Paper :	**Programming Fundamentals**
Program:	**BS (SE)**
Teacher Name:	**Dr. Fazal-e-Malik**
Student id no :	**16907**
Student Name:	**Abdul wahab**

Q 1 (a): What is the purpose of if statement? Discuss its two different forms with examples.

If Structures

Statement Type:

All forms of the if statements are executable branching statements.

Statement Purpose:

The various forms of if statements are Fortran's main branching tool. They give Fortran an ability to make decisions in a program. The different forms of if statements that can be used include the simple logical if, the if-then-else structure, and the arithmetic if.

Required Form:

There are a few different structures that must be used depending on what form of the if structure you are using. They are as follows: **The simple logical if:** In a simple logical if, all that is need is some LOGICAL expression enclosed in paraenthesis and some executable statement following the LOGICAL expression. The statement will only be executed if the LOGICAL expression evalutes to .true. An example of this form of if statement is shown below.

```
if (ldata.ge.100) print *,'There are to many data',
           & ' points'
```

If then - else structure:

```
If ( logical expression ) then
```

This is the first statement in an if-then-else structure. It contains the first logical expression to be evaluated. If the logical expression evaluates to .true. then only those statements beginning with the following one and ending before the next ELSE or ENDIF will be evaluated. All other statements in the structure will be ignored and not evaluated by the compiler.

Else if (logical expression) then

This is used to specifiy additional conditions if the preceeding if-then and else-if statements have not been met. It can only be used as a part of an if - then else structure.

Else

This is used to give a default condition that will be executed if none of the preceding conditions in the structure have been met.

End if

This is needed to tell the compiler that the end of the if then else block has been reached. See the examples section for an example of the if - then - else structure. **Arithmatic if** : This is a very old and obsolete structure in Fortran. **Never use this in any new programs you write.** The two requirements for this type of if statement are an arithmatic expression and three line numbers for the purpose of branching. It is best explained in the example section.

Examples:

The **if -then-else** structure was introduced with Fortran 77 and is a very powerful branching tool. It will allow the computer to make decisions based on a variety of different situations. Note however that once the computer evaluates one of the different conditions to be true, it will not evaluate any of the following conditions. An example of this structure is as follows:

```
if ( delta.ge.1.) then
print *, ' Too much deflection'
else if ( shear.ge.21.) then
print *, 'Beam fails in shear'
else if ( sigma.ge.36.) then
print *, 'Beam fails in tension'
else if ( sigma.le.-36.) then
print *, 'Beam fails in compression'
else
print *, 'Beam will not fail under these'
       &' conditions'
end if
```

The **arithmetic if**. It is **strongly recommended to never use this form of an if statement**. It is a carry over from an archaic form of Fortran and it is highly likely that it will be dropped from some future Fortran standard. It is only shown here so you

know what it is if you run into it in when modifying an early Fortran program. The following is an example of this structure.

```
      If (b**2-4*a*c) 100,200,300,
  100 print*, 'Roots are complex'
      Go to 400
       200 print *, ' single real root '
      Go to 400
  300 print *, 'two real roots'
  400 continue
```

In this type of structure the program branches to label 100 if the arithmetic expression in parenthesis evaluates to less than zero, or to 200 if it evaluates to be equal to zero or would branch to 300 if it evaluates to greater than zero.

Q 1 (b): Write a C++ program to read two numbers from keyboard and then find the LARGEST number of them.

```
#include <iostream>
using namespace std;

int main()
{
   int num1, num2;
   cout<<"Enter first number:";
   cin>>num1;
   cout<<"Enter second number:";
   cin>>num2;
   if(num1>num2)
   {
cout<<"First number "<<num1<<" is the largest";
   }
```

```
   else
   {
cout<<"Second number "<<num2<<" is the largest";
   }
   return 0;
}
```

Q 2 (a): What are the Logical Operators? Explain them

The Logical Operators

Logical operators are mainly used to control program flow. Usually, you will find them as part of an `if`, a `while`, or some other control statement (Chapter 6)

The Logical operators are:

`op1 && op2`

-- Performs a logical AND of the two operands.

`op1 || op2`

-- Performs a logical OR of the two operands.

`!op1`

-- Performs a logical NOT of the operand.

The concept of logical operators is simple. They allow a program to make a decision based on multiple conditions. Each operand is considered a condition that can be evaluated to a true or false value. Then the value of the conditions is used to determine the overall value of the `op1 operator op2` or `!op1` grouping. The following examples demonstrate different ways that logical conditions can be used.

The `&&` operator is used to determine whether both operands or conditions are true `and.pl`.

For example:

```
if ($firstVar == 10 && $secondVar == 9) {

    print("Error!");

};
```

If either of the two conditions is false or incorrect, then the print command is bypassed.

The || operator is used to determine whether either of the conditions is true.

For example:

```
if ($firstVar == 9 || $firstVar == 10) {

    print("Error!");
```

If either of the two conditions is true, then the print command is run.

Caution If the first operand of the || operator evaluates to true, the second operand will not be evaluated. This could be a source of bugs if you are not careful.

For instance, in the following code fragment:

```
if ($firstVar++ || $secondVar++) { print("\n"); }
```

variable `$secondVar` will not be incremented if `$firstVar++` evaluates to true.

The `!` operator is used to convert true values to false and false values to true. In other words, it inverts a value. Perl considers any non-zero value to be true-even string values. For example:

```
$firstVar = 10;

$secondVar = !$firstVar;



if ($secondVar == 0) {

   print("zero\n");

};
```

is equal to *0*- and the program produces the following output:

```
zero
```

You could replace the 10 in the first line with "ten," 'ten,' or any non-zero, non-null value.

Q 2 (b): Write a C++ program to get Temperature in Fahrenheit *F* and then find the Atmosphere according to the below rules:

- **If temperature *F* is above 40 degree Fahrenheit then display…….Very Hot.**
- **If temperature *F* is between 35 & 40 degree Fahrenheit then display……..Tolerable.**
- **If temperature *F* is between 30 & 35 degree Fahrenheit then display……..Warm.**
- **If temperature *F* is less than 30 degree Fahrenheit then display……..Cool.**

Answer:

```
#include<iostream>

   using namespace std;


   int main()

   {

      float fahrenheit, celsius;


      cout << "Enter the temperature in Celsius : ";

      cin >> celsius;

      fahrenheit = (celsius * 9.0) / 5.0 + 32;
```

```
    cout << "The temperature in Celsius    : " << celsius << endl;

    cout << "The temperature in Fahrenheit : " << fahrenheit << endl;

     if (fahrenheit > 40) {
    cout << "Very Hot: " << endl;
  }

  if (fahrenheit >= 35 & fahrenheit <= 40) {
    cout << "Tolerable: " << endl;

  } else if (fahrenheit >= 30 & fahrenheit <= 35) {
    cout << "Warm: " << endl;

  } else if (fahrenheit < 30) {
    cout << "Cool: " << endl;
  }

    return 0;

  }
```

Q 3 (a): What does Looping mean? Explain different loops in C++.

A loop is used for executing a block of statements repeatedly until a particular condition is satisfied. For example, when you are displaying number from 1 to 100 you may want set the value of a variable to 1 and display it 100 times, increasing its value by 1 on each loop iteration.

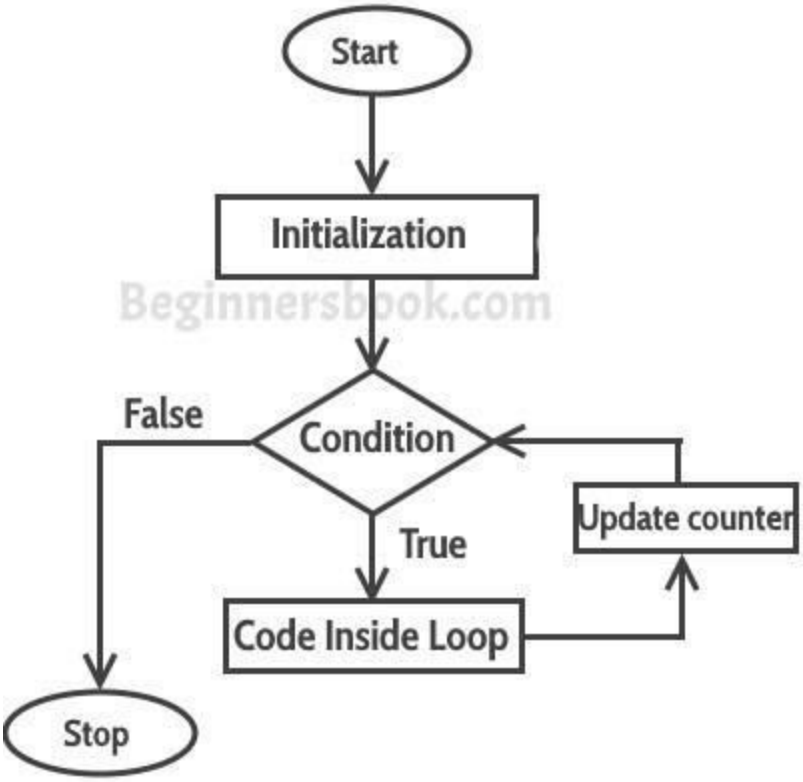
In C++ we have three types of basic loops: for, while and do-while. In this tutorial we will learn how to use “for loop” in C++.

Syntax of for loop

```
for(initialization; condition ; increment/decrement)
{
   C++ statement(s);
}
```

Flow of Execution of the for Loop

As a program executes, the interpreter always keeps track of which statement is about to be executed. We call this the control flow, or the flow of execution of the



program.

First step: In for loop, initialization happens first and only once, which means that the initialization part of for loop only executes once.

Second step: Condition in for loop is evaluated on each loop iteration, if the condition is true then the statements inside for for loop body gets executed. Once the condition returns false, the statements in for loop does not execute and the control gets transferred to the next statement in the program after for loop.

Third step: After every execution of for loop's body, the increment/decrement part of for loop executes that updates the loop counter.

Fourth step: After third step, the control jumps to second step and condition is re-evaluated.

The steps from second to fourth repeats until the loop condition returns false.

Example of a Simple For loop in C++

Here in the loop initialization part I have set the value of variable i to 1, condition is i<=6 and on each loop iteration the value of i increments by 1.

```
#include <iostream>
using namespace std;
int main(){
   for(int i=1; i<=6; i++){
      /* This statement would be executed
       * repeatedly until the condition
       * i<=6 returns false.
       */
      cout<<"Value of variable i is: "<<i<<endl;
   }
   return 0;
}
```

Output:

```
Value of variable i is: 1
Value of variable i is: 2
Value of variable i is: 3
Value of variable i is: 4
Value of variable i is: 5
Value of variable i is: 6
```

Infinite for loop in C++

A loop is said to be infinite when it executes repeatedly and never stops. This usually happens by mistake. When you set the condition in for loop in such a way that it never return false, it becomes infinite loop.

For example:

```
#include <iostream>
using namespace std;
int main(){
   for(int i=1; i>=1; i++){
      cout<<"Value of variable i is: "<<i<<endl;
   }
   return 0;
}
```

This is an infinite loop as we are incrementing the value of i so it would always satisfy the condition i>=1, the condition would never return false.

Here is another example of infinite for loop:

```
// infinite loop
for ( ; ; ) {
   // statement(s)
}
```

Example: display elements of array using for loop

```
#include <iostream>
using namespace std;
int main(){
   int arr[]={21,9,56,99, 202};
   /* We have set the value of variable i
    * to 0 as the array index starts with 0
    * which means the first element of array
    * starts with zero index.
    */
   for(int i=0; i<5; i++){
      cout<<arr[i]<<endl;
   }
   return 0;
}
```

Output:

```
21
9
56
99
202
```

While loop in C++ with example

As discussed earlier, loops are used for executing a block of program statements repeatedly until the given loop condition returns false.

Syntax of while loop

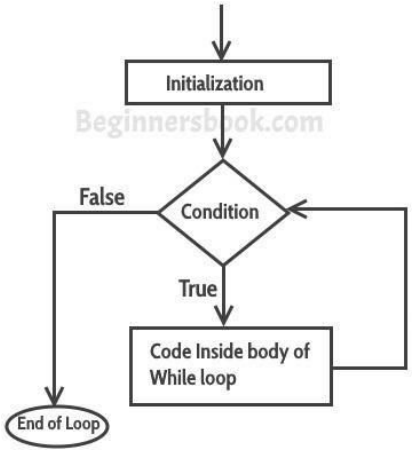
```
while(condition)
{
   statement(s);
}
```

How while Loop works?

In while loop, condition is evaluated first and if it returns true then the statements inside while loop execute, this happens repeatedly until the condition returns false. When condition returns false, the control comes out of loop and jumps to the next statement in the program after while loop.

Note: The important point to note when using while loop is that we need to use increment or decrement statement inside while loop so that the loop variable gets changed on each iteration, and at some point condition returns false. This way we can end the execution of while loop otherwise the loop would execute indefinitely.

Flow Diagram of While loop



While Loop example in C++

```
#include <iostream>
using namespace std;
int main(){
   int i=1;
   /* The loop would continue to print
    * the value of i until the given condition
    * i<=6 returns false.
    */
   while(i<=6){
      cout<<"Value of variable i is: "<<i<<endl; i++;
   }
}
```

Output:

```
Value of variable i is: 1
Value of variable i is: 2
Value of variable i is: 3
Value of variable i is: 4
Value of variable i is: 5
Value of variable i is: 6
```

Infinite While loop

A while loop that never stops is said to be the infinite while loop, when we give the condition in such a way so that it never returns false, then the loops becomes infinite and repeats itself indefinitely. **An example of infinite while loop:**
This loop would never end as I'm decrementing the value of i which is 1 so the condition i<=6 would never return false.

```
#include <iostream>
using namespace std;
int main(){
   int i=1; while(i<=6) {
      cout<<"Value of variable i is: "<<i<<endl; i--;
   }
}
```

Example: Displaying the elements of array using while loop

```
#include <iostream>
using namespace std;
int main(){
   int arr[]={21,87,15,99, -12};
   /* The array index starts with 0, the
    * first element of array has 0 index
    * and represented as arr[0]
    */
   int i=0;
   while(i<5){
      cout<<arr[i]<<endl;
      i++;
   }
}
```

Output:

```
21
87
15
99
```

-12

do-while loop in C++ with example

do-while loop is similar to while loop, however there is a difference between them: In while loop, condition is evaluated first and then the statements inside loop body gets executed, on the other hand in do-while loop, statements inside do-while gets executed first and then the condition is evaluated.

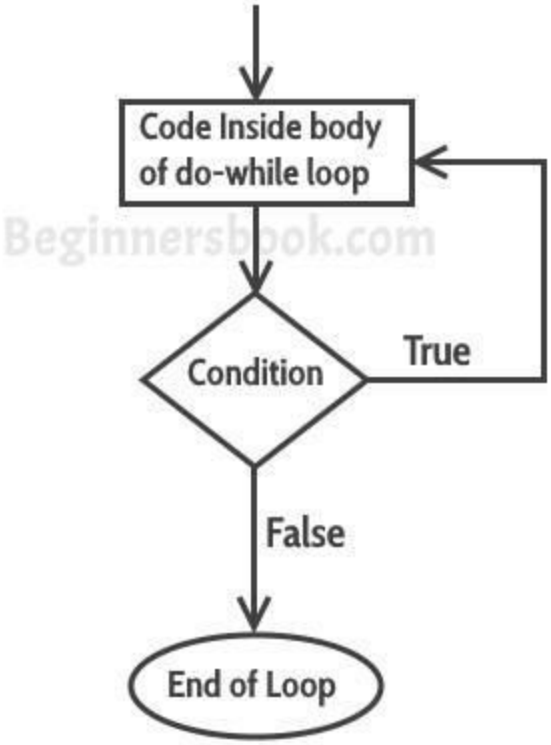
Syntax of do-while loop

```
do
{
   statement(s);
} while(condition);
```

How do-while loop works?

First, the statements inside loop execute and then the condition gets evaluated, if the condition returns true then the control jumps to the “do” for further repeated execution of it, this happens repeatedly until the condition returns false. Once

condition returns false control jumps to the next statement in the program after



do-while.

do-while loop example in C++

```
#include <iostream>
using namespace std;
int main(){
   int num=1;
   do{
      cout<<"Value of num: "<<num<<endl;
      num++;
   }while(num<=6);
   return 0;
}
```

Output:

```
Value of num: 1
Value of num: 2
Value of num: 3
Value of num: 4
Value of num: 5
Value of num: 6
```

Example: Displaying array elements using do-while loop

Here we have an integer array which has four elements. We are displaying the elements of it using do-while loop.

```
#include <iostream>
using namespace std;
int main(){
   int arr[]={21,99,15,109};
   /* Array index starts with 0, which
    * means the first element of array
    * is at index 0, arr[0]
    */
   int i=0;
   do{
      cout<<arr[i]<<endl;
      i++;
   }while(i<4);
   return 0;
}
```

Output:

```
21
99
15
109
```

Q 3 (b): Write a C++ program to read a number from keyboard and then determine whether it is Even or Odd number?

This C+ +Program checks if a given integer is odd or even. Here if a given number is divisible by 2 with the remainder 0 then the number is even number. If the number is not divisible by 2 then that number will be odd number.

```
#include<iostream>
using namespace std;

int main()
{
   int number, remainder;

   cout << "Enter the number : ";
   cin >> number;
   remainder = number % 2;
   if (remainder == 0)
      cout << number << " is an even integer " << endl;
   else
      cout << number << " is an odd integer " << endl;

   return 0;
```

Q 4 (A): What is the purpose of using break and continue statements?

Break Statement:

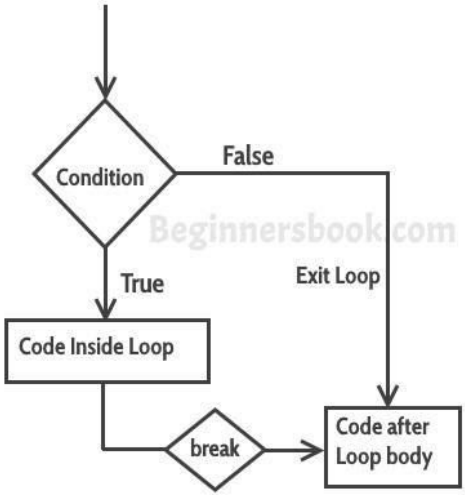
The **break statement** is used in following two scenarios:

a) Use break statement to come out of the loop instantly. Whenever a break statement is encountered inside a loop, the control directly comes out of loop terminating it. It is used along with if statement, whenever used inside loop(see the example below) so that it occurs only for a particular condition.

b) It is used in switch case control structure after the case blocks. Generally all cases in switch case are followed by a break statement to avoid the subsequent cases (see the example below) execution. Whenever it is encountered in switch-case block, the control comes out of the switch-case body.

Syntax of break statement

break statement flow diagram



Example – Use of break statement in a while loop

In the example below, we have a while loop running from 10 to 200 but since we have a break statement that gets encountered when the loop counter variable value reaches 12, the loop gets terminated and the control jumps to the next statement in program after the loop body.

```
#include <iostream>
using namespace std;
int main(){
   int num =10;
   while(num<=200) {
      cout<<"Value of num is: "<<num<<endl;
      if (num==12) {
         break;
      }
      num++;
   }
   cout<<"Hey, I'm out of the loop";
   return 0;
}
```

Output:

```
Value of num is: 10
Value of num is: 11
Value of num is: 12
Hey, I'm out of the loop
```

Example: break statement in for loop

```
#include <iostream>
using namespace std;
```

```
int main(){
   int var;
   for (var =200; var>=10; var --) {
      cout<<"var: "<<var<<endl;
      if (var==197) {
         break;
      }
   }
   cout<<"Hey, I'm out of the loop";
   return 0;
}
```

Output:

```
var: 200
var: 199
var: 198
var: 197
Hey, I'm out of the loop
```

Example: break statement in Switch Case

```
#include <iostream>
using namespace std;
int main(){
   int num=2;
   switch (num) {
      case 1: cout<<"Case 1 "<<endl;
      break;
      case 2: cout<<"Case 2 "<<endl;
      break;
      case 3: cout<<"Case 3 "<<endl;
      break;
      default: cout<<"Default "<<endl;
   }
   cout<<"Hey, I'm out of the switch case";
   return 0;
}
```

Output:

```
Case 2
Hey, I'm out of the switch case
```

In this example, we have break statement after each Case block, this is because if we don't have it then the subsequent case block would also execute. The output of the same program without break would be:

```
Case 2
Case 3
Default
```

```
Hey, I'm out of the switch case
```

Continue Statement

Continue statement is used inside loops. Whenever a continue statement is encountered inside a loop, control directly jumps to the beginning of the loop for next iteration, skipping the execution of statements inside loop's body for the current iteration.

Syntax of continue statement

Example: continue statement inside for loop

As you can see that the output is missing the value 3, however the for loop iterate though the num value 0 to 6. This is because we have set a condition inside loop in such a way, that the continue statement is encountered when the num value is equal to 3. So for this iteration the loop skipped the cout statement and started the next iteration of loop.

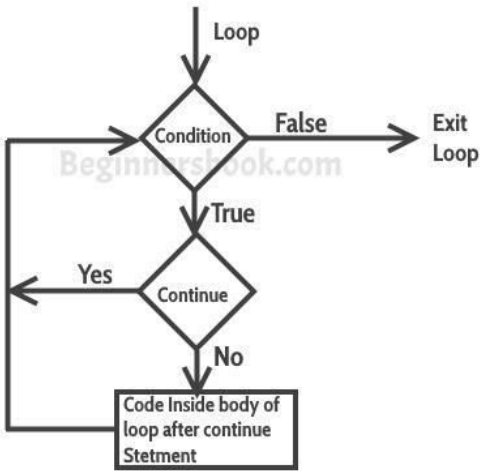
```
#include <iostream>
using namespace std;
int main(){
   for (int num=0; num<=6; num++) {
      /* This means that when the value of
       * num is equal to 3 this continue statement
       * would be encountered, which would make the
       * control to jump to the beginning of loop for
       * next iteration, skipping the current iteration
       */

      if (num==3) {
          continue;
      }
      cout<<num<<" ";
   }
   return 0;
}
```

Output:

```
0 1 2 4 5 6
```

Flow Diagram of Continue Statement



Example: Use of continue in While loop

```
#include <iostream>
using namespace std;
int main(){
   int j=6;
   while (j >=0) {
      if (j==4) {
         j--;
         continue;
      }
      cout<<"Value of j: "<<j<<endl;
      j--;
   }
   return 0;
}
```

Output:

```
Value of j: 6
Value of j: 5
Value of j: 3
Value of j: 2
Value of j: 1
Value of j: 0
```

Example of continue in do-While loop

```
#include <iostream>
using namespace std;
int main(){
```

```
   int j=4;
   do {
      if (j==7) {
         j++;
         continue;
      }
      cout<<"j is: "<<j<<endl;
      j++;
   }while(j<10);
   return 0;
}
```

Output:

```
j is: 4
j is: 5
j is: 6
j is: 8
j is: 9
```

Q 4 (b): Write a C++ program to find the sum of the following numbers:

1+2+3+……+10

```
#include <iostream>

using namespace std;

int main()

{

   int i,sum=0;

   for (i = 1; i <= 10; i++)

   {

      cout << i << " ";

sum=sum+i;

   }
```

```
    cout << "\n The sum of first 10 natural numbers: "<<sum << endl;
}
```

[redacted]

Q 5: What is an array? Explain On-Dimensional and Two-Dimensional Arrays with examples.

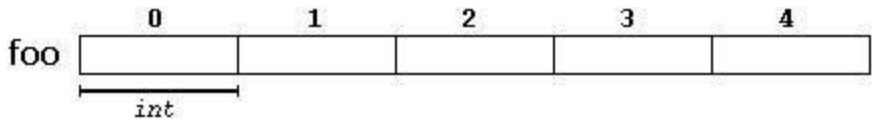
Arrays

ARRAYS IN C++

An array is a collection of elements of the same type placed in contiguous memory locations that can be individually referenced by using an index to a unique identifier.

Five values of type int can be declared as an array without having to declare five different variables (each with its own identifier).

For example, a five element integer array foo may be logically represented as;



where each blank panel represents an element of the array. In this case, these are values of type int. These elements are numbered from 0 to 4, with 0 being the first while 4 being the last; In C++, the index of the first array element is always zero. As expected, an n array must be declared prior its use. A typical declaration for an array in C++ is:

```
type name [elements];
```

where type is a valid type (such as `int, float` ...), name is a valid identifier and the elements field (which is always enclosed in square brackets `[]`), specifies the size of the array.

Thus, the foo array, with five elements of type `int`, can be declared as:

```
int foo [5];
```

NOTE

: The elements field within square brackets [], representing the number of elementsin the array, must be a constant expression, since arrays are blocks of static memory whose size must be known at compile time.

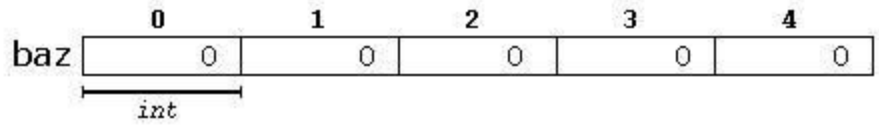
INITIALIZING ARRAYS

By default, are left uninitialized. This means that none of its elements are set to anyparticular value; their contents are undetermined at the point the array is declared.

The initializer can even have no values, just the braces:

```
int baz [5] = { };
```

This creates an array of five int values, each initialized with a value of zero:



But, the elements in an array can be explicitly initialized to specific values when it is declared, by enclosing those initial values in braces {}. For example:

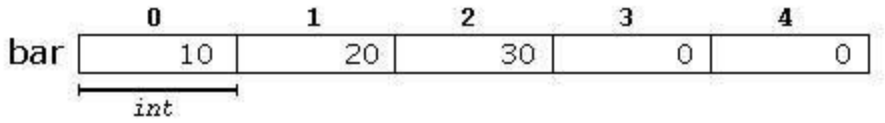
```
int foo [5] = { 16, 2, 77, 40, 12071 };
```

This statement declares an array that can be represented like this:

The number of values between braces {} shall not be greater than the number of elements in the array. For example, in the example above, foo was declared having 5 elements (as specified by the number enclosed in square brackets, []), and the braces {} contained exactly 5 values, one for each element. If declared with less, the remaining elements are set to their default values (which for fundamental types, means they are filled with zeroes). For example:

```
int bar [5] = { 10, 20, 30 };
```

Will create an array like this:



When an initialization of values is provided for an array, C++ allows the possibility of leaving the square brackets empty[]. In this case, the compiler will assume automatically a size for the array that matches the number of values included between the braces {}:

```
int foo [] = { 16, 2, 77, 40, 12071 };
```

After this declaration, array `foo` would be five `int` long, since we have provided five initialization values.

Finally, the evolution of C++ has led to the adoption of `universal initialization` also for arrays. Therefore, there is no longer need for the equal sign between the declaration and the initializer. Both these statements are equivalent:

```
int foo[] = { 10, 20, 30 };
int foo[] { 10, 20, 30 };
```

Here, the number of the array n is calculated by the compiler by using the formula n= #of initializers/sizeof(int).

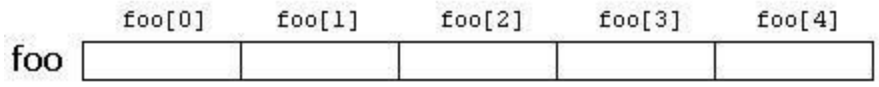
Static arrays, and those declared directly in a namespace (outside any function), are always initialized. If no explicit initializer is specified, all the elements are default-initialized (with zeroes, for fundamental types).

ARRAY ACCESSING

The values of any of the elements in an array can be accessed just like the value of a regular variable of the same type. The syntax is:

`name[index]`

Following the previous examples in which foo had 5 elements and each of those elements was of type int, the name which can be used to refer to each element is the following:



For example, the following statement stores the value 75 in the third element of `foo`:

```
foo [2] = 75;
```

and, for example, the following copies the value of the fourth element of `foo` to a variable called x:

```
x = foo[3];
```

Therefore, the expression `foo[2]` or `foo[4]` is always evaluated to an int. Notice that the third element of `foo` is specified `foo[2]`, the second one is `foo[1]`, since the first one is `foo[0]`. It's last element is therefore `foo[4]`. If we write `foo[5]`, we would be accessing the sixth element of `foo`, and therefore actually exceeding the size of the array.

In C++, it is syntactically correct to exceed the valid range of indices for an array. This can create problems, since accessing out-of-range elements do not cause errors on compilation, but can cause errors on runtime. The reason for this being allowed because index checking slows down program execution. At this point, it is important to be able to clearly distinguish between the two uses that brackets [] have related to arrays. They perform two different tasks: one is to specify the size of arrays when they are declared; and the second one is to specify indices for concrete array elements when they are accessed. Do not confuse these two possible uses of brackets `[]` with arrays.

```
int foo[5];         // declaration of a new array
foo[2] = 75;        // access to an element of the array.
```

The main difference is that the declaration is preceded by the type of the elements, while the access is not.

Some other valid operations with arrays:

```
foo[0] = a;
foo[i] = 75;
b = foo [i+2];
foo[foo[i]] = foo[i] + 5;
```

FOR EXAMPLE:

```
// arrays example
#include <iostream>
using namespace std;

int foo [] = {16, 2, 77, 40, 12071};
int i, result=0;

int main ()
{
  for ( i=0 ; i<5 ; i++ )
  {
    result += foo[i];
  }
  cout << result;
  return 0;
}
12206
```

MULTIDIMENSIONAL ARRAYS

Multidimensional arrays can be described as "arrays of arrays". For example, a bi-dimensional array can be imagined as a two-dimensional table made of elements, all of them hold same type of elements.

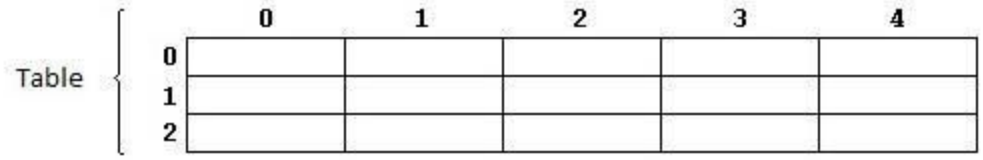
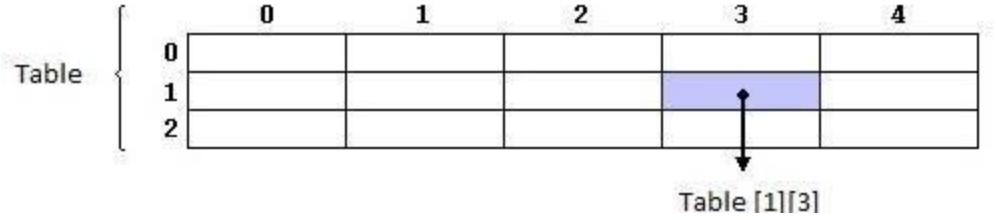


Table represents a bi-dimensional array of 3 per 5 elements of type int. The C++ syntax for this is

```
int Table [3][5];
```

and, for example, the way to reference the second element vertically and fourth horizontally in an expression would be:

```
Table[1][3]
```



(remember that array indices always begin with zero).

Multidimensional arrays are not limited to two indices (i.e., two dimensions). They can contain as many indices as needed. Although be careful: the amount of memory needed for an array increases exponentially with each dimension. For example:

```
char century [100][365][24][60][60];
```

Declares an array with an element of type char for each second in a century. This amounts to more than 3 billion char! So this declaration would consume more than 3 gigabytes of memory! And such a declaration is highly improbable and it underscores inefficient use of memory space.

At the end, multidimensional arrays are just an abstraction for programmers, since the same results can be achieved with a simple array, by multiplying its indices:

```
int Table [3][5];   // is equivalent to
int Table [15];     // (3 * 5 = 15)
```

With the only difference that with multidimensional arrays, the compiler automatically remembers the depth of each imaginary dimension. The following two pieces of code produce the exact same result, but one uses a bi-dimensional array while the other uses a simple array:

MULTIDIMENSIONAL ARRAY

```
const int WIDTH = 5;
const int HEIGHT = 3;

int Table [HEIGHT][WIDTH];
int n,m;

int main ()
{
  for (n=0; n<HEIGHT; n++)
    for (m=0; m<WIDTH; m++)
    {
      Table[n][m]=(n+1)*(m+1);
    }
}
```

PSEUDO-MULTIDIMENSIONAL ARRAY

```
const int WIDTH = 5;
const int HEIGHT = 3;

int Table [HEIGHT * WIDTH];
int n,m;
```

```
int main ()
{
  for (n=0; n<HEIGHT; n++)
    for (m=0; m<WIDTH; m++)
    {
      Table[n*WIDTH+m]=(n+1)*(m+1);
    }
}
```

None of the two code snippets above produce any output on the screen, but both assign values to the memory block called jimmy in the following way:

Table

	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

Note that the code uses named constants for the width and height, instead of using directly their numerical values. This gives the code a better readability, and allows changes in the code to be made easily in one place.

USING LOOP TO INPUT AN TWO-DIMENSIONAL ARRAY FROM USER

```
int mat[3][5], row, col ;
for (row = 0; row < 3; row++)
for (col = 0; col < 5; col++)
cin >> mat[row][col];
```

ARRAYS AS PARAMETERS

Two-dimensional arrays can be passed as parameters to a function, and they are passed by reference. This means that the function can directly access and modified the contents of the passed array. When declaring a two-dimensional array as a formal

parameter, we can omit the size of the first dimension, but not the second; that is, we must specify the number of columns. For example:

```
void print(int A[][3],int N, int M)
```

In order to pass to this function an array declared as:

```
int arr[4][3];
```

we need to write a call like this:

```
print(arr);
```

HERE IS A COMPLETE EXAMPLE:

```
#include <iostream>
using namespace std;
void print(int A[][3],int N, int M)
{
  for (R = 0; R < N; R++)
    for (C = 0; C < M; C++)
       cout << A[R][C];
}
int main ()
{
  int arr[4][3] ={{12, 29, 11},
                  {25, 25, 13},
                  {24, 64, 67},
                  {11, 18, 14}};
  print(arr,4,3);
  return 0;
}
```

Engineers use two dimensional arrays in order to represent matrices. The code for a function that finds the sum of the two matrices A and B are shown below.

FUNCTION TO FIND THE SUM OF TWO MATRICES

```
void Addition(int A[][20], int B[][20],int N, int M)
{
  for (int R=0;R<N;R++)
    for(int C=0;C<M;C++)
      C[R][C]=A[R][C]+B[R][C];
}
```

FUNCTION TO FIND OUT TRANSPOSE OF A MATRIX A

```
 void Transpose(int A[][20], int B[][20],int N, int M)
{
  for(int R=0;R<N;R++)
    for(int C=0;C<M;C++)
       B[R][C]=A[C][R];
}
```

ARRAYS AS PARAMETERS

At some point, we may need to pass an array to a function as a parameter. In C++, it is not possible to pass the entire block of memory represented by an array to a function directly as an argument. But what can be passed instead is its address. In practice, this has almost the same effect, and it is a much faster and more efficient operation.

To accept an array as parameter for a function, the parameters can be declared as the array type, but with empty brackets, omitting the actual size of the array. For example:

```
void procedure (int arg[])
```

This function accepts a parameter of type "array of `int`" called `arg`. In order to pass to this function an array declared as:

```
int myarray [40];
```

it would be enough to write a call like this:

```
procedure (myarray);
```

Here you have a complete example:

CODE

```
// arrays as parameters
#include <iostream>
using namespace std;

void printarray (int arg[], int length) {
  for (int n=0; n<length; ++n)
    cout << arg[n] << ' ';
  cout << '\n';
}

int main ()
{
  int firstarray[] = {5, 10, 15};
  int secondarray[] = {2, 4, 6, 8, 10};
  printarray (firstarray,3);
  printarray (secondarray,5);
}
```

SOLUTION

```
5 10 15
2 4 6 8 10
```

In the code above, the first parameter (`int arg[]`) accepts any array whose elements are of type int, whatever its length. For that reason, we have included a second parameter that tells the function the length of each array that we pass to it as its first parameter.

In a function declaration, it is also possible to include multidimensional arrays. The format for a tridimensional array parameter is:

```
base_type[][depth][depth]
```

For example, a function with a multidimensional array as argument could be:

```
void procedure (int myarray[][3][4])
```

Notice that the first brackets `[]` are left empty, while the following ones specify sizes for their respective dimensions. This is necessary in order for the compiler to be able to determine the depth of each additional dimension.

In a way, passing an array as argument always loses a dimension. The reason behind is that, for historical reasons, arrays cannot be directly copied, and thus what is really passed is a pointer. This is a common source of errors for novice programmers. Although a clear understanding of pointers, explained in a coming chapter, helps a lot.