C++ Data Types

While writing program in any language, you need to use various variables to store various information. Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

You may like to store information of various data types like character, wide character, integer, floating point, double floating point, boolean etc. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

Primitive Built-in Types

C++ offers the programmer a rich assortment of built-in as well as user defined data types. Following table lists down seven basic C++ data types −

Type	Keyword
Boolean	bool
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void
Wide character	wchar_t

Several of the basic types can be modified using one or more of these type modifiers −

- signed
- unsigned
- short
- long

The following table shows the variable type, how much memory it takes to store the value in memory, and what is maximum and minimum value which can be stored in such type of variables.

Type	Typical Bit Width	Typical Range
char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	2bytes	0 to 65,535
signed short int	2bytes	-32768 to 32767
long int	4bytes	-2,147,483,648 to 2,147,483,647
signed long int	8bytes	same as long int
unsigned long int	4bytes	0 to 4,294,967,295
long long int	8bytes	-(2^63) to (2^63)-1
unsigned long long int	8bytes	0 to 18,446,744,073,709,551,615
float	4bytes	
double	8bytes	
long double	12bytes	
wchar_t	2 or 4 bytes	1 wide character

The size of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.

Following is the example, which will produce correct size of various data types on your computer.

Live Demo

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

int main() {
   cout << "Size of char : " << sizeof(char) << endl;
   cout << "Size of int : " << sizeof(int) << endl;
```

```
   cout << "Size of short int : " << sizeof(short int) << endl;
   cout << "Size of long int : " << sizeof(long int) << endl;
   cout << "Size of float : " << sizeof(float) << endl;
   cout << "Size of double : " << sizeof(double) << endl;
   cout << "Size of wchar_t : " << sizeof(wchar_t) << endl;

   return 0;
}
```

This example uses **endl**, which inserts a new-line character after every line and << operator is being used to pass multiple values out to the screen. We are also using **sizeof()** operator to get size of various data types.

When the above code is compiled and executed, it produces the following result which can vary from machine to machine −

```
Size of char : 1
Size of int : 4
Size of short int : 2
Size of long int : 4
Size of float : 4
Size of double : 8
Size of wchar_t : 4
```

typedef Declarations

You can create a new name for an existing type using **typedef**. Following is the simple syntax to define a new type using typedef −

```
typedef type newname;
```

For example, the following tells the compiler that feet is another name for int −

```
typedef int feet;
```

Now, the following declaration is perfectly legal and creates an integer variable called distance −

```
feet distance;
```

Enumerated Types

An enumerated type declares an optional type name and a set of zero or more identifiers that can be used as values of the type. Each enumerator is a constant whose type is the enumeration.

Creating an enumeration requires the use of the keyword **enum**. The general form of an enumeration type is −

```
enum enum-name { list of names } var-list;
```

Here, the enum-name is the enumeration's type name. The list of names is comma separated.

For example, the following code defines an enumeration of colors called colors and the variable c of type color. Finally, c is assigned the value "blue".

```
enum color { red, green, blue } c;
c = blue;
```

By default, the value of the first name is 0, the second name has the value 1, and the third has the value 2, and so on. But you can give a name, a specific value by adding an initializer. For example, in the following enumeration, **green** will have the value 5.

```
enum color { red, green = 5, blue };
```

Here, **blue** will have a value of 6 because each name will be one greater than the one that precedes it.

C++ Variable Types

A variable provides us with named storage that our programs can manipulate. Each variable in C++ has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C++ is case-sensitive −

There are following basic types of variable in C++ as explained in last chapter −

Sr.No	Type & Description
1	**bool** Stores either value true or false.
2	**char** Typically a single octet (one byte). This is an integer type.
3	**int** The most natural size of integer for the machine.
4	**float** A single-precision floating point value.
5	**double** A double-precision floating point value.
6	**void** Represents the absence of type.
7	**wchar_t** A wide character type.

C++ also allows to define various other types of variables, which we will cover in subsequent chapters like **Enumeration, Pointer, Array, Reference, Data structures,** and **Classes**.

Following section will cover how to define, declare and use various types of variables.

Variable Definition in C++

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type, and contains a list of one or more variables of that type as follows −

```
type variable_list;
```

Here, **type** must be a valid C++ data type including char, w_char, int, float, double, bool or any user-defined object, etc., and **variable_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here −

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

The line **int i, j, k;** both declares and defines the variables i, j and k; which instructs the compiler to create variables named i, j and k of type int.

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows −

```
type variable_name = value;
```

Some examples are −

```
extern int d = 3, f = 5;    // declaration of d and f.
int d = 3, f = 5;           // definition and initializing d and f.
byte z = 22;                // definition and initializes z.
char x = 'x';               // the variable x has the value 'x'.
```

For definition without an initializer: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables is undefined.

Variable Declaration in C++

A variable declaration provides assurance to the compiler that there is one variable existing with the given type and name so that compiler proceed for further compilation without needing complete detail about the variable. A variable declaration has its meaning at the time of compilation only, compiler needs actual variable definition at the time of

linking of the program.

A variable declaration is useful when you are using multiple files and you define your variable in one of the files which will be available at the time of linking of the program. You will use **extern** keyword to declare a variable at any place. Though you can declare a variable multiple times in your C++ program, but it can be defined only once in a file, a function or a block of code.

Example

Try the following example where a variable has been declared at the top, but it has been defined inside the main function −

Live Demo

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

// Variable declaration:
extern int a, b;
extern int c;
extern float f;

int main () {
   // Variable definition:
   int a, b;
   int c;
   float f;

   // actual initialization
   a = 10;
   b = 20;
   c = a + b;

   cout << c << endl ;

   f = 70.0/3.0;
   cout << f << endl ;

   return 0;
}
```

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

```
30
23.3333
```

Same concept applies on function declaration where you provide a function name at the time of its declaration and its actual definition can be given anywhere else. For example −

```
// function declaration
int func();
int main() {
   // function call
   int i = func();
}

// function definition
int func() {
   return 0;
}
```

Lvalues and Rvalues

There are two kinds of expressions in C++ −

- **lvalue** − Expressions that refer to a memory location is called "lvalue" expression. An lvalue may appear as either the left-hand or right-hand side of an assignment.
- **rvalue** − The term rvalue refers to a data value that is stored at some address in memory. An rvalue is an expression that cannot have a value assigned to it which means an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned and can not appear on the left-hand side. Following is a valid statement −

```
int g = 20;
```

But the following is not a valid statement and would generate compile-time error −

```
10 = 20;
```

Variable Scope in C++

A scope is a region of the program and broadly speaking there are three places, where variables can be declared −

- Inside a function or a block which is called local variables,
- In the definition of function parameters which is called formal parameters.
- Outside of all functions which is called global variables.

We will learn what is a function and it's parameter in subsequent chapters. Here let us explain what are local and global variables.

Local Variables

Variables that are declared inside a function or block are local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using local variables −

Live Demo

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

int main () {
   // Local variable declaration:
   int a, b;
   int c;

   // actual initialization
   a = 10;
   b = 20;
   c = a + b;

   cout << c;

   return 0;
}
```

Global Variables

Global variables are defined outside of all the functions, usually on top of the program. The global variables will hold their value throughout the life-time of your program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. Following is the example using global and local variables −

Live Demo

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

// Global variable declaration:
int g;

int main () {
   // Local variable declaration:
   int a, b;

   // actual initialization
   a = 10;
   b = 20;
   g = a + b;

   cout << g;

   return 0;
}
```

A program can have same name for local and global variables but value of local variable inside a function will take preference. For example −

Live Demo

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

// Global variable declaration:
int g = 20;

int main () {
   // Local variable declaration:
   int g = 10;

   cout << g;

   return 0;
}
```

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

```
10
```

Initializing Local and Global Variables

When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you define them as follows −

Data Type	Initializer
int	0
char	'\0'
float	0
double	0
pointer	NULL

It is a good programming practice to initialize variables properly, otherwise sometimes program would produce unexpected result.

C++ Data Types

While writing program in any language, you need to use various variables to store various information. Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

You may like to store information of various data types like character, wide character, integer, floating point, double floating point, boolean etc. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

Primitive Built-in Types

C++ offers the programmer a rich assortment of built-in as well as user defined data types. Following table lists down seven basic C++ data types −

Type	Keyword
Boolean	bool
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void
Wide character	wchar_t

Several of the basic types can be modified using one or more of these type modifiers −

- signed
- unsigned
- short
- long

The following table shows the variable type, how much memory it takes to store the value in memory, and what is maximum and minimum value which can be stored in such type of variables.

Type	Typical Bit Width	Typical Range
char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	2bytes	0 to 65,535
signed short int	2bytes	-32768 to 32767
long int	4bytes	-2,147,483,648 to 2,147,483,647
signed long int	8bytes	same as long int
unsigned long int	4bytes	0 to 4,294,967,295
long long int	8bytes	-(2^63) to (2^63)-1
unsigned long long int	8bytes	0 to 18,446,744,073,709,551,615
float	4bytes	
double	8bytes	
long double	12bytes	
wchar_t	2 or 4 bytes	1 wide character

The size of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.

Following is the example, which will produce correct size of various data types on your computer.

Live Demo

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

int main() {
   cout << "Size of char : " << sizeof(char) << endl;
   cout << "Size of int : " << sizeof(int) << endl;
```

```
   cout << "Size of short int : " << sizeof(short int) << endl;
   cout << "Size of long int : " << sizeof(long int) << endl;
   cout << "Size of float : " << sizeof(float) << endl;
   cout << "Size of double : " << sizeof(double) << endl;
   cout << "Size of wchar_t : " << sizeof(wchar_t) << endl;

   return 0;
}
```

This example uses **endl**, which inserts a new-line character after every line and << operator is being used to pass multiple values out to the screen. We are also using **sizeof()** operator to get size of various data types.

When the above code is compiled and executed, it produces the following result which can vary from machine to machine −

```
Size of char : 1
Size of int : 4
Size of short int : 2
Size of long int : 4
Size of float : 4
Size of double : 8
Size of wchar_t : 4
```

typedef Declarations

You can create a new name for an existing type using **typedef**. Following is the simple syntax to define a new type using typedef −

```
typedef type newname;
```

For example, the following tells the compiler that feet is another name for int −

```
typedef int feet;
```

Now, the following declaration is perfectly legal and creates an integer variable called distance −

```
feet distance;
```

Enumerated Types

An enumerated type declares an optional type name and a set of zero or more identifiers that can be used as values of the type. Each enumerator is a constant whose type is the enumeration.

Creating an enumeration requires the use of the keyword **enum**. The general form of an enumeration type is −

```
enum enum-name { list of names } var-list;
```

Here, the enum-name is the enumeration's type name. The list of names is comma separated.

For example, the following code defines an enumeration of colors called colors and the variable c of type color. Finally, c is assigned the value "blue".

```
enum color { red, green, blue } c;
c = blue;
```

By default, the value of the first name is 0, the second name has the value 1, and the third has the value 2, and so on. But you can give a name, a specific value by adding an initializer. For example, in the following enumeration, **green** will have the value 5.

```
enum color { red, green = 5, blue };
```

Here, **blue** will have a value of 6 because each name will be one greater than the one that precedes it.

C++ Variable Types

A variable provides us with named storage that our programs can manipulate. Each variable in C++ has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C++ is case-sensitive −

There are following basic types of variable in C++ as explained in last chapter −

Sr.No	Type & Description
1	**bool** Stores either value true or false.
2	**char** Typically a single octet (one byte). This is an integer type.
3	**int** The most natural size of integer for the machine.
4	**float** A single-precision floating point value.
5	**double** A double-precision floating point value.
6	**void** Represents the absence of type.
7	**wchar_t** A wide character type.

C++ also allows to define various other types of variables, which we will cover in subsequent chapters like **Enumeration, Pointer, Array, Reference, Data structures,** and **Classes**.

Following section will cover how to define, declare and use various types of variables.

Variable Definition in C++

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type, and contains a list of one or more variables of that type as follows −

```
type variable_list;
```

Here, **type** must be a valid C++ data type including char, w_char, int, float, double, bool or any user-defined object, etc., and **variable_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here −

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

The line **int i, j, k;** both declares and defines the variables i, j and k; which instructs the compiler to create variables named i, j and k of type int.

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows −

```
type variable_name = value;
```

Some examples are −

```
extern int d = 3, f = 5;    // declaration of d and f.
int d = 3, f = 5;           // definition and initializing d and f.
byte z = 22;                // definition and initializes z.
char x = 'x';               // the variable x has the value 'x'.
```

For definition without an initializer: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables is undefined.

Variable Declaration in C++

A variable declaration provides assurance to the compiler that there is one variable existing with the given type and name so that compiler proceed for further compilation without needing complete detail about the variable. A variable declaration has its meaning at the time of compilation only, compiler needs actual variable definition at the time of

linking of the program.

A variable declaration is useful when you are using multiple files and you define your variable in one of the files which will be available at the time of linking of the program. You will use **extern** keyword to declare a variable at any place. Though you can declare a variable multiple times in your C++ program, but it can be defined only once in a file, a function or a block of code.

Example

Try the following example where a variable has been declared at the top, but it has been defined inside the main function −

Live Demo

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

// Variable declaration:
extern int a, b;
extern int c;
extern float f;

int main () {
   // Variable definition:
   int a, b;
   int c;
   float f;

   // actual initialization
   a = 10;
   b = 20;
   c = a + b;

   cout << c << endl ;

   f = 70.0/3.0;
   cout << f << endl ;

   return 0;
}
```

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

```
30
23.3333
```

Same concept applies on function declaration where you provide a function name at the time of its declaration and its actual definition can be given anywhere else. For example −

```
// function declaration
int func();
int main() {
   // function call
   int i = func();
}

// function definition
int func() {
   return 0;
}
```

Lvalues and Rvalues

There are two kinds of expressions in C++ −

- **lvalue** − Expressions that refer to a memory location is called "lvalue" expression. An lvalue may appear as either the left-hand or right-hand side of an assignment.
- **rvalue** − The term rvalue refers to a data value that is stored at some address in memory. An rvalue is an expression that cannot have a value assigned to it which means an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned and can not appear on the left-hand side. Following is a valid statement −

```
int g = 20;
```

But the following is not a valid statement and would generate compile-time error −

```
10 = 20;
```

C++ Constants/Literals

Constants refer to fixed values that the program may not alter and they are called **literals**.

Constants can be of any of the basic data types and can be divided into Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.

Again, constants are treated just like regular variables except that their values cannot be modified after their definition.

Integer Literals

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals −

```
212         // Legal
215u        // Legal
0xFeeL      // Legal
078         // Illegal: 8 is not an octal digit
032UU       // Illegal: cannot repeat a suffix
```

Following are other examples of various types of Integer literals −

```
85         // decimal
0213       // octal
0x4b       // hexadecimal
30         // int
30u        // unsigned int
30l        // long
30ul       // unsigned long
```

Floating-point Literals

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

While representing using decimal form, you must include the decimal point, the exponent, or both and while representing using exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some examples of floating-point literals −

```
3.14159       // Legal
314159E-5L    // Legal
510E          // Illegal: incomplete exponent
210f          // Illegal: no decimal or exponent
.e55          // Illegal: missing integer or fraction
```

Boolean Literals

There are two Boolean literals and they are part of standard C++ keywords −

- A value of **true** representing true.
- A value of **false** representing false.

You should not consider the value of true equal to 1 and value of false equal to 0.

Character Literals

Character literals are enclosed in single quotes. If the literal begins with L (uppercase only), it is a wide character literal (e.g., L'x') and should be stored in **wchar_t** type of variable . Otherwise, it is a narrow character literal (e.g., 'x') and can be stored in a simple variable of **char** type.

A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

There are certain characters in C++ when they are preceded by a backslash they will have special meaning and they are used to represent like newline (\n) or tab (\t). Here, you have a list of some of such escape sequence codes −

Escape sequence	Meaning
\\	\ character
\'	' character
\"	" character
\?	? character
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\ooo	Octal number of one to three digits
\xhh . . .	Hexadecimal number of one or more digits

Following is the example to show a few escape sequence characters −

Live Demo

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

int main() {
   cout << "Hello\tWorld\n\n";
   return 0;
}
```

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

```
Hello   World
```

String Literals

String literals are enclosed in double quotes. A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separate them using whitespaces.

Here are some examples of string literals. All the three forms are identical strings.

```
"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"
```

Defining Constants

There are two simple ways in C++ to define constants −

- Using **#define** preprocessor.
- Using **const** keyword.

The #define Preprocessor

Following is the form to use #define preprocessor to define a constant −

```
#define identifier value
```

Following example explains it in detail −

Live Demo

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

#define LENGTH 10
#define WIDTH  5
#define NEWLINE '\n'

int main() {
   int area;

   area = LENGTH * WIDTH;
   cout << area;
   cout << NEWLINE;
```

```
   return 0;
}
```

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

```
50
```

The const Keyword

You can use **const** prefix to declare constants with a specific type as follows −

```
const type variable = value;
```

Following example explains it in detail −

Live Demo

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

int main() {
   const int  LENGTH = 10;
   const int  WIDTH  = 5;
   const char NEWLINE = '\n';
   int area;

   area = LENGTH * WIDTH;
   cout << area;
   cout << NEWLINE;
   return 0;
}
```

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

```
50
```

Note that it is a good programming practice to define constants in CAPITALS.

C++ Modifier Types

C++ allows the **char, int,** and **double** data types to have modifiers preceding them. A modifier is used to alter the meaning of the base type so that it more precisely fits the needs of various situations.

The data type modifiers are listed here −

- signed
- unsigned
- long
- short

The modifiers **signed, unsigned, long,** and **short** can be applied to integer base types. In addition, **signed** and **unsigned** can be applied to char, and **long** can be applied to double.

The modifiers **signed** and **unsigned** can also be used as prefix to **long** or **short** modifiers. For example, **unsigned long int**.

C++ allows a shorthand notation for declaring **unsigned, short,** or **long** integers. You can simply use the word **unsigned, short,** or **long,** without **int**. It automatically implies **int**. For example, the following two statements both declare unsigned integer variables.

```
unsigned x;
unsigned int y;
```

To understand the difference between the way signed and unsigned integer modifiers are interpreted by C++, you should run the following short program −

Live Demo

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

/* This program shows the difference between
   * signed and unsigned integers.
*/
int main() {
   short int i;           // a signed short integer
   short unsigned int j;  // an unsigned short integer

   j = 50000;

   i = j;
   cout << i << " " << j;

   return 0;
}
```

When this program is run, following is the output −

```
-15536 50000
```

The above result is because the bit pattern that represents 50,000 as a short unsigned integer is interpreted as -15,536 by a short.

Type Qualifiers in C++

The type qualifiers provide additional information about the variables they precede.

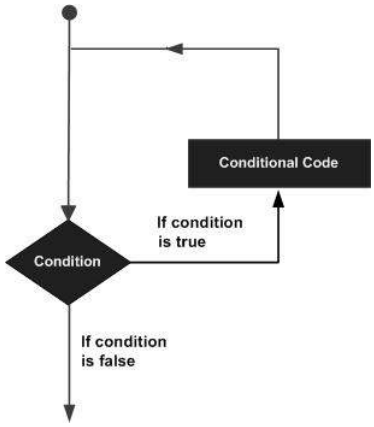
Sr.No	Qualifier & Meaning
1	**const** Objects of type **const** cannot be changed by your program during execution.
2	**volatile** The modifier **volatile** tells the compiler that a variable's value may be changed in ways not explicitly specified by the program.
3	**restrict** A pointer qualified by **restrict** is initially the only means by which the object it points to can be accessed. Only C99 adds a new type qualifier called restrict.

C++ Loop Types

There may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages −



C++ programming language provides the following type of loops to handle looping requirements.

Sr.No	Loop Type & Description
1	while loop Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2	for loop Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	do...while loop Like a 'while' statement, except that it tests the condition at the end of the loop body.
4	nested loops You can use one or more loop inside any another 'while', 'for' or 'do..while' loop.

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C++ supports the following control statements.

Sr.No	Control Statement & Description
1	break statement Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
2	continue statement Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	goto statement Transfers control to the labeled statement. Though it is not advised to use goto statement in your program.

The Infinite Loop

A loop becomes infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the 'for' loop are required, you can make an endless loop by leaving the conditional expression empty.

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

int main () {
   for( ; ; ) {
      printf("This loop will run forever.\n");
   }

   return 0;
}
```

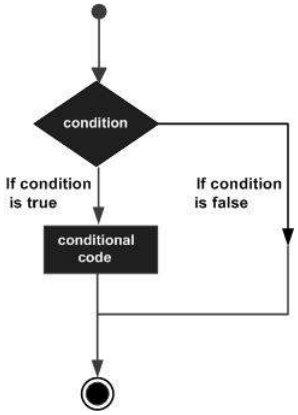
When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C++ programmers more commonly use the 'for (;;)' construct to signify an infinite loop.

NOTE − You can terminate an infinite loop by pressing Ctrl + C keys.

C++ 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 form of a typical decision making structure found in most of the programming languages −



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

Sr.No	Statement & Description
1	if statement An 'if' statement consists of a boolean expression followed by one or more statements.
2	if...else statement An 'if' statement can be followed by an optional 'else' statement, which executes when the boolean expression is false.
3	switch statement A 'switch' statement allows a variable to be tested for equality against a list of values.
4	nested if statements You can use one 'if' or 'else if' statement inside another 'if' or 'else if' statement(s).
5	nested switch statements You can use one 'switch' statement inside another 'switch' statement(s).

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. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

C++ Basic Input/Output

The C++ standard libraries provide an extensive set of input/output capabilities which we will see in subsequent chapters. This chapter will discuss very basic and most common I/O operations required for C++ programming.

C++ I/O occurs in streams, which are sequences of bytes. If bytes flow from a device like a keyboard, a disk drive, or a network connection etc. to main memory, this is called **input operation** and if bytes flow from main memory to a device like a display screen, a printer, a disk drive, or a network connection, etc., this is called **output operation**.

I/O Library Header Files

There are following header files important to C++ programs −

Sr.No	Header File & Function and Description
1	**<iostream>** This file defines the **cin, cout, cerr** and **clog** objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, respectively.
2	**<iomanip>** This file declares services useful for performing formatted I/O with so-called parameterized stream manipulators, such as **setw** and **setprecision**.
3	**<fstream>** This file declares services for user-controlled file processing. We will discuss about it in detail in File and Stream related chapter.

The Standard Output Stream (cout)

The predefined object **cout** is an instance of **ostream** class. The cout object is said to be "connected to" the standard output device, which usually is the display screen. The **cout** is used in conjunction with the stream insertion operator, which is written as << which are two less than signs as shown in the following example.

Live Demo

```
#include <iostream>

using namespace std;

int main() {
   char str[] = "Hello C++";

   cout << "Value of str is : " << str << endl;
}
```

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

```
Value of str is : Hello C++
```

The C++ compiler also determines the data type of variable to be output and selects the appropriate stream insertion operator to display the value. The << operator is overloaded to output data items of built-in types integer, float, double, strings and pointer values.

The insertion operator << may be used more than once in a single statement as shown above and **endl** is used to add a new-line at the end of the line.

The Standard Input Stream (cin)

The predefined object **cin** is an instance of **istream** class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The **cin** is used in conjunction with the stream extraction operator, which is written as >> which are two greater than signs as shown in the following example.

Live Demo

```
#include <iostream>

using namespace std;

int main() {
   char name[50];

   cout << "Please enter your name: ";
   cin >> name;
   cout << "Your name is: " << name << endl;

}
```

When the above code is compiled and executed, it will prompt you to enter a name. You enter a value and then hit enter to see the following result −

```
Please enter your name: cplusplus
Your name is: cplusplus
```

The C++ compiler also determines the data type of the entered value and selects the appropriate stream extraction operator to extract the value and store it in the given variables.

The stream extraction operator >> may be used more than once in a single statement. To request more than one datum you can use the following −

```
cin >> name >> age;
```

This will be equivalent to the following two statements −

```
cin >> name;
cin >> age;
```

The Standard Error Stream (cerr)

The predefined object **cerr** is an instance of **ostream** class. The cerr object is said to be attached to the standard error device, which is also a display screen but the object **cerr** is un-buffered and each stream insertion to cerr causes its output to appear immediately.

The **cerr** is also used in conjunction with the stream insertion operator as shown in the following example.

Live Demo

```
#include <iostream>

using namespace std;

int main() {
   char str[] = "Unable to read....";

   cerr << "Error message : " << str << endl;
}
```

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

```
Error message : Unable to read....
```

The Standard Log Stream (clog)

The predefined object **clog** is an instance of **ostream** class. The clog object is said to be attached to the standard error device, which is also a display screen but the object **clog** is buffered. This means that each insertion to clog could cause its output to be held in a buffer until the buffer is filled or until the buffer is flushed.

The **clog** is also used in conjunction with the stream insertion operator as shown in the following example.

Live Demo

```
#include <iostream>

using namespace std;

int main() {
   char str[] = "Unable to read....";

   clog << "Error message : " << str << endl;
}
```

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

```
Error message : Unable to read....
```

You would not be able to see any difference in cout, cerr and clog with these small examples, but while writing and executing big programs the difference becomes obvious. So it is good practice to display error messages using cerr stream and while displaying other log messages then clog should be used.

C++ 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 { } can not 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 −

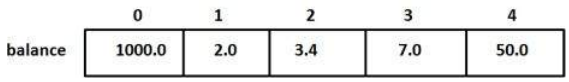
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

```
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 representaion 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 −

Live Demo

```
#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;

   // 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 −

Sr.No	Concept & Description
1	Multi-dimensional arrays C++ supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
2	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.
3	Passing arrays to functions You can pass to the function a pointer to an array by specifying the array's name without an index.
4	Return array from functions C++ allows a function to return an array.