:

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

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

In the program in Listing 1 on the facing page, after control returns to `main()` from `begin()`, it invokes this function in the following way:

```c
if (form_triangle(side_1, side_2, side_3))
```

The call is executed from within the if(); the arguments `side_1`, `side_2` and `side_3` are “injected” into the three parameters of `form_triangle()`: `side_1` is passed to `alpha`, `side_2` to `beta` and `side_3` to `gamma`. The instructions within the function itself are written in terms of its formal parameters; but the values used in the function’s computation are the values that came from `main()`. You might be wondering about the difference in meaning between arguments and parameters: the function call from `main()` uses arguments; the function definition itself is in terms of parameters. `side_1`, `side_2` and `side_3` are arguments; `alpha`, `beta` and `gamma` are parameters. It should be obvious that the types of formal parameters in a function’s definition should match the type of the arguments used in the call, each to each.

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

```c
return (expression);
```

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

```c
return;
```

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

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

```c
if (form_triangle (side_1, side_2, side_3))
    printf("Area of triangle is: \%f\n",
    area_triangle (side_1, side_2, side_3));
```

the function area_triangle() illustrates a further optional feature of function: local variables:

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

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

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

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

area_triangle() returns this value to main(), via its local variable, area. The program is given in Listing 1.

/* Program 8.1; File name:unit8-prog1.c */
#include <stdio.h>
#include <math.h>
#include <iostream>

void begin(void); /* function prototypes */
int form_triangle(double alpha, double beta, double gamma);
double area_triangle(double x, double y, double z);
int main(void)
{
    double side_1, side_2, side_3;
    begin();
    scanf("%lf,%lf,%lf", &side_1, &side_2, &side_3);
    if (form_triangle(side_1, side_2, side_3))
        printf("Area of triangle is: \%f\n",
        area_triangle(side_1, side_2, side_3));
    else
        printf("The lengths that you entered cannot form a triangle.\n";
        return (0);
}
void begin(void)
{
    printf("This program determines whether three numbers\n");
    printf("that you enter can form the three sides of a\n");
    printf("triangle. If they can, the program prints\n");
    printf("its area...Enter values for the three sides:";
}
int form_triangle(double alpha, double beta, double gamma)
{
    if ((alpha + beta > gamma) &&
        (beta + gamma > alpha) &&
        (gamma + alpha > beta))
        return 1;
    else
Programming in C

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

Listing 1: Program for finding the area of a triangle.

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

int form_triangle();

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

double area_triangle();

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

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

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

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

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

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

E1) Write a C function that determines whether an integer passed to it is a prime number; if it is not, get main() to print its smallest factor.
E2) Write a C function that determines whether an integer passed to it is a perfect square; if it is, get main() to print the square root.

E3) Write a C function to return the number of primes less than a positive integer passed to it.

What happens to the value of variables used in functions after the control leaves the function? Which are the variables that a function can access apart from the ones declared in the functions? These are some of the questions that we will discuss in the next section.

8.3 FUNCTIONS AND SCOPE

When one function is invoked by another, the arguments used by the “caller” are transmitted, as we have seen, into the formal parameters of the “called” function. However, the called function is sent only copies of the caller’s variables; it can use them in its computation, but it cannot change them, except locally, within its own boundary. Such changes are not reflected in the values of the caller’s variables. The called function has no power to change them. The reason for this is that the function’s variables are created in the invocation at addresses different from those of the caller’s variables. The called function has no access to the caller’s address space, which remains insulated from it. This is illustrated in Program 8.2 in which the function

float inflation() is sent the prices of milk, potatoes and bread of a few years ago. It inflates them, prints them, and returns control back to main(). However, the corresponding variables in main remain unaltered.

#include <stdio.h>
void inflation(float m, float p, float b);
int main()
{
    float milk_price = 2.50, potatoes_prices = 0.50, bread_price = 2.85;
    printf("Unit prices of milk, potatoes and bread in 1984\nand bread in 1984\n");
    printf("were %.2f, %.2f and %.2f respectively,\n\n", milk_price, potatoes_prices, bread_price);
    printf("Let’s see what inflation did to them...\n\n");
    printf("Transferring these \nprices to inflation ()\n\n");
    inflation(milk_price, potatoes_prices, bread_price);
    printf("\nNow control is back in main ()\n\n");
    printf("Print those prices once more...\n\n");
    printf("milk: %.2f, potatoes: %.2f and bread: %.2f\n\n", milk_price, potatoes_prices, bread_price);
    printf("What?? They haven’t changed at all \nin main ()!!!\n");
    return (0);
}
void inflation(float milk, float potatoes, float bread)
{
    float percent_rise, annual_price_ratio;
    int i, num_years;
    printf("We welcome you to inflation ()\n\n");
    printf("enter a % rise for inflation (range 0...100):\n");
    scanf("%f", &percent_rise);
    printf("\nEnter number of years inflation occurred: ");
    scanf("%d", &num_years);
    annual_price_ratio = (100 + percent_rise) / 100;
for (i = 1; i <= num_years; i++) {
    milk *= annual_price_ratio;
    potatoes *= annual_price_ratio;
    bread *= annual_price_ratio;
}

printf("As a result of %.0f%% inflation, \
prices %d years later are...\n", percent_rise, num_years);
printf("milk: %.2f, potatoes: %.2f and bread: %.2f\n\n",
milk, potatoes, bread);
printf("Thank you for bearing with inflation()...\n\n");
printf("Now return to main ()...\n\n");
}

/* Program 8.2: Output */
Unit prices of milk, potatoes and bread in 1984
were 2.50, 0.50 and 2.85 respectively.
Let's see what inflation did to them...
transferring these prices to inflation ()...
We welcome you to inflation ()...
Enter a % rise for inflation (range 0...100): 10
Enter number of years inflation occurred: 10
As a result of 10% inflation, prices 10 years
later are...
milk 6.48, potatoes 1.30, bread 7.39
Thank you for bearing with inflation()...
Now return to main ()...
Now control is back in main ()...
Print those prices once more...
milk: 2.50, potatoes: 0.50 and bread: 2.85
What?? The haven't changed at all in main ()!!!

The automatic variables declared within a function's body are local to it. They are created when control passes to the function, are in existence until control remains within the function, and are annihilated when it departs therefrom. In Program 8.3 the function f1() is called twice in succession by main(). In its first invocation f1()'s automatic variable x is assigned a value which it returns to main(). But conditions are such that in its second invocation x cannot be assigned a value; one might suppose then that the last value of x, the value it had when the function was first invoked, may still be available: not so. That variable was destroyed and its value was lost when control returned to main(). The second call to f1() returns an unpredictable value for x. In fact, by a little modification of Program 8.3 you can easily show that each call to f1() creates x at a different address.

/* Program 8.3; File name:unit8-prog3.c */
#include <stdio.h>
int f1(int a, int b);
int main()
{
    int alpha, beta, gamma;
    alpha = 10;
    beta = 20;
gamma = f1(alpha, beta);
printf("alpha = %d, beta =%d, gamma = %d\n", alpha, beta, gamma);
alpha = 20;
beta = 30;
gamma = f1(alpha, beta);
printf("alpha = %d, beta = %d, gamma = %d\n", alpha, beta, gamma);
return (0);
}

int f1(int p, int q)
{
    int x;
    if (p * q < 300)
        x = p * q;
    return (x);
    return (0);
}

/* Program 8.3: Output */
alpha = 10, beta = 20, gamma = 200
alpha = 20, beta = 30, gamma = 30

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

static int x;

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

/* Program 8.4; File name:unit8-prog4.c */
#include <stdio.h>
int f1(int a, int b);
int main()
{
    int alpha, beta, gamma;
    alpha = 10;
    for (beta = alpha; beta > alpha / 3; beta--){
        printf("Proceeding to f1 ()...\n");
        gamma = f1(alpha, beta);
        printf("alpha = %d, beta = %d, gamma =%d\n", alpha, beta, gamma);
    }
    return (0);
}
int f1(int p, int q)
{
    static int x, count = 0;  /* initialisation unnecessary, count is by default 0 */
    printf("...Entered f1 ()...\n");
    printf("Current number of calls to function: %d\n", ++count);
    printf("On entry x was found to have the value: %d\n", x);
    if (p * q > 70)
        x = p * q;
    printf("...Returning to main ()...\n");
Programming in C

\[
\text{return } (x);
\]
\[
\text{return } (0);
\]

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

\begin{verbatim}
/* Program 8.5; File name:unit8-prog5.c */
#include <stdio.h>
#define LEAP_YEAR(Year) Year % 4 == 0 && Year % 100 != 0
    || Year % 400 == 0
int Day, Month, Year;
int valid_date();
int main()
{
    printf("This program finds the day corresponding \\
to a given date.\n");
    printf("Enter date, month, year ... format \\
is dd-mm-yyyy.\n");
    printf("Enter a 1 or 2-digit number for day, \ \\
followed by a\n");
    printf("dash, followed by a 1 or 2-digit number \ \\
for month,\n");
    printf("followed by a dash, followed by a 2 or \ \\
4-digit number\n");
    printf("for the year. Valid year range is 1582 to\ \\
4902, inclusive.\n");
    printf("A 2-digit number for the year will imply \ \\
20th century\n");
    printf("years.\nEnter dd-mm-yyyy: ");
    while (scanf("%d-%d-%d", &Day, &Month, &Year) != 3)
    {
        printf("Invalid number of arguments or hypens \ \\
mismatched\n");
        printf("Re-enter: ");
    }
    if (valid_date())
        printf("Date entered is OK.\n");
    else
        printf("Date entered is not OK.\n");
    return (0);
}
int valid_date()
{
    if (Year < 0)
        return 0;
    if (Year < 100)
        Year += 1900;
    if (Year < 1582 || Year > 4902)
        return 0;
    if (!(LEAP_YEAR(Year)) && (Month == 2) && (Day > 28))
        return 0;
    if (Month < 1 || Month > 12)
        return 0;
    if (Day < 1 || Day > 31)
        return 0;
    if ((Day > 30)
        && (Month == 4 || Month == 6 || Month == 9 || Month == 11))
        return 0;
}
\end{verbatim}
This example nicely illustrates the major advantage of functions. valid_date may be used with any program that processes dates (not of course the kind that we eat!) Most of it has been adapted from Program 5.16.

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

The function is2or5factor() updates the two global variables int two_occurs and five_occurs declared before main(). They are not passed to the function by main(), they’re already visible in it. The ternary operator in main() yields the number of zeros in 1000!

/* Program 8.6; File name:unit8-prog6.c */
#include <stdio.h>
void is2or5factor(int n);
int two_occurs, five_occurs;
int main()
{
    int number;
    for (number = 1000; number >= 2; number --)
        is2or5factor(number);
    printf("Number of zeros in 1000! is: %d\n",
        two_occurs > five_occurs ? five_occurs : two_occurs);
    return (0);
}

void is2or5factor(int number)
{
    int quotient;
    quotient = number;
    while (quotient % 2 == 0){
        two_occurs++;
        quotient /= 2;
    }
    while (quotient % 5 == 0){
        five_occurs++;
        quotient /= 5;
    }
}

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

One important feature of global variables that should be kept in mind is that if a function has a local variable of the same name as a global variable, the local variable has precedence over the global. Analogously, a variable declared within a block has precedence over an external variable of the same name. Recall the behaviour of
Programming in C

Programs 6.16 - 6.19. The local is “in scope” when control enters the function. The external definition is ignored.

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

```c
extern int x;
```

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

```c
extern int x = 5; /* WRONG */
```

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

There is one last scope rule relating to externally defined `static` variables that must be stated here for completeness (we hope that you will christen your variables carefully, and may never need to use it): suppose there is a situation in a multi-file program in which it is required to use a variable within a file, and within it only; suppose also that a variable of the same name is defined globally in another file. To restrict the scope of a variable to the functions contained in a single file, and to prevent name conflicts across other files, the variable is defined `static` externally in the file to which it must be confined. For example, suppose that your program and the functions that it uses are stored in three files, file_1, file_2 and file_3:

<table>
<thead>
<tr>
<th>contents of file_1</th>
<th>contents of file_2</th>
<th>contents of file_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x;</td>
<td>int y;</td>
<td>static int x, y;</td>
</tr>
<tr>
<td>main ()</td>
<td>func_2()</td>
<td>funct_4()</td>
</tr>
<tr>
<td>{}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>extern int y;</td>
<td>extern int x;</td>
<td></td>
</tr>
<tr>
<td>func_1()</td>
<td>func_3()</td>
<td>func_5()</td>
</tr>
<tr>
<td>{}</td>
<td>{} int y;</td>
<td>{}</td>
</tr>
</tbody>
</table>

Then `int x` defined in file_1 is available in `main()` and `func_1()` because it is external to it in the same file, and in `func_3()`, because of the `extern` qualifier before its definition in file_2. `int y`, defined in file_2 at the top of `func_2()` is available also in `func_1()`; it’s redefined inside `func_3()`, where the automatic variable has precedence over its externally defined namesake; the external `static` `int` variables `x` and `y` in file_3 are accessible only inside that file; they have nothing to do with other variables of the
same names in file_1 and file_2. Precisely how multi-file programs are compiled and linked is a compiler-operating system feature that differs from compiler to compiler. We can only refer you to your system’s manuals.

To understand these rules let’s look at Program 8.7, which consists of three functions other than main(), func_1(), func_2 and func_3() each of type int. global_var is a global variable and therefore accessible by every function except func_3, wherein it’s redefined:

```c
int global_var = 2;
```

The int variable a is defined externally to func_2() and func_3(). Naturally it is accessible in these functions; but it is also available in main() and in func_1(), because it is redeclared in these functions:

```c
extern int a;
```

when main() calls func_1() it sends in the values 5, 7 and 5 to L, M and N respectively. The function changes global_var, M and a: the change in M is not reflected back in main(). M is only a local variable, a country cousin, so to speak. And the function returns a value to main()’s int variable, c.

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

```c
/* Program 8.7; File name: unit8-prog7.c */
#include <stdio.h>
int func_1(int l, int m, int n);
int func_2(int p, int q, int r);
int func_3(int x, int y, int z);
int global_var = 5;
int main()
{
    extern int a;
    int b = 5, c;
    a++;
    printf("main () changes extern a to %d\n", a);
    c = func_1(global_var, a, b);
    printf("\nOn return from func_1 ()\n");
    printf("a = %d\n", a);
    printf("b = %d\n", b);
    printf("c = %d\n", c);
    printf("global_var = %d\n", global_var);
    c = func_2(a, b, c);
    printf("\nOn return from func_2 ()\n");
    printf("a = %d\n", a);
    printf("b = %d\n", b);
    printf("c = %d\n", c);
    printf("global_var = %d\n", global_var);
    c = func_3(a, b, c);
    printf("\nOn return from func_3 ()\n");
    printf("a = %d\n", a);
    printf("b = %d\n", b);
    printf("c = %d\n", c);
    printf("global_var = %d\n", global_var);
    return (0);
}
```
Programming in C

int func_1(int L, int M, int N)
{
    extern int a;
    printf("ln func_1 ()\n");
    printf("L gets main ()'s value of global_var: %d\n", L);
    global_var++;
    --a;
    M += a;
    printf("global_var, a, M in func_1 (): %d, %d, %d
", global_var, a, M);
    printf("Returning...\n");
    return (L * global_var + M * global_var - N * a);
}

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

int func_3(int X, int Y, int Z)
{
    int global_var = 2;
    printf("ln func_3 ()\n");
    printf("X gets main ()'s value of a: %d\n", X);
    Z -= 2 - --a;
    Y += a - Z;
    printf("global_var, a in func_3 (): %d, %d\n", global_var, a);
    printf("Returning...\n");
    return (X + Y + Z);
    return (0);
}

Here is the output:

main () changes extern a to 7

In func_1 ()
L gets main ()'s value of global_var: 5
global_var, a, M in func_1 (): 6, 6, 13
Returning...
On return from func_1 ()
a = 6
b = 5
c = 78

In func_2 ()
P, Q, R gets main ()'s a, b, c: 6, 5, 78
Returning...
On return from func_2 ()
a = 1
b = 5
c = 363
global_var = 5

In func_3()...
X gets main()’s value of a: 5
global_var, a in func_3(): 2, 1

Returning...
On return from func_3()
a = 1
b = 5
c = 7
global_var = 5

Here are some exercises for you.

E4) Predict the outputs of Program 8.4 and 8.6. Verify your results by creating, compiling and executing these programs.

E5) What does the following program print?

/* Program 8.8; File name:unit8-prog8.c */
#include <stdio.h>
int alpha(int a1, int a2, int a3);
int beta(int b1, int b2, int b3);
int x = 2, y, z;
int main()
{
    auto int y = 4, z = 6;
    printf("%d\n", (x = alpha(x, y, z)));
    printf("%d\n", (x = beta(x, y, z)));
}
printf("%d\n", (x = alpha(x, y, z)));
printf("%d\n", (x = beta(x, y, z)));
return (0);
}
int alpha(int A1, int A2, int A3)
{
    extern int w;
    return (y = w = A1 + A2 + A3);
}
int w = 5;
int beta(int B1, int B2, int B3)
{
    x = 7;
    return (x = x + y + w + B1 + B2 + B3);
}

E6) What does the following program, spread over three files, print? The contents of file_1 are:

#include <stdio.h>
int x = 5;
void func_1(void);
We saw that a function cannot modify the variables of the calling function and it can return at most one value. How to overcome these shortcomings? This is the topic of discussion in the next section.

8.4 POINTERS AS FUNCTION ARGUMENTS

One apparent drawback of functions is that they can return at most one value back to the calling program. We say apparent with some justification, because the reality is that...
Functions

A function may be made to send back as many different values as may be required (though not via the return statement, whose load carrying capacity is limited to but one expression). A C function has in fact all the power of a FORTRAN function and subroutine combined. One mechanism by which a function may circumvent the limitation imposed by the return statement lies in C’s use of global variables. This corresponds to Fortran’s COMMON statement. As we have seen in program 8.8, a function can assign the results of its computations to global variables available to main() to use: two_occurs and five_occurs. Another mechanism for sending back more than one value is provided by pointers.

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

```c
/* Program 8.10; File name: unit8-prog10.c */
#include <stdio.h>

void cant_change(int a, int b, int c);
void can_change(int *l, int *m, int *n);

int main()
{
    int x = 4, y = 5, z = 6;
    printf("In main (): x = %d, y = %d, z = %d.\n", x, y, z);
    cant_change(x, y, z);
    printf("Back in main (): x = %d, y = %d, z = %d.\n", x, y, z);
    printf("cant_change () couldn't change x, y and z.\n");
    can_change(&x, &y, &z);
    printf("Back in main (): x = %d, y = %d, z = %d.\n", x, y, z);
    printf("can_change () could change x, y and z.\n");
    return (0);
}

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

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

/* Program 8.10: Output */
int main (): x = 4, y = 5, z = 6.
Entered cant_change ()...
```
The function can_change() does a subroutine’s work: it sends back three different values into three different variables in main(). This program should solve one mystery that may have plagued you; why scanf() uses pointers and printf() doesn’t. The reason is that scanf() must write new values into existing memory locations. If it used identifiers instead of pointers, it wouldn’t have been able to alter the contents of those locations.

Program 8.11 in Listing 2 on the facing page determines the length, slope, and y-intercept of a line whose end-points are given. The function line()’s argument list includes the co-ordinates of the end-points, as well as pointers to the variables that must be computed. Note that the function has a return type of int; it returns a value to main(), the value of error_trap, which indicates the success or failure of the computation. The function performs its error checking before attempting to execute an operation that could be illegal, e.g. divide by zero: this would occur if the co-ordinates of the end-points were the same, or the line were parallel to the y-axis.

Here are some exercises for you.

E7) Write a C program that uses pointers and a function to compute both the area and the perimeter of a triangle whose sides are given.

E8) Write a C program that uses pointers and a single function to compute the co-ordinates of the ex-centre of a triangle, given the co-ordinates of its vertices’s. Your program should return the radius of the ex-circle.

E9) Repeat exercise 7 for the in-circle of the triangle. (In exercises 7, 8 and 9 be sure to check that you are not dealing with non-existent triangles!)

E10) Write a C program that uses pointers and a single function to compute the two roots of a quadratic equation, given its coefficients. Your function must be able to cope with the four cases: non-existent roots (a = 0); real and different roots; equal roots; and complex roots.

So far, we have passed ordinary variables as arguments to functions. How to pass an array as an argument to a function? This is the topic of discussion in the next section.

### 8.5 UNIDIMENSIONAL ARRAYS AS FUNCTION ARGUMENTS: STRING FUNCTIONS

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

/* Program 8.11; File name:unit8-prog11.c */
#include <stdio.h>
#include <math.h>
int main(void)
{
    float x1, y1, x2, y2, length, slope, y_intercept;
    int error_trap;
    printf("Enter co-ordinates in the form x, y for\n");
    printf("point where line begins: ");
    scanf("%f, %f", &x1, &y1);
    printf("Co-ordinates at end of line: ");
    scanf("%f, %f", &x2, &y2);
    error_trap = line(x1, y1, x2, y2, &length, &slope,
                     &y_intercept);
    if (error_trap == -2)
        printf("No line possible between (%f, %f) and (%f, %f)\n", x1, y1, x2, y2);
    else if (error_trap == -1)
        printf("Line has infinite slope. Length is: %f\n", length);
    else
        printf("Length = %f, Slope = %f, Y_intercept = %f:\n", length, slope, y_intercept);
}
int line(float x1, float y1, float x2, float y2,
         float *length, float *slope, float *y_intercept)
{
    if ((x1 == x2) && (y1 == y2))
        return -2;
    else if (x1 == x2) {
        *length = y2 > y1 ? (y2 - y1) : (y1 - y2);
        return -1;
    }
    else{
        *slope = (y2 - y1) / (x2 - x1);
        *length = sqrt((x2 - x1) * (x2 - x1)
                        + (y2 - y1) * (y2 - y1));
        *y_intercept = y1 - *slope * x1;
        return 0;
    }
}

Listing 2: Functions to find the slope a line joining 2 points and the distance between the points.

/* Program 8.12; File name:unit8-prog12.c */
#include <stdio.h>
void uppercase(char *pointer);

int main(void)
{
    char string_holder[100];
    puts("Type a few good words, and press return.");
    gets(string_holder);
    uppercase(string_holder);
    puts(string_holder);
    return (0);
}
void uppercase(char *ptr)

81
Listing 3: A function to change all the characters in a string to upper case.

C permits the following alternative declaration inside the function header:

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

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

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

```c
int howlong (char string[])
{
    int i = 0;
    while (*string++)
        ++ i;
    return i;
}
```

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

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

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

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

```c
void copyover (char * strin_1, char * string_2)
{
    for (; *string_1 ++ = *string_2 ++; )
}
```

These functions are available in the standard library where they are known by other names: `howlong()` is called `strlen()`, `joinem()` is called `strcat()` (from string concatenation), and `copyover()` is called `strcpy()`. ANSI C compliant compilers come with a header `<string.h>` which contains declarations for these and several other functions for string manipulation. Older compilers do not have `<string.h>`; the
Table 1: Some `<string.h>` functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char *strcpy(s1, s2)</td>
<td>copies s2 to s1, returns s1</td>
</tr>
<tr>
<td>char *strncpy(s1, s2, n)</td>
<td>copies at most n characters of s2 to s1, returns s1</td>
</tr>
<tr>
<td>char *strcat(s1, s2)</td>
<td>appends s2 to s1, returns s1</td>
</tr>
<tr>
<td>char *strncat(s1, s2, n)</td>
<td>appends at most n characters of s2 to s1, returns s1</td>
</tr>
<tr>
<td>int strcmp(s1, s2)</td>
<td>Compares s1 to s2, returns a negative, zero or positive value depending on s1 being less than, equal to or greater than s2</td>
</tr>
<tr>
<td>int strncmp(s1, s2, n)</td>
<td>compares at most n characters of s1 to s2, returns a negative, zero or positive value depending on s1 being less than, equal to or greater than s2</td>
</tr>
<tr>
<td>char *strchr(s1, c)</td>
<td>returns pointer to first occurrence of char c in s1, or NULL if c is not in s1</td>
</tr>
<tr>
<td>char *strrchr(s1, c)</td>
<td>returns pointer to last occurrence of char c in s1, or NULL if c is not in s1</td>
</tr>
<tr>
<td>char *strbrk(s1, s2)</td>
<td>returns pointer to first occurrence in string s1 of any character in string s2, NULL if none is present</td>
</tr>
<tr>
<td>char *strstr(s1, s2)</td>
<td>returns to first occurrence of string s2 in s1, NULL if not present</td>
</tr>
<tr>
<td>size_t strlen(s)</td>
<td>returns length of s as an unsigned int (size_t is the type of result produced by the sizeof operator, and unsigned int.)</td>
</tr>
</tbody>
</table>

Functions are directly available. Some of the functions defined in `<string.h>` are listed in Table. 1 (your compiler may provide several others): Though string arrays are by far the most commonly used arrays in C programs, it is frequently required to process lists of numbers such as student’s marks, a savings account’s transactions, etc. Program 8.13 illustrates how an array of `ints` is passed to a function; the function returns the maximum and minimum elements of the array and the mean of all its elements. The interesting thing about Program 8.13 is that it can work for arrays of any size; on doesn’t have to keep track of the number of elements. The `sizeof` operator applied to an array yields the number of bytes in it. Dividing this number of `sizeof (int)` gives the number of elements in the array:

```c
num_el = sizeof array / sizeof (int);
```

Program 8.13 illustrates how an array of `ints` is passed to a function; the function returns the maximum and minimum elements of the array and the mean of all its elements. The interesting thing about Program 8.13 is that it can work for arrays of any size; on doesn’t have to keep track of the number of elements. The `sizeof` operator applied to an array yields the number of bytes in it. Dividing this number of `sizeof (int)` gives the number of elements in the array:

```c
num_el = sizeof array / sizeof (int);
```

```c
/* Program 8.13; File name:unit8-prog13.c */
#include <stdio.h>
float minmax(int arrayofints [], int num_el, int *max_el, int *min_el);
int main()
{
  int *min, *max;
  float avg;

  static int array[24] = {
-1, -2, -3, -4, -5, -6,
  7,  8,  9, 10, 11, 12,
-13, -14, -15, -16, -17, -18,
 19, 20, 21, 22, 23, 24,
};

  avg = minmax (array, sizeof array / sizeof (int), min, max);
  printf("Max is %d, Min is %d, Avg of elements is %.2f\n",
      *max, *min, avg);
  return (0);
}
```

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

Here are some exercises for you.

E11) Write a function that returns the day which fell on a given date passed to it. Make
sure that your function returns a message “Invalid date”, if an illegal date is
passed to it.

E12) Write a function that accepts two one-dimensional arrays A [] and B [] and returns
their scalar product:

E13) Write a function that accepts two one-dimensional arrays A[] and B[] and returns
their sum in a third array C[].

E14) Write a function that accepts a string and determines the number of times a given
character appears in it. For example, if the string is “Malayalam”, and the
character is ‘a’, your program should print the value 4.

E15) Modify Program 8.13 so that it returns to main() the subscripts of the maximum
and minimum elements in it.

E16) A set of marks in the range 0 - 100 is given. Write a C function to determine how
many of the scores fell in the ranges 0 - 10, 11 - 20, 21 - 30, ..., 91 - 100.

E17) You are given a list of numbers, some of which may occur more than once in the
list. Write a C function to remove duplicate numbers from the list.

E18) Write a C functions to sort an array of numbers using the interchange sort. Your
function should be able to handle any “reasonably sized” array of ints.

Suppose we want to solve a set of simultaneous equations. We will use a matrix to
represent the co-efficients of the equations. How can we pass a matrix to a function that
computes the inverse of a matrix, for example? We will discuss this in the next section.

8.6 MULTI-DIMENSIONAL ARRAYS AS FUNCTION
ARGUMENTS

If a two-dimensional array, say int numbers [4] [6] is to be passed to a function
addrows(), the parameter declaration must specify rather more than merely:

int numbers []; /* or, */
int numbers [][];

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

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

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

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

/* Program 8.14 */
#include <stdio.h>
void addrows(int mat[][6], int s[]);
int main()
{
    static int sums[4];
    static int matrix[4][6] = {
        {1, 2, 3, 4, 5, 6},
        {7, 8, 9, 10, 11, 12},
        {13, 14, 15, 16, 17, 18},
        {19, 20, 21, 22, 23, 24}
    };
    addrows(matrix, sums);
    printf("Row sums are: %d, %d, %d and %d\n",
           sums[0], sums[1], sums[2], sums[3]);
    return (0);
}
void addrows(int mat[][6], int *s1)
{
    int i, j;
    for (i = 0; i < 4; i++)
        for (j = 0; j < 6; j++)
            s1[i] = s1[i] + mat[i][j];
}

E19) Write a C functions to interchange any two rows of a two-dimensional array passed to it.

E20) Write a function to determine if a two-dimensional array passed to it is symmetric, that is, if

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

for any valid i and j.

E21) Write a C function to transpose a matrix, that is, your function should make its rows into columns, and vice-versa.
E22) Write a C function to multiply two conformable matrices. Use your function to find the product $P = M \times M \times M$ for the matrix $M_{3\times3}$ given by

\[
\begin{bmatrix}
4 & 9 & 2 \\
3 & 5 & 7 \\
8 & 1 & 6
\end{bmatrix}
\]

E23) Write a C function to determine if a given matrix input to it is a magic square, that is, if all its rows, columns and diagonals sum to the same number. Use your function to prove that both $M$ and $P$ in exercise 22 above are magic squares.

We conclude this unit here. You may use the summary given in the next section to recapitulate what we have discussed regarding functions.

### 8.7 SUMMARY

In this Unit we have discussed the following:

1. We have discussed how the functions are helpful in breaking up large programs into small, manageable tasks and encourage modular programming.
2. We have discussed the concept of prototype of a function and how to declare functions and how to use them.
3. We discussed the scope of variables declared in functions and the rules governing the use of variables declared externally to a function.
4. We have discussed how to call functions by reference using pointers.
5. We have discussed how to pass arrays as arguments to functions.

### 8.8 SOLUTIONS/ANSWERS

E1) The program is given in Listing 4.

```c
#include <stdio.h>

int isprime(int number);

int main()
{
    int number, factor;
    printf("Enter a Number, I'll check if it is a prime.\n");
    scanf("%d", &number);
    if ((factor = isprime(number)) > 1) {
        printf("The number is composite.\n");
        printf("%d is the smallest prime factor.",factor);
    } else
        printf("The number is prime.\n");
    return (0);
}

int isprime(int number)
/* This program returns the 1 if a number is prime
and the smallest prime factor if it is not a prime. */
{
    int divisor;
    /* We first dispose of 2 and 3. */
    if (number == 2 || number == 3)
        return (1);
```
Next we dispose even numbers.*/

```c
if (number % 2 == 0) {
    divisor = 2;
    return (divisor);
}
for (divisor = 3;; divisor += 2)
    if (number % divisor == 0) {
        return (divisor);
    } else if (divisor * divisor > number)
        return (1);
}
```

Listing 4: Solution to exercise 1.

Note that we have used the variable `factor` to save the prime factor returned by `isprime()`. The assignment statement `factor = isprime(number)` has the value returned by `isprime()`. We use this to check whether the value returned by `isprime()` is one or greater than one. If it is greater than one, the number is composite and the value saved in `factor` is its smallest prime factor.

E2) See Listing 5.

```c
#include <stdio.h>
int issquare(int number);
int main ()
{
    int number, sq_root;
    printf("Enter a number. I will tell you \n");
    printf("if it is a perfect square.\n");
    scanf("%d", &number);
    if ((sq_root=issquare(number))>0){
        printf ("The number is a perfect square.\n");
        printf ("The square root is %d.,sq_root);
    } else
        printf ("The number is not a perfect square.");
    return (0);
}
```

Listing 5: Solution to exercise 2.

E3) See Listing 6 for the program.

```c
/*Answer to exercise 3. File name:unit8-ans-ex-3.c*/
#include <stdio.h>
int isprime(int n);
int main()
{
    int number=0, i = 3, noof_primes = 0;
    printf("Enter a positive integer greater than 3.I will \
give\n");
    printf("the number of primes less than that number.\n");
    scanf("%d", &number);
    while (i < number) {
        if (isprime(i) == 1)
            noof_primes++;
        i++;
    }
```
Programming in C

```c
int isprime(int number)
/* This function returns the value 1 if a number is prime
and the smallest prime factor if it is not a prime. */
{
    int divisor;
    /* We first dispose of 2 and 3. */
    if (number == 2 || number == 3)
        return (1);
    /* Next we dispose even numbers. */
    if (number % 2 == 0) {
        divisor = 2;
        return (divisor);
    }
    for (divisor = 3;; divisor += 2)
        if (number % divisor == 0) {
            return (divisor);
        } else if (divisor * divisor > number)
            return (1);
}
Listing 6: Solution to exercise 3.

E7) The solution is in Listing 7.
/* Answer to exercise 7; File name: unit8-ans-ex-7.c */
#include <stdio.h>
#include <math.h>
void begin(void);
/* function prototypes */
int form_triangle(double a, double b, double c);
double area_perimeter(double x, double y, double z, double *perimeter);
int main(void)
{
    double side_1, side_2, side_3, perimeter;
    begin();
    scanf("%lf,%lf,%lf", &side_1, &side_2, &side_3);
    if (form_triangle(side_1, side_2, side_3)){
        printf("Area of triangle is: %f.\n",
            area_perimeter(side_1, side_2, side_3, &perimeter));
        printf("The perimeter is %f.\n", perimeter);
    } else
        printf("The lengths that you entered cannot form a triangle.\n"),
            return (0);
}
void begin(void)
{
    printf("This program determines whether three \n");
    printf("numbers that you enter can form the three \n");
    printf("sides of a triangle. If they can, the \n");
    printf("program prints its area and perimeter...\n");
    printf("Enter values for the three sides:\n");
}
int form_triangle(double alpha, double beta, double gamma)
{
    if ((alpha + beta > gamma) &&
        (beta + gamma > alpha) &&
        (gamma + alpha > beta))
```
```c
Functions

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

Listing 7: Solution to exercise 7.

E12) See Listing 8.

```c
#include <stdio.h>
#define MAX_SIZE 20
float dotproduct(float A[], float B[], int size);
int main()
{
    static float A[MAX_SIZE], B[MAX_SIZE];
    int i, size;
    printf("Enter the length of the vector.\n");
    scanf("%d", &size);
    for (i=0; i < size; i++){
        printf("Enter the values of A[%d] and B[%d] separated by commas.\n", i, i);
        scanf("%f,%f", &A[i], &B[i]);
    }
    printf("Their dot product is %f.", dotproduct(A, B, size));
    return (0);
}
float dotproduct(float A[], float B[], int size)
{
    float ans = 0;
    int i;
    for (i=0; i < size; i++)
        ans = ans + A[i] * B[i];
    return (ans);
}

Listing 8: Solution to exercise 12.

E14) See Listing 9.

```c
#include <stdio.h>
int getcount(char *input, char c);
int main()
{
    char *string="Malayalam";
    printf("l occurs %d times.", getcount(string, 'l'));
    return (0);
}
int getcount(char *input, char c)
{
    int i=0;
    while (*input){
        if (*input == c){
            i++;
        }
        input++;
    }
```
Programming in C

Listing 9: Solution to exercise 14.

You can also use `strchr()` function given in Table 1 on page 83 to write such a function. Try it!

E19) See Listing 10.

```c
/* Solution to exercise 19. */
#include <stdio.h>
int interchange_rows(float array[][3], int i, int j);
void printarray(float array[][3]);
int main()
{
  int i=1, j=3;
  float array[3][3];
  for (i = 0; i < 3; i++)
    for (j = 0; j < 3; j++)
      array[i][j] = (i+j)/0.5;
  printarray(array);
  interchange_rows(array, 0, 2);
  printf("After interchange the array is:\n");
  printarray(array);
  return (0);
}
void printarray(float array[][3])
{
  int i, j;
  for (i = 0; i < 3; i++)
    for (j = 0; j < 3; j++)
      printf("%.2f, ", array[i][j]);
  printf("\n");
}
int interchange_rows(float array[][3], int i, int j)
{
  int a;
  float temp;
  for (a = 0; a < 3; a++)
    temp = array[i][a];
  array[i][a] = array[j][a];
  array[j][a] = temp;
  return (0);
}
```

Listing 10: Solution to exercise 19.

E20) See Listing 11.

```c
#include <stdio.h>
int issymmetric(float a[][3]);
int main()
{
  int i = 1, j = 3;
  float array[3][3];
  for (i = 0; i < 3; i++)
    for (j = 0; j < 3; j++)
      array[i][j] = i < j ? (i+j)/0.5 : 0;
  if (issymmetric(array) == -1)
    printf("The array is not symmetric.\n");
  else
    printf("The array is symmetric.\n");
```
functions

return (0);
}

int issymmetric(float a[][3])
{
    int i,j,result=1;
    for(i=1; i < 3; i++)
        for(j=0; j < i; j++)
            if(a[i][j] != a[j][i])
                result = -1;
    return(result);
}

Listing 11: Solution to exercise 20.
UNIT 9  FUNCTIONS -II

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 discuss 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);
    }
}
```
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_disks, char *A, char *B, char * C)
{
    if (num_disks == 1)
        printf("Move disc from %s to %s\n", A, C);
    else {
        hanoi(num_disks - 1, A, C, B);
        printf("Move disc from %s to %s\n", A, C);
        hanoi(num_disks - 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
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:

$$\text{hcf (hcf (a, b), c);}$$

The HCF of four numbers is given by the call:

$$\text{hcf (hcf (hcf (a, b), c), d);}$$

or equivalently and more simply by

$$\text{hcf (hcf (a, b),hcf(c, d));}$$

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:

```c
/* 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;
```
Programming in C

```c
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:

```c
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:

```c
#include <stdio.h>

char lifo(char c);

int main(void)
{
    char anykey;
    lifo(anykey = getchar());
    putchar(anykey);
    return (0);
}

char lifo(char C)
{
    if (C == '\n')
        return '\0';
    C = getchar();
    lifo(C);
    putchar(C);
    return (0);
}
```

Can you explain its output?

E3) The Fibonacci numbers are most easily evaluated iteratively. However, their definition:

```c
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:
    /* Program 9.4; File name: unit9-prog4.c */
    #include <stdio.h>
    int x, y;
    void func(void);
    void main(void)
    {
      int z = 1;
      printf("Entering main().\n");
      while (x++ <= 3) {
        printf("x = %d, y = %d, z = %d,\n", x, y, z);
        func();
        printf("a = %d, b = %d,\n", a, b);
        printf("x = %d, y = %d, z = %d,\n", x, y, z);
        printf("Finally...:\n");
        y--; z++; printf("x = %d, y = %d, z = %d,\n", x, y, z);
      }
      printf("Finally...:\n");
      y--; z++; printf("x = %d, y = %d, z = %d,\n", x, y, z);
    }

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

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.
Programming in C

Typically, a macro definition has the form:

```c
#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 `#define` 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:

```c
#define h 1
#define e 2
#define n 3
```

Then, no replacements will be made in the statements below:

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

The `#undef` preprocessor directive cancel a previous `#define` statement:

```c
#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:

```c
/* 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 `#defines` 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`:

```c
#define DO_THE_FOLLOWING BEGIN
#define BEGIN {
#define WRITE_THIS_STRING printf("%s\n",
#define WRITE_THIS_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.

`#include` 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`ed in a program; however, if in a `#include`ed file _A_ there is a reference to quantities defined in another `#include`ed file, say file _B_, the the inclusion of file _B_ must precede that of file _A_. More, a `#include`ed 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 conditional 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 `#defined`:

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

These statements assume that there was a previous `#define`ion, 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:

```c
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:

```c
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:

```c
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:

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

but it wouldn’t work in every situation: if you had an expression:
\[ q = \frac{600.0}{\text{VOL}_\text{CUBOID}(3, 4, 5)}; \]

it would translate to:

\[ \frac{600.0}{(3) \times (4) \times (5)}; \]

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 \texttt{define} \texttt{VOL}_\text{CUBOID} would be:

\texttt{\texttt{define} VOL}_\text{CUBOID}(A, B, C) ((A) \times (B) \times (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:

\texttt{\texttt{define} VOL}_\text{CUBE}(X) ((X) \times (X) \times (X))

\texttt{int side, volume;}

\texttt{side = 17;}

\texttt{volume = VOL}_\text{CUBE}(\text{side});

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

\texttt{volume = VOL}_\text{CUBE}(++ \text{side});

the macro processor would in all innocence translate it:

\texttt{volume = ((++ side) \times (++ side) \times (++ side));}

\texttt{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 \texttt{int}s \texttt{X} and \texttt{Y}. Under what conditions may your macro fail?

E8) Write macros:

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

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

1) \texttt{VOL}_\text{SPHERE}(\texttt{R}) to compute the volume of a sphere
2) \texttt{AREA}_\text{SPHERE}(\texttt{R}) to compute the area of a sphere
3) \texttt{VOL}_\text{CYL}(\texttt{R}, \texttt{H}) to compute the volume of a cylinder
4) \texttt{AREA}_\text{CYL}(\texttt{R}, \texttt{H}) to compute the area of a cylinder
5) \texttt{VOL}_\text{CONE}(\texttt{R}, \texttt{H}) to compute the volume of a cone
6) \texttt{AREA}_\text{CONE}(\texttt{R}, \texttt{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:

```c
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:

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

Because `argv[]` is an array of strings, it could equivalently be declared:

```c
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'.

```c
//...
Its ordinal value will be ('d' - '0'). If the next digit encountered is 'e', the number built so far will have the value $10 \times ('d' - '0') + ('e' - '0$). 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:

```c
/* 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.

```c
/* 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.)
like `scanf()` and `printf()`, we knew beforehand 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()`:

```c
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`:

```c
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()`.

```c
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()`.

```c
int addem (int * how_many, ...);
```

106
{  
  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
Programming in C

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.

```c
/* 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
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

```c
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

```c
putchar ((*func_2) (first_string));
```

is equivalent to

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

According to ANSI standard, the usage

```c
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.

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

You will find that it is equivalent to

```c
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:

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

and

```c
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 `ints`. Similarly, the declaration:

```c
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 ints.

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

```c
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:

\[
2, 3, 5, 7, 11, 13, \ldots \text{(up to or including } N)\]

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:

```c
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 discoverers) 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 
\( n \times \text{sizeof prime} \).

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, \text{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 \times \text{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:

```c
primes_ptr = (\text{int *} \text{calloc (num_primes, \text{sizeof (int)})});
```

`(\text{int *})` casts the value returned by `calloc()` to a pointer to `\text{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 function uses the trapezoidal rule to integrate the 
expressions \( \frac{1}{\ln (x)} \) 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 \( b \), then the width of any strip is \( h = (b - a) / n \). The \( i \)th strip 
has x-values \( a + (i - 1) \times h \) and \( a + i \times h \), and thus its area is 
\( 0.5 \times h \times (f(a + (i - 1) \times h) + f(a + i \times h)) \). 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 \times \text{sizeof int}`), 
and returns a pointer to it if it is available, else the NULL pointer.

```c
if ((primes_ptr - (\text{int *} \text{calloc (num_primes, \text{sizeof (int)})})) == \text{NULL})
```
```c
/* 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;
```
Functions-II

```c
for (primes_ptr = top_ptr; primes_ptr <= bottom_ptr; primes_ptr++)
    printf("%d\n", *primes_ptr);
return (0);
}
```

```c
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:

```c
*(++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

```c
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().

```c
/* 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));
```
Programming in C

```c
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 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 `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.

```c
#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);
```

114
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 returns 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.

```c
#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.

```c
/*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.

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 twelfth 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 the 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)
Programming in C

```c
{ 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.

```c
#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.
10.1 INTRODUCTION

If you have used any computer application, whether it is a word processor or a spreadsheet or a presentation package, you would have saved your work in files in the hard disk. This is a very convenient feature because you have a permanent record of your work that you can change or update and you can also share your work with others. Also, sometimes very large programs use a large amount of data again and it is rather a waste of effort to generate it again and again. So, you need to know how to work with files, how to save data from files, read data and modify files. We will discuss this in the Sec. 10.2.

Let us now turn our attention to the organisation of information within files. Suppose that you want to store data about the passbook entries for a savings bank account. For each transaction you would want to store a variety of items of different types; to make our example more concrete, suppose that you’ve received a cheque, and you’d like to record the following details about it: by whom given, of what amount, on which bank, whether local or outstation, and date and number of cheque. How will this information be organised? It would be convenient if there was a single data receptacle that could hold all this information in one place. Clearly that receptacle couldn’t be an array. An array can only hold items of just one type, int or char or whatever; the information that you need to store happens to involve a variety of types: char arrays, int and a float. What you’re looking for is a data structure that in C parlance is called the struct.

A struct, similar to Pascal’s record, can hold several kinds of items, and enables you to maintain related information of different types in a single object. Structs are introduced in Sec. 10.3 of this Unit. In Sec. 10.4, Sec. 10.5 and Sec. 10.6, we discuss how to store and access data in structs.

A union is a C device that allows you to view a single storage area in more than one way: a variable declared as a union can store different kinds of data at different times. We discuss union in Sec. 10.7 of this unit.

C provides a variety of operators to manipulate the contents of a packed field of bits. In systems programs there is very often a need to set up a number of “flags”—values that must be 0 or 1: sensing these tells about the status of the system, and helps decide on a suitable course of action. Now, since a single bit is enough to represent a status, it makes little sense to use a byte or word for the purpose. In C it’s possible to “pack” bits into a “field”, where they may be set and tested. Bit operations are the final topic covered in this Unit.
Objectives

After studying this unit, you should be able to

• declare files and perform file I/O;

• declare, create and operate on instances of structs;

• declare and operate on unions; and

• use the bit operators on packed fields.

10.2 FILES AND FILE I/O

Files are the most visible and the most valuable feature of any computer system to its users because they provide a mechanism for the long term storage of data; they remain unaffected even if a computer’s power is switched off. Computers would be of no use at all if they did not provide a mechanism for the permanent storage of information.

Files are stored in the concentric circular tracks of a floppy or a “hard” disk. A hard disk, unlike a floppy, consists of several platters rigidly mounted on a common spindle; each surface of platter may have several hundred tracks. These tracks contain an even distribution of particles of a magnetic oxide, each of which can be permanently magnetised in one of two ways, to store a 0 or a 1. So the bi-stable circuits of a computer’s RAM find an analogue in the dual states of magnetisation of particles of iron oxide on the disk’s surface. As a disk spins in its “drive” (this is a motor driven platform on which the disk is fixed), a “read-write” head can be made to home in to any of its tracks. In a hard (or even a floppy) disk drive, there’s a head for every surface. These heads are joined to a common actuator arm; the arm (and so the heads) can move radially along the disk’s surface, towards or away from its centre. Any head can be positioned on any track on the surface it serves. When a file is read the contents of the track spinning past the head (the individual little magnets) induce corresponding electric signals inside a coil in the read-write head; these signals flow in the RAM, where they are stored as bits. Thus reading a file means picking up the particulate magnetisations on a disk’s tracks and storing them as electric signals of 0 or 1 in the computer’s RAM. Conversely, when a file is written the contents of RAM—electric signals of low or high voltage—are made to travel as currents to the coil in the read-write head; these currents magnetise the particles flowing by under the head, while the disk spins, creating in them an image precisely similar to the bits from which the signals emanated. Thus are the contents of RAM captured on disk, and files are “written”. Disk files are permanent to the extent that the magnetism of the particles in which they are stored is permanent.

The tracks on a disk’s surface are divided into a number of sectors of equal size. Just as each word of RAM has an address, so also has each sector of a disk. These addresses are known to the computer’s operating system. The operating system also knows which file is written on which sector(s)—these need not be contiguous—and it also keeps a list of the free sectors on the disk. When a file is to be read, the operating system moves the read-write head to the tracks which contains the sector on which the file begins; the time that this head movement takes is called the “seek time”. Having reached its designated track, the head waits there until the desired sector has appeared under it: this waiting period is called the “latency time”. Finally the data from disk can be transferred to the RAM; this third delay is called the “transfer time”. Large files may occupy several sectors, which need not be contiguous. The read-write head may have to be directed to sectors of different tracks on several surfaces to read large fragmented files. You can see that because mechanical motions are involved, a disk read or write must by CPU standards be a very slow operation; it is in fact hundreds of thousands of times slower than a CPU cycle. So it’s just as well that in a read operation an entire sector is
read at a time. The contents of a sector are placed in a block of RAM called the input buffer. Intelligent programmers organise their data in disk files so that when further data items are needed by their program, they may already be in the input buffer from the last read operation.

In a read operation, precisely how the read-write head is directed to a disk’s track from which it must pick the chosen finole from the thousands that may be resident on the disk, and how it performs the read while the magnetised particles rush past it (sometimes as fast as two hundred km. per hour!) is a matter of enormous and quite awe-inspiring complexity. Fortunately for us, as users of the computer we have nothing whatsoever to do with this complexity. How C finds where exactly our files are on a given disk, and how it finds free tracks therein to store the new files that we may create, is a matter that it must negotiate with the computer’s operating system. That’s not our concern. What is nice for us is that C provides us with a very convenient interface to deal with files. Its convenience lies in the fact that the functions for I/O that we have so far used—printf(), scanf(), puts(), gets(), putchar() and getchar() for monitor and keyboard I/O—have quite similar counterparts for disk I/O. There is a special reason for this: the functions for keyboard input and monitor output that we are familiar with, are themselves implemented via files. These functions depend upon three files that work in the background, generally invisible to us. The files, declared in <stdio.h>, are the **standard input**, the **standard output** and the **standard error**: we shall have more to say about them later. The good news is, if you know how to use scanf() and printf(), you know how to read from the disk and write to disk files! Designers of operating systems call this uniformity of interface “device independence”.

Before a file can be used for reading or writing, it must be **opened**. This is done by a call to the **fopen**() function. **fopen**() takes two string arguments, therefore enclosed in double quotes. The first of these is the file’s name; the second is an option (r, w, a, etc.) that tells C what we want to do with the file: read it, or write to it, or append to it, etc. File handling functions are prototyped in <stdio.h>, which also includes other needed declarations. Naturally this header must be **#include**d in all your programs that work with files. Table 1 lists the options available with **fopen**(). A call to **fopen**() to read the file passbook.dat looks like:

```c
fopen ("passbook.dat", "r")
```

If **fopen**() is successful in its mission of opening the file for reading, it returns a new type of pointer, a pointer of type FILE; FILE is a **typedef** defined in <stdio.h>. If
fopen() cannot successfully open the file, it returns instead the NULL pointer.

To connect your program to the file that fopen() opens, your program must also declare a pointer of type FILE: that declaration is easy:

```c
FILE * file_ptr;
```

Now you can assign to file_ptr the pointer that fopen() returned:

```c
file_ptr = fopen ("passbook.dat", "r");
```

In this way file_ptr provides you read access to the file passbook.dat. You need never again refer to the file itself by name: file_ptr will do all the required book-keeping for you, such as holding your place in the file, etc.

In typical concise fashion C programmers combine the file opening operation with a check to see if the pointer returned by fopen() was NULL:

```c
if ((file_ptr = fopen ("passbook.dat", "r")) != NULL)
    printf("Had no difficulty in opening passbook.dat\n");
else
    printf("Regrets...couldn't open passbook.dat\n");
```

Even more concisely, one writes:

```c
if (file_ptr = fopen ("passbook.dat", "r")) etc.
```

A call to fclose() closes a previously opened file, flushing as it does so any associated buffer; the single argument of fclose() is the relevant file pointer. Let’s now look at Programs 10.1 - 10.5.

### fprintf()

Program 10.1 illustrates how the three parts of the argument list of fprintf(), the function to write to a file opened by fopen(), are written: the first is the file pointer, the second is the control string, and the third is the list of variables or expressions to be written:

```c
fprintf(file_1, "%s", "Writing to a disk file is\n");
```

Upon successful return fprintf() returns the number of characters printed.

```c
/* Program 10.1; file name: unit10-prog1.c */
#include <stdio.h>
ing main(void)
{
    FILE *file_1;
    if ((file_1 = fopen("newfile.dat", "w")) != NULL) {
        fprintf(file_1, "%s", "Writing to a disk file is\n");
        fprintf(file_1, "%s", "quite easy, as no doubt you\n");
        fprintf(file_1, "%s", "will readily agree.\n");
        fclose(file_1);
    } else
        printf("Unable to open newfile.dat for writing");
    return (0);
}
```

Executing Program 10.1 creates the 77 byte file named newfile.dat. Here are its contents:

```
Writing to a disk file is
quite easy, as no doubt you
will readily agree.
```

Program 10.2 reads 5 int values from the keyboard, and stores them in the file numbers.dat:
/* Program 10.2; file name: unit10-prog2.c */
#include <stdio.h>
int main(void)
{
    FILE *ptr;
    int numbers[5];
    int i;
    if ((ptr = fopen("numbers.dat", "w")) != NULL) {
        printf("Enter 5 numbers, to be stored in numbers.dat...");
        for (i = 0; i < 5; i++) {
            scanf("%d", &numbers[i]);
            fprintf(ptr, "%d\t", numbers[i]);
        }
        fprintf(ptr, "\n");
        fclose(ptr);
    }
    else
        printf("Unable to open numbers.dat for writing...\n");
        return (0);
}

Here is the result of executing Program 10.2:

Enter 5 numbers, to be stored in numbers.dat...1 2 3 4 5

The file numbers.dat was created:
1 2 3 4 5

fscanf ()

Similarly fscanf() reads files. Its arguments are written as shown below:

fscanf(file_ptr, "control string", list of pointers);

Each successive pointer argument must correspond properly with each successive conversion specifier. Program 10.3 reads the file numbers.dat, and sums its entries.

/* Program 10.3; file name: unit10-prog3.c */
#include <stdio.h>
int main(void)
{
    FILE *ptr;
    int val, i, sum = 0;
    if ((ptr = fopen("numbers.dat", "r")) != NULL) {
        for (i = 0; i < 5; i++) {
            fscanf(ptr, "%d", &val);
            sum += val;
        }
        printf("The sum of the entries in numbers.dat is %d\n", sum);
        fclose(ptr);
    } else
        printf("Unable to open numbers.dat for writing...\n");
            return (0);
}

/* Program 10.3: Output: */

The sum of the entries in numbers.dat is 15

stdin, stdout and stderr

You might think that calls to fprintf() and fscanf() differ significantly from calls to printf() and scanf(); these latter functions didn’t seem to require file pointers.
As a matter of fact they do. The file pointer associated with printf() is a constant pointer named stdout, defined in <stdio.h>. Similarly scanf() has an associated constant pointer named stdin. scanf() reads from stdin and printf() writes to stdout, as you may verify by executing Program 10.4 below.

/* Program 10.4; file name unit10-prog4.c */
#include <stdio.h>
int main (void)
{
    int first, second;
    fprintf(stdout, "Enter two ints in this line: ");
    fscanf(stdin, "%d %d", &first, &second);
    fprintf(stdout, "Their sum is: %d.\n", first + second);
    return (0);
}

There is a third constant file pointer defined in <stdio.h> named stderr. This is associated with the standard error file. stderr has the following use: In some systems such as PC-DOS and Unix, you can redirect the output of your programs to files, by using the redirection operator, >. In DOS, for example, if f1.exe is an executable file that writes to the monitor, then you can redirect its output to a disk file f1.o by the command:

f1 > f1.o<CR>

Output that would normally appear on the monitor can thus be sent to a file. On the other hand, if you were redirecting output, you wouldn’t want any error messages such as:

"Unable to open newfile.dat for writing"

to be redirected; you’d want them to appear on the screen. Writing error messages to stderr:

fprintf(stderr, "Unable to open newfile.dat for writing");

does not ensure that normal output will be redirected, but error messages will still appear on the screen.

sprintf () and sscanf ()

printf() and fprintf() are therefore close cousins, as are scanf() and fscanf(); but there are two other useful members of this family, namely sprintf() and sscanf(). sprintf() writes its output in a string, whose address is its first argument; as before, its second argument is the format control string, and this is followed by a list of variables or expressions:

sprintf(output_buffer, control_string, list);

Likewise, sscanf() can read a string and store its contents into the pointers constituting list:

sscanf(input_buffer, control_string, list);

fgets ()

Recall from Unit 7 that scanf() is not the most convenient function to read strings with—it regards white space as a string delimiter—and fscanf() is not different in this regard. But the fgets() function, like the gets(), is easy to use if you wish to read a whole line from a file at a time. It is generally considered unsafe to use gets(). This is because there is no restriction on the length of the string read. Because of this, if a very long string is read in, the program can crash if there isn’t enough memory to store it. This doesn’t happen in fgets() because the size of the string is specifically mentioned. (Another alternative is to use scanf() with the format specifier %20s, for example, to limit the length of the string read in to 20.) fgets() is called with three
arguments: a pointer to a char array that will store the string read, an integer specifying the maximum number of characters to be read, say n, and a file pointer specifying the file to be read. Program 10.5 uses an array of 50 chars, named input_buffer [], in which fgets() deposits the contents of newfile.dat, a line at a time. fgets() reads characters until a newline occurs in the input, which is stored, or until it has read n - 1 characters. It appends the null character after the last character read. As a function, fgets() returns the value of its first argument (the pointer input_buffer if a read was successful, otherwise the NULL pointer. This property is exploited in the while() loop of Program 10.5.

/* Program 10.5; file name:unit10-prog5.c */
#include <stdio.h>
int main(void)
{
    FILE * file_1;
    char input_buffer[50];
    if ((file_1 = fopen("newfile.dat", "r"){" ! = NULL){
        while "fgets (input_buffer,
             sizeof (input_buffer), file_1)) ! = NULL) |
            printf(input_buffer);
        fclose (file_1);
    } else
        printf(" Sorry, unable to open newfile.dat \n ");
    return (0);
}

fgets ()

Similarly, fputs(), used in Program 10.6, writes a string in a file. Its two arguments are the memory address of an input string and the file pointer. The input string is written up to (but not including) its null character. Program 10.6 accepts the names of two files in its command line. It copies the first file into the second, line by line.

/* Program 10.6; file name:unit10-prog6.c */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    FILE * file_1, *file_2;
    char input_buffer[50];
    if ((file_1 = fopen(argv[1], "r")) == NULL)
            printf("Unable to open %s for reading
", argv[1]);
    else if ((file_2 = fopen(argv[2],"w")) == NULL)
        printf("Unable to open %s for writing\n",argv[2]);
    else{
        while "fgets(input_buffer,
             sizeof(input_buffer), file_1)) ! = NULL) |
            fputs(input_buffer, file_2);
        fclose(file_1);
        fclose(file_2);
    }
    return (0);
}

getc () and putc ()

getc() and putc() are two character I/O functions for files, very similar to getchar() and putchar() respectively. The single arguments of getc() is an input file pointer, it fetches the next character from the file, and returns EOF at end of file, or if an error occurs. putc() has two arguments: the character to be written, and a pointer to the output file. (Why has the variable next_char in Program 10.7 been declared as
/* Program 10.7; file name: unit10-prog7.c */
#include <stdio.h>
int main (void)
{
    FILE *file_1;
    int next_char;
    if (file_1 = fopen("newfile.dat", "r")) {
        while ((next_char = getc(file_1)) != EOF)
            putc(next_char, stdout);
        fclose(file_1);
    } else 
        printf("Sorry, unable to open newfile.dat\n");
    return (0);
}

Realise from Program 10.7 that putchar(x) is equivalent to putc (x, stdout),
and getchar() is equivalent to getc (stdin).

fread (), fwrite (), rewind () and feof()

Program 10.8 uses two ANSI C functions for direct I/O. The fread() and fwrite() functions do not interpret (i.e. format) what they read or write. You will therefore not in
general be able to display, i.e. type or cat to the screen a file created by fwrite():
instead, fread() must be used to read it, as Program 10.8 illustrates. The fread() and fwrite() functions have four arguments. The first of these is the address of a
memory buffer from which fwrite() writes into a file, (or into which fread() reads
from a file). The second argument of each function is the size of the items to be
written or read; the third, the number of items; and the fourth, the file pointer. The call:
fwrite (numbers, sizeof (double), NUM_EL, f1);
writes NUM_EL doubles from a buffer numbers into a file associated with the pointer
f1.

The call:
fread (copy, sizeof (double), NUM_EL f1);
reads NUM_EL doubles from a file associated with the pointer f1 into a buffer named
copy.

Program 10.8 opens (creates) a file double.dat for both writing and reading: observe
the ‘‘w+’’ option in the call to fopen(). numbers is the name (i.e. address) of a
buffer of dimension NUM_EL. The doubles stored in numbers are written into
doubles.dat; but this file cannot be displayed on the screen because the objects written
in it are in the format in which they were stored in memory. To look at the contents of
the file, you’ll have to access it via an fread(). However, the file pointer after the last
write is stationed at the bottom of the file; before you can begin to read it you must
bring its pointer back to the top. The call:
rewind (f1);
brings the read pointer back to the beginning of the file.

The call to fread() in Program 10.8 reads the file into the buffer copy. The for (;;) loop prints the contents of copy [].
FILE * f1;
static double numbers[NUM_EL] =
{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
};
static double copy[NUM_EL];
int i;
if ((f1 = fopen("doubles.dat", "w+")) != NULL)
    fwrite(numbers, sizeof(double), NUM_EL, f1);
else
    printf("Error writing in file doubles.dat...\n");
rewind(f1);
fwrite(copy, sizeof(double), NUM_EL, f1);
for (i = 0; copy[i] > 0; i++)
    printf("%.0f\n", copy[i]);
fclose(f1);
return (0);

The end-of-file condition can be tested by a call to feof(); this function, whose single
argument is a file pointer, returns a non-zero value if an attempt is made to read beyond
the end of a file:
if (feof(fp))
    printf("Error: cannot read beyond EOF");

Here are some exercises for you.

E1) Write a C program to count the number of characters, words, and in lines in a text
file whose name is supplied in the command line.

You may have wondered that we have not written any program involving complex
numbers. We will discuss struct in the next section, which will enable us to work with
complex numbers.

10.3 STRUCTS

Arrays, as we have seen in the previous units, provide a means to bundle homogeneous
data items into a single named unit. But, more often than not, “real world” applications
require the grouping together of heterogeneous data items. Consider once more the
information that one would want to store about depositing a cheque in one’s savings
account:
i) by WHO issued;
ii) on what bank;
iii) whether cheque was local or outstation;
iv) cheque number
v) date of issue
vi) date of deposit
vii) amount of cheque

Of the items in our list, the first four are strings; the fifth and sixth, the dates of issue
and deposit, consist of triplets of ints; and the seventh is a float. If we considered using
arrays to store the records of several similar transactions, then we would need eleven in
all: four two-dimensional arrays for the strings; three int arrays each for the two dates;
and one of floats to hold the amounts of the cheques. Every transaction would require
us to access these eleven arrays; nor could we know the appropriate number of elements
to specify for them: altogether an approach to be avoided.

**Structs** are a C facility that permit several heterogeneous but interrelated items to be
treated as a single unit. The keyword **struct** introduces such a unit:

```c
struct cheque_deposited
{
    char by_whom_given [30];
    char bank [30];
    char local_or_not [2];
    char cheque_number [10];
    int dd_issued;
    int mm_issued;
    int yy_issued;
    int dd_deposited;
    int mm_deposited;
    int yy_deposited;
    float amount;
};
```

This **struct** declaration names eleven **members**; (arrays have elements; structs have
members). But it reserves no storage. It creates a template that can be used to define
objects of type **struct** `cheque_deposited`:

```c
struct cheque_deposited nobel_prize, copley_medal, fields_medal;
```

Here `nobel_prize`, `copley_medal` and `fields_medal` are structure identifiers. Each of these structures can store the details of a cheque, just as much as an `int` can store an integer value. The scientist who wins a Nobel Prize, a Copley Medal and a Fields Medal can store the details of the cheques he receives in these structures. The name `cheque_deposited` is a structure tag, and is optional: **structs** may be defined without a tag, as for example the following **structs** to store the dates of our national holidays:

```c
struct
{
    int dd;
    int mm;
    int yy;
} gandhi_jayanti, independence_day, republic_day;
```

Here `gandhi_jayanti`, `independence_day`, `republic_day` are three structures that can store dates. Because the **struct** declaration is not tagged, a type has not been created to store any other dates.

The advantage of a tag is that further instances of structures can subsequently be created by simple definitions using the tag as a type. Consider a **struct** `date` declared externally to `main()`:

```c
struct date
{
    int dd;
    int mm;
    int yy;
};
```

Then other structures of type `date` may be defined in the functions that follow:

```c
struct date gandhi_jayanti, independence_day, republic_day;
struct date Abhishek_birtday, Aparajita_birthday;
```

Arrays of structures may also be defined:

```c
struct cheque_deposited bunch [1000];
```

bunch is an array of 1000 structs of type `cheque_deposited`.!
A structure tag may therefore be thought of as a user defined type, like `float` and `double`. The `sizeof` operator can be used to determine the size of a `struct` type, or of its instances:

```c
printf("%d\n", sizeof (struct date));
printf("%d\n", sizeof nobel_prize);
```

Remember, in this last usage the `sizeof` operator will not return the value of the Nobel Award in dollars—only the size of the structure `noble_prize` in bytes! Here are some exercise to check your understanding of declaring and using `struct`s.

**E2)** Show how you will declare a struct to hold employee payroll data: name, department, date of birth, date of joining, basic salary, dearness, house-rent and other allowances, provident fund, life insurance and income tax deduction, and contributions towards CGHS, ESI, etc.

**E3)** Show how you will declare a struct to hold the fee-bills of students in school. The structures must be able to record the following information: student name, date of birth, class and section, father’s name, address and telephone number, whether student uses the school bus, distance she travels on the bus (this information is used to compute the bus bill), tuition fee, and fees chargeable for extracurricular activities, such as swimming, skating, etc.

**E4)** Show how you will declare a struct to record information about students enrolled for a degree program at a University. The University offers, B.A., B.Sc., M.A., M.Sc. and Ph.D. degrees in several subjects. Each student is identified by a unique enrollment number.

We have discussed in this section how to store heterogeneous data using `struct`. In the next section, we will see how to access the data stored in `struct`s.

## 10.4 THE DOT OPERATOR

In order to gain access to a member of a structure, say `float amount` in the cheque for the Nobel Prize, we use the dot operator, `"."`

```c
structure_name.member_name
```

as in the `printf()` below:

```c
printf("Congratulations on your award! Amount deposited = %.2f",
nobel_prize.amount);
```

Though the dot operator may appear to you as a binary operator in the above expression, it has in fact a priority as high as the parentheses or the subscript operators, and like them it groups from left to right.

The members of a structure can be initialised to constant values by enclosing the values to be assigned in braces after the structure’s definition (non ANSI C compilers may require the `static` keyword in the lead):

```c
/* static */ struct date Gandhiji_birthday = {
   2, 10, 1869
},
Nehruji_Birthday = {
   14, 11, 1889
};
/* static */ struct cheque_deposited delhi_lottery = {
   "Delhi Administration",
   "State Bank of India",
```

129
If there are fewer values listed in an initialisation than the number of members present, then the remaining members are initialised to zero by default.

To print the approximate difference in the ages of Gandhiji and Nehruji we can write:

```
printf("Age difference = %d\n", Nehruji_birthday.yy - Gandhiji_birthday.yy);
```

Program 10.9 illustrates how the dot operator is used to extract the members of a structure:

```
/* Program 10.9; file name: unit10-prog9.c*/
#include <stdio.h>

struct cheque_deposited
{
    char by_whom_given [30];
    char bank [30];
    char local_or_not [2];
    char cheque_number [10];
    int dd_issued;
    int mm_issued;
    int yy_issued;
    int dd_deposited;
    int mm_deposited;
    int yy_deposited;
    float amount;
};

int main (void)
{
    /* static */
    struct cheque_deposited delhi_lottery =
    {"Delhi Administration",
     "State Bank of India",
     "Y",
     "381654729",
     2, 5, 1994, 3, 5, 1994,
     50000000.00};
    printf ("Cheque date: %d-%d-%d\n",
            delhi_lottery.dd_issued,
            delhi_lottery.mm_issued,
            delhi_lottery.yy_issued);
    printf("Cheque number: %s\n", delhi_lottery.cheque_number);
    printf("Drawn on: %s\n", delhi_lottery.bank);
    printf("Amount: Rs. %.2f", delhi_lottery.amount);
    return (0);
}
```

/* Program 10.9: Output: */

Cheque date: 2-5-1994
Cheque number: 381654729
Drawn on: State Bank of India
Amount Rs. 50000000.00

Finally, structures may be nested. Given the prior declaration of `struct date`,

```
struct cheque_deposited may be declared:
```
struct cheque_deposited
{
    char by_whom_given [30];
    char bank [30];
    char local_or_not [2];
    char cheque_number [10];
    struct date issued;
    struct date deposited;
    float amount;
};

Program 10.10 - 10.11 illustrate struct declarations and the use of the dot operator.

Program 10.10, which declares struct complex, implements the addition of complex numbers. Its members store the real and imaginary parts of three complex numbers, val_1, val_2 and their complex sum, name result.

/* Program 10.10 ; file name: unit10-prog10.c*/
#include <stdio.h>
struct complex
{
    float real;
    float imaginary;
};
int main ()
{
    struct complex val_1, val_2, result;
    printf("This program adds two complex numbers.\n");
    printf("Enter real and imaginary parts of first\number: ");
    scanf("%f %f", &val_1.real, &val_1.imaginary);
    printf("Enter real and imaginary parts of second\number: ");
    scanf("%f %f", &val_2.real, &val_2.imaginary);
    result.real = val_1.real + val_2.real;
    result.imaginary = val_1.imaginary + val_2.imaginary;
    printf("The sum is %f + i * %f\n", result.real,
result.imaginary);
    return (0);
}

Program 10.11 uses nested structures: struct date is used in the declaration of struct cheque_deposited. Because the dot operator associates from left to right, in order to access the month parameter of struct date of structure nobel_prize, we write:

nobel_prize.issued.mm;
/* Program 10.11; file name: unit10-prog11.c */
#include <stdio.h>
struct date
{
    int dd;
    int mm;
    int yy;
};
struct cheque_deposited
{
    char by_whom_given [30];
    char bank [30];
    char local_or_not [2];
    char cheque_number [10];
    struct date issued;
    struct date deposited;
    float amount;
}
Programming in C

nobel_prize;

int main (void)
{
    printf("Size in bytes of struct cheque_deposited is %d\n",
    sizeof (struct cheque_deposited));
    printf("Size in bytes of struct date is %d\n",
    sizeof (struct date)); /*Continued from previous page*/
    printf("Size of month parameter in struct date of ");
    printf("structure nobel_prize is: %d\n",
    sizeof nobel_prize.issued.mm);
    return (0);
}

Here are some exercises for you.

E5) Write a C language program to create an array of structures to record and print
employee payroll data. (Refer to exercise 2.)

E6) Write a C language program to create an array of structures to record and print
student fee payment data. (Refer to exercise 3.) Bus fees will depend upon actual
distance between home and school, which must be input to the program in each
case. Fees may be paid in cash or by cheque. If fees are paid by cheque, your
program should record relevant details.

E7) Write a C program to record and print data about students enrolled in the various
courses at a University. (Refer to exercise 4.)

In the next section, we will discuss the function fseek() that can be used to randomly
access any part of the file.

10.5 STRUCTS AND FILES: fseek()

Structs are customarily associated with files and databases. Our program to record the
transactions listed in a passbook must store the information in files if it’s to be any use
at all. Naturally, because details about cheque issues and deposits must be different
from those for cash withdrawals and deposits, it may be simplest to have separate files
for each of these activities. Program 10.12 below is a “quick and dirty” way of writing
the information about deposited cheques to a file named cheqdpst.dat, and fetching
the data about any cheque already deposited. It’s quick and dirty because it uses the
same char array, namely value_holder [], to read the number of records to be
entered, the amount on each cheque deposited, and the number of the record to be
fetched; nor does the program store the number of transactions currently held in
cheqdpst.dat. Moreover, it has no error checking, no facility for data correction
after entry, no input for date variables, etc. But these features are easily built in. The
new, and extremely useful feature that this program introduces, is the C capability to
seek out an arbitrarily chosen record from the hundreds or thousands that may exist in
cheqdpst.dat. The fseek() function allows a file to be treated as though it were an
array of bytes: we can position ourselves anywhere inside that we want, and read any
number of bytes forwards or backwards from it!

The ‘‘a+’’ option in the call to fopen() opens the file cheqdpst.dat for
appending and reading. If the file does not exist, this call to fopen() creates it.

/* Program 10.12 */
#include <stdio.h>
#include <stdlib.h>
#define NUM_CHEQUES 100
struct cheque_deposited
{
    char by_whom_given[30];
    char bank[30];
    char local_or_not[2];
    char cheque_number[10];
    float amount;
};

int main (void)
{
    FILE * fp;
    struct cheque_deposited cheques[NUM_CHEQUES], cheque_holder;
    int number, rcrd_nmb, i;
    char value_holder[30];
    float amt_deposited = 0.0;
    if ((fp = fopen("cheqdpst.dat", "a+")) != NULL){
        printf("How many cheques to be deposited? ");
        gets(value_holder);
        number = atoi(value_holder);
        for (i = 0; i < number; i++){
            printf("\t\t\t...Enter details for cheque %d...\n", i + 1);
            printf("Cheque issued by: ");
            gets((cheques[i]).by_whom_given);
            printf("Bank name: ");
            gets((cheques[i]).bank);
            printf("Local bank (Y/N): ");
            gets((cheques[i]).local_or_not);
            printf("Cheque number: ");
            gets((cheques[i]).cheque_number);
            printf("Cheque amount: ");
            gets(value_holder);
            cheques[i].amount = atof(value_holder);
            amt_deposited += cheques[i].amount;
        }
        fwrite(cheques, sizeof(struct cheque_deposited), number, fp);
        printf("Amount deposited was: %.2f\n", amt_deposited);
        gets(value_holder);
        rcrd_nmb = atoi(value_holder);
        rewind(fp);
        if ((fseek(fp, (long)(rcrd_nmb - 1) * sizeof(struct cheque_deposited)), 0)) == 0){
            fread(&cheque_holder, sizeof(struct cheque_deposited), 1, fp);
            printf("Cheque details are as under:\n");
            printf("cheque issued by: ");
            puts(cheque_holder.by_whom_given);
            printf("Bank name: ");
            puts(cheque_holder.bank);
            printf("Local bank (Y/N): ");
            puts(cheque_holder.local_or_not);
            printf("Cheque number: ");
            puts(cheque_holder.cheque_number);
            printf("Cheque amount: ");
            printf("%.2f\n", cheque_holder.amount);
        }
        else
            printf("Sorry, unable to locate record %d\n", rcrd_nmb);
    }
    fclose(fp);
else
    printf("Error opening passbook.dat for \nappending records...\n\n");
return (0);
}

fseek()
fseek() need three arguments: the first is a file pointer, the second, a long offset value, which is the number of bytes the position pointer must move in order to home in to the chosen byte; the third argument of fseek() specifies the origin from which offset is measured: this can be 0, 1 or 2. 0 sets the file beginning as the origin of the search, 1 sets the current position in the files as the search origin, and 2 sets the end of the file as the search origin. fseek() returns a non-zero value on error.

if ((fseek (fp, (long)
 ((rcrd_nmb - 1) * sizeof (struct cheque_deposited)), 0)) == 0)
 etc.

10.6 STRUCTS AND FUNCTIONS: THE ARROW OPERATOR

Error checking is extremely important to any program that writes data to files. Program 10.13 validates a date read in as a struct via the function int valid_date(). Each member of the structure any_date is passed to valid_date() as a separate parameter. The function cannot alter the values of the arguments sent to it. The function zeller() is invoked only after valid_date() confirms that the three date parameters sent to it are good. As usual, it reports the day corresponding to the given date. Note that it does so without the switch statement.

/* Program 10.13; file name: unit10-prog13.c */
#include <stdio.h>
#define LEAP_YEAR(Year) (Year %4 == 0 && Year % 100 != 0) || Year % 400 == 0
struct date
{
    int dd;
    int mm;
    int yy;
};
int valid_date(int D, int M, int Y);
int Zeller(int D, int M, int Y);
char *days[] =
{
    "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"
};
int main()
{
    struct date any_date;
    printf("This program reads a date in the \nformat dd-mm-yyyy\n");
    printf("and determines if its valid. \nIf it is, it returns the\n");
    printf("Corresponding day. Enter a date: ");
    scanf("%d-%d-%d", &any_date.dd, &any_date.mm, &any_date.yy);
    if (valid_date(any_date.dd, any_date.mm, any_date.yy))
        printf("That was a %s", ...
days[Zeller(any_date.dd, any_date.mm, any_date.yy)];

    } else
    
    printf("Not a valid date.\n");

return (0);
}

int valid_date(int dd, int mm, int yy)
{
    if (yy < 0)
return 0;
    if (yy < 100)
        yy += 1900;
    if (yy < 1582 || yy > 4902)  
return 0;
    if (!(LEAP_YEAR(yy)) && (mm == 2) && (dd > 28))
return 0;
    if (!(LEAP_YEAR(yy)) && (mm == 2) && (dd > 29))
return 0;
    if (mm < 1 || mm > 12)
return 0;
    if (dd < 1 || dd > 31)
return 0;
    if ((dd > 30)
        && (mm == 4 || mm == 6 || mm == 9 || mm == 11))
return 0;

return 1;
}

int Zeller(int DD, int MM, int YY)
{
    int ZM, ZY;
    if (MM < 3)
        ZM = MM + 10;
else
        ZM = MM - 2;
    if (ZM > 10)
        ZY = YY - 1;
else
        ZY = YY;

return (((int) ((13 * ZM - 1) / 5) + DD + ZY % 100 +
(int)) (ZY % 100) / 4) - 2 * (int) (ZY / 100) +
(int) (ZY / 400) + 77) % 7;

return (0);
}

Instead of passing the members of a structure separately to a function (as was done in
the program above), one can pass a structure to a function in toto, provided that the
struct declaration is known to the function. As usual, if the values of the members are
changed in the function, then those changes are restricted to the function. Program
10.14 uses a function to implement complex multiplication. The function has two
structures for its parameters, each of type struct complex, and returns a value of the
same type.

/* Program 10.14; file name: unit10-progl4.c */
#include <stdio.h>
struct complex
{
    float real;
    float imaginary;
};
struct complex result (struct complex val_1, struct complex val_2);  
int main (void)
{
struct complex num_1, num_2, product;
printf("This program multiplies two complex numbers.\n");
printf("Enter real and imaginary parts of first number: ");
scanf("%f %f", &num_1.real, &num_1.imaginary);
printf("Enter real and imaginary parts of second number: ");
scanf("%f %f", &num_2.real, &num_2.imaginary);
product = result (num_1, num_2);
printf("Result: %f + i * %f\n", product.real, product.imaginary);
return (0);
}

struct complex result (struct complex v1, struct complex v2)
{
    struct complex answer;
    answer.real = v1.real * v2.real
    - v1.imaginary * v2.imaginary;
    answer.imaginary = v1.real * v2.imaginary
    + v2.real * v1.imaginary;
    return answer;
}

Pointers to structures are declared in the same way as pointers to any other variable:

struct complex * pc;

pc is now a pointer to a structure of type complex. The address of a structure of this
type may be assigned to it.

pc = &num_1; (For ANSI C compliant compilers pc would now point to the first
member of num_1; for pre-ANSI C compilers this may not be true.) Structure pointers
may be passed to functions. This is often more convenient than passing (i.e. copying)
an entire structure into the function, but there is an inherent risk of unintended changes
to the structure. Within the function the real and imaginary parts of the complex
number referenced by pc are given by (*pc).real and(*pc).imaginary.

The parentheses are necessary. Though the “contents of” operator binds very tightly
to its operand, the membership operator has a higher priority. If the parentheses were
absent the order of evaluation would have been *(pc.real) and
* (pc.imaginary). But pc is not a struct, it’s a pointer. It doesn’t have members;
pc.real and pc.imaginary are therefore illegal constructs. n the other hand, the
notation (*pc).real is cumbersome: so C has as an alternative—the
arrow operator -> (which consists of a minus sign followed by the greater than
sign) for the member reference operator. The member reference:

pc->real has the same effect as: (*pc).real

Program 10.15 shows how the arrow operator is used. The program determines the
distance between two points (x1, y1) and (x2, y2). Each point is represented as a
structure, and the addresses of these structures are passed to the function distance().
The arrow operator is used to extract the value of each of the members—the x and y
c-ordinates—of the structures from their pointers. These values are used in
Pythagoras’s formula for the distance between (x1,y1) and (x2,y2).

/* Program 10.15; file name: unit10-prog15.c*/
#include <stdio.h>
#include <math.h>
struct point
{
    float x;
    float y;
};
float distance(struct point *p1, struct point *p2);
int main(void)
{

```c
struct point p1, p2;
printf("This program determines the distance\n");
printf("between two points in the X-Y plane.\n");
printf("Enter X and Y co-ordinates of first point: ");
scanf("%f %f", &p1.x, &p1.y);
printf("Enter X and Y co-ordinates of second point: ");
scanf("%f %f", &p2.x, &p2.y);
printf("Distance between (%f, %f) and (%f, %f) is: %f\n",
p1.x, p1.y, p2.x, p2.y, distance(&p1, &p2));
return (0);
}
float distance(struct point *p1, struct point *p2)
{
    return (sqrt((p1->x - p2->x) * (p1->x - p2->x) +
        (p1->y - p2->y) * (p1->y - p2->y)));
}
```

Here are some exercises for you.

E8) Write a C language program to print a file backwards.

E9) Write a C language program to maintain a personal telephone directory. Your
    program should support the following functions: 1) add a new number to the
directory; 2) locate and display a number, given the owner’s name; 3) change a
number; 4) delete a number; and 5) exit from the program gracefully.

In the next section, we will discuss another variable type called `union` which can hold
variables of different types.

### 10.7 UNIONS

A `union` is a C variable typed by the keyword `union` that can accommodate at any time
just one of its listed members, which may be objects of different types, i.e. of different
widths:

```c
union hold_all
{
    char alpha;
    int beta;
    float gamma;
    double delta [3];
    int * ptr;
}
```

This instances `box_1`, `box_2` and `box_3` of the `union` `hold_all` can hold variables
of each of its listed types. Obviously they will be wide enough to hold the biggest of its
members, in this case `double` `delta [3]`. Each member of a `union` is stored at the
same base address. Similarly to structures, the dot and arrow operators are used to
access any of the `union`’s members, via the mechanisms:

```c
union_name.member_name;
```

as in:

```c
box_1.alpha = 'x';
box_2.beta = 3;
box_3.delta [0] = 3.14;
```

or `union_pointer->member`, as in:

```c
x->gamma = 2.71
```
where \( x \) is a pointer to a `union` variable, say `box_1`.

A `union` may be initialised by a value of the type of its first member. If a `union` has been assigned a value of a certain type, then it is not correct to attempt to retrieve from it a value of a different type. Program 10.16 and its output illustrate some properties of `unions`. \( x \) is a pointer of type `union` `hold_all`, and is a parameter of the function `func()` whose two other parameters are an `int` and a `double`. The `union` `hold_all` is just wide enough to accommodate the three `doubles` of the array `delta [ ]`. The unions `box_1`, `box_2` and `box_3` are assigned a `char`, and `int` and a `double` value respectively. `func()` is invoked with the following arguments: a pointer to `box_1`, and the current values of `box_2.beta` and `box_3.delta[0]`. Within `func()` itself the expression `P->alpha` reference the member `alpha` of `box_1`, which has the value 'x'. This is changed to 'y'.

/* Program 10.16 */
#include <stdio.h>

union hold_all
    {
    char alpha;
    int beta;
    float gamma;
    double delta [3];
    int * ptr;
    }
box_1, box_2, box_3, * x;

void func (union hold_all *p, int q, double r);

int main (void)
{
    printf("The size of the union hold_all is: %d\n",
    sizeof (union hold_all));
    printf("The size of an instance of it, box_1 is: %d\n",
    sizeof box_1);
    box_1.alpha = 'x';
    box_2.beta = 3;
    box_3.delta[0] = 3.14;
    printf("Can func () change box_1.alpha, which noe is: %c?\n", box_1.alpha);
    printf("Let\'s see...sending \ncontrol of func ()...\n");
    func(&box_1, box_2.beta, box_3.delta[0]);
    printf("Yes it can: box_1.alpha after func()\nis: %c\n", box_1.alpha);
    return (0);
}

void func (union hold_all *P, int Q, double R)
{
    P->alpha += R/Q;
}

/* Program 10.16: Output */
The size of the union hold_all is: 24
the size of an instance of it, box_1 is: 24
Can func () change box_1.alpha, which now is: x?
Let\'s see... sending control to func()...
Yes it can: box_1.alpha after func () is: y

In the next section, we will discuss how to manipulate the individual bits of values integer variables.
No doubt it has occurred to you that Booleans should be representable by individual bits in memory, rather than by entire bytes or words. When several flags (i.e. Boolean values) must be stored, the savings made possible by using bits within a word rather than whole words can be considerable. This idea directly motivates the concept of packed fields of bits, and operations on individual bits.

In C there are two ways in which the individual bits within a word can be manipulated. The first depends upon storing a value as an int and then to use the bitwise operators, explained below, to mask or set specific bits of the word. The second uses the bit field approach, in which the syntax of definition and the access method are based on structures.

C has six bitwise operators, of which five are binary and one unary. They may be applied to any signed or unsigned integer operands, char, short, int or long. The unary operator, a tilde sign, one’s complements its operand, i.e. it changes the 1-bits to 0s, and vice versa. It associates from right to left. Suppose the int \( x \) stores the value -1 on a two’s complement machine. On such a machine this int value is represented by all bits set to 1s. Then \( \sim x \) will have all its bits set to 0s. Of the binary bitwise operators, three are logical connectives: in order of decreasing priority they are:

- & the bitwise AND operator
- | the bitwise OR operator, and
- ^ the bitwise EXCLUSIVE OR operator

The bitwise AND operator produces a result every bit of which is the result of ANDing the corresponding bits of its two operands. The bitwise OR operator produces the result of ORing the bits of its operands. Similarly the bitwise exclusive OR yields the value resulting from XORing its operands. (The difference between the OR and the exclusive OR is that in the latter case if both operands are true, the result is false. For example, if I’m going to Jaipur from Delhi by bus or by train, then it’s false that I’m simultaneously travelling by both forms of transport: I can be either on the bus, or on the train, but not on both.)

The two shift operators << and >> push their left operand to the left or to the right, respectively, a number of bit positions given by their right operand.

To illustrate how these operators are used, suppose a 16-bit int \( x \) has the octal value 012345, i.e. 000101001100101 binary, and another 16-bit int \( y \) has the octal value 023456, i.e. 0010011100101110 binary, then \( x \& y \) is produced by ANDing the corresponding bits of \( x \) and \( y \), as follows:

\[
\begin{align*}
x &= \text{000101001100101, octal 12345} \\
y &= \text{0010011100101110, octal 23456} \\
x \& y &= \text{0000010000100100, i.e. octal 2044}
\end{align*}
\]

Similarly, ORing the bits of \( x \) and \( y \) yields:

\[
\begin{align*}
x &= \text{00010100111100101} \\
y &= \text{0010011100101110} \\
x | y &= \text{0011011111111111, i.e. octal 33757}
\end{align*}
\]

The exclusive OR of the bits of \( x \) and \( y \) yields:

\[
\begin{align*}
x &= \text{0001010011100101} \\
y &= \text{0010011100101110} \\
x^y &= \text{0011001111001011, i.e. octal 31713}
\end{align*}
\]
x << 2 pushes out the 2 leftmost bits of x, and replaces the vacated bits on the right by zeroes; this is equivalent to multiplying x by 4. Similarly y >> 2 pushes out to the right the 2 rightmost bits of y. If y is an unsigned quantity, as it is in the present instance, the vacated bits will be stuffed with 0s; if y is a signed value, then an arithmetic shift will replace the vacated bits by 1s, but a logical shift will replace them by 0s. Your compiler’s manual should tell you which of the two shifts is implemented on your system; if not, a simple program can help you find out. Program 10.17 illustrates the bit fiddling operators:

```c
/* Program 10.17; file name: unit10-prog17.c */
#include <stdio.h>
int main (void)
{
    int w = -1;
    int x = 012345, y = 023456, z;
    printf("w = %d, one's complement = ~w = %d\n", w, ~w);
    printf("x =\t\t%o\ny =\t\t%o\nz = x & y = %o\n", x, y, x & y);
    printf("x =\t\t%o\ny =\t\t%o\nz = x | y = %o\n", x, y, x | y);
    printf("x =\t\t%o\ny =\t\t%o\nz = x ^ y = %o\n", x, y, x ^ y);
    printf("w = %d, w << 2 = %d, w >> 2 = %d\n", w, w << 2, w >> 2);
    return (0);
}
```

/* Program 10.17: Output: */
w = -1, one’s complement = ~w = 0
x = 12345
y = 23456
z = x & y = 2044
x = 12345
y = 23456
z = x | y = 33757
x = 12345
y = 23456
z = x ^ y = 31713
w = -1, w << 2 = -4, w >> 2 = -1

The second way to manipulate bits within a word is by creating a field structure. Groups of contiguous bits within a word are called a field. A field is assigned both a name as well as a width, this being the number of bits in the field. The value stored within a field must be an int-like quantity:

```c
struct bitfields
{
    unsigned x : 2;
    unsigned y : 2;
    unsigned z : 3;
    unsigned z : 1;
} psw;
```

This defines a variable called psw which contains four unsigned bit fields.
The number following the colon is the field width. Field variables may be assigned values; be careful however that the value assigned to a field is not greater than the maximum storable inside it. Individual fields are referenced as though they were structure members. The assignments:

```
psw.z = 5;
```

set the bits in the field z as 101. The **signed** or **unsigned** specification makes for portability; this is important because bit fields are extremely implementation dependent. For example, C does not specify whether fields must be stored left to right within a word, or vice versa. Some compilers may not allow fields to cross a word boundary. Unnamed fields may be used as fillers. In declaring the structure pc as follows we have forced a 3 bit gap between the fields x and y:

```
struct
{
  unsigned x : 2;
  : 3;
  unsigned y : 4;
} pc;
```

An unnamed field of width 0 will cause the next field to begin in the following word instead of at the boundary of the last field. The fields within a word do not have addresses. It is incorrect to use the `&` (address of) operator with them.

---

E10) Write a C language program to count the number of bits in your computer’s word.

E11) Write a C program to implement right or left rotation of bits within a word. n bits (where n is a positive integer less than the word size) are pushed out to the right or left, caught as they fall, and pushed in from the left or right respectively.

---

We now end our discussion by recalling what we have discussed so far in the next section.

### 10.9 SUMMARY

In this Unit we have studied how disk files may be read and written by C programs. File I/O functions are for the most part similar to the keyboard/monitor I/O functions that we have already seen in the preceding Units; they are therefore particularly easy to use.

**fopen()**

```
fopen(string_pointer, options)
```

where the string_pointer is either the file name itself as a string(like "password.dat" for example) or a pointer to a string containing the file name. The options are given in Table. 1 on page 121.

**fprintf()**

```
fprintf(file pointer, control string, expressions)
```

This is similar to `printf()` but for the extra argument, the file pointer.

**fscanf()**

```
fscanf(file pointer, control string, expressions)
```

This is similar to `scanf()` but for the extra argument, the file pointer.
### sprintf() and sscanf()

`sprintf(output_buffer, control_string, list);`

`sscanf(input_buffer, control_string, list);`

are for reading strings writing output to strings.

### fgets() and fputs()

`fgets(pointer to a char array, N chars, file pointer)`

is for reading an entire line from a file pointed to by the `file pointer`. If the number of characters in the line is more than `N - 1` it reads the `N - 1` characters; otherwise it reads the entire line.

`fputs(pointer to a char array, file pointer)`

writes the string to the file.

### getc() and putc()

`getc(file pointer)`

reads a character from the file that `file pointer` points to. It returns EOF if it reaches the end of the file or there is an error.

`putc(Character to be written., file pointer)`

writes the `Character to be written` to the file pointed to by the `file pointer`.

### fread() and fwrite()

Both functions have four arguments. The first of these is the address of a memory buffer from which `fwrite()` writes into a file, (or into which `fread()` reads from a file). The second argument of each functions is the size of the items to be written or read; the third, the number of items; and the fourth, the file pointer.

### rewind()

The function `rewind(file pointer)` makes the `file pointer` to point to the beginning of the file.

### feof()

The function `feof(file pointer)` can be used to check if the file pointer is pointing to the end of the file.

### fseek()

`fseek(file pointer, offset, origin)`

allows us to access any portion of the file. The first argument is the `file pointer`. The third is the position in the file, 0 for the beginning, 1 for the current position and 2 for the end of the file. The second is the offset from the origin specified in argument 3.

Structs are a C facility that allow the storage of heterogeneous but related information. A struct variable has members which can be accessed by the dot and arrow operations.

A union is a type of struct that can hold at any time just one of its members, which may be of various types. It is similar to Pascal’s variant records.

It is possible to manipulate the bits within an int-like word. There are two ways by which this may be accomplished. The first method depends upon the six bitwise operators of C: The second uses the approach of bit fields, which are groups of...
contiguous bits within a word, and which can be assigned small integer values. Bit fields are accessed like structure members.

### 10.10 SOLUTIONS/ANSWERS

E1) See the program in Listing 1. This is just a modified version of the program in page 20 of Kernighan and Ritchie’s book, second edition. Create a small file to test the program and work through the program to find what happens when your file is processed by the programme.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define IN 1 /*inside a word.*/
#define OUT 0 /*outside a word.*/
int main(int argc, char *argv[])
{  
    FILE *file;
    char name[20];
    int c, nl, nw, nc, state;
    nl=nw=nc=0;
    state = OUT;
    strcpy(name,argv[1]);
    printf("File name is %s.,name);
    if((file=fopen(name,"r")) == NULL){
        printf("Unable to open file.");
        exit(1);
    }
    else{
        while((c=getc(file)) != EOF){
            ++nc;
            if (c == '
')
                ++nl;
            if (c == ' ' || c == '
' || c == '	')
                state = OUT;
            else if(state == OUT){
                state = IN;
            }
```
E8) The program is given in Listing 2. As in the program to count the number of lines, words and characters in a file, create a small example file and work through it.

Listing 1: Solution to exercise 1.

Listing 2: Solution to exercise 8.