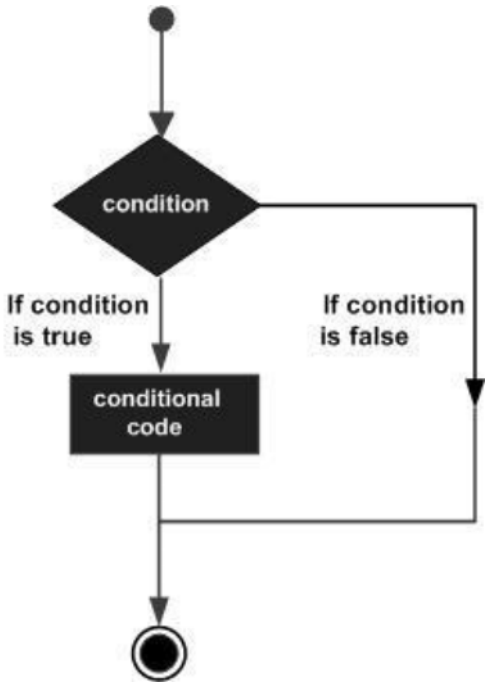
13. DECISION-MAKING STATEMENTS

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general from of a typical decision making structure found in most of the programming languages:



C++ programming language provides following types of decision making statements.

Statement	Description
if statement	An 'if' statement consists of a boolean expression followed by one or more statements.
if...else statement	An 'if' statement can be followed by an optional 'else' statement, which executes when the boolean expression is false.
switch statement	A 'switch' statement allows a variable to be tested for equality against a list of values.

nested if statements	You can use one 'if' or 'else if' statement inside another 'if' or 'else if' statement(s).
nested switch statements	You can use one 'switch' statement inside another 'switch' statement(s)

If Statement

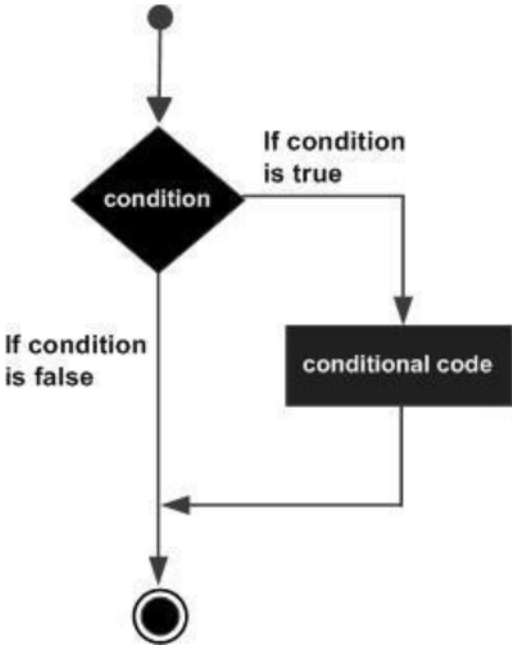
An if statement consists of a boolean expression followed by one or more statements.

Syntax

The syntax of an if statement in C++ is:

```
if(boolean_expression)
{
// statement(s) will execute if the boolean expression is true
}
```

If the boolean expression evaluates to true, then the block of code inside the if statement will be executed. If boolean expression evaluates to false, then the first set of code after the end of the if statement (after the closing curly brace) will be executed. Flow Diagram



Example

```
#include <iostream>
using namespace std;
int main ()
{
// local variable declaration:
int a = 10;
// check the boolean condition
if( a < 20 )
{
// if condition is true then print the following
cout << "a is less than 20;" << endl;
}
cout << "value of a is : " << a << endl;
return 0;
} C++
76
```

When the above code is compiled and executed, it produces the following result:

```
a is less than 20;
value of a is : 10
```

if…else Statement

An if statement can be followed by an optional else statement, which executes when the boolean expression is false.

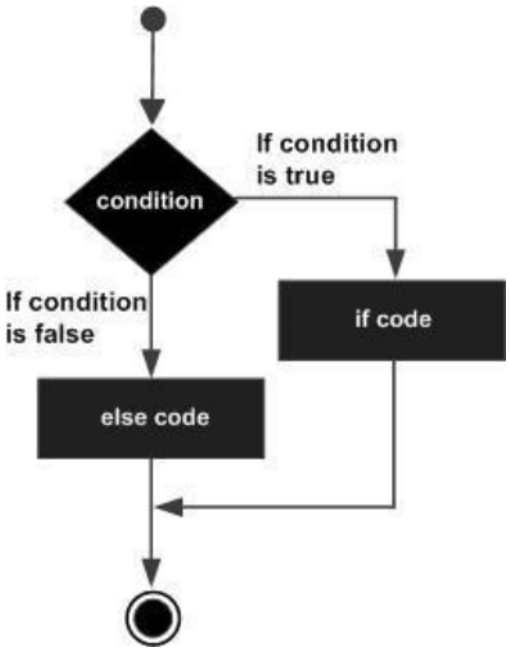
Syntax

The syntax of an if...else statement in C++ is:

```
if(boolean_expression)
{
// statement(s) will execute if the boolean expression is true
}
else
{
// statement(s) will execute if the boolean expression is false
}
```

If the boolean expression evaluates to true, then the if block of code will be executed, otherwise else block of code will be executed.

Flow Diagram



Example

```
#include <iostream>
using namespace std;
int main ()
{
// local variable declaration:
int a = 100;
// check the boolean condition
if( a < 20 )
{
// if condition is true then print the following
```

```
cout << "a is less than 20;" << endl;
}
else
{
// if condition is false then print the following
cout << "a is not less than 20;" << endl;
}
cout << "value of a is : " << a << endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
a is not less than 20;
value of a is : 100
```

if...else if...else Statement

An if statement can be followed by an optional else if...else statement, which is very usefull to test various conditions using single if...else if statement.

When using if , else if , else statements there are few points to keep in mind.

☐ An if can have zero or one else's and it must come after any else if's.

 An if can have zero to many else if's and they must come before the else.

☐ Once an else if succeeds, none of he remaining else if's or else's will be tested.

Syntax

The syntax of an if...else if...else statement in C++ is:

```
if(boolean_expression 1)
{
// Executes when the boolean expression 1 is true
}
else if( boolean_expression 2)
{
// Executes when the boolean expression 2 is true
}
else if( boolean_expression 3)
{
// Executes when the boolean expression 3 is true
}
else
{
// executes when the none of the above condition is true.
}
```

Example

```
#include <iostream>
using namespace std;
int main ()
{
// local variable declaration:
int a = 100;
// check the boolean condition
if( a == 10 ) C++
```

```
{
// if condition is true then print the following
cout << "Value of a is 10" << endl;
}
else if( a == 20 )
{
// if else if condition is true
cout << "Value of a is 20" << endl;
}
else if( a == 30 )
{
// if else if condition is true
cout << "Value of a is 30" << endl;
}
else
{
// if none of the conditions is true
cout << "Value of a is not matching" << endl;
}
cout << "Exact value of a is : " << a << endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of a is not matching
Exact value of a is : 100
```

Switch Statement

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax

The syntax for a switch statement in C++ is as follows:

```
switch(expression){
case constant-expression :
statement(s);
break; //optional
case constant-expression :
statement(s);
break; //optional
// you can have any number of case statements.
default : //Optional
statement(s);
}
```

The following rules apply to a switch statement:

☐ The expression used in a switch statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.

☐ You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.

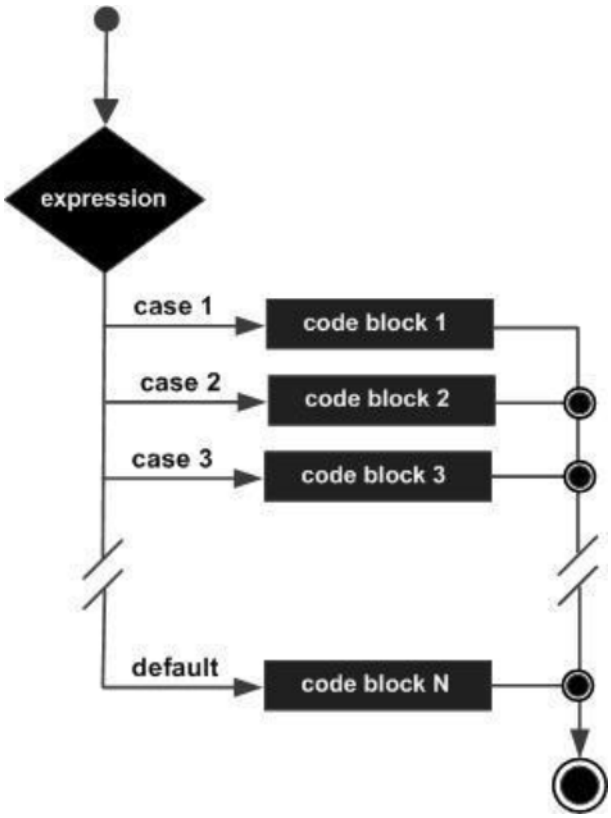
☐ The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.

☐ When the variable being switched on is equal to a case, the statements following that case will execute until a `break` statement is reached.

☐ When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

☐ Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.

☐ A `switch` statement can have an optional `default` case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.



Flow Diagram

Example

```
#include <iostream>
using namespace std;
int main ()
{
// local variable declaration:
char grade = 'D';
switch(grade)
{
case 'A' :
cout << "Excellent!" << endl;
break;
case 'B' :
case 'C' :
```

```
cout << "Well done" << endl;
break;

case 'D' :
cout << "You passed" << endl;
break;
case 'F' :
cout << "Better try again" << endl;
break;
default :
cout << "Invalid grade" << endl;
}
cout << "Your grade is " << grade << endl;
return 0;
}
```

This would produce the following result:

```
You passed
Your grade is D
```

Nested if Statement

It is always legal to nest if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).

Syntax

The syntax for a nested if statement is as follows:

```
if( boolean_expression 1)
{
// Executes when the boolean expression 1 is true
if(boolean_expression 2)
{
// Executes when the boolean expression 2 is true
}
}
```

You can nest else if...else in the similar way as you have nested *if* statement.

Example

```
#include <iostream>
using namespace std;
int main ()
{
// local variable declaration:
int a = 100;
int b = 200;
// check the boolean condition
if( a == 100 )
{
// if condition is true then check the following
if( b == 200 )
{
// if condition is true then print the following
cout << "Value of a is 100 and b is 200" << endl;
}
}
cout << "Exact value of a is : " << a << endl;
cout << "Exact value of b is : " << b << endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```

Nested switch Statements

It is possible to have a switch as part of the statement sequence of an outer switch. Even if the case constants of the inner and outer switch contain common values, no conflicts will arise. C++

C++ specifies that at least 256 levels of nesting be allowed for switch statements.

Syntax

The syntax for a nested switch statement is as follows:

```
switch(ch1) {
case 'A':
cout << "This A is part of outer switch";
switch(ch2) {
case 'A':
cout << "This A is part of inner switch";
break;
case 'B': // ...
}
break;
case 'B': // ...
}
```

Example

```
#include <iostream>
using namespace std;
int main ()
{
// local variable declaration:
int a = 100;
int b = 200;
switch(a) {
case 100:
cout << "This is part of outer switch" << endl;
switch(b) {
case 200:
cout << "This is part of inner switch" << endl; C++
```

```
}
}
cout << "Exact value of a is : " << a << endl;
cout << "Exact value of b is : " << b << endl;
return 0;
}
```
This would produce the following result:
```
This is part of outer switch
This is part of inner switch
Exact value of a is : 100
Exact value of b is : 200
```
The ? : Operator
We have covered conditional operator “? :” in previous chapter which can be used to replace if...else statements. It has the following general form:
```
Exp1 ? Exp2 : Exp3;
```
Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.
The value of a ‘?’ expression is determined like this: Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ‘?’ expression.

14. FUNCTIONS
program has at least one function, which is main(), and all the most trivial programs can define additional functions.
You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task.
A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.
The C++ standard library provides numerous built-in functions that your program can call. For example, function strcat() to concatenate two strings, function memcpy() to copy one memory location to another location, and many more functions.
A function is known with various names like a method or a sub-routine or a procedure etc.
Defining a Function
The general form of a C++ function definition is as follows:
```
return_type function_name( parameter list )
{
body of the function
}
```
A C++ function definition consists of a function header and a function body. Here are all the parts of a function:
☐ Return Type: A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.

☐ Function Name: This is the actual name of the function. The function name and the parameter list together constitute the function signature.

☐ Parameters: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

Function Body: The function body contains a collection of statements that define what the function does.

Example:

Following is the source code for a function called max(). This function takes two parameters num1 and num2 and returns the maximum between the two:

```
// function returning the max between two numbers
int max(int num1, int num2)
{
// local variable declaration
int result;
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
```

Function Declarations

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts:

```
return_type function_name( parameter list );
```

For the above defined function max(), following is the function declaration:

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so following is also valid declaration:

```
int max(int, int); C++
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a Function

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when it's return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example:

```
#include <iostream>
using namespace std;
// function declaration
int max(int num1, int num2);
int main ()
{
// local variable declaration:
int a = 100;
int b = 200;
int ret;
// calling a function to get max value.
ret = max(a, b);
cout << "Max value is : " << ret << endl;
return 0;
}
// function returning the max between two numbers C++
```

```
int max(int num1, int num2)
{
// local variable declaration
int result;
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
```
I kept max() function along with main() function and compiled the source code. While running final executable, it would produce the following result:
```
Max value is : 200
```
Function Arguments
If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.
The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit. While calling a function, there are two ways that arguments can be passed to a function:

Call Type	Description
Call by value	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Call by pointer	This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter

```
}
```
For now, let us call the function swap() by passing values by reference as in the following example:
```
#include <iostream>
using namespace std;
// function declaration
void swap(int &x, int &y);
int main ()
{
// local variable declaration:
int a = 100;
int b = 200;
cout << "Before swap, value of a :" << a << endl;
cout << "Before swap, value of b :" << b << endl;
```

```
/* calling a function to swap the values using variable reference.*/
swap(a, b);
cout << "After swap, value of a :" << a << endl;
cout << "After swap, value of b :" << b << endl;
return 0;
}
```

When the above code is put together in a file, compiled and executed, it produces the following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100 C++
```

By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

Default Values for Parameters

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead. Consider the following example:

```
#include <iostream>
using namespace std;
int sum(int a, int b=20)
{
int result;
result = a + b;
return (result);
}
int main ()
{
// local variable declaration:
int a = 100;
int b = 200;
int result;
// calling a function to add the values.
result = sum(a, b);
cout << "Total value is :" << result << endl; C++
```

```
// calling a function again as follows.
result = sum(a);
cout << "Total value is :" << result << endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Total value is :300
Total value is :120 C++
```

Normally, when we work with Numbers, we use primitive data types such as int, short, long, float and double, etc. The number data types, their possible values and number ranges have been explained while discussing C++ Data Types.

Defining Numbers in C++

You have already defined numbers in various examples given in previous chapters. Here is another consolidated example to define various types of numbers in C++:

```
#include <iostream>
using namespace std;
int main ()
{
// number definition:
short s;
int i;
long l;
float f;
double d;
// number assignments;
s = 10;
i = 1000;
l = 1000000;
f = 230.47;
d = 30949.374;
// number printing;
cout << "short s :" << s << endl;
cout << "int i :" << i << endl;
cout << "long l :" << l << endl;
```

15. NUMBERS C++

```
cout << "float f :" << f << endl;
cout << "double d :" << d << endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
short s :10
int i :1000
long l :1000000
float f :230.47
double d :30949.4
```

Math Operations in C++

In addition to the various functions you can create, C++ also includes some useful functions you can use. These functions are available in standard C and C++ libraries and called built-in functions. These are functions that can be included in your program and then use.

C++ has a rich set of mathematical operations, which can be performed on various numbers. Following table lists down some useful built-in mathematical functions available in C++.

To utilize these functions you need to include the math header file <cmath>. S.N.	Function & Purpose
1	double cos(double); This function takes an angle (as a double) and returns the cosine.
2	double sin(double); This function takes an angle (as a double) and returns the sine.
3	double tan(double); This function takes an angle (as a double) and returns the tangent.
4	double log(double); This function takes a number and returns the natural log of that number.
5	double pow(double, double); The first is a number you wish to raise and the second is the power you wish to raise it t
6	double hypot(double, double); If you pass this function the length of two sides of a right triangle, it will return you the length of the hypotenuse.
7	double sqrt(double); You pass this function a number and it gives you the square root.
8	int abs(int); This function returns the absolute value of an integer that is passed to it.

9	double fabs(double); This function returns the absolute value of any decimal number passed to it.
10	double floor(double); Finds the integer which is less than or equal to the argument passed to it.

Following is a simple example to show few of the mathematical operations:

```
#include <iostream>
#include <cmath>
using namespace std;
int main ()
{
// number definition:
short s = 10; C++
```

```
int i = -1000;
long l = 100000;
float f = 230.47;
double d = 200.374;
// mathematical operations;
cout << "sin(d) :" << sin(d) << endl;
cout << "abs(i) :" << abs(i) << endl;
cout << "floor(d) :" << floor(d) << endl;
cout << "sqrt(f) :" << sqrt(f) << endl;
cout << "pow( d, 2) :" << pow(d, 2) << endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
sign(d) :-0.634939
abs(i) :1000
floor(d) :200
sqrt(f) :15.1812
pow( d, 2 ) :40149.7
```

Random Numbers in C++

There are many cases where you will wish to generate a random number. There are actually two functions you will need to know about random number generation. The first is `rand()`, this function will only return a pseudo random number. The way to fix this is to first call the `srand()` function.

Following is a simple example to generate few random numbers. This example makes use of `time()` function to get the number of seconds on your system time, to randomly seed the rand() function:

```
#include <iostream>
#include <ctime>
#include <cstdlib>
using namespace std; C++
101
```

```
int main ()
{
int i,j;
// set the seed
srand( (unsigned)time( NULL ) );
/* generate 10 random numbers. */
for( i = 0; i < 10; i++ )
{
// generate actual random number
j= rand();
cout <<" Random Number : " << j << endl;
}
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Random Number : 1748144778
Random Number : 630873888 Random Number : 2134540646
Random Number : 219404170
Random Number : 902129458
Random Number : 920445370
Random Number : 1319072661
Random Number : 257938873
Random Number : 1256201101
Random Number : 580322989 C++
```


16. ARRAYS

C++ provides a data structure, the array, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.
Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.
All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Declaring Arrays

To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows:

```
type arrayName [ arraySize ];
```

This is called a single-dimension array. The arraySize must be an integer constant greater than zero and type can be any valid C++ data type. For example, to declare a 10-element array called balance of type double, use this statement:

```
double balance[10];
```

Initializing Arrays

You can initialize C++ array elements either one by one or using a single statement as follows:

```
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

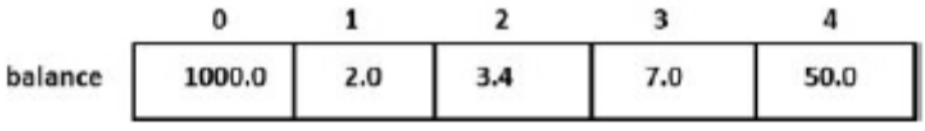
The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array:
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

```
double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

You will create exactly the same array as you did in the previous example.

```
balance[4] = 50.0;
```

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th, i.e., last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representation of the same array we discussed above:



Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

```
double salary = balance[9];
```

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example, which will use all the above-mentioned three concepts viz. declaration, assignment and accessing arrays:

```
#include <iostream>
using namespace std;
#include <iomanip>
using std::setw;
int main ()
{
int n[ 10 ]; // n is an array of 10 integers
// initialize elements of array n to 0
for ( int i = 0; i < 10; i++ )
{
n[ i ] = i + 100; // set element at location i to i + 100
}
cout << "Element" << setw( 13 ) << "Value" << endl; C++
104
```

```
// output each array element's value
for ( int j = 0; j < 10; j++ )
{
cout << setw( 7 )<< j << setw( 13 ) << n[ j ] << endl;
}
return 0;
}
```

This program makes use of `setw()` function to format the output. When the above code is compiled and executed, it produces the following result:

```
Element Value
0 100
1 101
2 102
3 103
4 104
5 105
6 106
7 107
8 108
9 109
```

Arrays in C++

Arrays are important to C++ and should need lots of more detail. There are following few important concepts, which should be clear to a C++ programmer:

Concept	Description
Multi-dimensional arrays	C++ supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
Pointer to an array	You can generate a pointer to the first element of an array by simply specifying the array name, without any index.
Passing arrays to functions	You can pass to the function a pointer to an array by specifying the array's name without an index.
Return array from functions	C++ allows a function to return an array.

Multi-dimensional Arrays

C++ allows multidimensional arrays. Here is the general form of a multidimensional array declaration:

```
type name[size1][size2]...[sizeN];
```

For example, the following declaration creates a three dimensional 5 . 10 . 4 integer array:

```
int threedim[5][10][4];
```

Two-Dimensional Arrays

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y, you would write something as follows:

```
type arrayName [ x ][ y ];
```

Where type can be any valid C++ data type and arrayName will be a valid C++ identifier.

A two-dimensional array can be think as a table, which will have x number of rows and y number of columns. A 2-dimensional array a, which contains three rows and four columns can be shown as below:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in array a is identified by an element name of the form a[i][j], where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

Initializing Two-Dimensional Arrays

Multidimensioned arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row have 4 columns.

```
int a[3][4] = {
{0, 1, 2, 3} , /* initializers for row indexed by 0 */
{4, 5, 6, 7} , /* initializers for row indexed by 1 */
{8, 9, 10, 11} /* initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements

An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example:

```
int val = a[2][3];
```

The above statement will take 4th element from the 3rd row of the array. You can verify it in the above digram. `#include <iostream>`

```
using namespace std;
int main ()
{
// an array with 5 rows and 2 columns.
int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
// output each array element's value
for ( int i = 0; i < 5; i++ )
for ( int j = 0; j < 2; j++ )
```

{ C++

```
cout << "a[" << i << "][" << j << "]: ";
cout << a[i][j]<< endl;
}
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

Pointer to an Array

It is most likely that you would not understand this chapter until you go through the chapter related C++ Pointers.

So assuming you have bit understanding on pointers in C++, let us start: An array name is a constant pointer to the first element of the array. Therefore, in the declaration:

```
double balance[50];
```

balance is a pointer to &balance[0], which is the address of the first element of the array balance. Thus, the following program fragment assigns p the address of the first element ofbalance:

```
double *p;
```

balance is a pointer to &balance[0], which is the address of the first element of the array balance. Thus, the following program fragment assigns p the address of the first element ofbalance:

```
double *p;
double balance[10]; C++
```

```
p = balance;
```
It is legal to use array names as constant pointers, and vice versa. Therefore, *(balance + 4) is a legitimate way of accessing the data at balance[4].
Once you store the address of first element in p, you can access array elements using *p, *(p+1), *(p+2) and so on. Below is the example to show all the concepts discussed above:
```
#include <iostream>
using namespace std;
int main ()
{
// an array with 5 elements.
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
double *p;
p = balance;
// output each array element's value
cout << "Array values using pointer " << endl;
for ( int i = 0; i < 5; i++ )
{
cout << "*(p + " << i << ") : ";
cout << *(p + i) << endl;
}
cout << "Array values using balance as address " << endl;
for ( int i = 0; i < 5; i++ )
{
cout << "*(balance + " << i << ") : ";
cout << *(balance + i) << endl;
}
return 0; }
```
When the above code is compiled and executed, it produces the following result:
```
Array values using pointer
*(p + 0) : 1000
*(p + 1) : 2
*(p + 2) : 3.4
*(p + 3) : 17
*(p + 4) : 50
Array values using balance as address
*(balance + 0) : 1000
*(balance + 1) : 2
*(balance + 2) : 3.4
*(balance + 3) : 17
*(balance + 4) : 50
```
In the above example, p is a pointer to double which means it can store address of a variable of double type. Once we have address in p, then *p will give us value available at the address stored in p, as we have shown in the above example.

Passing Arrays to Functions

C++ does not allow to pass an entire array as an argument to a function. However, You can pass a pointer to an array by specifying the array's name without an index.
If you want to pass a single-dimension array as an argument in a function, you would have to declare function formal parameter in one of following three ways

and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received.

Way-1

Formal parameters as a pointer as follows:

```
void myFunction(int *param)
{
.
.
. C++ Formal parameters as a pointer as follows:
void myFunction(int *param)
{
.
.
. C++
```

```
}
```

Way-2

Formal parameters as a sized array as follows:

```
void myFunction(int param[10])
{
.
.
.
}
```

Way-3

Formal parameters as an unsized array as follows:

```
void myFunction(int param[])
{
.
.
.
}
```

Now, consider the following function, which will take an array as an argument along with another argument and based on the passed arguments, it will return average of the numbers passed through the array as follows:

```
double getAverage(int arr[], int size)
{
int i, sum = 0;
double avg;
for (i = 0; i < size; ++i)
{
sum += arr[i];
}
avg = double(sum) / size;return avg;
}
```

Now, let us call the above function as follows:

```
#include <iostream>
using namespace std;
// function declaration:
double getAverage(int arr[], int size);
int main ()
{
// an int array with 5 elements.
int balance[5] = {1000, 2, 3, 17, 50};
double avg;
// pass pointer to the array as an argument.
avg = getAverage( balance, 5 ) ;
// output the returned value
cout << "Average value is: " << avg << endl;
return 0;
}
```

When the above code is compiled together and executed, it produces the following result:

```
Average value is: 214.4
```

As you can see, the length of the array doesn't matter as far as the function is concerned because C++ performs no bounds checking for the formal parameters. C++

Return Array from Functions

C++ does not allow to return an entire array as an argument to a function. However, you can return a pointer to an array by specifying the array's name without an index.

If you want to return a single-dimension array from a function, you would have to declare a function returning a pointer as in the following example:

```
int * myFunction()
{
.
.
.
}
```

Second point to remember is that C++ does not advocate to return the address of a local variable to outside of the function so you would have to define the local variable as staticvariable.

Now, consider the following function, which will generate 10 random numbers and return them using an array and call this function as follows:

```
#include <iostream>
#include <ctime>
using namespace std;
// function to generate and retrun random numbers.
int * getRandom( )
{
static int r[10];
// set the seed
srand( (unsigned)time( NULL ) );
for (int i = 0; i < 10; ++i)
{
r[i] = rand();
cout << r[i] << endl;
} C++
113
```

```
return r;
}
// main function to call above defined

int main ()
{
// a pointer to an int.
int *p;
p = getRandom();
for ( int i = 0; i < 10; i++ )
{
cout << "*(p + " << i << ") : ";
cout << *(p + i) << endl;
}
return 0;
}
```

When the above code is compiled together and executed, it produces result something as follows:

```
624723190
1468735695
807113585
976495677
613357504
1377296355
1530315259
1778906708
1820354158
667126415
*(p + 0) : 624723190
*(p + 1) : 1468735695 C++
114
```

```
*(p + 2) : 807113585
*(p + 3) : 976495677
*(p + 4) : 613357504
*(p + 5) : 1377296355
*(p + 6) : 1530315259
*(p + 7) : 1778906708
*(p + 8) : 1820354158
*(p + 9) : 667126415
```