UNIT 6 CONTROL STRUCTURES-II

Structure **Page No.**

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:

for(;;)

```
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:

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

sets i = 3 in the loop body. But

```
i = 3;
```

```
for(;;)
printf("%d\n", i);
```

does this just as well. In each case, the `printf()` statement is executed forever, because an absent Boolean(the *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_exp* is `false`, the loop is not entered.

The `printf ()` below can never be executed:

```
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:

```
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:

```
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

```
/* 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 ++)
```

```c
/* 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...");
    printf("Was the loop entered?\n");
    return (0);
}
```

Listing 4: Exercise 2.

```c
/* 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...j = %d\n", j);
    printf("Is the Boolean j < 6 still true ?\n%s",
           j < 6 ? "Yes," : "No, ");
    printf("because j is now %d.\n", j);
    return (0);
}
```

Listing 5: Exercise 2.

```c
/* 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 \
quite easy to use.\n");
        printf("I\'ve said this %s.\n",
j == 0 ? "once" : "twice");
    };                              /*End for */
    printf("What I say %d times must be true!\n", j);
    return (0);
}
```

Listing 6: Exercise 2.

```c
/* 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 %d is \n %d and %d, respectively.\n", i - 1,
sum_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.

```
/* 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

$$3, 5, 7, \cdots \text{(int)} \sqrt{n}$$

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

```
/* 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;

the **switch** statement traps each kind of prime.

```
/* Program 6.10; File name:unit6-prog10.c */
#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 \
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:

```
if      (number % factor == 0)
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:

```
while   (search for a factor has been successful)
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:

```
factor += 2             /* if last factor was odd */
```

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

```
1  /* Program 6.11; File name: unit6-prog11.c */
2  #include <stdio.h>
3  #include<stdlib.h>
4  int main()
5  {
6      int number, quotient, factor, no_pfactors = 1;
7      printf("Enter a number, I'll print it\'s prime factors:");
8      scanf("%d", &number);
9      quotient = number;
10     if (number == 2) {
11         printf("2 is a prime. The factors are 1, 2.");
12         exit(0);
13     }
14     printf("Prime factors with multiplicity are: 1");
15 /* cast out factors of 2, if any */
16     while (quotient % 2 == 0) {
17         no_pfactors++;
18         printf(", %d", 2);
19         quotient /= 2;
20     }
21 /* only odd factors can remain, if any */
22     for (factor = 3; factor <= quotient; factor += 2)
23         while (quotient % factor == 0) {
24             no_pfactors++;
25             printf(", %d", factor);
26             quotient /= factor;
27         };
28     if (no_pfactors == 2)
29         printf("\n The number is prime.");
30     printf("\n");
31     return (0);
32 }
```

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 \
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:

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

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 you 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.

E14) 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 **double**; `n` is an **int**. All functions return **double**. The expressions in Table. 1, `int * exp, double * ip`

Table 1: Mathematical Functions in math.h

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

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 `int` variables–`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) `ints` together, in one go, by a single statement, e.g.

```
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 **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 **ith** student will then be accessible in the `int marks [i]`, where the **index** (or **subscript) 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 `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 `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 `int` variables, say `mrks_7` against `mrks_8`, one could compare the value of `marks [i]` against `marks [j]`, for any `i` and any `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

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

reserves 40 consecutive storage locations to hold 40 variables of type `int`. See the illustration below. Though the addresses of the array elements are only illustrative, note

Memory Address	Array Element	Contents of array element
6700	marks [0]	77
⋮	⋮	⋮
6728	marks [7]	53
6732	marks [8]	32
6736	marks [9]	81
⋮	⋮	⋮
6856	marks [39]	64

Table 2: The array `marks [40]`, consisting of 40 consecutive `int`s

that they increase in steps of four bytes, the width of `int` in all the 32-bit machines.

Program in Listing 13 on the next page illustrates how values are read in into the array `marks []`. These marks are then averaged (`avg`), and the numbers of students who received marks greater than or equal to the average mark (`abv_avg`) is printed, as well as the number of those who obtained below average marks (`bel_avg`). Finally the array is sorted to arrange the entries in descending order. The declaratory statement of Program 6.13

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

declares `marks[]`; to be an `int` array of dimension 40. The `for (;;)` loop which follows immediately is used to read in a value into each of these elements. The variable `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 `printf()` is written to print student's numbers from 1 through 40. Note in the `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 **interchange sort** is the simplest of all sorting algorithms. It uses two `for (;;)` loops. The first picks out, turn by turn, each of the forty elements of `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("\n\nIn descending order marks are:\n\n");
    for (i = 0; i < 40; i++)
        printf("%d\n", marks[i]);
    printf("\n\nThe average mark is %f\n", avg);
    printf("\n\n%d students obtained above average\
marks.\n", abv_avg);
    printf("%d students obtained below average\
marks.\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];
```

will be quite acceptable.

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 **long int**, to pile the primes in:

```
long primes [1000]; /* assuming there are 1000 such primes */
```

After the **ith** prime has been generated and assigned to the array element `primes [i]`, the next prime generated must naturally be placed in the next longword, `primes[i+1]`. So this time we'd want that incrementing the index `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 **int**s or **long**s: 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 **char**s or **int**s or **float**s or **double**s, or of a user defined data type.

This is a powerful feature of arrays: incrementing the index in the array `marks []`, where each student's mark occupies an **int**, enables one to move from one item of type **int** to the next. But incrementing the index in the array `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 `i` as desired. Arrays bestow the power of random access to their elements, through the subscript `i`.

```
/* Program 6.14; File name: unit6-prog14.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);
}
```

Listing 14: A program to list primes of the form $k^2 + 1$.

Note that in the last example we dimensioned the array `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:

```
seats [6] = ‘B’;
```

Freeing the 23rd seat after a cancellation would mean:

```
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 **char**s 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:

```
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’.

> ANSI C
>
> 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 **int**s, as they are generated during an executing program; and suppose also that the numbern 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:

$$14 = 3 + 11 = 7 + 7$$
$$16 = 3 + 13 = 5 + 11$$
$$30 = 7 + 23 = 11 + 19 = 13 + 17 \text{ etc.}$$

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

The program in Listing 16 illustrates the syntax of the **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>
int main()
{
    char a = 'b';
    static char vowels[] = { 'a', 'e', 'i', 'o', 'u' };
    static char vowel_string[] = "aeiou";
    static int five_primes[] = { 2, 3, 5, 7, 11 };
    static double tenbignums[10] = { 3816.54729, 1654.72938 };
    printf("The size of char on this system \
is: %d\n", sizeof(char));
    printf("Therefore the size of char a is: %d\n", sizeof a);
    printf("However, the size of\'b\' is: %d\n", sizeof 'b');
    printf("The size of short on this system \
is: %d\n", sizeof(short));
    printf("The size of int on this system \
is: %d\n", sizeof(int));
    printf("The size of long  on this system \
is: %d\n", sizeof(long));
    printf("The size of float on this system \
is: %d\n", sizeof(float));
    printf("The size of double on this system \
is: %d\n", sizeof(double));
    printf("The number of bytes in vowels [] \
is: %d\n", sizeof vowels);
    printf("The number of bytes in vowel_string [] \
is: %d\n", sizeof vowel_string);
    printf("The number of bytes in five_primes [] \
is: %d\n", sizeof five_primes);
    printf("The number of bytes in tenbignums [] \
is: %d\n", sizeof tenbignums);
    printf("The number of bytes in STRING is: \
%d\n", sizeof STRING);
    return (0);
}
```

Listing 16: The 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'b' 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 `sum_n`; these variables, and the variables `number, factor, primes [1000]` and `i` from the previous block, in fact all program variables that we've encountered thus far are examples of **automatic** variables.

```
/* 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 envoloping 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 \
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 `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 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 `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 `primes[]` can be added to `sum_n`. Variables defined externally to a block are accessible inside it, and inside any nested blocks. (That is why our **#define**s have been written outside of `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 `main`(): then they will be accessible in all the code that follows it. 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 **auto** in their declarations, thus:

```
auto int i;
```

to remind their readers (and themselves) that the local value of the variable is to be used in the present instance. **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 \
when control\n");
            printf("entered the inner block for \
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", i, j);
            k++;
            goto endprogram;
        }
        printf
            (" Local value of i has precedence over \
global value ... \n ");
        printf(" Therefore its value is:%d \ n ", i);
        prinf(" 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 \
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 \
retained ? \n ");
    printf(" No, because...i = %d, j = %d \ n ", i, j);
    printf
        (" However, the last value of k is still \
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**s.

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;
int beta;
int main()
{
    auto int alpha = 10;
    printf("%d, %d\n", alpha, beta);
    beta = 2;
  innerblock:
    {
        int alpha = 15;
        static int gamma = 20;
```

```
            gamma /= beta;
            printvalues:
            printf("%d, %d, %d\n", alpha, beta, gamma);
            alpha++;
            beta++;
            gamma++;
        }
        printf("%d, %d\n", alpha, beta);
        if (beta == 3)
            goto innerblock;
        else if (beta == 4)
            goto printvalues;
        return (0);
    }
```

Listing 19: Exercise 24

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 **for** (;;) loop, and its application to a variety of situations. The **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 **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 **static**.

3. The **sizeof** operator is a unary operator which, associating from right to left, yields as an **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: **auto**, **register** and **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 `first_expression` sets the value of i to 1. Next, the boolean expression is checked. Recall that the expression 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 i. Since the `third_expression` is missing, it will evaluate the second expression, whose value is `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("\nEnter multiplier: ");
    scanf("%d", &val_1);
    printf("\nEnter 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 {
        printf("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 $n^2 - n + 41$ 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 ...\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("For i = %d,  f(i) = %d is composite...\n",
                       i, number);
                break;
            } else if (factor * factor > number) {
                printf("For i = %d, f(i) = %d is a prime...\n", i,
number);
                break;
            }
    }           /*End For */
    return (0);
}
```

Listing 25: Solution to exercise 10.

We leave the next polynomial as an exercise to you.

E11) See Listing 26.

```
/* 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;
            };
        printf("\n");
        printf("Sum of the factors is %d:\n", so_factors);
        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.

```
/*Answer to exercise 12, unit6; File name unit6-ans-ex-12.c*/
#include<stdio.h>
int main()
{
    int i, po_two = 0, po_five = 0, dummy;
    for (i = 1; i <= 1000; i++) {
        dummy = i;
        while (dummy % 2 == 0) {
            po_two++;
            dummy /= 2;
        }
        while (dummy % 5 == 0) {
            po_five++;
            dummy /= 5;
        }
    }
    printf("Power of 2 = %d, Power of 5 = %d\n",
po_two, po_five);
    printf("The number of zeros in 10000! is %d",
           po_five < po_two ? po_five : po_two);
    return (0);
}
```

Listing 27: Solution to exercise 12.

E13) The program is in Listing 28.

```
/*Solution to exercise 13, unit 6.
File name:unit6-ans-ex-13.c*/
#include <stdio.h>
#include <math.h>
#define TOL .000001
int main()
{
    int i, no_iterations = 2500;
    double x_zero = 1.0, x_one;
    for (i = 0;; i++) {
        x_one = 8.99999 / (6.0 - x_zero);
        if (fabs(x_one - x_zero) <= TOL) {
            printf("One root of eqn. is: %10.5f\n", x_one);
            printf("convergence achieved after %d \
iterations.\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 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 \
iterations.\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++)
```

```
            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.