Higher Secondary Course

COMPUTER APPLICATIONS

(Commerce)

CLASS - XII

Government of Kerala

DEPARTMENT OF EDUCATION

State Council of Educational Research and Training (SCERT), Kerala

2015

1 Review of C++ Programming

Significant Learning Outcomes

After the completion of this chapter the learner

- uses input statements in programs to enter data into the computer.
- uses output statements in programs to display various forms of output.
- applies various forms of if statements to make decisions while solving problems.
- compares else if ladder and switch statement.
- distinguishes different looping statements of C++.
- selects appropriate loop in programs for solving problems.
- uses the concept of nested loop in problem solving and predicts the output.
- identifies the effect of break and continue statements in loops by explaining their effect on the program flow.

The basic concepts of C++ language were discussed in Class XI. A good understanding of these concepts is very essential to attain the significant learning outcomes envisaged in the following chapters. This chapter is a quick tour to refresh the concepts and skills you acquired in C++ language in Class XI. Each concept will be presented with necessary details only. The most important aspects like selection statements and looping statements are explained with the help of programs. Some advanced features like nested loops and the effect of `break` and `continue` statements in loops are also introduced in this chapter.

Since we use GNU Compiler Collection (GCC) with Geany IDE for developing C++ programs, we should be aware of the structure of program, format of specifying the header file, size of the data types, etc.

1.1 Basics of C++

C++ being a programming language, we started by learning its character set followed by tokens, expressions and statements. We also discussed data types and their modifiers. While constructing expressions, we identified the need of data type conversions. Table 1.1 provides a brief idea of these elements.

An overview of C++	
Character set	Fundamental unit of C++ language. Classified into letters (a - z, A - Z), digits (0 - 9), special characters (# , ; : > { + etc.), white spaces (space bar, tab, new line) and some other characters whose ASCII code fall in the range from 0 to 255.
Tokens	Basic building blocks of C++ programs. Constituted by one or more characters. Classified into keywords, identifiers, literals, punctuators and operators.
Keywords	Reserved words that convey specific meaning to the language compiler.
Identifiers	User-defined words to identify memory locations, statements, functions, data types, etc. Certain rules are to be followed to ensure the validity of identifiers. Identifiers include variables, labels, function names, etc.
Literals	Tokens that do not change their value during the program run. They are also known as constants. Classified into integer constants, floating point constants, character constants and string constants. Integer constant is constituted by digits only with an optional plus (+) or minus (-) sign as the first character. Floating point constant is expressed in fractional form and exponential form. Character constant is a single character of C++ enclosed within single quotes. There are some special character constants, called escape sequences. They represent some non-printable or non-graphic characters like new line ('\n'), tab space ('\t') and punctuation marks like single quote ('\' '), double quotes ('\"'), question mark ('\?') etc. String constant is a sequence of characters enclosed by a pair of double quotes.
Operators	Symbols that tell the compiler about some operations. Each of them actually triggers a specific operation. Based on the number of operands (data on which operation is carried out), operators are classified into unary, binary and ternary. Another classification is based on the type of operation

An overview of C++	
	performed. They consist of arithmetic (+, -, *, /, %), relational (<, <=, >, >=, ==, !=) and logical (&&, ||, !) operators. These operators give some value as the result of the operation. There are some special operators named *get from* (>>) for input, *put to* (<<) for output and assignment (=) for setting a value in a variable. Another category of operators implicitly performs an assignment operation after an arithmetic operation. They include increment (++), decrement (--) and arithmetic assignment (+=, -=, *=, /=, %=) operators.
Punctuators	Special characters like comma (,), semi colon (;), hash (#), braces ({}) etc. used for the perfection of syntax of various constructs of the language used in programs. They have semantic and syntactic meaning to the compiler.
Data types	These are means to identify the type of data and associated operations handling these data. Data types are classified into fundamental and user-defined data types. Fundamental data types represent atomic values and they include `int`, `char`, `float`, `double` and `void`. Each data type excluding void has its own size and range for the values they represent. Data type void represents an empty set of data and hence its size is zero.
Type modifiers	The keyword `signed`, `unsigned`, `short` and `long` are the type modifiers. They are used with data types to modify the size of memory space and range of data supported by the basic data types.
Expressions	Expressions are constituted by operators and required operands to perform an operation. Based on the operators used, they are classified into arithmetic expressions, relational expressions and logical expressions. Arithmetic expression is divided into integer expression and real expression. Integer expression consists only of integer data as operands and it returns integer value. In real expressions, the operands and the return-value are floating point data.

An overview of C++	
Type conversion	Relational expressions consist of numeric or character data as operands and they return True or False as outputs. Logical expressions use relational expressions as operands in practice and return True or False value as results. When different types of operands are involved in an arithmetic expression, type conversion takes place. It is the process of converting the current data type of a value into another type. It may be implicitly and explicitly converted. In implicit type conversion, complier is responsible for the conversion. It always converts a lower type into higher one and hence it is also known as type promotion. In explicit conversion, user is responsible for the conversion. Here the user determines the destination data type and hence it is known as type casting.

Table 1.1: Basic elements of C++ language

1.1.1 Various statements in a C++ program

Usually every program begins with pre-processor directives. We use `#include` statement, the pre-processor directive to attach a header file to provide information about predefined identifiers and functions used in the program. The pre-processor directive statements are followed by `using namespace` statement. Usually we use the predefined namespace `std` to specify the scope of identifiers `cin` and `cout`. Then the `main()` function appears. It is an essential function in a C++ program, where the program execution starts and ends. It consists of declaration statements and a set of executable statements required for solving the problem. Let us have a close look at these statements.

Declaration statement

Variables are identifiers of memory locations and are used in programs to refer to data. They should be declared prior to their use in the program and data types are required for this. The following statements are examples for variable declaration:

```
int n, sum;
float rad, area;
signed int a,b,c;
```

Values can be provided to the variables along with the declaration as shown below:

```
int n=10;
```

This kind of statement is known as variable initialisation statement. The value assigned to the variable can be replaced by some other value later in the program. But the following statement does not allow changing the value of the variable.

```
const int n=10;
```

Here the variable initialisation begins with the access modifier `const`, a keyword that restricts a change in the value of the variable.

Input statement

C++ provides the operator **>>**, called extraction operator or get from operator. It is a binary operator and hence it requires two operands. The first operand is the pre-defined identifier `cin` that identifies keyboard as input object. The second operand is strictly a variable. We can use more than one variable in the same statement to receive more than one input. The following are valid examples:

```
cin>>rad;
cin>>a>>b>>c;
```

Output statement

To perform output operation, C++ gives the operator **<<**, called insertion operator or put to operator. It is also a binary operator. The first operand is the pre-defined identifier `cout` that identifies monitor as the output object. The second operand may be a constant, a variable or an expression. The following are some examples for output statements:

```
cout<< "hello";
cout<< area;
cout<< 25;
cout<< a+b+c;
cout<< "Sum of " << n << "numbers = " << sum;
```

Assignment statement

A specific data is stored in memory locations (variables) using assignment operator (**=**). The statement that contains = is known as assignment statement. It is also a binary operator and the operand to the left of = should be a variable. The operand after = may be a numeric constant, a variable or an expression of numeric type. The following assignment statements are valid:

```
n = 253;
area = 3.14*rad*rad;
a = b = c;
```

There are special assignment operators, called arithmetic assignment operators. They are **+=**, **-=**, ***=**, **/=** and **%=**. These are all binary operators and the operand to left of these operators should be variables. Their operations may be illustrated as follows:

```
n+=2;          // It is equivalent to n=n+2;
a*=b;          // It is equivalent to a=a*b;
sum-=n%10;     // It is equivalent to sum=sum-n%10;
```

The operators **++** and **--** are special operators of C++, which are, in a way, similar to assignment statements. These are unary operators and the operand should be a variable. The following statements illustrate the operations associated with them:

```
n++;           // It is equivalent to n=n+1;
a--;           // It is equivalent to a=a-1;
```

There are two versions for these operators: postfix form and prefix form. `a++;` and `a--;` are the postfix form of increment and decrement operators respectively. `++a;` and `--a;` are the prefix form. Whatever be the form, ++ operator adds 1 to the content of the operand variable and the result is stored in it.

The two versions differ when used with an assignment or output statement. Assume that **a** is an integer variable with value 5 and **b** is another integer variable. After the execution of the statement: `b=a++;` the value of `b` will be 5 and that of `a` will be 6. That is, `b=a++;` is equivalent to the statement sequence: `b=a;  a=a+1;`. So this method of incrementing is known as use and change method.

But the statement, `b=++a;` is equivalent to the statement sequence: `a=a+1; b=a;`. So the values of both `a` and `b` will be 6. This method of incrementing is known as change and use method.

Similarly, the statement `cout<<a--;` will display 5, but the value of `a` will be 4. It is equivalent to the statement sequence `cout<<a;  a=a-1;`. But the statement `cout<<--a;` is equivalent to `a=a-1;  cout<<a;`.

The input, output and assignment operators (>>, << and =) may appear more than once in the respective statements. It is known as cascading. The following are examples in each category for cascading of input, output and assignment operators respectively.

```
cin >> a >> b >> c;
cout << "Sum of " << n << "numbers = " << sum;
a = b = c;
```

1.1.2 Structure of a C++ program

Program 1.1 shows the basic structure of a C++ program. It accepts the length and breadth of a rectangle and computes its area and perimeter.

Program 1.1: To find the area and perimeter of a rectangle

```
#include <iostream>
using namespace std;
int main()
{
    float length, breadth, peri, area;
    cout << "Enter the length and breadth of rectangle: ";
    cin >> length >> breadth;
    peri = 2*(length + breadth);
    area = length * breadth;
    cout << "Perimeter = " << peri << endl;
    cout << "Area = " << area << endl;
    return 0;
}
```

Program 1.1 uses the header file `iostream`, since the identifiers `cin` and cout are used. The second line is also essential to use `cin` and `cout` independently. The `using namespace` statement uses `std` to make `cin` and `cout` available in `main()`. In GCC, the function name `main()` is preceded by the data type `int`. Variables are declared using `float` data type. Cascading of input operator and output operators are utilised. Formulae are used in the assignment statements to solve the problem. The `endl` is used instead of `'\n'` to print a new line after each of the results. Figure 1.1(a) shows the screenshot of Program 1.1 in Geany IDE and Figure 1.1(b) shows the output on execution.

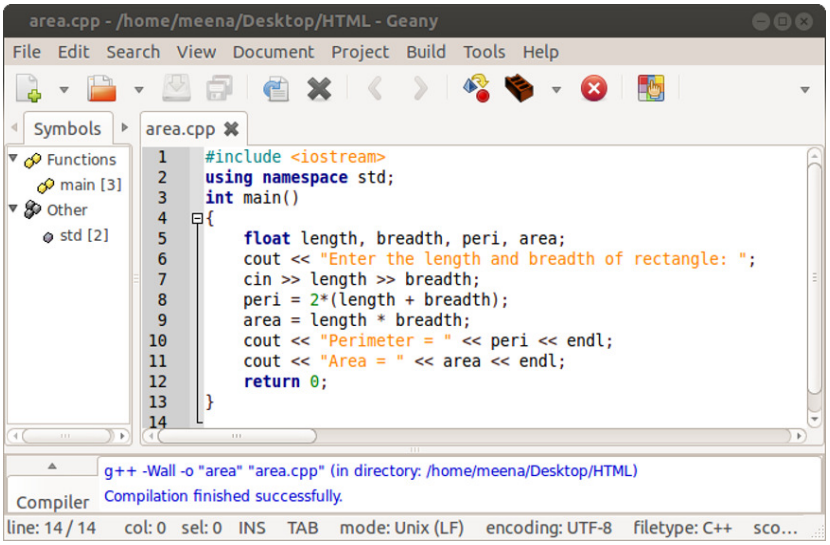


Fig. 1.1(a): Program 1.1.in Geany IDE

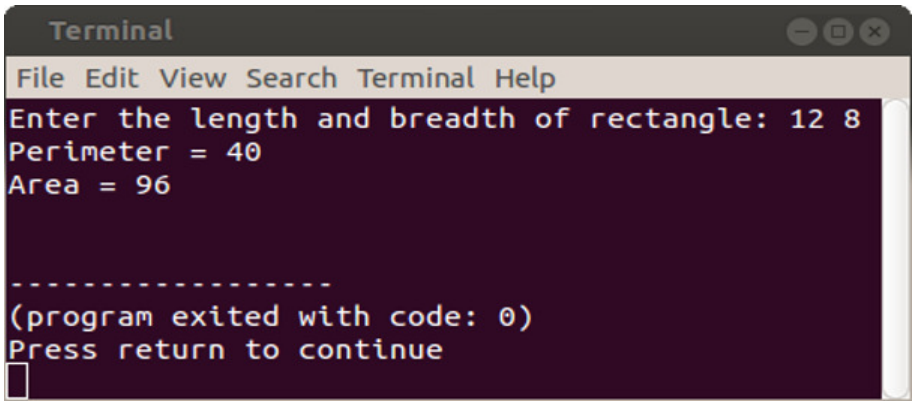


Fig. 1.1(b): Output of Program 1.1 in the terminal window of Geany IDE

This program follows a sequential structure. That is, all statements in the program are executed in a sequential fashion.

While writing a C++ program, we use the statement "**`using namespace std;`**". Why?

A program cannot have the same name for more than one identifier (variables or functions) in the same scope. In our home two or more persons (or even living beings) will not have the same name. If there are, it will surely make conflicts in the identity within the home. So, within the scope of our home, a name should be unique. But our neighbouring home may have a person (or any living being) with the same name as that of one of us. It will not make any confusion of identity within the respective scopes. But an outsider cannot access a particular person by simply using the name; but the house name is also to be mentioned.

The concept of namespace is similar to a house name. Different identifiers are associated to a particular namespace. It is actually a group name in which each item is unique in its name. User is allowed to create own namespaces for variables and functions. The keyword `using` technically tells the compiler about a namespace where it should search for the elements used in the program. In C++, `std` is an abbreviation of 'standard' and it is the standard namespace in which `cout`, `cin` and a lot of other things are defined. So, when we want to use them in a program, we need to follow the format `std::cout` and `std::cin`. This kind of explicit referencing can be avoided with the statement `using namespace std;` in the program. In such a case, the compiler searches this namespace for the elements `cin`, `cout`, `endl`, etc. So whenever the computer comes across `cin`, `cout`, `endl` or anything of that matter in the program, it will read it as `std::cout`, `std::cin` or `std::endl`.

The statement `using namespace std;` doesn't really add a function, it is the include `<iostream>` that "loads" `cin`, `cout`, `endl` and all the like.

Know your progress

1. What is meant by token in C++?
2. Read the following tokens and identify the type of token to which each of these belongs:

i. number	ii. 23.98	iii. "\0"	iv. cin	v. ++
vi. void	vii. '\\'	viii. ;	ix. =	x. a

3. What is the role of #include statement in C++ program?
4. What is wrong in the statement: cin>>25; ?
5. List the type modifiers of C++.

1.2 Control statements

The sequential flow of execution in a program may need to be altered while solving problems. It may be in the form of selection, skipping or repeated execution of one or more statements. Usually this decision will be based on some condition(s). C++ provides statements to facilitate this requirement with the help of control transfer statements. These are classified into two: (i) decision making/selection statements and (ii) iteration statements. Let us see how these statements help problem solving.

1.2.1 Selection statements

C++ provides two statements to select a task from the alternatives based on a condition. They are **`if`** statement and **`switch`** statements. `if` statement has different versions: simple if, `if - else` and `else if` ladder. The following program illustrates the mode of execution of these statements. This program accepts the CE scores of Computer Applications in three terms and finds the highest as the final CE score.

Program 1.2: To find the best CE score from the three given scores

```
#include <iostream>
using namespace std;
int main()
{
    short int ce1, ce2, ce3, final_ce;
    cout<<"Enter three CE scores: ";
    cin>>ce1>>ce2>>ce3;
    if (ce1>ce2)
      final_ce=ce1;//Will be executed, if the condition is true
    else
      final_ce=ce2;//Will be executed, if the condition is false
```

```
    if (ce3>final_ce) final_ce=c3;//No else block for this if
    cout<<"Final CE Score is "<<final_ce;
    return 0;
}
```

Program 1.2 uses `short int` data type for the variables. In GCC, `int` takes 4 bytes of memory, whereas `short` takes only 2 bytes. In this problem we need only the values within the range of `short`. The program also uses `if-else` statement and a simple `if` statement. Initially, the test expression `ce1>ce2` will be evaluated. If it evaluates to true, the value of `ce1` will be assigned to `final_ce`, otherwise that of `ce2` will be assigned. After that, the third score stored in `ce3` is compared with the content of `final_ce`. If the score in `ce3` is higher, it will be stored in `final_ce`, otherwise `final_ce` will not be changed.

We learned that `else if` ladder is a multi-branching statement. Program 1.3 illustrates the working of `else if` ladder. It accepts a character and prints whether it is an alphabet, digit or any other character.

Program 1.3: To check whether the character is uppercase letter, lowercase letter, digit or other characters

```
#include <iostream>
using namespace std;
int main()
{
    char ch;
    cout<<"Enter a character: ";
    cin>>ch;
    if (ch>='A' && ch<='Z')
        cout<<"Uppercase letter";
            else if (ch>='a' && ch<='z')
                cout<<"Lowercase letter";
                    else if (ch>='0' && ch<='9')
                        cout<<"Digit";
                            else
                                cout<<"Other character";
    return 0;
}
```

Program 1.3 uses `else if` ladder or `else if` staircase to select a statement from four different alternatives. Three conditions are provided for selecting one among the three actions. The fourth alternative will be selected when all the three conditions are evaluated to false.

In some cases, where integer equality conditions are used for decision making, `switch` statement can replace `else if` ladder. Since `char` type data are treated as numeric, they are also used in equality checking. Program 1.4 illustrates this concept. This program accepts any one of the four letters a, b, c and d. If 'a' is the input, the word 'Abacus' will be displayed. Similarly, 'Binary' for 'b', 'Computer' for 'c' and 'Debugging' for 'd' will be displayed. For the given problem, actually the `default` statement is not required. In that case, there will not be any response from the program for an input other than the four specified characters. So, to make the program user-friendly, `default` case is mentioned in this program.

Program 1.4: To display a word for a given character

```
#include <iostream>
using namespace std;
int main()
{
    char ch;
    cout<<"Enter a, b, c or d: ";
    cin>>ch;
    switch(ch)
    {
      case 'a': cout<<"Abacus";
                break;
      case 'b': cout<<"Binary";
                break;
      case 'c': cout<<"Computer";
                break;
      case 'd': cout<<"Debugging";
                break;
      default : cout<<"Invalid input!!";
    }
    return 0;
}
```

Program 1.4 also uses the concept of multi branching. Different cases are given, out of which only one will be executed. Selection will be based on the match between the value of the expression provided with switch and the constant attached with any one case. If none of the constants is matched with the value of `ch`, the `default` case will be executed.

Let us do

Replace the switch statement used in Program 1.4 with else if ladder.

In Program 1.4, if break; statements are removed what will be the output?

The else if ladder used in Program 1.3 cannot be replaced with switch. Why?

Conditional operator (? :)

It is a ternary operator of C++ and it requires three operands. It can substitute if - else statement. The `if – else` statement used in Program 1.2 can be replaced by the following:

```
final_ce = (ce1>ce2) ? ce1 : ce2;
```

If we have to find the largest among three scores, nesting of conditional operator can be used as follows:

```
final_ce = (ce1>ce2) ? ((ce1>ce3)?ce1:ce3) :
                                    ((ce2>ce3)?ce2:ce3);
```

1.2.2 Looping statements

C++ provides three looping statements: **`while`**, **`for`** and **`do-while`**. A looping statement has four components: initialisation expression, test expression, update expression and loop-body. The loop-body is the set of statements for repeated execution. The execution is continued as long as the test expression (condition) is true. The variable used in the test expression, called loop control variable, gets its initial value through the initialisation expression (or statement). Update expression changes the value of the loop control variable. Usually, it takes place after each execution of the loop-body.

Looping statements, also called iteration statements, are classified into two: entry-controlled and exit-controlled. In entry-controlled loops, test expression is evaluated before the execution of the loop-body. Program control enters the loop-body only if the condition is true. `while` and `for` are examples of entry-controlled loops. But in exit-controlled loop, condition is checked only after executing the loop-body. So it is certain that the loop-body will be executed at least once in the case of exit-controlled loop. `do-while` statement belongs to this category.

All the three expressions (initialization, test and update) are placed together in for loop. But in the case of `while` and `do-while`, initialisation expression has to be given before the loop and update expression within the loop-body. Test expression appear along with the word `while`. Let us see some programs to understand the working of these loops.

Program 1.5: To find the sum of digits of a number

```
#include <iostream>
using namespace std;
int main()
{
    int num, sum=0, dig;
    cout<<"Enter a number: ";
    cin>>num;
    while (num>0)
    {
        dig=num%10;
        sum=sum+dig;
        num=num/10;
    }
    cout<<"Sum of the digits of the input number = "<<sum;
    return 0;
}
```

In Program 1.5, `num` is the loop control variable as it is involved in the test expression. The initialisation of this variable is through an input statement. Note that the updation expression is within the loop-body. If the input is not a positive number, the body will never be executed.

Let us see another program that uses `for` statement to constitute a loop.

Program 1.6: To find the sum of the first N natural numbers

```
#include <iostream>
using namespace std;
int main()
{
    int n, sum=0;
    cout<<"Enter the limit: ";
    cin>>n;
    for(int i=1; i<=n; i++)
        sum=sum+i;
    cout<<"Sum of the first "<<n<<" natural numbers = "<<sum;
    return 0;
}
```

In Program 1.6, the variable `i` is the loop control variable and is initialised with an assignment statement. The updation is done with increment operation. The program needs a loop based on counting. The for statement is more appropriate in such situations.

As mentioned earlier, `do-while` loop ensures the execution of loop-body at least once. Its application is illustrated in the following program.

Program 1.7: To find the average height of a group of students

```
#include <iostream>
using namespace std;
int main()
{
    float hgt, sum=0, avg_hgt;
    short n=0;
    char ch;
    do
    {
        cout<<"Enter the height: ";
        cin>>hgt;
        n++;
        sum=sum+hgt;
        cout<<"Any more student (Y/N)? ";
        cin>>ch;
    }while (ch=='Y' || ch=='y');
    avg_hgt=sum/n;
    cout<<"Average Height = "<<avg_hgt;
    return 0;
}
```

The execution of the loop-body in Program 1.7 is repeated as long as user responds by inputting 'Y' or 'y'.

Let us do

Rewrite the looping segment of Program 1.5 using `for` and `do-while` statements.

Rewrite the looping segment of Program 1.6 using `while` and `do-while` statements.

Rewrite the looping segment of Program 1.7 using `for` and `while` statements.

Know your progress

1. What are the selection statements of C++?
2. What are the four components of a loop?
3. Give examples for entry-controlled loop.
4. Which control transfer statement is equivalent to the conditional operator (`?:`)?
5. All `switch` statements can be replaced by any form of `if` statement. State whether it is true or false.

1.3 Nested Loops

Placing a loop inside the body of another loop is called nesting of a loop. In a nested loop, the inner loop statement will be executed repeatedly as long as the condition of the outer loop is true. Here the loop control variables for the two loops should be different.

Let us observe how a nested loop works. Take the case of a minute-hand and second-hand of a clock. Have you noticed the working of a clock? While the minute-hand stands still at a position, the second-hand moves to complete one full rotation (say 1 to 60). The minute hand moves to the next position (that is, the next minute) only after the second hand completes one full rotation. Then the second-hand again completes another full rotation corresponding to the minute-hand's current position. For each position of the minute-hand, the second-hand completes one full rotation and the process goes on. Here the second hand movement can be treated as the execution of the inner loop and the minute-hand's movement can be treated as the execution of the outer loop. Figure 1.2 shows the concept of nested loops with the help of digital watch.

Fig. 1.2: Concept of nested loop using digital watch

As in Figure 1.2, the value of second changes from 0 to 59 keeping the minute fixed. Once the value of seconds reached 59, the next change will be in minutes. Once the minute is changed, the second resets its value to 0.

All types of loops in C++ allow nesting. An example is given to show the working procedure of a nested for loop.

```
for( i=1; i<=2; ++i)
{
    for(j=1; j<=3; ++j)
    {
        cout<< "\n" << i << " and " << j;
    }
}
```

Outer loop

Inner loop

Initially value 1 is assigned to the outer loop variable `i`. Its test expression is evaluated to be True and hence the body of the loop is executed. The body contains the inner loop with the control variable `j` and it begins to execute by assigning the initial value 1 to `j`. The inner loop is executed 3 times, for `j=1`, `j=2`, `j=3`. Each time it evaluates the test expression `j<=3` and displays the output since it is True.

```
1 and 1
1 and 2
1 and 3
```

The first 1 is of `i` and the second 1 is of `j`

When the test expression `j<=3` is False, the program control comes out of the inner loop. Now the update statement of the outer loop is executed which makes `i=2`. Then the test expression `i<=2` is evaluated to True and once again the loop body (i.e. the inner loop) is executed. Inner loop is again executed 3 times, for `j=1`, `j=2`, `j=3` and displays the output.

```
2 and 1
2 and 2
2 and 3
```

After completing the execution of the inner loop, the control again goes back to the update expression of the outer loop. Value of `i` is incremented by 1 (now `i=3`) and the test expression `i<=2` is now evaluated to be False. Hence the loop terminates its execution. Table 1.2 illustrates the execution of the above given program segment:

Iterations	Outer loop (i)	Inner loop (j)	Output
1	1	1	1 and 1
2	1	2	1 and 2
3	1	3	1 and 3
4	2	1	2 and 1
5	2	2	2 and 2
6	2	3	2 and 3

Table 1.2: Execution of a nested loop

When working with nested loops, the control variable of the outer loop changes its value only after the inner loop is terminated. Let us write a program to display the given triangle using nested loop.

```
*
* *
* * *
* * * *
* * * * *
```

Program 1.8: To display a triangle of stars

```
#include<iostream>
using namespace std;
int main()
{
    short int i, j;
    for(i=1; i<=5; ++i)     //outer loop
    {
        cout<< "\n" ;
        for(j=1; j<=i; ++j) // inner loop
            cout<< '*';
    }
    return 0;
}
```

Let us do

In Program 1.8, what will the output be if we use the statement `cout<<i;` instead of `cout<<'*';`?

Similarly, what will the output be if we use the statement `cout<<j;`?

1.4 Jump statements

The statements that facilitate the transfer of program control from one place to another are called jump statements. C++ provides four jump statements that perform unconditional control transfer in a program. They are `return`, `goto`, `break` and `continue` statements. All of these are keywords. In addition, C++ provides a standard library function `exit()` that helps us to terminate a program.

The `return` statement is used to transfer control back to the calling program or to come out of a function. It will be explained in detail later in Chapter 3. Now, we will discuss the other jump statements.

1.4.1 goto statement

The goto statement can transfer the program control to anywhere in the function. The target destination of a goto statement is marked by a label, which is an identifier. The syntax of goto statement is:

```
        goto  label;
        ............;
        ............;
label:  ............;
        ............;
```

where the `label` can appear in the program either before or after `goto` statement. The label is followed by a colon (:) symbol. For example, consider the following code fragment which prints numbers from 1 to 50.

```
    int i=1;
start:
    cout<<i;
    ++i;
    if (i<=50)
        goto start;
```

Here, the `cout` statement prints the value 1. After that `i` is incremented by 1 (now `i=2`), then the test expression `i<=50` is evaluated. Since it is True the control is transferred to the statement marked with the label `start`. When the test expression evaluates to False, the process terminates and transfers the program control following the `if` statement. It is to be noted that the usage of `goto` is not encouraged in structured programming.

1.4.2 break statement

We have already discussed the effect of `break` in `switch` statement. It transfers the program control outside the `switch` block. When a `break` statement is encountered in a loop (`for`, `while`, `do-while`), it takes the program control outside the immediate enclosing loop. Execution continues from the statement immediately after the control structure. Let us see how it affects the execution of loops. Consider the following two program segments.

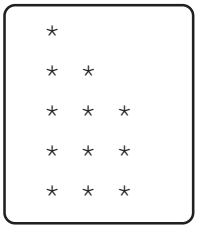
Code segment 1:

```
i=1;
while(i<=10)
{
    cin>>num;
    if (num==0)
        break;
    cout<<"Entered number is: "<<num;
    cout<<"\nInside the loop";
    ++i;
}
cout<<"\nComes out of the loop";
```

The above code fragment allows to input 10 different numbers. During the input if any number happens to be 0, the program control comes out of the loop by skipping the rest of the statements within the loop-body and displays the message "Comes out of the loop" on the screen. Let us consider another code segment that uses break within a nested loop.

Code segment 2:

```
for(i=1; i<=5; ++i)     //outer loop
{
    cout<<"\n";
    for(j=1; j<=i; ++j) //inner loop
    {
        cout<<"* ";
        if (j==3) break;
    }
}
```



This code segment will display the given pattern:

The nested loop executes normally for the value of i=1, i=2, i=3. For each value of i, the variable j takes values from 1 to i. When the value of i becomes 4, the inner loop executes for the value of j=1, j=2, j=3 and comes out from the inner loop on executing the break statement.

1.4.3 continue statement

The statement continue is another jump statement used for skipping over a part of the code within the loop-body and forcing the next iteration. The break statement forces termination of the loop, but the continue statement forces next iteration of the loop. The following program segment explains the working of a continue statement:

```
for (i=1; i<=10; ++i)
{
    if (i==6)
        continue;
    cout<<i<<"\t";
}
```

This code gives the following output:

1 2 3 4 5 7 8 9 10

Note that 6 is not in the list. When the value of i becomes 6 the continue statement is executed. As a result, the output statement is skipped and program control goes to the update expression for next iteration.

A `break` statement inside a loop will abort the loop and transfer control to the statement following the loop. A `continue` statement will just abandon the current iteration and let the loop continue with next iteration. When a `continue` statement is used within `while` and `do-while` loops, care should be taken to avoid infinite execution.

Let us do

Let us compare `break` and `continue` statements. Table 1.3 is designed to show the comparison aspects. Some of the entries are filled. The remaining entries are left for you to fill with proper comparison points.

continue	break
•	• Used only with loops
• Takes the control outside the loop by skipping the remaining part of the loop	• ....
•	• Program control goes outside only when the test expression of the loop returns false

Table 1.3: Comparison between `break` *and* `continue`

Let us write a program that requires the use of nested loops. Program 1.9 can display all prime numbers below 100.

Program 1.9: To display all prime numbers below 100

```
#include<iostream>
using namespace std;
int main()
{   short int n, i, flag;
    cout<<"Prime numbers below 100 are...\n";
    for(n=2; n<=100; n++)  //Outer loop
    {   flag=1;
        for(i=2; i<=n/2; i++)     //Inner loop
            if(n%i==0)
            {   flag=0;
                break;//Takes the control outside the iiner loop
            }
        if(flag==1)  cout<<n<<'\t';
    }
    return 0;
}
```

In Program 1.8, the variable `n` is given the values from 2 to 100 through the outer loop. Each value is checked for prime using the inner loop. If any of the values from 2 to `n/2` is found to be a factor of `n`, inner loop is terminated by changing the value of `flag` from 1 to 0. The value of `n`, for which `flag` remains 1 after the termination of the inner loop, is prime and is printed.

Let us conclude

We have refreshed ourselves with the basic concepts of C++ language. Character set, tokens, data types, type modifiers, expressions and type conversions are presented in capsule form. Different types of C++ statements are recollected with the help of examples. Various control transfer statements are briefly explained with programs. Nested loops and the two jump statements `break` and `continue` are introduced as the new concepts. A clear-cut idea about these topics is very much essential to learn the concepts covered in Chapters 2 and 3 of this book.

Let us practice

1. Write a C++ program to display all palindrome numbers between 100 and 200.
2. Write a C++ program to display all Armstrong numbers below 1000.
3. Write a C++ program to display all perfect numbers below 1000.
4. Write a C++ program to display the multiplication table of a number.
5. Write a C++ program to prepare electricity bill for a group of consumers. The previous meter reading and current reading are the inputs. The payable amount is to be calculated using the following criteria:

   Up to 300 units : Rs. 5.00/- per unit
   Up to 350 units : Rs. 5.70/- per unit
   Up to 400 units : Rs. 6.10/- per unit
   Up to 500 units : Rs. 6.70/- per unit
   Above 500 units : Rs. 7.50/- per unit

   The program should provide facility to input the details of any number of consumers as the user wants.

Let us assess

1. What is wrong with the following code segment?

```
for (short i=1; i<5; ++i)
    for (i=5; i>0; --i)
        cout<<i<<"\t";
```

2. Distinguish between `break` and `continue` statements.

3. The `default` case in `switch` is equivalent to the `else` block of `else if` ladder. Justify this statement with the help of example.
4. What will be the output of the following code fragment?

```
for (outer=10; outer>5; --outer)
    for (inner=1; inner<4; ++inner)
        cout<<outer<<"\t"<<inner<<endl;
```

5. Write a program to produce the following output using nested loop:

```
A
A    B
A    B    C
A    B    C    D
A    B    C    D    E
```

6. Read the following C++ code snippet:

```
for (n=1; n<5; ++n)
{
    cout<<i;
    if (i==2) continue;
    if (i%3==0) break;
    cout<<"Hello";
}
```

Choose the correct output of this code from the following:

a. 1Hello2Hello3Hello4Hello b. 1Hello2Hello3

c. 1Hello23 d. 1Hello23Hello

7. Read the following C++ code segment and replace it with a looping statement:

```
        cin>>n;
loop:   r=n%10;
        s=s*10+r;
        n=n/10;
        if (n!=0) goto loop;
        cout<<s;
```

8. Which of the following is not a character constant in C++?

a. `'\t'` b. `'a'` c. `'9'` d. `'9a'`

9. Some of the following identifiers are invalid. Identify them and give reason for the invalidity.

a. `unsigned` b. `cpp` c. `2num` d. `cout`

10. What happens if `break` is not used in `switch` statement?

2 Arrays

Significant Learning Outcomes

After the completion of this chapter, the learner

- identifies the scenarios where an array can be used.
- uses arrays to refer to a group of data.
- familiarises with the memory allocation for arrays.
- accesses any element in an array while solving problems.
- solves problems in which large amount of data is to be processed.
- represnts strings using character arrays.
- carries out various word processing operations using character arrays.

We use variables to store data in programs. But if the quantity of data is large, more variables are to be used. This will cause difficulty in accessing the required data. We have revised the concept of C++ data types in Chapter 1 and we used basic data types to declare variables and perform type conversion. In this chapter, a derived data type in C++, named 'array' is introduced. The word 'array' is not a data type name, rather it is a kind of data type derived from fundamental data types to handle large number of data easily. We will discuss the creation and initialisation of arrays, and operations like traversal.

2.1 Array and its need

An **array** is a collection of elements of the same type placed in contiguous memory locations. Arrays are used to store a set of values of the same type under a single variable name. Each element in an array can be accessed using its position in the list, called index number or subscript.

Why do we need arrays? We will illustrate this with the help of an example. Let us consider a situation where we need to store the scores of 20 students in a class and has to find their class

average. If we try to solve this problem by making use of variables, we will need 20 variables to store students' scores. Remembering and managing these 20 variables is not an easy task and the program may become complex and difficult to understand.

```
int a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t;
float avg;
cin>>a>>b>>c>>d>>e>>f>>g>>h>>i>>j>>k>>l>>m>>n>>o>>p>>q>>r>>s>>t;
avg = (a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t)/20.0;
```

As it is, this code is fine. However, if we want to modify it to deal with the scores of a large number of students, say 1000, we have a very long and repetitive task at hand. We have to find a way to reduce the complexity of this task.

The concept of array comes as a boon in such situations. As it is a collection of elements, memory locations are to be allocated. We know that a declaration statement is needed for memory allocation. Let us see how arrays are declared and used.

2.1.1 Declaring arrays

Just like the ordinary variable, the array is to be declared properly before it is used. The syntax for declaring an array in C++ is as follows.

```
data_type array_name[size];
```

In the syntax, `data_type` is the type of data that the array variable can store, `array_name` is an identifier for naming the array and the `size` is a positive integer number that specifies the number of elements in the array. The following is an example:

```
int num[10];
```

The above statement declares an array named `num` that can store 10 integer numbers. Each item in an array is called an `element` of the array. The elements in the array are stored sequentially as shown in Figure 2.1. The first element is stored in the first location; the second element is stored in the second location and so on.

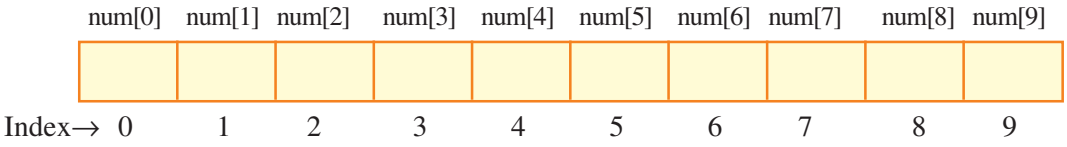


Fig. 2.1: Arrangement of elements in an array

Since the elements in the array are stored sequentially, any element can be accessed by giving the array's name and the element's position. This position is called the **index** or **subscript** value. In C++, the array index starts with zero. If an array is

declared as `int num[10];` then the possible index values are from 0 to 9. In this array, the first element can be referenced as `num[0]` and the last element as `num[9]`. The subscripted variable, `num[0]`, is read as "num of zero" or "num zero". It's a shortened way of saying "the num array subscripted by zero". So, the problem of referring the scores of 1000 students can be resolved by the following statement:

```
int score[1000];
```

The array, named `score`, can store the scores of 1000 students. The score of the first student is referenced by `score[0]` and that of the last by `score[999]`.

2.1.2 Memory allocation for arrays

The amount of storage required to hold an array is directly related to its type and size. Figure 2.2 shows the memory allocation for the first five elements of array **num**, assuming 1000 as the address of the first element. Since `num` is an integer type array, size of each element is 4 bytes (in a system with 32 bit integer representation using GCC) and it will be represented in memory as shown in Figure 2.2.

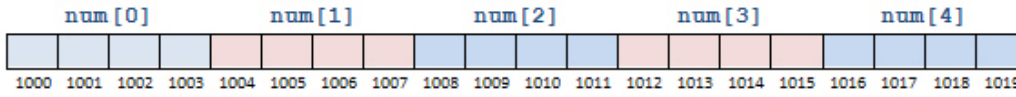


Fig. 2.2: Memory allocation for an integer array

The memory space allocated for an array can be computed using the following formula:

total_bytes = `sizeof`(array_type) × size_of_array

For example, total bytes allocated for the array declared as `float a[10];` will be 4 × 10 = 40 bytes.

2.1.3 Array initialisation

Array elements can be initialised in their declaration statements in the same manner as in the case of variables, except that the values must be included in braces, as shown in the following examples:

```
int score[5] = {98, 87, 92, 79, 85};
char code[6] = {'s', 'a', 'm', 'p', 'l', 'e'};
float wgpa[7] = {9.60, 6.43, 8.50, 8.65, 5.89, 7.56, 8.22};
```

Initial values are stored in the order they are written, with the first value used to initialize element 0, the second value used to initialize element 1, and so on. In the first example, `score[0]` is initialized to 98, `score[1]` is initialized to 87, `score[2]` is initialized to 92, `score[3]` is initialized to 79, and `score[4]` is initialized to 85.

If the number of initial values is less than the size of the array, they will be stored in the elements starting from the first position and the remaining positions will be initialized with zero, in the case of numeric data types. For `char` type array, such positions will be initialised with `' '` (space bar) character. When an array is initialized with values, the size can be omitted. For example, the following declaration statement will reserve memory for five elements:

```
int num[] = {16, 12, 10, 14, 11};
```

2. 1.4 Accessing elements of arrays

Array elements can be used anywhere in a program as we do in the case of normal variables. We have seen that array is accessed element-wise. That is, only one element can be accessed at a time. The element is specified by the array name with the subscript. The following are some examples of using the elements of the array named score:

```
score[0] = 95;
score[1] = score[0] - 11;
cin >> score[2];
score[3] = 79;
cout << score[2];
sum = score[0] + score[1] + score[2] + score[3] + score[4];
```

The subscript in brackets can be a variable, a constant or an expression that evaluates to an integer. In each case, the value of the expression must be within the valid subscript range of the array. An important advantage of using variable and integer expressions as subscripts is that, it allows sequencing through an array by using a loop. This makes statements more structured keeping away from the inappropriate usage as follows:

```
sum = score[0] + score[1] + score[2] + score[3] + score[4];
```

The subscript values in the above statement can be replaced by the control variable of `for` loop to access each element in the array sequentially. The following code segment illustrates this concept:

```
sum = 0;
for (i=0; i<5; i++)
     sum = sum + score[i];
```

An array element can be assigned a value interactively by using an input statement, as shown below:

```
for(int i=0; i<5; i++)
    cin>>score[i];
```

When this loop is executed, the first value read is stored in the array element `score[0]`, the second in `score[1]` and the last in `score[4]`.

Program 2.1 shows how to read 5 numbers and display them in the reverse order. The program includes two `for` loops. The first one allows the user to input array values. After five values have been entered, the second `for` loop is used to display the stored values from the last to the first.

Program 2.1: To input the scores of 5 students and display them in reverse order

```
#include <iostream>
using namespace std;
int main()
{
    int i, score[5];
    for(i=0; i<5; i++)   //  Reads the scores
    {
        cout<<"Enter a score: ";
        cin>>score[i];
    }
    for(i=4; i>=0; i--)   //  Prints the scores
        cout<<"score[" << i << "] is " << score[i]<<endl;
    return 0;
}
```

The following is a sample output of program 2.1:

```
Enter a score: 55
Enter a score: 80
Enter a score: 78
Enter a score: 75
Enter a score: 92
score[4] is 92
score[3] is 75
score[2] is 78
score[1] is 80
score[0] is 55
```

Accessing each element of an array at least once to perform any operation is known as ***traversal*** operation. Displaying all the elements of an array is an example of traversal. If any operation is performed on all the elements in an array, it is a case of traversal. Program 2.2 shows how traversal is performed in an array.

Program 2.2: Traversal of an array

```
#include <iostream>
using namespace std;
int main()
{
    int a[5], i;
    cout<<"Enter the elements of the array :";
    for(i=0; i<5; i++)
        cin >> a[i];  //Reading the elements
    for(i=0; i<5; i++)
        a[i] = a[i] + 1; // A case of traversal
    cout<<"\nNow value of elements in the array are...\n";
    for(i=0; i<5; i++)
        cout<< a[i]<< "\t";  // Another case of traversal
    return 0;
}
```

The following is a sample output of program 2.2:

```
Enter the elements of the array : 12  3   6   1   8
Now value of elements in the are...
13  4  7  2  9
```

Let us do

1. Write array declarations for the following:
   a. Scores of 100 students
   b. English letters
   c. A list of 10 years
   d. A list of 30 real numbers
2. Write array initialization statements for the following:
   a. An array of 10 scores: 89, 75, 82, 93, 78, 95, 81, 88, 77, and 82
   b. A list of five amounts: 10.62, 13.98, 18.45, 12.68, and 14.76
   c. A list of 100 interest rates, with the first six rates being 6.29, 6.95, 7.25, 7.35, 7.40 and 7.42.
   d. An array of 10 marks with value 0.
   e. An array with the letters of VIBGYOR.
   f. An array with number of days in each month.
3. Write a C++ code to input values into the array: `int ar[50];`
4. Write a C++ code fragment to display the elements in the even positions of the array: `float val[100];`

Let us solve another problem that requires traversal operation. Program 2.3 accepts five numbers from a user and finds the sum of these numbers.

Program 2.3: To find the sum of the elements of an array

```
#include <iostream>
using namespace std;
int main()
{
    int a[5], i, sum;
    cout<<"Enter the elements of the array :";
    for(i=0; i<5; i++)
        cin >> a[i]; //Reading the elements
    sum = 0;
    for(i=0; i<5; i++)
        sum = sum + a[i]; // A case of traversal
    cout<<"\nSum of the elements of the array is "<< sum;
    return 0;
}
```

The following is a sample output of Program 2.3:

```
Enter the elements of the array : 12  3   6   1   8
Sum of the elements of the array is 30
```

Program 2.4 illustrates another case of traversal to find the largest element in an array. In this program a variable `big` is used to hold the largest value. Initially it is assigned with the value in the first location. Then it is compared with the remaining elements. Whenever a larger value is found, it repalces the value of `big`.

Program 2.4: To find the largest element in an array

```
#include <iostream>
using namespace std;
int main()
{
    int a[5], i, big;
    cout<<"Enter the elements of the array :";
    for(i=0; i<5; i++)
        cin >> a[i];
    big = a[0];
    for(i=1; i<5; i++)
        if (a[i] > big)  // A case of traversal
            big = a[i];
```

```
    cout<<"\nThe biggest element is " << big;
    return 0;
}
```

The following is a sample output of program 2.4:

```
Enter the elements of the array : 12  3    6    1    8
The biggest element is 12
```

2.2 String handling using arrays

We know that string is a kind of literal in C++ language. It appears in programs as a sequence of characters within a pair of double quotes. Imagine that you are asked to write a program to store your name and display it. We have learned that variables are required to store data. Let us take the identifier `my_name` as the variable. Remember that in C++, a variable is to be declared before it is used. A declaration statement is required for this and it begins with a data type. Which data type should be used to declare a variable to hold string data? There is no basic data type to represent string data. We may think of **`char`** data type. But note that the variable declared using `char` can hold only one character. Here we have to input string which is a sequence of characters.

Let us consider a name "Niketh". It is a string consisting of six characters. So it cannot be stored in a variable of `char` type. But we know that an array of `char` type can hold more than one character. So, we declare an array as follows:

```
char my_name[10];
```

It is sure that ten contiguous locations, each with one byte size, will be allocated for the array named `my_name`. If we follow the usual array initialization method, we can store the characters in the string "Niketh" as follows:

```
char my_name[10]={'N', 'i', 'k', 'e', 't', 'h'};
```

Figure 2.3 shows the memory allocation for the above declared character array. Note that, we store the letters in the string separated by commas. If we want to input the same data, the following C++ statement can be used:

```
for (int i=0; i<6; i++)
    cin >> my_name[i];
```

During the execution of this statement, we have to input six letters of "Niketh" one after the other separated by **Space bar**, **Tab** or **Enter** key. The memory allocation in both of these cases will be as shown in Figure 2.3.

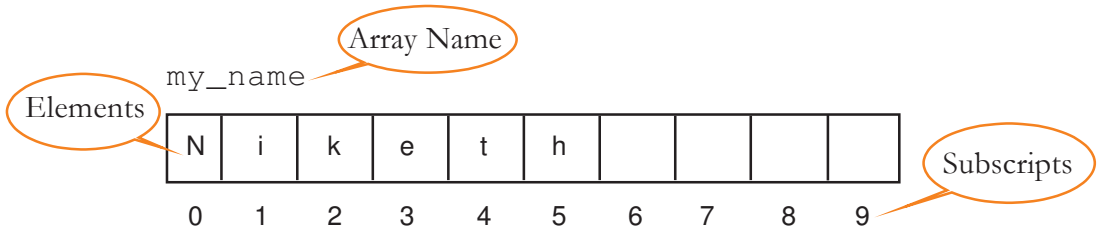


Fig. 2.3 : Memory allocation for the character array

So, let us conclude that a character array can be used to store a string, since it is a sequence of characters. However, it is true that we do not get the feel of inputting a string. Instead, we input the characters constituting the string one by one.

In C++, character arrays have some privileges over other arrays. Once we declare a character array, the array name can be considered as an ordinary variable that can hold string data. Let's say that a character array name is equivalent to a string variable. Thus your name can be stored in the variable `my_name` (the array name) using the following statement:

```
cin >> my_name;
```

It is important to note that this kind of usage is wrong in the case of arrays of other data types. Now let us complete the program. It will be like the one given in Program 2.5.

Program 2.5 To input a string and display

```
#include <iostream>
using namespace std;
int main()
{
     char my_name[10];
     cout << "Enter your name: ";
     cin >> my_name;
     cout << "Hello " << my_name;
     return 0;
}
```

On executing this program we will get the output as shown below.

```
Enter your name: Niketh
Hello Niketh
```

Note that the string constant is not `"Hello"`, but `"Hello "` (a white space is given after the letter `'o'`).

Let us do

Run Program 2.5 and input your full name by expanding the initials if any, and check whether the output is correct or not. If your name contains more than 10 characters, increase the size of the array as needed.

2.3 Memory allocation for strings

We have seen how memory is allocated for an array of characters. As Figure 2.3 shows, the memory required depends upon the number of characters stored. But if we input a string in a character array, the scene will be different. If we run Program 2.5 and input the string `Niketh`, the memory allocation will be as shown in Figure 2.4.

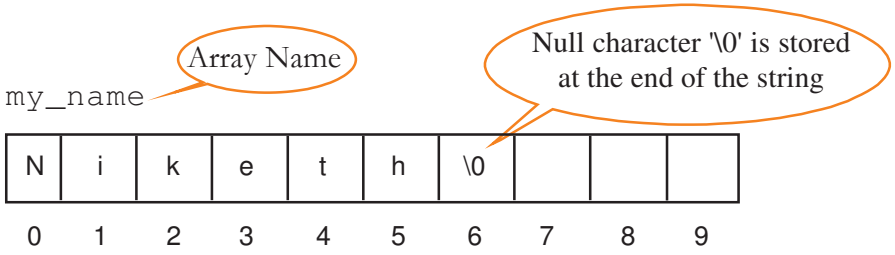


Fig. 2.4 : Memory allocation for the character array

Note that a null character `'\0'` is stored at the end of the string. This character is used as the string terminator and added at the end automatically. Thus we can say that memory required to store a string will be equal to the number of characters in the string plus one byte for null character. In the above case, the memory used to store the string `Niketh` is seven bytes, but the number of characters in the string is only six.

As in the case of variable initialization, we can initialize a character array with a string as follows:

```
char my_name[10] = "Niketh";
char str[] = "Hello World";
```

In the first statement 10 memory locations will be allocated and the string will be stored with null character as the delimiter. The last three bytes will be left unused. But, for the second statement, size of the array is not specified and hence only 12 bytes will be allocated (11 bytes for the string and 1 for `'\0'`).

2.4 Input/Output operations on strings

Program 2.5 contains input and output statements for string data. Let us modify the declaration statement by changing the size of the array to 20. If we run the program by entering the name `Maya Mohan`, the output will be as follows:

```
Enter your name: Maya Mohan
Hello Maya
```

Note that though there is enough size for the array, we get only the word "Maya" as the output. Why does this happen?

Let us have a close look at the input statement: `cin>>my_name;`. We have experienced that only one data item can be input using this statement. A white space is treated as a separator of data. Thus, the input `Maya Mohan` is treated as two data items, `Maya` and `Mohan` separated by white space. Since there is only one input operator (**>>**) followed by a variable, the first data (i.e., `Maya`) is stored. The white space after "Maya" is treated as the delimiter.

So, the problem is that we are unable to input strings containing white spaces. C++ language gives a solution to this problem by a function, named **`gets()`**. The function `gets()` is a console input function used to accept a string of characters including white spaces from the standard input device (keyboard) and store it in a character array.

The string variable (character array name) should be provided to this function as shown below:

```
gets(character_array_name);
```

When we use this function, we have to include the library file `cstdio.h` in the program. Let us modify Program 2.5, by including the statement `#include<cstdio>`, and replacing the statement `cin>>my_name;` by `gets(my_name);` After executing the modified program, the output will be as follows:

```
Enter your name: Maya Mohan
Hello Maya Mohan
```

The output shows the entire string that we input. See the difference between `gets()` and `cin`.

Though we do not use the concept of subscripted variable for the input and output of strings, any element in the array can be accessed by specifying its subscript along with the array name. We can access the first character of the string by `my_name[0]`, fifth character by `my_name[4]` and so on. We can even access the null character (`'\0'`) by its subscript. Program 2.6 illustrates this idea.

Program 2.6: To input a string and count the vowels in a string

```
#include <iostream>
#include <cstdio>    //To use gets() function
using namespace std;
int main()
{
    char str[20];
```

```
    int vow=0;
    cout<<"Enter a string: ";
    gets(str);
    for(int i=0; str[i]!='\0'; i++)
        switch(str[i])
        {   case 'a':
            case 'e':
            case 'i':
            case 'o':
            case 'u': vow++;
        }
    cout<<"No. of vowels in the string "<<str<<" is "<<vow;
    return 0;
}
```

If we run Program 2.6 by inputting the string "hello guys", the following output can be seen:

```
Enter a string: hello guys
No. of vowels in the string hello guys is 3
```

Now, let us analyse the program and see how it works to give this output.

- In the beginning, the `gets()` function is used and so we can input the string `"hello guys"`.
- The body of the `for` loop will be executed as long as the element in the array, referenced by the subscript `i`, is not the null character (`'\0'`). That is, the body of the loop will be executed till the null character is referenced.
- The body of the loop contains only a `switch` statement. Note that, no statements are given against the first four cases of the `switch`. In the last case, the variable `vow` is incremented by 1. You may think that this is required for all the cases. Yes, you are right. But, you should use the `break` statement for each case to exit the `switch` after a match. In this program the action for all the cases are the same and that is why we use this style of code.
- While the `for` loop iterates, the characters will be retrieved one by one for matching against the constants attached to the cases. Whenever a match is found, the variable `vow` is incremented by 1.
- As per the input string, matches occur when the value of `i` becomes 1, 4 and 7. Thus, the variable `vow` is incremented by 1 three times and we get the correct output.

We have seen how `gets()` function facilitates input of strings. Just like the other side of a coin, C++ gives a console function named **`puts()`** to output string data.

The function `puts()` is a console output function used to display a string data on the standard output device (monitor). Its syntax is:

```
puts(string_data);
```

The string constant or variable (character array name) to be displayed should be provided to this function. Observe the following C++ code fragment:

```
char str[10] = "friends";
puts("hello");
puts(str);
```

The output of the above code will be as follows:

```
hello
friends
```

Note that the string `"friends"` in the character array `str[10]` is displayed in a new line. Try this code using `cout<<"hello";` and `cout<<str;` instead of the `puts()` functions and see the difference. The output will be in the same line without a space in between them in the case of `cout` statement.

Let us do

Predict the output, if the input is "HELLO GUYS" in Program 2.6. Execute the program with this input and check whether you get the correct output. Find out the reason for difference in output. Modify the program to get the correct output for any given string.

Let us conclude

We have discussed array as a data type to refer to a group of same type of data. Memory allocation for arrays is explained with the help of schematic diagrams. The use of looping statements, especially `for` loop in manipulating the elements of an array are also illustrated through programs. We have also seen that how arrays help to handle strings effectively in programs.

Let us practice

1. Write a C++ program to input the amount of sales for 12 months into an array named SalesAmt. After all the input, find the total and average amount of sales.
2. Write a C++ program to create an array of N numbers, find the average and display those numbers greater than the average.

3. Write a C++ program to swap the first and the last elements of an integer array.
4. Write a C++ program to input 10 integer numbers into an array and determine the maximum and minimum values among them.
5. Write a C++ program to input a string and find the number of uppercase letters, lowercase letters, digits, special characters and white spaces.
6. Write a C++ program to count the number of words in a sentence.
7. Write a C++ program to find the length of a string.

Let us assess

1. The elements of an array with ten elements are numbered from ____ to ____.
2. An array element is accessed using _____.
3. If `AR` is an array, which element will be referenced using `AR[7]`?
4. Consider the array declaration `int a[3]={2,3,4};` What is the value of `a[1]`?
5. Consider the array declaration `int a[]={1,2,4};` What is the value of `a[1]`?
6. Printing all the elements of an array is an example for _____ operation.
7. Write down the output of the following code segment:

```
puts("hello");
puts("friends");
```

8. Write the initialisation statement to store the string "GCC".
9. Define an Array.
10. What does the declaration `int studlist[1000];` mean?
11. How is memory allocated for a single dimensional array?
12. Write C++ statements to accept an array of 10 elements and display the count of even and odd numbers in it.
13. Read the following statements:

```
char name[20];
cin>>name;
cout<<name;
```

What will be the output if you input the string "Sachin Tendulkar"? Justify your answer.

14. Write C++ statements to accept two single dimensional arrays of equal length and find the difference between corresponding elements.
15. Write a program to check whether a string is a palindrome or not.