Programming Fundamentals Using C++

Your first C++ program

- A C++ program is a collection of commands or statements.
- Example:

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

int main()
{
  cout << "Hello world!";
  return 0;
}
```

Let's break down the parts of the code:

#include <iostream>

- C++ offers various headers, each of which contains information needed for programs to work properly.
- This program calls for the header <iostream>.
- The number sign (#) at the beginning of a line targets the compiler's pre-processor.
- In this case, #include tells the pre-processor to include the <iostream> header.
- The <iostream> header defines the standard stream objects that input and output data.
- The C++ compiler ignores blank lines.
- In general, blank lines serve to improve the code's readability and structure.
- Whitespace, such as spaces, tabs, and newlines, is also ignored, although it is used to enhance the program's visual attractiveness.
-

using namespace std;

- In our code, the line using namespace std; tells the compiler to use the std (standard) namespace.
- The std namespace includes features of the C++ Standard Library.

Main

- Program execution begins with the main function, int main().
- Curly brackets { } indicate the beginning and end of a function, which can also be called the function's body.
- The information inside the brackets indicates what the function does when executed.
- The entry point of every C++ program is main(), irrespective of what the program does.

cout

- The next line, cout << "Hello world!"; results in the display of "Hello world!" to the screen.
- In C++, streams are used to perform input and output operations.
- **cout** is the used for the output operations.
- **cout** is used in combination with the insertion operator.
- Write the insertion operator as << to insert the data that comes after it into the stream that comes before.
- In C++, the semicolon is used to terminate a statement. Each statement must end with a semicolon. It indicates the end of one logical expression.

Statements

- A block is a set of logically connected statements, surrounded by opening and closing curly braces.
- For example:

```
{
cout << "Hello world!";
return 0;
}
```

- You can have multiple statements on a single line, if you remember to end each statement with a semicolon.
- Failing to do so will result in an error.

Return

- The last instruction in the program is the return statement.
- The line return 0; terminates the main() function and causes it to return the value 0 to the calling process.
- A non-zero value (usually of 1) signals abnormal termination.

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

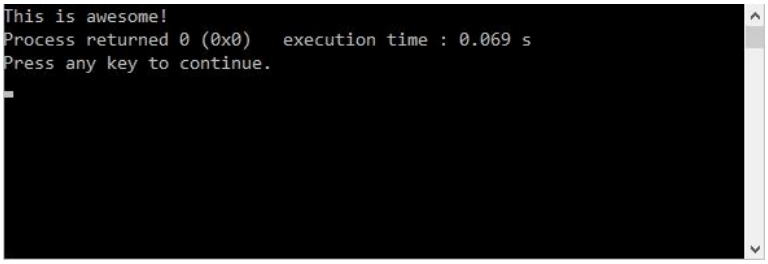
```
int main()
{
  cout << "Hello world!";
  return 0;
}
```

- If the return statement is left off, the C++ compiler implicitly inserts "return 0;" to the end of the main() function.
- **Note:** we can remove the (return 0) line from the code and still we get the same result.

Your First C++ Program

- You can add multiple insertion operators after cout.

```
cout << "This " << "is " << "awesome!";
```



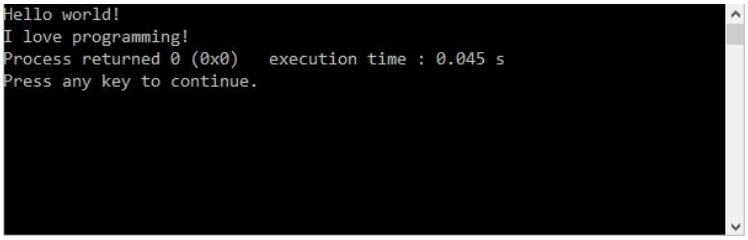
New Line

- The **cout** operator does not insert a line break at the end of the output.
- One way to print two lines is to use the **endl** manipulator, which will put in a line break.

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

int main()
{
cout << "Hello world!" << endl;
cout << "I love programming!";
return 0;
}
```

- The **endl** manipulator moves down to a new line to print the second text.
- **Result:**



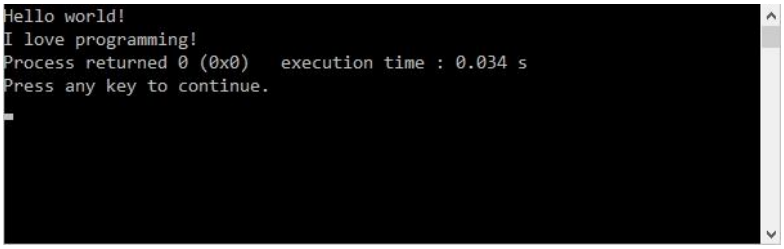
New Lines

- The new line character **\n** can be used as an alternative to **endl**.
- The backslash (\) is called an **escape character**, and indicates a "special" character.
- **Example:**

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

int main()
{
cout << "Hello world! \n";
cout << "I love programming!";
return 0;
}
```

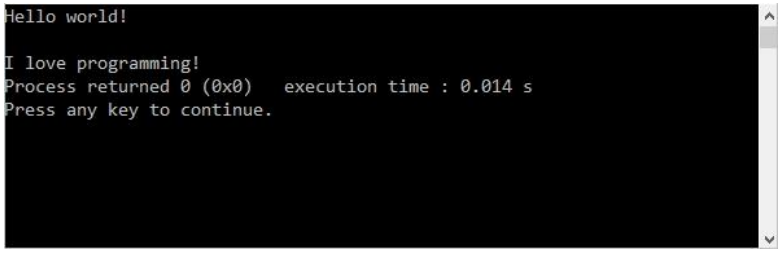
- **Result:**



Two newline characters placed together result in a blank line.

```
#include <iostream>
using namespace std;
int main()
{
cout << "Hello world! \n\n";
cout << "I love programming!";
return 0;
}
```

- **Result:**



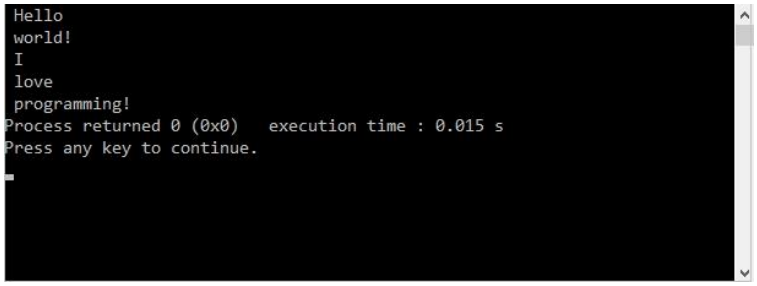
Multiple New Lines

- Using a single **cout** statement with as many instances of **\n** as your program requires will print out multiple lines of text.

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

int main()
{
cout << " Hello \n world! \n I \n love \n programming!";
return 0;
}
```

- **Result:**



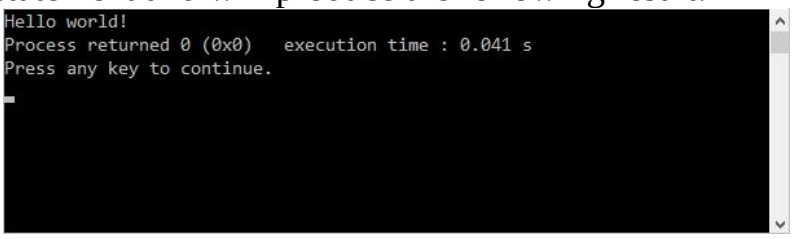
Comments

- **Comments** are explanatory statements that you can include in the C++ code to explain what the code is doing.
- The compiler ignores everything that appears in the comment, so none of that information shows in the result.
- A comment beginning with **two slashes (//)** is called a single-line comment.
- The slashes tell the compiler to ignore everything that follows, until the end of the line. **For example:**

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

```
int main()
{
// prints "Hello world"
cout << "Hello world!";
return 0;
}
```

- When the above code is compiled, it will ignore the **// prints "Hello world"** statement and will produce the following result:



Multi-Line Comments

- Comments that require multiple lines begin with **/*** and end with ***/**
- You can place them on the same line or insert one or more lines between them.

```
/* This is a comment */

/* C++ comments can
span multiple lines
*/
```

Using Comments

- Comments can be written anywhere and can be repeated any number of times throughout the code.
- Within a comment marked with /* and */, // characters have no special meaning, and vice versa. This allows you to "nest" one comment type within the other.

```
/* Comment out printing of Hello world!

cout << "Hello world!"; // prints Hello world!

*/
```

- Adding comments to your code is a good practice.
- It facilitates a clear understanding of the code for you and for others who read it.

Variables

- Creating a variable reserves a space in memory for storing values.
- Each variable must have a data type.
- C++ have various data types.

Integer

- Integer, a built-in type, represents a whole number value.
- Define integer using the keyword **int**.
- C++ requires that you specify the type and the identifier for each variable defined.
- An identifier is a name for a variable, function, class, module, or any other user-defined item.
- An identifier starts with a letter (A-Z or a-z) or an underscore (_), followed by additional letters, underscores, and digits (0 to 9).
- For example, define a variable called myVariable that can hold integer values as follows: **int myVariable = 10;**
- Now, let's assign a value to the variable and print it.

```
#include <iostream>
using namespace std;
int main()
{
int myVariable = 10;
cout << myVariable;
return 0;
}
// Outputs 10
```

- **Note:** the C++ programming language is case-sensitive, so myVariable and myvariable are two different identifiers.
- Define all variables with a **name** and a **data type** before using them in a program.
- In cases in which you have multiple variables of the same type, it's possible to define them in one declaration, separating them with **commas**.

```
int a, b; // defines two variables of type int
```

- A variable can be assigned a value and can be used to perform operations.
- For example, we can create an additional variable called **sum**, and add two variables together.

```
#include <iostream>
using namespace std;
int main()
{
int a = 30;
int b = 12;
int sum = a + b;
cout << sum;
return 0;
}
//Outputs 42
```

Declaring Variables

- You have the option to assign a value to the variable at the time you declare the variable or to declare it and assign a value later.
- You can also change the value of a variable.
- Specifying the data type is required just once, at the time when the variable is declared.
- After that, the variable may be used without referring to the data type.
- Specifying the data type for a given variable more than once results in a **syntax error**.
- A variable's value may be changed as many times as necessary throughout the program.
- Examples:

```
int a;
int b = 42;
a = 10;
b = 3;
```

User Input

- To enable the user to input a value, use **cin** in combination with the extraction operator (>>).
- The variable containing the extracted data follows the operator.
- The following example shows how to accept user input and store it in the num variable:

```
int num;
cin >> num;
```

- As with **cout**, extractions on **cin** can be chained to request more than one input in a single statement: **cin >> a >> b;**
- **Example 1:**

```
#include <iostream>
using namespace std;
int main()
{
int a, b;
cout << "Enter a number \n";
cin >> a;
cout << "Enter another number \n";
cin >> b;
return 0;
}
```

- **Example 2:**

```
#include <iostream>
using namespace std;
int main()
{
int a, b;
int sum;
cout << "Enter a number \n";
cin >> a;
cout << "Enter another number \n";
cin >> b;
sum = a + b;
cout << "Sum is: " << sum << endl;
return 0;
}
```

Arithmetic Operators

- C++ supports these arithmetic operators.

Operator	Symbol	Form
Addition	+	x + y
Subtraction	-	x - y
Multiplication	*	x * y
Division	/	x / y
Modulus	%	x % y

- The addition operator adds its operands together.
- Example:

```
int x = 40 + 60;
cout << x;

// Outputs 100
```

- Dividing by 0 will crash your program.
- The modulus operator (%) returns the remainder after an integer division.

Operator Precedence

- Operator **precedence** determines the grouping of terms in an expression, which affects how an expression is evaluated.
- Certain operators take higher precedence over others; for example, the multiplication operator has higher precedence over the addition operator.
- Example:

```
int x = 5+2*2;
cout << x;
// Outputs 9
```

- The program evaluates 2*2 first, and then adds the result to 5.
- As in mathematics, using **parentheses** alters operator precedence.

```
int x = (5 + 2) *2;
cout << x;
// Outputs 14
```

- Parentheses force the operations to have higher precedence.
- If there are parenthetical expressions nested within one another, the expression within the innermost parentheses is evaluated first.
- If none of the expressions are in parentheses, multiplicative (multiplication, division, modulus) operators will be evaluated before additive (addition, subtraction) operators.

Assignment Operators

- The simple **assignment** operator (=) assigns the right side to the left side.
- C++ provides shorthand operators that have the capability of performing an operation and an assignment at the same time.
- **Example:**

```
int x = 10;
x += 4; // equivalent to x = x + 4
x -= 5; // equivalent to x = x – 5
```

- The same shorthand syntax applies to the multiplication, division, and modulus operators.

```
x *= 3; // equivalent to x = x * 3
x /= 2; // equivalent to x = x / 2
x %= 4; // equivalent to x = x % 4
```

- The increment operator is used to increase an integer's value by one and is a commonly used C++ operator.

```
x++; //equivalent to x = x + 1
```

- Example:

```
int x = 11;
x++;
cout << x;

//Outputs 12
```

- The increment operator has two forms, prefix and postfix.

```
++x; //prefix
x++; //postfix
```

- **Prefix** increments the value, and then proceeds with the expression.
- **Postfix** evaluates the expression and then performs the incrementing.
- Prefix example:

```
x = 5;
y = ++x;
// x is 6, y is 6
```

- Postfix example:

```
x = 5;
y = x++;
// x is 6, y is 5
```

- The prefix example increments the value of x, and then assigns it to y.
- The postfix example assigns the value of x to y, and then increments it.

Decrement Operator

- The decrement operator (--) works in much the same way as the increment operator, but instead of increasing the value, it decreases it by one.

```
--x; // prefix
x--; // postfix
```

Data Types

- The operating system allocates memory and selects what will be stored in the reserved memory based on the variable's data type.
- The data type defines the proper use of an identifier, what kind of data can be stored, and which types of operations can be performed.
- There are several built-in types in C++.

Expressions

- The examples below show legal and illegal C++ expressions.
- 55+15 // **legal** C++ expression (Both operands of the + operator are integers)
- 55 + "John" // **illegal** (The + operator is not defined for integer and string)
- "Hello," + "John" // **legal** (The + operator is used for string concatenation)

Numeric Data Types

- Numeric data types include:
  - Integers (whole numbers), such as -7, 42.
  - Floating point numbers, such as 3.14, -42.67.
  - Strings & Characters
  - Booleans

Integer

- **The integer** type holds non-fractional numbers, which can be positive or negative.
- **Examples:** 42, -42, and similar numbers.
- The size of the integer type varies according to the architecture of the system on which the program runs, although 4 bytes is the minimum size in most modern system architectures.
- Use the int keyword to define the integer data type. **int a = 42;**
- Several of the basic types, including integers, can be modified using one or more of these type modifiers:
  - **signed:** can hold both negative and positive numbers.
  - **unsigned:** can hold only positive values.
  - **short:** Half of the default size.

  - **long:** Twice the default size.
- Example: **unsigned long int a;**

Floating Point Numbers

- A floating-point type variable can hold a real number, such as 420.0, -3.33, or 0.03325.
- There are three different floating-point data types: float, double, and long double.
- In most modern architectures, a float is 4 bytes, a double is 8, and a long double can be equivalent to a double (8 bytes), or 16 bytes.
- For example: double temp = 4.21;
- Floating point data types are always signed, which means that they have the capability to hold both positive and negative values.

Strings

- **A string** is composed of numbers, characters, or symbols.
- String literals are placed in double quotation marks.
- **Examples:** "Hello", "My name is David".
- You need to include the <string> library to use the string data type.
- You don't need to include <string> separately, if you already use <iostream>.

```
#include <string>
using namespace std;
int main() {
string a = "I am learning C++";
return 0;
}
```

Characters

- **Characters** are single letters or symbols, and must be enclosed between single quotes, like 'a', 'b', etc.
- In C++, single quotation marks indicate a character;
- double quotes create a string literal.
- While 'a' is a single a character literal, "a" is a string literal.
- A char variable holds a 1-byte integer.
- However, instead of interpreting the value of the char as an integer, the value of a char variable is typically interpreted as an ASCII character.
- **Example:** char test = 'S';

- American Standard Code for Information Interchange (ASCII) is a character-encoding scheme that is used to represent text in computers.

Booleans

- **Boolean** data type returns just two possible values: true (1) and false (0).
- To declare a boolean variable, we use the keyword **bool**.
- bool online = false;
- bool logged_in = true;
- Conditional expressions are an example of Boolean data type.

Variable Naming Rules

- Use the following rules when naming variables:
    - All variable names must begin with a letter of the alphabet or an underscore(_).
    - After the initial letter, variable names can contain additional letters, as well as numbers.
    - Blank spaces or special characters are not allowed in variable names.
- C++ keyword (reserved word) cannot be used as variable names. Example, int, float, double, cout.
- There are two known naming conventions:
    - **Pascal case:** The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. For example: BackColor
    - **Camel case:** The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized. For example: backColor

Case-Sensitivity

- C++ is case-sensitive, which means that an identifier written in uppercase is not equivalent to another one with the same name in lowercase.
- For example, **myvariable** is not the same as **MYVARIABLE** and not the same as **MyVariable**.
- *These are three different variables.*
- Choose variable names that suggest the usage, for example: firstName, lastName.

Conditionals and loops

The if Statement

- The **if** statement is used to execute some code if a condition is true.

Syntax:

```
if (condition) {
//statements
}
```

- The **condition** specifies which expression is to be evaluated.
- If the condition is **true**, the statements in the curly brackets are executed.
- If the condition is **false**, the statements in the curly brackets are ignored, and the program continues to run after the if statements body.
- Use **relational operators** to evaluate conditions.
- **Example:**

```
if (7 > 4) {
cout << "Yes";
}
// Outputs "Yes"
```

- The **if** statement evaluates the condition (7>4), finds it to be **true**, and then executes the **cout** statement.
  If we change the greater operator to a less than operator (**7<4**), the statement will
- Note: A condition specified in an if statement does not require a semicolon.

Relational operators:

Operator	Description	Example
>=	Greater than or equal to	7 >= 4 True
<=	Less than or equal to	7 <= 4 False
==	Equal to	7 == 4 False
!=	Not equal to	7 != 4 True

Example:

```
if (10 == 10) {
cout << "Yes";
}
// Outputs "Yes"
```

- The **not equal to** operator evaluates the operands, determines whether they are equal or not.
- If the operands are not equal, the condition is evaluated to **true**.
- **Example:**

```
if (10 != 10) {
cout << "Yes";
}
```

- The above condition evaluates to **false** and the block of code is not executed.
- You can use relational operators to compare variables in the **if** statement.
- **Example:**

```
int a = 55;
int b = 33;
if (a > b) {
cout << "a is greater than b";
}

// Outputs "a is greater than b"
```

The else Statement

- An **if** statement can be followed by an optional **else** statement, which executes when the condition is **false**.

Syntax:

```
if (condition) {
//statements
}
else {
//statements
}
```

- The compiler will test the condition:
  - If it evaluates to **true**, then the code inside the **if** statement will be executed.
  - If it evaluates to **false**, then the code inside the **else** statement will be executed.
- When only **one** statement is used inside the if/else, then the curly braces can be omitted.
- **Example:**

```
int mark = 90;
if (mark < 50) {
cout << "You failed." << endl;
}
else {
cout << "You passed." << endl;
}

// Outputs "You passed."
```

- In all previous examples only one statement was used inside the if/else statement, but you may include as many statements as you want.
- **Example:**

```
int mark = 90;
if (mark < 50) {
cout << "You failed." << endl;
cout << "Sorry" << endl;
}
else {
cout << "Congratulations!" << endl;
cout << "You passed." << endl;
cout << "You are awesome!" << endl;
}
/* Outputs
Congratulations!
You passed.
You are awesome!
*/
```

Nested if Statements

- You can also include, or **nest**, if statements within another if statement.
- **Example:**

```
int mark = 100;
if (mark >= 50) {
cout << "You passed." << endl;
if (mark == 100) {
cout <<"Perfect!" << endl;
}
}
else {
cout << "You failed." << endl;
```

```
}

/*Outputs
You passed.
Perfect!
*/
```

- C++ provides the option of nesting an unlimited number of if/else statements.
- **Example:**

```
int age = 18;
if (age > 14) {
if(age >= 18) {
cout << "Adult";
}
else {
cout << "Teenager";
}
}
else {
if (age > 0) {
cout << "Child";
}
else {
cout << "Something's wrong";
}
}
```

- Remember that all **else** statements must have corresponding **if** statements.
- In if/else statements, **a single statement** can be included without enclosing it into curly braces.
- Example:

```
int a = 10;
if (a > 4)
cout << "Yes";
else
cout << "No";
```

The while Loop

- A **loop** repeatedly executes a set of statements until a particular condition is satisfied.
- A **while** loop statement repeatedly executes a target statement as long as a given condition remains **true**.

Syntax:

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

- The loop iterates while the condition is **true**.
- At the point when the condition becomes **false**, program control is shifted to the line that immediately follows the loop.
- The loop's **body** is the block of statements within curly braces.
- **Example:**

```
int num = 1;
while (num < 6) {
cout << "Number: " << num << endl;
num = num + 1;
}

/* Outputs
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
*/
```

- The example above declares a variable equal to 1 (int num = 1).
- The **while** loop checks the condition (num < 6), and executes the statements in its body, which increment the value of **num** by one each time the loop runs.
- After the 5th iteration, **num** becomes 6, and the condition is evaluated to **false**, and the loop stops running.
- The increment value can be changed.
- If changed, the number of times the loop is run will change, as well.

```
int num = 1;
while (num < 6) {
cout << "Number: " << num << endl;
num = num + 3;
}

/* Outputs
Number: 1
Number: 4
*/
```

- Without a statement that eventually evaluates the loop condition to **false**, the loop will continue indefinitely.
- The increment or decrement operators are used to change values in the loop.
- **Example:**

```
int num = 1;
while (num < 6) {
cout << "Number: " << num << endl;
num++;
}
/* Outputs
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
*/
```

- num++ is equivalent to num = num + 1.
- A loop can be used to obtain multiple inputs from the user.
- Let's create a program that allows the user to enter a number 5 times, each time storing the input in a variable.

```
int num = 1;
int number;
while (num <= 5) {
cin >> number;
```

```
num++;
}
```

- The above code asks for user input 5 times, and each time saves the input in the **number** variable.
- Now let's modify our code to calculate the sum of the numbers the user has entered.

```
int num = 1;
int number;
int total = 0;
while (num <= 5) {
cin >> number;
total += number;
num++;
}
cout << total << endl;
```

- The code above adds the number entered by the user to the **total** variable with each loop iteration.
- Once the loop stops executing, the value of **total** is printed.
- This value is the sum of all the numbers the user entered.
- Note that the variable **total** has an initial value of 0.

The for loop

- A **for** loop is a repetition control structure that allows you to efficiently write a loop that executes a specific number of times.

Syntax:

```
for ( init; condition; increment ) {
statement(s);
}
```

- The **init** step is executed first and does not repeat.
- Next, the **condition** is evaluated, and the body of the loop is executed if the condition is true.
- In the next step, the **increment** statement updates the loop control variable.
- Then, the loop's body repeats itself, only stopping when the condition becomes **false**.
- **Example:**

```
for (int x = 1; x < 10; x++) {
// some code
}
```

- The **init** and **increment** statements may be left out, if not needed, but remember that the **semicolons** are mandatory.
- The example below uses a **for** loop to print numbers from 0 to 9.

```
for (int a = 0; a < 10; a++) {
cout << a << endl;
}

/* Outputs
0
1
2
3
4
5
6
7
8
9
*/
```

- In the **init** step, we declared a variable **a** and set it to equal 0.
  a < 10 is the **condition**.
- After each iteration, the a++ **increment** statement is executed.
- When **a** increments to 10, the condition evaluates to **false**, and the loop stops.
- It's possible to change the increment statement.

```
for (int a = 0; a < 50; a+=10) {
cout << a << endl;
}
/* Outputs
0
10
20
30
```

```
40
*/
```

- You can also use decrement in the statement.

```
for (int a = 10; a >= 0; a -= 3) {
cout << a << endl;
}

/* Outputs
10
7
4
1
*/
```

- When using the for loop, don't forget the **semicolon** after the **init** and **condition** statements.

The do...while Loop

- Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop checks its condition at the bottom of the loop.
- A **do...while** loop is like a **while** loop. The one difference is that the **do...while** loop is guaranteed to execute **at least one time**.

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

- Example:

```
int a = 0;
do {
cout << a << endl;
a++;
} while(a < 5);

/* Outputs
0
1
2
```

```
3
4
*/
```

- Don't forget the **semicolon** after the while statement.

while vs. do...while

- If the condition evaluated to **false**, the statements in the **do** would still run once:

```
int a = 42;
do {
cout << a << endl;
a++;
} while(a < 5);

// Outputs 42
```

- The **do...while** loop executes the statements at least once, and then tests the condition.
- The **while** loop executes the statement after testing condition.

The do...while Loop

- As with other loops, if the condition in the loop never evaluates to **false**, the loop will run forever.
- Example:

```
int a = 42;
do {
cout << a << endl;
} while (a > 0);
```

- This will print 42 to the screen **forever**.
- Always test your loops, so you know that they operate in the manner you expect.

Multiple Conditions

- Sometimes there is a need to test a variable for equality against multiple values.
- That can be achieved using multiple if statements.
- **Example:**

```
int age = 42;
if (age == 16) {
cout <<"Too young";
}
if (age == 42) {
cout << "Adult";
}
if (age == 70) {
cout << "Senior";
}
```

- The **switch** statement is a more elegant solution in this scenario.

The switch Statement

- The **switch** statement tests a variable against a list of values, which are called **cases**, to determine whether it is equal to any of them.

```
switch (expression) {
case value1:
statement(s);
break;
case value2:
statement(s);
break;
...
case valueN:
statement(s);
break;
}
```

- Switch evaluates the expression to determine whether it's equal to the value in the case statement.
- If a match is found, it executes the statements in that case.
- A switch can contain any number of **case** statements, which are followed by the **value** in question and a **colon**.
- Here is the previous example written using a single **switch** statement:

```
int age = 42;
switch (age) {
case 16:
cout << "Too young";
```

```
break;
case 42:
cout << "Adult";
break;
case 70:
cout << "Senior";
break;
}
```

- The code above is equivalent to three **if** statements.
- Notice the keyword **break;** that follows each case. That will be covered shortly.

The default Case

- In a switch statement, the optional **default** case can be used to perform a task when none of the cases is determined to be true.
- **Example:**

```
int age = 25;
switch (age) {
case 16:
cout << "Too young";
break;
case 42:
cout << "Adult";
break;
case 70:
cout << "Senior";
break;
default:
cout << "This is the default case";
}

// Outputs "This is the default case"
```

- The **default** statement's code executes when none of the cases matches the switch expression.
- The **default** case must appear at the end of the switch.

The break Statement

- The **break** statement's role is to terminate the switch statement.
- In instances in which the variable is equal to a case, the statements that come after the case continue to execute until they encounter a **break** statement.
- In other words, leaving out a **break** statement results in the execution of all of the statements in the following cases, even those that don't match the expression.
- **Example:**

```
int age = 42;
switch (age) {
case 16:
cout << "Too young" << endl;
case 42:
cout << "Adult" << endl;
case 70:
cout << "Senior" << endl;
default:
cout <<"This is the default case" << endl;
}
/* Outputs
Adult
Senior
This is the default case
*/
```

- As you can see, the program executed the matching case statement, printing "Adult" to the screen.
- With no specified **break** statement, the statements continued to run after the matching case.
- Thus, all the other case statements printed.
- This type of behavior is called **fall-through**.
- As the switch statement's final case, the **default** case requires no **break** statement.
- The **break** statement can also be used to break out of a loop.

Logical Operators

- Use **logical operators** to combine conditional statements and return **true** or **false**.

Operator	Name of Operator	Form
&&	**AND** Operator	y && y
||	**OR** Operator	x || y
!	**NOT** Operator	! x

- The **AND** operator works the following way:

Left Operand	Right Operand	Result
false	false	**false**
false	true	**false**
true	false	**false**
true	true	**true**

- In the AND operator, both operands must be **true** for the entire expression to be **true**.
- **Example:**

```
int age = 20;
if (age > 16 && age < 60) {
cout << "Accepted!" << endl;
}

// Outputs "Accepted"
```

- In the example above, the logical AND operator was used to combine both expressions.
- The expression in the if statement evaluates to **true** only if both expressions are **true**.
-

Within a single if statement, logical operators can be used to combine **multiple** conditions.

```
int age = 20;
int grade = 80;
if (age > 16 && age < 60 && grade > 50) {
cout << "Accepted!" << endl;
}
```

- The entire expression evaluates to **true** only if all of the conditions are **true**.

The OR Operator

- The **OR** (||) operator returns true if any one of its operands is **true**.

Left Operand	Right Operand	Result
false	false	**false**
false	true	**true**
true	false	**true**
true	true	**true**

- Example:

```
int age = 16;
int score = 90;
if (age > 20 || score > 50) {
cout << "Accepted!" << endl;
}

// Outputs "Accepted!"
```

- You can combine any number of logical **OR** statements you want.
- In addition, multiple **OR** and **AND** statements may be chained together.

Logical NOT

- The logical **NOT** (!) operator works with just a single operand, reversing its logical state.
- Thus, if a condition is **true**, the NOT operator makes it **false**, and vice versa.

Right Operand	Result
true	**false**
false	**true**

```
int age = 10;
if ( !(age > 16) ) {
cout << "Your age is less than 16" << endl;
}

// Outputs "Your age is less than 16"
```

Functions

- A **function** is a group of statements that perform a particular task.
- You may define your own functions in C++.
- Using functions can have many advantages, including the following:
  - You can reuse the code within a function.
  - You can easily test individual functions.
  - If it's necessary to make any code modifications, you can make modifications within a single function, without altering the program structure.
  - You can use the same function for different inputs.
- Every valid C++ program has at least one function - the **main()** function.

The Return Type

- The **main** function takes the following general form:

```
int main()
{
// some code
return 0;
}
```

- A function's **return type** is declared before its name.
- In the example above, the return type is **int**, which indicates that the function returns an integer value.
- Occasionally, a function will perform the desired operations without returning a value.
- Such functions are defined with the keyword **void**.
- **void** is a basic data type that defines a valueless state.

Defining a Function

- Define a C++ function using the following syntax:

```
return_type function_name( parameter list )
{
body of the function
}
```

- **return-type**: Data type of the value returned by the function.
- **function name**: Name of the function.

- **parameters**: 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.
- **body of the function**: A collection of statements defining what the function does.
- Parameters are **optional**; that is, you can have a function with no parameters.
- As an example, let's define a function that does not return a value, and just prints a line of text to the screen.

```
void printSomething()
{
cout << "Hi there!";
}
```

- Our function, entitled **printSomething**, returns **void**, and has no parameters.
- Now, we can use our function in **main()**.

```
int main()
{
printSomething();
return 0;
}
```

- To **call** a function, you simply need to pass the required parameters along with the function name.

Functions

- You must declare a function prior to calling it.
- Example:

```
#include <iostream>
using namespace std;
void printSomething() {
cout << "Hi there!";
}

int main() {
printSomething();
```

```
return 0;
}
```

- Putting the declaration after the **main()** function results in an error.
- A function **declaration**, or **function prototype**, tells the compiler about a function name and how to call the function.
- The actual body of the function can be defined separately.
- **Example:**

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

//Function declaration
void printSomething();

int main() {
printSomething();

return 0;
}

//Function definition
void printSomething() {
cout << "Hi there!";
}
```

- 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.

Function Parameters

- For a function to use **arguments**, it must declare formal **parameters**, which are variables that accept the argument's values.
- **Argument:** a piece of data that is passed into a function or a program.
- Example:

```
void printSomething(int x)
{
```

```
cout << x;
}
```

- This defines a function that takes one **integer** parameter and prints its value.
- Formal parameters behave within the function similarly to other local variables.
- They are created upon entering the function and are destroyed upon exiting the function.
- Once parameters have been defined, you can pass the corresponding arguments when the function is called.
- **Example:**

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

void printSomething(int x) {
cout << x;
}

int main() {
printSomething(42);
}

// Outputs 42
```

- The value 42 is passed to the function as an **argument** and is assigned to the formal **parameter** of the function: **x**.
- Making changes to the parameter within the function does not alter the argument.
- You can pass different arguments to the same function.
- For example:

```
int timesTwo(int x) {
return x*2;
}
```

- The function defined above takes one integer parameter and returns its value, multiplied by 2.
- We can now use that function with different arguments.

```
int main() {
cout << timesTwo(8);
// Outputs 16
```

```
cout <<timesTwo(5);
// Outputs 10

cout <<timesTwo(42);
// Outputs 84
}
```

Functions with Multiple Parameters

- You can define as many parameters as you want for your functions, by separating them with **commas**.
- Let's create a simple function that returns the sum of two parameters.

```
int addNumbers(int x, int y) {
// code goes here
}
```

- As defined, the **addNumbers** function takes two parameters of type **int**, and returns **int**.
- **Data type** and **name** should be defined for each parameter.
- Now let's calculate the sum of the two parameters and return the result:

```
int addNumbers(int x, int y) {
int result = x + y;
return result;
}
```

- Now we can call the function.

```
int addNumbers(int x, int y) {
int result = x + y;
return result;
}
int main() {
cout << addNumbers(50, 25);
// Outputs 75
}
```

- You can also assign the returned value to a variable.

```
int main() {
int x = addNumbers(35, 7);
cout << x;
// Outputs 42
}
```

- You can add as many parameters to a single function as you want.
- **Example:**

```
int addNumbers(int x, int y, int z, int a) {
int result = x + y + z + a;
return result;
}
```

- If you have multiple parameters, remember to separate them with **commas**, both when declaring them and when passing the arguments.

C++ Examples with Solutions

- Write a C++ program to input a number from the keyboard. Then, you find and print its double.

```
#include <iostream>
using namespace std;
int main ()
{
  int i;
  cout << "Please enter an integer value: ";
  cin >> i;
  cout << "The value you entered is " << i;
  cout << " and its double is " << i*2 << ".\n";
  return 0;
}
```

- Write a C++ program to perform a countdown using a while-loop.

```
#include <iostream>
```

```
using namespace std;
int main ()
{
  int n = 10;

  while (n>0) {
    cout << n << ", ";
    --n;
  }
  cout << "liftoff!\n";
}
```

- Write the same above C++ program using for loop.

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

int main ()
{
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
  cout << "liftoff!\n";
}
```

- Write a C++ program which has a function that adds any two integers. Then, print the result.

```
#include <iostream>
using namespace std;
int addition (int a, int b)
{
  int r;
  r=a+b;
  return r;
}
int main ()
{
  int z;
  z = addition (5,3);
```

```
cout << "The result is " << z;}
```