Chapter 5

REPITITION STATEMENTS, ARRAYS AND STRUCTURED PROGRAMMING

This chapter explores the different methods programmers use to construct repeating sections of code and how that code can be implemented in C++. More commonly, a section of code that is repeated is referred to as a *loop*, because after the last statement in the code is executed, the program branches, or loops, back to the first statement and start another repetition through the code. Each repetition is also referred to as *an iteration*.

5.1 <u>BASIC LOOP STRUCTURES</u>

The real power of a program is realized when the same type of operation must be made over and over.

Constructing a repetitive section of code requires that four elements be present. The first necessary element is a repetition statement. This repetition statement defines the boundaries containing the repeating section of code and also controls whether the code is executed or not. C++ provides three different forms of repetition statements:

1. *while* structure
2. *for* structure
3. *do-while* structure

Each of these statements requires a condition that must be evaluated, which is the second required element for constructing repeating sections of code. Valid conditions are similar to those used in selection statements. If the condition is true, the code is executed; otherwise, it is not.

The third required element is a statement that initially sets the condition. This statement must always be placed before the condition is first evaluated to ensure correct loop execution the first time the condition is evaluated.

Finally, there must be a statement within the repeating section of code that allows the condition to become false. This is necessary to ensure that, at some point, the repetition stop.

The condition being tested can be evaluated at either (1) the beginning or (2) the end of the repeating section of code.

If the test occurs at the beginning of the loop, the type of loop is called a *pre-test loop* or *entrance-controlled loop*. If the test occurs at the end of the loop, the type of loop is called a *post-test loop* or *exit-controlled-loop*.

In addition to where the condition is tested (pretest or posttest), repeating sections of code are also classified. In a *fixed count loop*, the condition is used to keep track of how many repetitions have occurred. In this kind of loops, a fixed number of repetitions are performed, at which point the repeating section of code is exited.

In many situations, the exact number of repetitions are not known in advance or the items are too numerous to count beforehand. In such cases, a *variable condition loop* is used. In a variable condition loop, the tested condition does not depend on a count being achieved, but rather on a variable that can change interactively with each pass through the loop. When a specified value is encountered, regardless of how many iterations have occurred, repetitions stop.

5.2 while LOOPS

The *while* statement is used for repeating a statement or series of statements as long as a given conditional expression is evaluated to true.

The syntax for the *while* statement:

```
while( condition expression){
        statements;
}
```

The flow chart of the *while* statement is given in Figure 5.1.

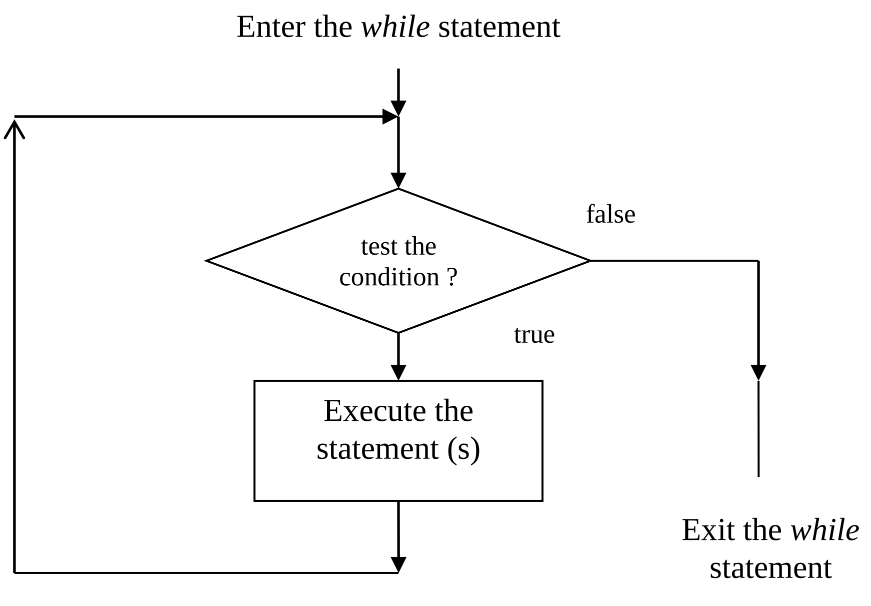

Figure 5.1 The flow chart of the *while* statement

Example 5.2.1

```
// this program computes the sum of 10 first integers starting from 1
#include <iostream.h>
int main()
{
  const int N = 10
  int sum = 0;
  int count = 1;            // initialize count
  while (count <= N){
    sum = sum + count;
    count++;            // increment count
  }
  cout << “The sum is “ << sum << endl;
  return 0;
}
```

The output of the above program:

```
The sum is  55
```

In the above program, the loop incurs a *counter-controlled repetition*. Counter-controlled repetition requires:

1) the name of a control variable (the variable *count* in this case);
2) the initial value of the control variable (*count* is initialized to 1 in this case)
3) the condition that tests for the final value of the control variable (i.e., whether looping should continue) ;
4) the *increment* (or *decrement*) by which the control variable is modified each time through the loop.

Example 5.2.2

```
#include <iostream.h>
int main()
{
  int i;
  i = 10;
  while (i >= 1)
  {
    cout << i << "  ";
    i--;            // subtract 1 from i
  }
  return 0;
 }
```

The output of the above program:

```
    10  9  8  7  6  5  4  3  2  1
```

5.3 INTERACTIVE while LOOPS

Combining interactive data entry with the repetition capabilities of the *while* statement produces very adaptable and powerful programs.

Example 5.3.1

```
// Class average program with counter-controlled repetition
#include <iostream.h>
int main()
{
  int total,       // sum of grades
      gradeCounter, // number of grades entered
      grade,       // one grade
      average;     // average of grades

  // initialization phase
  total = 0;
  gradeCounter = 1;                  // prepare to loop

  while ( gradeCounter <= 10 ) {       // loop 10 times
    cout << "Enter grade: ";          // prompt for input
    cin >> grade;                    // input grade
    total = total + grade;           // add grade to total
    gradeCounter = gradeCounter + 1;  // increment counter
  }

  // termination phase
  average = total / 10;              // integer division
  cout << "Class average is " << average << endl;
  return 0;
}
```

The following is a sample run of the above program:

```
Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82
Enter grade: 94
Class average is  81
```

Sentinels

In programming, data values used to indicate either the start or end of a data series are called *sentinels*. The sentinel values must be selected so as not to conflict with legitimate data values.

Example 5.3.2

```
#include <iostream.h>
```

```
int main()
{
  float grade, total;
  grade = 0;
  total = 0;
  cout << "\nTo stop entering grades, type in any number less than 0.\n\n";
  cout << "Enter a grade: ";
  cin  >> grade;
  while (grade >= 0 )
  {
    total = total + grade;
    cout << "Enter a grade: ";
    cin  >> grade;
  }
  cout << "\nThe total of the grades is " << total << endl;
  return 0;
}
```

The following is a sample run of the above program:

```
To stop entering grades, type in any number less than 0.

Enter a grade: 95
Enter a grade: 100
Enter a grade: 82
Enter a grade: -2

The total of the grades is  277
```

***break* statement**

The *break* statement causes an exit from the innermost enclosing loop statement.

Example:

```
while( count <= 10)
{
    cout << “Enter a number: “:
    cin >> num;
    if (num > 76){
         cout << “you lose!\n”;
         break;
    }
    else
       cout << “Keep on trucking!\n”;
    count++;
}
//break jumps to here
```

The *break* statement violates pure structured programming principles because it provides a second, nonstandard exit from a loop.

However, it is useful and valuable for breaking out of loops when an unusual condition is detected.

***continue* Statements**

The *continue* statement halts a looping statement and restarts the loop with a new iteration.

```
while( count < 30)
{
   cout << “Enter a grade: “;
   cin >> grade;
   if (grade < 0 || grade > 100)
      continue;
   total = total + grade;
   count++;
}
```

In the above program, invalid grades are simply ignored and only valid grades are added to the total.

The null statement

All statements must be terminated by a semicolon. A semicolon with nothing preceding it is also a valid statement, called the *null statement*. Thus, the statement

```
  ;
```

is a null statement.

Example:

```
if (a > 0)
   b = 7;
else ;
```

The null statement is a do-nothing statement.

5.4 <u>for LOOPS</u>

The *for* statement is used for repeating a statement or series of statements as long as a given conditional expression evaluates to true.

One of the main differences between *while* statement and *for* statement is that in addition to a conditional expression, you can also include code in the *for* statement

- to initialize a counter variable and
- changes its value with each iteration

The syntax of the *for* statement:

```
for ( initialization expression; condition; update statement){
    statement(s);
}
```

In its most common form, the initialization expression consists of a single statement used to set the starting value of a *counter variable,* the condition contains the maximum or minimum value of the counter variable can have and determines when the loop is finished, and the update statement provides the increment value that is added to or subtracted from the counter variable each time the loop is executed.

The flowchart of the *for* statement is given in Figure 5.2.

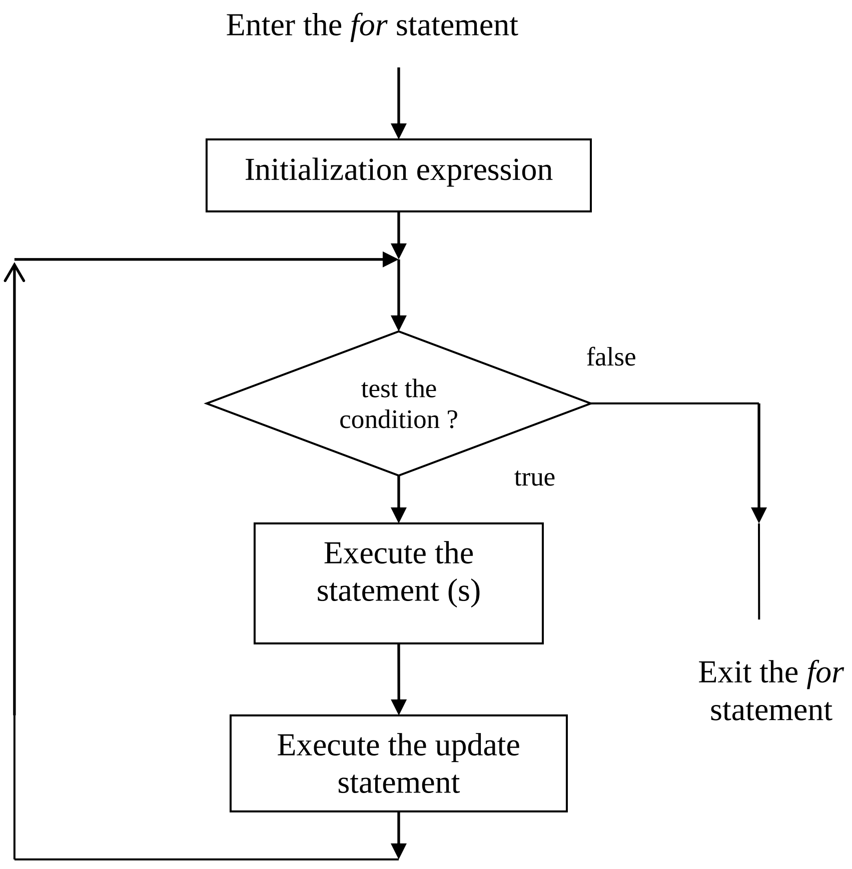
Enter the *for* statement
Initialization expression
test the condition ?
false
true
Execute the statement (s)
Execute the update statement
Exit the *for* statement

Figure 5.2 The flow chart of the *for* statement

Example 5.4.1

```
#include <iostream.h>
int main()
{
  int sum = 0;
  for (int number  = 2; number <= 100; number += 2)
    sum += number;
  cout << “Sum is “ << sum << endl;
  return 0;
}
```

The output of the above program:

```
 Sum is  2550
```

Example 5.4.2
In this example, we have to solve the following problem:
A person invests $1000.00 in a saving account with 5 percent interest. Assuming that all interest is left on deposit in the account, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula for determining these amounts:

$$a = p(1 + r)^n$$

where *p* is the original amount invested, *r* is the annual interest rate and n is the number of years and *a* is the amount on deposit at the end of the *n*th year.

```
#include <iostream.h>
#include <iomanip.h>
#include <math.h>
int main()
{
  double amount,
              principal = 1000.0,
              rate = 0.05;

  cout << "Year” << setw(21)
       << "Amount on deposit" << endl;

  cout << setiosflags(ios::fixed | ios::showpoint) << setprecision(2);
  for (int year = 1; year <= 10; year++)
 {
   amount = principal*pow(1.0 + rate, year);
   cout << setw(4) << year
        << setw(21) << amount << endl;
 }
  return 0;
}
```

The output of the above program:

```
Year   Amount on deposit
   1             1050.00
   2             1102.50
   3             1157.62
   4             1215.51
   5             1276.28
   6             1340.10
   7             1407.10
```

```
  8          1477.46
  9          1551.33
  10         1628.89
```

5.5 NESTED LOOPS

In many situations, it is convenient to use a loop contained within another loop. Such loops are called *nested loops*.

Example 5.5.1

```
#include <iostream.h>

int main()
{
  const int MAXI = 5;
  const int MAXJ = 4;
  int i, j;
  for(i = 1; i <= MAXI; i++)     // start of outer loop
  {
    cout << "\ni is now " << i << endl;

    for(j = 1; j <= MAXJ; j++)   // start of inner loop
    cout << "  j = " << j;     // end of inner loop
  }                              // end of outer loop
  cout << endl;
  return 0;
}
```

The output of the above program:

```
i is now  1
  j = 1  j = 2  j = 3  j = 4
i is now  2
  j = 1  j = 2  j = 3  j = 4
i is now  3
  j = 1  j = 2  j = 3  j = 4
i is now  4
  j = 1  j = 2  j = 3  j = 4
i is now  5
  j = 1  j = 2  j = 3  j = 4
```

5.6 do-while LOOPS

The *do..while* statement executes a statement or statements once, then repeats the execution as long as a given conditional expression evaluates to true.

The *do..while* statement is used to create *post-test loops*.

The syntax for the *do..while* statement:

```
do {
    statements;
} while (conditional expression);
```

Example:

```
do {
    cout<< "\nEnter an identification number:";
    cin >> idNum;
} while (idNum < 1000|| idNum> 1999);
```

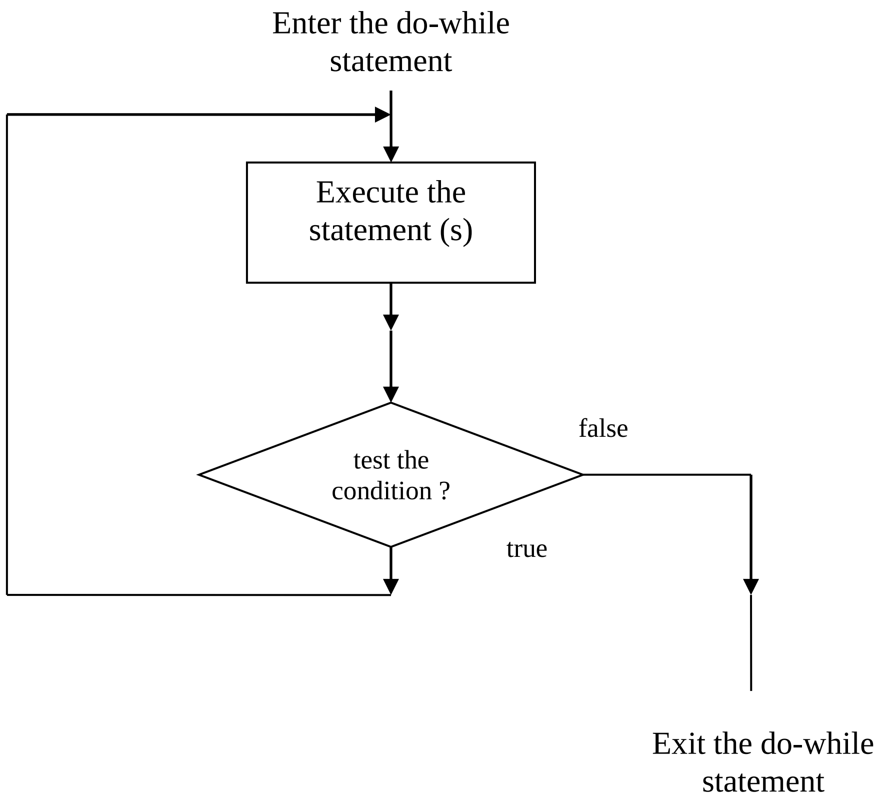
Enter the do-while statement
Execute the statement (s)
test the condition ?
false
true
Exit the do-while statement

Figure 5.3 The flowchart of the *do-while* statement

Here, a request for a new id-number is repeated until a valid number is entered.

```
do {
    cout<< "\nEnter an identification number:";
    cin >> idNum;
    if (idNum < 1000 || idNum > 1999)
   {
        cout << "An invalid number was just entered\n";
        cout << "Please reenter an ID number /n";
    }
    else break;
} while (true);
```

5.7 STRUCTURED PROGRAMMING WITH C++

The *goto* Statement

In C++, *goto* statement – an unconditional branch, is just a legacy code from C language. The result of the *goto* statement is a change in the flow of control of the program to the first statement after the *label* specified in the *goto* statement.

Example:

```
start:                  // label
    if (cout > 10) go to end;
    …
    …
    go to start;
end:  cout << endl;
```

The *goto* statement can lead to programs that are more difficult to debug, maintain, and modify.

Structured Programming

During the 1960s, it became clear that the indiscriminate use of transfers of control through *goto statements* was the root of much difficulty experienced by programmer groups. The notion of so-called *structured programming* became almost synonymous with "*goto elimination*."

Bohm and Jacopini's work demonstrated that all programs could be written in terms of only three control structures:

- sequence structure
- selection structure
- repetition structure

The sequence structure is built into C++. Unless directed otherwise, the computer executes C++ statements one after the other in the order in which they are written. Figure 5.4 shows a sequence structure.

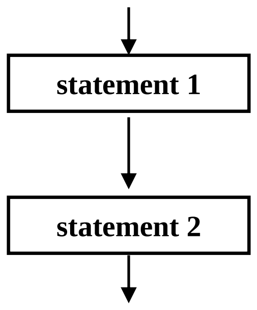

Figure 5.4 A sequence structure

C++ provides three types of selection structures:
- *if* statement (single-selection structure)
- *if-else* statement (double-selection structure)
- *switch* statement. (multiple-selection structure)

C++ provides three types of repetition structures:
- *while* statement
- *do-while* statement
- *for* statement

So C++ has only seven control structures: sequence, three types of selection and three types of repetition. Each C++ program is formed by combining as many of each type of control structures as is appropriate for the algorithm the program implements.

We will see that each control structure has only one *entry point* and one *exit point*. These *single-entry/single-exit control structures* make it easy to build programs.

One way to build program is to connect the exit point of one control structure to the entry point of the next. This way is called *control-structure-stacking*.

Another way is to place one control structure inside another control structure. This way is called *control-structure-nesting*.

Consistent applying reasonable *indentation* conventions throughout your programs greatly improves program readability. We suggest a fixed-size tab of about ¼ inch or three blanks per indent.

For example, we indent both body statements of an *if..else* structure as in the following statement:

```
if (grade >= 60)
   cout << "Passed";
else
   cout << "Failed";
```

Top-down Stepwise Refinement

Using good control structures to build programs is one of the main principles of structured programming. Another principle of structured programming is *top-down, stepwise refinement*.

Consider the following problem:
Develop a class-averaging program that will process an arbitrary number of grades each time the program is run.

We begin with a pseudocode representation of the *top*:

```
Determine the class average for the exam
```

Now we begin the refinement process. We divide the top into a series of smaller tasks and list these in the order in which they need to be performed. This results in the following *first refinement*.

First Refinement:

```
Initialize variables
Input, sum and count the exam grades
Calculate and print the class average
```

Here only the sequence structure has been used.

To proceed to the next level of refinement, we need some variables and a repetition structure. We need a running total of the numbers, a count of how many numbers have been processed, a variable to receive the value of each grade as it is input and a variable to hold the calculated average. We need a loop to calculate the total of the grades before deriving the average. Because we do not know in advance how many grades are to be processed, we will use sentinel-controlled repetition. The program will test for the sentinel value after each grade is input and will terminate the loop when the sentinel value is entered by the user. Now we come to the pseudocode of the second refinement.

Second Refinement:

```
Input the first grade(possibly the sentinel)
While the user has not as yet entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Input the next grade(possibly the sentinel)
Calculate and print the class average
```

The pseudocode statement

```
Calculate and print the class average
```

can be refined as follows:

```
If the counter is not equal to zero
    set the average to the total divided by the counter
    print the average
else
    Print "No grades were entered".
```

Notice that we are being careful here to test for the possibility of division by zero – a fatal error, if undetected, would cause the program to fail. Now we come to the pseudocode of the third refinement.

Third Refinement:

```
Initialize total to zero
Initialize counter to zero
Input the first grade
While the user has not as yet entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Input the next grade
If the counter is not equal to zero
    set the average to the total divided by the counter
    print the average
```

```
        else
            Print "No grades were entered".
```

Final step: After coding, we come to the following C++ program.

```
#include <iostream.h>
#include <iomanip.h>
int main()
{
  int total,       // sum of grades
      gradeCounter, // number of grades entered
      grade;        // one grade
  double average;  // number with decimal point for average

  // initialization phase
  total = 0;
  gradeCounter = 0;

  // processing phase
  cout << "Enter grade, -1 to end: ";
  cin >> grade;
  while ( grade != -1 ) {
    total = total + grade;
    gradeCounter = gradeCounter + 1;
    cout << "Enter grade, -1 to end: ";
    cin >> grade;
  }
  // termination phase
  if ( gradeCounter != 0 ) {
    average =  double ( total ) / gradeCounter;
    cout << "Class average is " << setprecision( 2 )
         << setiosflags( ios::fixed | ios::showpoint )
         << average << endl;
  }
  else
    cout << "No grades were entered" << endl;
  return 0;
  }
```

5.8 ARRAYS

An *array* is an advanced data type that contains a set of data represented by a single variable name.

An element is an individual piece of data contained in an array.

5.8.1 Array Declaration

The syntax for declaring an array is

type name[elements];

Array names follow the same naming conventions as variable names and other identifiers.

Example:

```
int MyArray[4];
char StudentGrade[5];
```

The declaration *int MyArray[3];* tells the compiler to reserve 4 elements for integer array *MyArray*.

The numbering of elements within an array starts with an index number of 0. An *index number* is an element's numeric position within an array. It is also called *a subsript*.

Each individual element is referred to as an *indexed variable* or a *subscripted variable* because both a variable name and an index or subscrip value must be used to reference the element.

<u>Example</u>:

StudentGrade[0] refers to the first element in the *StudentGrade* array.
StudentGrade[1] refers to the second element in the *StudentGrade* array.
StudentGrade[2] refers to the third element in the *StudentGrade* array.
StudentGrade[3] refers to the fourth element in the *StudentGrade* array.
StudentGrade[4] refers to the fifth element in the *StudentGrade* array.

Subscripted variables can be used anywhere scalar variables are valid. Examples using the elements of the MyArray array are:

```
MyArray[0] = 17;
MyArray[1] = MyArray[0] – 11;
MyArray[2] = 5*MyArray[0];
MyArray[3] = (MyArray[1] + MyArray[2] –3)/2;
Sum = MyArray[0] +MyArray[1] +MyArray[2] + MyArray[3];
```

<u>Example 5.8.1</u>

```
#include <iostream.h>
int main(){
   char StudentGrade[5]= {‘A’, ‘B’, ‘C’,  ‘D’, ‘F’};
   for ( int i = 0; i < 5; i++)
          cout << StudentGrade[i] << endl;
   return 0;
}
```

The output is:

```
A
B
```

```
C
D
F
```

Example 5.8.2

```
// Compute the sum of the elements of the array
#include <iostream.h>

int main()
{
  const int arraySize = 12;
  int a[ arraySize ] = { 1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45 };
  int total = 0;

  for ( int i = 0; i < arraySize; i++ )
    total += a[ i ];

  cout << "Total of array element values is " << total << endl;
  return 0 ;
}
```

The output of the above program is as follows :

```
Total of array element values is  383
```

5.8.2 Multi-Dimensional Arrays

The C++ language allows arrays of any type, including arrays of arrays. With two bracket pairs we obtain a two-dimensional array. The idea can be iterated to obtain arrays of higher dimension. With each bracket pair we add another array dimension.

Some examples of array declarations

```
int a[1000];          // a one-dimensional array
int b[3][5];          // a two-dimensional array
int c[7][9][2];       // a three-dimensional array
```

In these above example, *b* has 3 × 5 elements, and *c* has 7 × 9 × 2 elements. Starting at the base address of the array, all the array elements are stored contiguously in memory.

For the array *b*, we can think of the array elements arranged as follows:

	col 1	col2	col3	col4	col5
row 1	b[0][0]	b[0][1]	b[0][2]	b[0][3]	b[0][4]
row 2	b[1][0]	b[1][1]	b[1][2]	b[1][3]	b[1][4]
row 3	b[2][0]	b[2][1]	b[2][2]	b[2][3]	b[2][4]

<u>Example 5.8.3</u>
This program checks if a matrix is symmetric or not.

```
#include<iostream.h>
const int N = 3;
int main()
{
    int i, j;
    int a[N][N];
    bool symmetr = true;
    for(i= 0; i<N; i++)
       for (j = 0; j < N; j++)
         cin >> a[i][j];
    for(i= 0; i<N; i++)
         for (j = 0; j < N; j++)
           cout << a[i][j]<< endl;
    for(i= 0; i<N; i++){
         for (j = 0; j < N; j++)
           if(a[i][j] != a[j][i]){
              symmetr = false;
              break;
           }
         if(!symmetr)
            break;
     }
     if(symmetr)
         cout<<"\nThe matrix is symmetric"<< endl;
     else
          cout<<"\nThe matrix is not symmetric"<< endl;
    return 0;
}
```

5.8.3 Strings and String Built-in Functions

In C++ we often use character arrays to represent strings. A string is an array of characters ending in a null character ('\0'). A string may be assigned in a declaration to a character array. The declaration

```
char strg[] = "C++";
```

initializes a variable to the string "C++". The declaration creates a 4-element array *strg* containing the characters 'C', '+', '+' and *'\0'*. The null character (\0) marks the end of the text string. The declaration determines the size of the array automatically based on the number of initializers provided in the initializer list.

C++ does not provide built-in operations for strings. In C++, you must use a string built-in functions to manipulate *char* variables. Some commonly used string functions are listed in Table 5.1.

Tabe 5.1 Common string functions

Function	Description
strcat()	Append one string to another
strchr()	Find the first occurrence of a specified character in a string
strcmp()	Compare two strings
strcpy()	Replaces the contents of one string with the contents of another
strlen()	Returns the length of a string

The *strcpy()* function copies a literal string or the contents of a *char* variable into another *char* variable using the syntax:

```
strcpy(destination, source);
```

where *destination* represents the char variable to which you want to assign a new value to and the *source* variable represents a literal string or the char variable contains the string you want to assign to the destination.

The *strcat()* function combines two strings using the syntax:

```
strcat(destination, source);
```

where *destination* represents the char variable whose string you want to combine with another string. When you execute *strcat()*, the string represented by the source argument is appended to the string contained in the destination variable.

Example:

```
        char FirstName[25];
        char LastName[25];
        char FullName[50];
        strcpy(FirstName, “Mike”);
        strcpy(LastName, “Thomson”);
        strcpy(FullName, FirstName);
        strcat(FullName, “ “);
        strcat(FullName, LastName);
```

Two strings may be compared for equality using the *strcmp()* function. When two strings are compared, their individual characters are compared a pair at a time. If no differences are found, the strings are equal; if a difference is found, the string with the first lower character is considered the smaller string.

The function listed in Table 5.1 are contained in the *string.h* header file. To use the functions, you must add the statement *#include<string.h>* to your program.

Example 5.8.4

```
#include<iostream.h>
#include<string.h>
int main()
{
        char FirstName[25];
```

```
            char LastName[25];
            char FullName[50];
            strcpy(FirstName, "Mike");
            strcpy(LastName, "Thomson");
            strcpy(FullName, FirstName);
            strcat(FullName, " ");
            strcat(FullName, LastName);
            cout << FullName << endl;
            int n;
            n = strcmp(FirstName, LastName);
            if(n<0)
                        cout<< FirstName << " is less than "<< LastName<<endl;
            else if(n ==0)
                        cout<< FirstName << " is equal to "<< LastName<<endl;
            else
                        cout<< FirstName << " is greater than "<< LastName<<endl;
           return 0;
}
```

The output of the program:

```
Mike Thomson
Mike is less than Thomson
```

5.9. **<u>STRUCTURES</u>**

A *structure*, or *struct*, is an advanced, user-defined data type that uses a single variable name to store multiple pieces of related information.

The individual pieces of information stored in a structure are referred to as *elements*, *field*, or *members*.

You define a structure using the syntax:

```
            struct struct_name{
            data_type field_name;
            data_type field_name;
            ……..
            } variable_name;
```

For example, the statement

```
            struct emloyee{
               char idnum[5];
               char name[40];
               long salary;
            };
```

declares the form of a structure named *employee* and reserves storage for the individual data items listed in the structure. The *employee* structure consists of three data items or fields.

And the statement

```
        struct emloyee{
           char idnum[5];
           char name[40];
           long salary;
        } Emp;
```

declares that *Emp* is a structure variable which has the form of the structure *employee*. To access the field inside a structure variable, you append a period to the variable name, followed by the field name using the syntax:

        variable.field;

When you use a period to access a structure fields, the period is referred to as the *member selection operator*.

Example 5.9.1

```
#include <iostream.h>
struct Date    // this is a global declaration
{
  int month;
  int day;
  int year;
};
int main()
{
  Date birth;
  birth.month = 12;
  birth.day = 28;
  birth.year = 1986;
  cout << "\nMy birth date is "
       << birth.month << '/'
       << birth.day   << '/'
       << birth.year % 100 << endl;
   return 0;
}
```

The ouput of the above program is:

```
My birth date is  12/28/86
```

Arrays of Structures

The real power of structures is realized when the same structure is used for lists of data. Declaring an array of structures is the same as declaring an array of any other variable type.

<u>Example 5.9.2</u>:
The following program uses array of employee records. Each of employee record is a structure named *PayRecord*. The program displays the first five employee records.

```
#include <iostream.h>
#include <iomanip.h>
const int MAXNAME = 20;
                    // maximum characters in a name
struct PayRecord      // this is a global declaration
{
  long id;
  char name[MAXNAME];
  float rate;
};

int main()
{
  const int NUMRECS = 5;
                      // maximum number of records
  int i;
  PayRecord employee[NUMRECS] = {
                            { 32479, "Abrams, B.", 6.72 },
                            { 33623, "Bohm, P.", 7.54},
                             { 34145, "Donaldson, S.",  5.56},
                             { 35987, "Ernst, T.", 5.43 },
                             { 36203, "Gwodz, K.", 8.72 }
                                };

  cout << endl;   // start on a new line
  cout << setiosflags(ios::left);
                              // left justify the output
  for ( i = 0; i < NUMRECS; i++)
    cout << setw(7)  << employee[i].id
             << setw(15) << employee[i].name
             << setw(6)  << employee[i].rate << endl;
  return 0;
}
```

The output of the program is:

```
        32479       Abrams, B.          6.72
        33623       Bohm, P.            7.54
        34145       Donaldson, S.       5.56
        35987       Ernst, T.           5.43
        36203       Gwodz, K            8.72
```