UNIT 6: CONTROL STRUCTURES - II

Structure

6.0 Introduction
6.1 Objectives
6.2 The for (;;) Loop
6.3 Unidimensional Arrays
6.4 The Initialization of Arrays and the size of Operator
6.5 Storage Classes and Scope
6.6 Summary

6.0 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. 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. This idea becomes immensely powerful when arrays are used in conjunction with loops: one can simply match the index of the loop with the subscript of the datum. Then the set of statements to process a single datum can be repeatedly used, in a loop. to process each succeeding array element the array subscript varying as the loop index varies. A new data item can be processed for each new value of the loop index. Typically, suppose there is an organization with eight thousand workers that wishes to award a bonus to each worker equal to 10% of her wages: then wages may be declared as an array of 8000 floats, and each element of this array - accessible via its subscript - can be processed by the same set of statements. These can be executed in a loop with the subscript (the array index) coinciding with the loop index.

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 exist and (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.

6.1 OBJECTIVES

After working through the examples and exercises of this unit you will be able to:

- write programs using the for (;;) loop
- create and make use of the array data structure
- use the size of operator
- apply the register, auto and static storage class rules.

6.2 THE for (;;) LOOP

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 of 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 (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 initialize die loop index: first-expr is evaluated before the loop is entered. Though it is customary to initialize the loop index via first _ expr it is not a requirement: as in the while () loop, the loop index may be initialized 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 die 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-initialize 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 (;;)
 printt'("%d\n". i);
```

does this just as well. In each case, the printf () statement is executed forever. because an absent Boolean is regarded as true. Nor is the presence of third-expr a syntactical requirement. 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. 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:

```
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);
```

What values will it print?

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-initializes the loop index after each iteration; generally it increments the variable initialized by first expr. but it may change (increase or decrease) the index by any amount. Second _expr determines whether this 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 is re-initialized, so that it gets the value 4. The Boolean is found to he 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 Program 6.4. In Program 6.1 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-initialization 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.

```
/* Program 6.1 */
#include
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);
    }
```

In Program 6.2 below, we rise the for (;;) property that the loop re-initialization is performed after each execution of the loop body, which in this case consists of the null statement:

```
/* Program 6.2 */
#Include
  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);
    }
```

In Program 6.3 depends on the for (;;) property that the Boolean is evaluated each time before the loop is entered.

```
/* Program 6.3 */
#include
main ( )
        {
          int i, sum = 0;
          for (i = 1; i < 101 && (sum +:r- i ++);)
             ;
         printf ("The sum of the numbers from 1 to %d is %d\n", i - 1,sum);
        }
```

Check Your Progress 1

Question 1: State the out put of Program 6.4 , 6.5 and 6.6 below :

```
 /* Program 6.4 */
 # include
 main ( )
        {
          int j = 5;
          for ( prientf ("will this loop be entered? \n" ); j < 2;j = 1)
             prientf ( prent this , if the loop is entered....");
          prientf ( "Was the loop entered? \n");
        }

 /* Program 6.5*/
 # include
 main ( )
        {
          int j = 5;
          for ( prientf (" Will this loop be entered ?\n"); j < 6; j = 7 )
             prientf ( " Prient this, if the loop is entered ....j=%d\n",j)
          prientf ( "Is the Boolean j 6 still true ?\n%s",j < 6 ? "Yes," :
        "No,");
          prientf ( "because j is now % d.\n" , j );
        }

/* Program 6.6 */
# include
 main ( )
        {
          int j;
          for ( j = O; j < 2; j ++)
            {
              Prientf ( " The for (;;) loop is really quite easy to use.\n");
              Prientf ( " I\'ve said this %s.\n",j=0? " once P: "twice");
            }
          Prientf (" What I say % d times must be true!\n",j);
        }
```

Question 2: Replace the for (;;) statement inProgram 6.1 above by:

```
For (I = 1, sum = 0; i < 101; I++)
```

Realise that the comma oprator 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 related on exit from the loop .

Question 3: In Program 6.3 above, can the && operator be realised by the || operator ? Can it be replaced by the comma operator ? Are the parentheses around

Sum +=i++

required?

Question 4: Program 6.7 uses the if -then else operator to sum the odd and even integer from 1 through 100 separately. Why are the parentheses in the if- then - else operator required?

```
/* Program 6.7 */
# include
main ( )
   {
     int I, sum -odd = 0 , sum - even = 0;
     for ( I=1; I<=100; I ++ )
       I % 2? (sum-odd + =I): (sum - even +=I);
     Prientf( "The sum of the even numbers and odd number from 1\
     To % d is and %d , respectively . \ n" , I=1 even ,sum -odd);
   }
```

Question 5: Create, compile and execute Programs 6.1 - 6.7 above, replacing for (;;)statements by while ()statements.

Question 6: Rewrite the programs for the Russian Peasant Multiplication Algorithm and the Collatz problem using for (;;) loops.

Let's work through Program 6.8 below to quickly recapitulate what we've learnt about the for (;;) loop.

```
/* Program 6.8 */
#include
main ( )
   {
    int a, b, c;
    for (b=O;b++ < 2;c=b++)
       a=b;
     printf ("a = %d, b = %d, c = %d\n", a, b, c);
     for (a=b=O,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=O;a=(b++<5)&&c++<=6;a=b++,c++)
       printf ("a = %d, b = %d, c = %d\n", a, b, c);
     for (a=O,b=2;++b<20-(c=b/2>5?-a:++a);)
       b += a;
     priiitf ("a = %d, b = %d, c = %d\n", a, b, c);
   }
```

In the first for (;;) loop b is initialized to 0. The Boolean h ++ 2 is true, the loop statement is executed so that a gets the current value of h, 1. In the re-initialization, 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, h and c arc assigned the values 1, 3 and 1 respectively.

in the next for (;;) loop. note that a and h are each 0 to begin with, while c is 38. In both the testing as well as the re-initialization phase the value of c is decreased by a factor of 3; and the loop is exiled when c becomes 0. Before the loop is first entered, c is reduced to 37: h is post-incremented to 1 in this iteration of the loop; then in the re-initialization, it's again post-incremented, while c is reduced to 36. Before the loop is re-entered, a gets the value 2, and c becomes 35. The process continues, until b has the value 8, and a the value 20.

In the third for (;;) loop, the Boolean post-increments h and c each to 1, and is itself true, so a gets the value 1, and die loop is entered. The printf ( ) prints the current values of a, h and c. Before the Boolean is tested for the next round of execution, h and c are each post-incremented to 2, while a remains at 1. The Boolean is still true, but h and c are post-incremented to 3 and a is reassigned the value 1. In the next re-initialization, a gets the value 3, while h 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-initialization 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 number. 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 5.8 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,. ....,(int) square-root (n)..

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

```
/* Program 6.9 */
#include
main ( )
        {
          int number. factor,
          prititf (.'Enter a number,I\II tell you if it\'s a prime:")
          scanf ("%d", &number);
          if (number == 2 11 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;
           }
    }
```

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. Program 6.10 is a modified version of Program 6.9; the switch statement traps each kind of prime.

```
/* Program 6.10 */
#include
main ( )
    {
      unsigned limit, number, factor, endigit1 = 0.
      endigit3 = 0, endigit7 = 0, endigit9 = 0;
      printf ("Enter number upto which test will run\n");
      do
        {
           printf ("Value entered must be greater than 7. less than
        65533: ");
           scanf ("%u",&limit);
        }
    while ((limit < 7) 11 (limit > 65533));
    for (number = 7; number <= limit; number += 2)
        {
          for (factor = 3; ; factor +=2)
            if (number % factor == 0)
              break;
            else
              if (factor * factor > number)
    /* contd.*/

        {
                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);
 }
```

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

```
Enter number upto which test will run Value entered must be greater than 7, less than 65533: 65533

Primes upto 65533 ending with 1 = 1625
Primes upto 65533 ending with 3 = 1645
Primes upto 65533 ending with 7 = 1647
Primes upto 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 itslogic:

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 with in 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. There fore, beginning with 3, other factors, if any, can only be generated by:

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

Since the largest prime factor of a number can be no larger than its square root our search must end as soon as:

```
                factor * factor >= number
```

with these preliminaries out of the way let's look at Program 6.11:

```
/* Program 6.11*/
    #include
    main ( )
        {
          int number, quotient. factor.
          printf ("Enter a number, I'll print it\'s prime factors:");
          scanf ("%d", &number);
          quotient = number;
          printf ("Factors are: 1");
            /* cast out factors of 2, if any*/
          while (quotient % 2 me 0)
            {
              printf (",%d",2);
              quotient /= 2;
            }
          /* only odd factors can remain, If any*/
          for (factor = 3; factor <= quotient; factor += 2)
             {
              if (factor * factor > number)
            {
              printf(",%d; number is prime\n", number);
              break;
            }
          while (quotient % factor = 0)
            {
               printf (". %d",factor);
               quotient /= factor;
            }
            if (quotient== 1)
               break;
            if (factor * factor > quotient)
                {
                printf(",%d", quotient);
                break;
                }
            }
        printf ("\n");
        }
```

Exercise: Execute Program 6.11 for various values of input.

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 * x + 6.3 * x - 2.7 = 0

We can write this equation as

x * x + 6.3)'= 2.7

or

x=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

xn+l = 2.7 / (xn + 6.3)

Setting a first guess for xo, we can determine x1; having found x1, we can substitute it back in the formula to find X2, 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? There fore it's a good idea to set a limit on the number of iterations to be performed, so you don't get "caught in the loop", going round and round In circles looking for a root that isn't there1 Program 6.12 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 your 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).

```
/* Program 6.12 */
#include
#include
#define TOL 1.0e-5
main ( )
      {
       int i;
       double x_zero = 1.0, x_onc;
       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;
        }
      }
```

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 crises 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

$$x = F(x)$$

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.

Check Your Progress 2

Question 1: Modify Program 6.9 above to verify the claim that the numbers generated by the formula:

Number = n * n - n + 41

For all n ranging from 1 through 40 are primes.

Repeat for:

Number = n * n - 79 * n + 1601 [1 n < 79]

Question 2: Execute program 6.11 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 factors 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 equal the number itself. A number is called abundant if the sum of its factors exceeds the number, define if this sum is less than the number. Modify your program to determine if a number is abundant, perfect or define.

Question 3: Write a C program to determine how many zeroes there are at the end of number 1000! = 1*2*3*.........* 1000.

Question 4: Verify that nearly 1300 iterations are required to find one of the equation:

x*x-6.0* x + 8.99999=0

Correct to six places of decimals.

Question 5: Find one root of

$$3 * x - \sqrt{(1 + \sin(x))} = 0$$

in the neighbourhood of 0.4.

table 6.1

Mathematical functions in math.h

All arguments x, y in the following are assumed to be double; n is an int. All functions return double.

sin (x)	sine of x, x in radians
cos (x)	cosine of x, x in radians
tan (x)	tangent of x, x in radians
asin (x)	inverse sine of x,-1.O <= 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
log 10 (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 * 2n
frexp (X, int *exp)	returns the mantissa of a double value and stores the characteristic as a as a power of 2 in *exp
modf (x, double* ip) part in *ip	returns the positive fractional part of its first argument. Stores integral
fmod (x, y)	floating point remainder of x / y, with the same sign as x.

Note: The expressions in Table 6.1 above: int*px, double*ip are pointers, discussed in a later Unit.

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 students 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 pro ram. 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 4(X),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 realized, 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.

Memory Address	Array Element	Contents of array element
6700	marks. [0]	77
		.
		.
6714	marks [7]	53
6716	marks [8]	32
6718	marks [9]	81
		.
		.
6778	marks [39]	64

Fig. 1 The array marks [40], consisting of 40 consecutive ints. Though the addresses of the array elements are only illustrative, note that they increase in steps of two bytes, the width of int.

Program 6.13 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.

```
/* Program 6.13 */
#include
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);
    }
```

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 upto 39 to cover all the forty students' marks. But because we arc more used to counting from 1 onwards, the printf () is written to print students' 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 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 arc 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: die compiler will revolt. On the other hand, a macro definition for 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 upto hundred million of the form k * k + 1. 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 fil, the next prime generated must naturally be placed in the next long word, primes [i+1]. So this time we'd want that incrementing the index i should lead us from the current long word to the next long word. C provides us this convenience; we don't have to worry about whether array elements are ints or 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 chars or ints or floats or doubles, 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 */
#define TRUE 1
#define FALSE 0
#include
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 ("%1d\n", primes [i]);
              }
          }
      }
```

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 language considerately do, it would let you overwrite on 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 840. and the largest is 99800101: prove 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 Program 6.15 below, 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').

```
/* Program 6.15*/
#include
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 1 denominator + '0';
              next-num %= denominator;
            }
         printf ("%d", integer part);
         for (i = 0; i <1001; i ++)
           putchar (result [i]);
        }
```

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 travelers) 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 scat is available, the contents of the corresponding byte are set to the char 'A'; but when that scat is booked, the value in the byte is changed 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 scat 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 scat after a cancellation would mean:

```
                    Seats [22] = 'A';
```

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

6.4. THE INITIALISATION OF ARRAYS, AND THE size of 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 die 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 a char array called example [], the appropriate C statement is:

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

In most pre-ANSI C's 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 initialization 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 initialized 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 initialized 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 initialized to non-zero values, each element of it must be explicitly defined. 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 scat 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 initializing 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 initialized. One can determine the size in bytes of an undimensioned array, and of a variety of other objects, using the size of 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), gf 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:

Size of (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.

[Goldbach conjectured (1742) that any even number greater than 4 can be decomposed into the sum of two old primes, thus:

14=3+ 11 =7+7
16=3+ 13=5+ 11
30=7+23= 11 + 19= 13+ 17 etc.

Though this conjecture has never been. found to fail, a proof is still awaited. Nor do we know precisely how many such decompositions are possible for an aribitrary even integer. Interestingly enough, numbers divisible by 15, seem to have significantly more partitions into two primes thin oilier, approximately equal, numbers.]

The number of bytes to be set aside will then be 2 * 2 * n on a machine on which the size of int is two byes (two bytes per prime), but 2 * 4 * n on one in which the size of int is 4 bytes (a VAX or Macintosh). Neither of these expressions is portable. But 2 * size of (int) *n can be used without modification on any system.

Program 6.16 illustrates the syntax of the size of operator. Note carefully the parentheses around the basic types. Observe that the program yields the expected sizes of dimensioned as well as undimensioned arrays.

```
/* Program 6.16 */
#define STRING "What is the number of bytes in this string?"
#include
main ( )
       {
         char a = 'b'.,
         static char vowels ('a', 'e', 'i', 'o', 'u'};
         static char vowel-suing = "aciou";
         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", size of (char));
         printf ("There fore the size of char a is: %d\n', size of a);
         printf ("However, the size of \'b\' is: %d\n", size of 'b');
         printf ("The size of short on this system is: %d\n", size of (short)),
         printf ("The size of int on this system is: %d\n", size of (int));
         printf ("The size of long on this system is: %d\n", size of (long));
       /*contd */

          printf ("The size of float on this system is: %d\n", size of(float));
          printf ("The size of double on this system is: %ft", size of (double));
         printf ("The number of bytes in vowels [] is: %d\n", size of vowels);
         printf ("The number of bytes in' vowel-string [] is: %d\n", size of vowel-string);
         printf ("The number of bytes in five _ primes [] is: %d\n", size of five _ primes);
         printf ("The number of bytes in tenbignums [] is: %d\n", size of tenbignums);
         printf ("The number of bytes in STRING is: %d\n", size of STRING);
       }
```

The output of the program on an IBM PC 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: 2
The size of short or. this system is: 2
The size of int on this system is: 2
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: 10
The number of bytes in tenbignums [] is: 80
The number of bytes in STRING is: 44
```

Check Your Progress 3

Question 1: Explain the third line in the out put above.

Question 2: Count carefully the number of bytes in STRING. Why is there a discrepancy between the number of bytes you counted, and the number in the ninth line of output?

Question 3: Modify the above program to prove that the null string is precisely one byte long.

Question 4: Execute program 6.16 on different compilers, and architecturally different computers if possible. Do you get the same result each time? Or the same result as above?

Question 5: Examine the following declaration:

```
Static char hello []=
                    {

                    h','e','l','l','o'

                    }
```

Why is hello [] not a string?

Question 6: 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]?

Question 7: What is the difference between an assignment statement and an initialization?

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. Program 6.17 is baseion such a logical division. It consists of two blocks, the first of which contains code to generate the array primes [10001; while the second tests every other value of sum n (because half the values of sum-n are even, anyway, and there- fore composite) and lists lose that are prime. (In the next unit, where functions are intro- duced, we shall learn to encode each separate activity as a function, as a more appropriate division of labour.)

The second block of Program 6.17 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 automat- ic variables.

```
/* Program 6.17 *1
#include
main ( )
      {
      /* Outermost block
      int number, factor, primes [1000], i;
      /* This section generates a thousand primes
      and stores them in primes [1 */
```

```
    primes [01 = 2; primes [11 = 3; i = 2;
    for (number=5;;number+=2)
        {
          for (factor = 3; ; factor += 2)
             if (number % factor == 0)
            break;
             else
            if (fictor * factor > number)
                {
                  primes [i ++] = number;
                  break;
                }
        if ( i == 1000)
            break;
        continue;
    }

/* 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
    [10001 and i are
    available here.*/
    int n;
    unsigned sum n = 2.
    for (n = 1; (suin_n += primes [n]) < 32767; n ++)
        {
        if (n % 2 == 0)
           continue;
        for (factor = 3; ; factor += 2)
           if (sum_n % factor == 0)
            break;
           else
        if (factor * factor > sum_n)
            {
            printf ("The sum of the first %d primes \
            is %u and is itself a primes", n + 1, sum~n);
            break:;
        }
    continue;
    }
  }
/* The variables of the ncsted block are not available here. */
}
```

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 0 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 exist. 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 Program 6.17 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 #defines 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 initialized 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 Program 6.18 and its output to become familiar with the properties of automatic variables.

```
                    /* Program 6.18*/
                    # include <stdio.h>
                            int k=7;
                    main ()
                            {
                    nt i = 5, j = 6;
            printf (" Control s presently in outer block, where\n");
                    printf ("i = % d, j = %d\n", i, j , k):
                    redefineij:
                                    {
                            int i= 1, j:
                            innerblock;
            printf ("Control has now ent ered the nner block...... \n");
                            if ( k== 8)
                                    {
            printf ("i and j were not initialised when control\n"):
    printf )"enteed the nner block for t  he second time...\n");
            prntf ("Are ther former values retaned ?\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"):
            /* contd.*/
```

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... k8
Control has now entered the inner block....
i and j were not initialized 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 he applied to register variables.

Another C storage declaration 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 initialized to 0, in contrast to autos.

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 initialized.

Check Your Progress 4

Question 1: Change the declarations in the inner block of Program 6.18, execute it and verify that static variables behave as expected.

Question 2: State the output of the following program:

```
/* Program 6.19*/
#include <stdio.h
int alpha=5;
int beta;
main ( )
               {
                 auto int alpha=10;
                 printf ("%d, %d\n", alpha, beta);
                 beta =2;
                 innerblock:
                         {
                           int alpha = 15;
                           static int gamma = 20;
                           gamma/=beta;
                           print values;
                           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;
                    }
```

6.6 SUMMARY

In this unit we have studied 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 initializes the loop variable; the second determines whether the loop be entered; and the third reinitializes 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.

In this Unit we have also 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. Is identified by its 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 initialized must be declared as static.

The size of 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 army.

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 variables may take arbitrary values if they are not initialized externally defined variables or static variables are by default initialized to zero. A variable external to a block is accessible inside the block.