YOBE STATE UNIVERSITY DAMATURU

FACULTY OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

LECTURES NOTE

COURSE CODE: CSC 2302

COURSE TITLE: COMPUTER PROGRAMMING USING C++

COURSE LECTURER: YA'U NUHU

EMAIL: yaunuhu20@gmail.com

Before We Begin

Before we begin the actual coding part, we want to clarify a few things.

1. Fear of Coding

Programming may feel intimidating for beginners. However, always remember that learning to code is just another skill. If you give time, you will learn it for sure.

2. Programming is a Language

Similar to languages like English, programming is also a language to interact with computers.

You will just need to remember a few instructions and rules. And, you can use the same instructions (syntax) to create different programs.

3. Math for Programming

Learning to code involves a lot of logic and trial-and-error. However, nothing beyond basic arithmetic is required.

INTRODUCTION

C++ is a general-purpose, high-level programming language. It was developed by Bjarne Stroustrup as an extension of the C programming language, and it retains most of C's syntax. However, C++ adds object-oriented features to C, making it a more powerful and flexible language.

The main difference between C and C++ is that C++ support classes and objects, while C does not.

C++ is used to develop a wide variety of software, including operating systems, browsers, games, and embedded systems. It is a popular language for systems programming because it gives programmers a high level of control over system resources and memory.

Here are some of the key features of C++

- Performance: C++ is a compiled language, which means that it is converted into machine code before it is executed. This makes C++ programs very fast.
- Power: C++ is a powerful language that can be used to create complex and sophisticated software.
- Flexibility: C++ supports a variety of programming paradigms, including procedural, object-oriented, and generic programming.
- Portability: C++ code can be compiled and run on a variety of platforms.
- Object-oriented programming: C++ supports object-oriented programming, which allows programmers to create reusable code and modularize their programs.
- Low-level control: C++ gives programmers a high level of control over system resources and memory, which makes it a good choice for systems programming.
- **Fast and efficient**: C++ is a compiled language, which means that it is converted into machine code before it is executed. This makes C++ programs very fast.

C++ is considered an object-oriented programming language because it supports the four pillars of OOP:

- **Encapsulation:** Encapsulation is the bundling of data and code into a single unit called an object. This allows for data to be hidden from the outside world, making it more secure and easier to maintain.
- **Abstraction:** Abstraction is the process of hiding the implementation details of an object from the user. This allows the user to focus on the object's behavior, rather than how it works.
- **Inheritance:** Inheritance is the ability for one object to inherit the properties and methods of another object. This allows for code reuse and makes it easier to create complex objects.
- **Polymorphism:** Polymorphism is the ability for an object to behave differently depending on its context. This allows for code to be more flexible and adaptable.

In addition to these four pillars, C++ also supports other OOP features, such as classes, objects,

Here are some specific examples of where C++ is used:

- **Operating systems:** Many operating systems, including Windows, macOS, and Linux, are written in C++.
- **Browsers:** The web browsers Chrome, Firefox, and Safari are all written in C++.
- **Games:** Many popular games, such as Counter-Strike, Dota 2, and World of Warcraft, are written in C++.
- **Embedded systems:** C++ is often used to develop embedded systems, such as those found in cars, smartphones, and medical devices.

If you are interested in a career in software development, learning C++ is a great way to get started. C++ is a powerful and versatile language that can be used to create a wide variety of software. It is also a well-respected language in the industry, so learning C++ will make you a more marketable candidate for jobs.

C++ Get Started

To start using C++, you need two things:

- A text editor, like Notepad, to write C++ code
- A compiler, like GCC, to translate the C++ code into a language that the computer will understand

There are many text editors and compilers to choose from. For COM 313, we will use an IDE (see below).

C++ Install IDE

An IDE (Integrated Development Environment) is used to edit AND compile the code.

Popular IDE's include Code::Blocks, Eclipse, and Visual Studio. These are all free, and they can be used to both edit and debug C++ code.

Note: Web-based IDE's can work as well, but functionality is limited.

We will use **Code::Blocks** in our tutorial, which we believe is a good place to start.

You can find the latest version of Codeblocks at http://www.codeblocks.org/.

Download the mingw-setup.exe file, which will install the text editor with a compiler.

C++ Quickstart

Let's create our first C++ file.

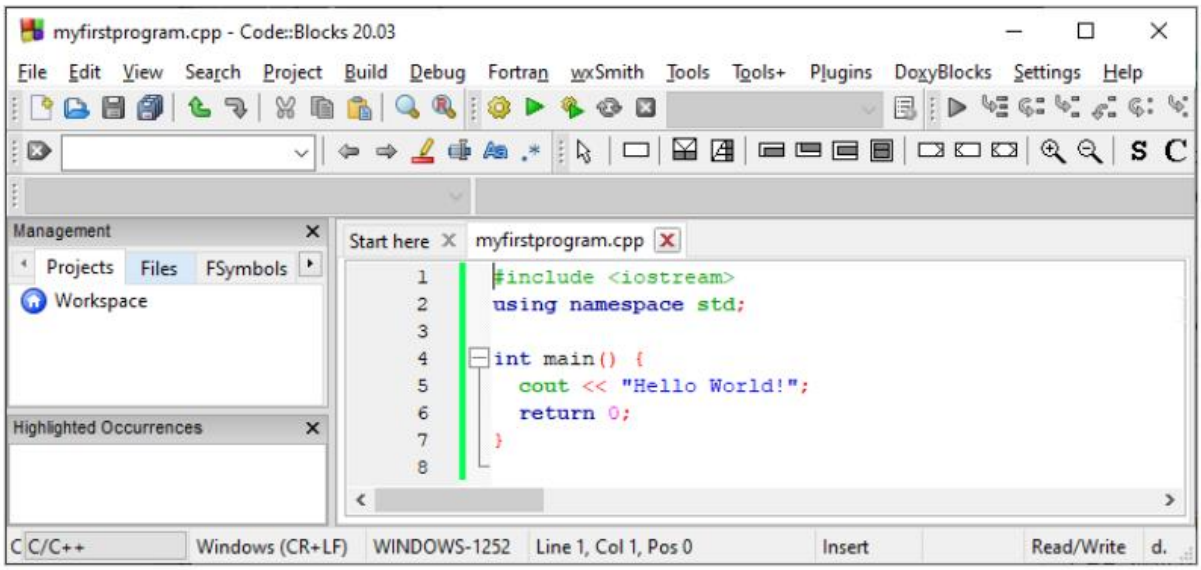
Open Codeblocks and go to **File > New > Empty File**.

Write the following C++ code and save the file as myfirstprogram.cpp (**File > Save File as**):

myfirstprogram.cpp

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

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

Don't worry if you don't understand the code above - we will discuss it in detail in later chapters. For now, focus on how to run the code.
In **Codeblocks,** it should look like this:



Then, go to **Build > Build and Run** to run (execute) the program. The result will look something to this:

```
Hello World!
Process returned 0 (0x0) execution time : 0.011 s
Press any key to continue.
```

Congratulations! You have now written and executed your first C++ program.

Data Types and Variable IN C++

Data types are used to define the size and format of data, and they also determine the operations that can be performed on the data.

There are two main types of data types in C++:

- Primitive data types are the basic building blocks of data in C++. They are predefined by the language, and they have a fixed size and format. The most common primitive data types in C++ are:

- int: Integers are whole numbers, without decimals.
- float: Floating-point numbers are numbers with decimals.
- double: Double-precision floating-point numbers are more precise than floats.
- char: Characters are single letters or symbols.
- bool: Boolean values can be either true or false.

- Derived data types are data types that are created from primitive data types. They can be used to store more complex data, such as arrays, strings, and structures.

  Here is a table of the primitive data types in C++, along with their size and range:

Data type	Size	Range
int	4 bytes	-2147483648 to 2147483647
float	4 bytes	3.402823466e+38 to 1.175494351e-38
double	8 bytes	1.7976931348623157e+308 to 2.2250738585072014e-308
char	1 byte	0 to 255
bool	1 byte	true or false

Here is an example of how data types are used in C++:

```
int my_number = 10;
float my_float = 1.23;
char my_character = 'a';
bool my_boolean = true;
```

In this example, the variable my_number is an integer, the variable my_float is a floating-point number, the variable my_character is a character, and the variable my_boolean is a Boolean value.

Data types are an important part of C++, and they are essential for storing and manipulating data in C++ programs.

A variable in C++ is a named location in memory that stores a value. Variables are used to store data that can be changed during the execution of a program.

Variables are declared in C++ using the var_name data_type syntax. For example, the following code declares a variable named my_number of type int:

```
int my_number;
```

The data_type of a variable determines the size and format of the data that can be stored in the variable. For example, an int variable can store a whole number, a float variable can store a number with decimals, and a char variable can store a single character.

Once a variable is declared, it can be assigned a value using the var_name = value syntax. For example, the following code assigns the value 10 to the variable my_number:

```
int my_number;
my_number = 10;
```

Variables can be used to store data of any type, including primitive data types, derived data types, and user-defined data types.

Here are some of the benefits of using variables:

- Variables make code more readable and maintainable. Instead of having to hard-code values throughout the code, variables can be used to store values that can be changed as needed. This makes the code more readable and easier to maintain.
- Variables allow for code reuse. Once a variable has been declared and assigned a value, it can be used in other parts of the code. This allows for code reuse and makes the code more efficient.
- Variables can be used to store complex data. Derived data types, such as arrays, strings, and structures, can be used to store complex data. This allows for more flexibility and power in C++ programs.

Here are some of the rules for declaring variables in C++:

- Variable names must start with a letter or an underscore.
- Variable names can contain letters, numbers, and underscores.
- Variable names must be unique within the scope of the variable.
- Variable names are case-sensitive.

Here are some valid and invalid C++ names:

```
int poodle;     // valid
int Poodle;     // valid and distinct from poodle
int POODLE;     // valid and even more distinct
Int terrier;    // invalid -- has to be int, not Int
int my_stars3   // valid
int _Mystars3; // valid but reserved -- starts with underscore
int 4ever;      // invalid because starts with a digit
int double;     // invalid -- double is a C++ keyword
int begin;      // valid -- begin is a Pascal keyword
int __fools;    // valid but reserved - starts with two underscores
int the_very_best_variable_i_can_be_version_112;  // valid
int honky-tonk;         // invalid -- no hyphens allowed
```

If you want to form a name from two or more words, the usual practice is to separate the words with an underscore character, as in my_onions, or to capitalize the initial character of each word after the first, as in myEyeTooth.

FLOATIN POINT OBJECT TYPES AND OPERATOR FOR FUNDAMENTAL TYPES

A float is a data type that can be used to store floating-point numbers. Floating-point numbers are numbers that have a decimal point and can be represented in a variety of ways, such as scientific notation.

The float data type is a primitive data type, which means that it is predefined by the C++ language. The size and format of a float variable is determined by the compiler. However, in general, a float variable is 4 bytes long and can store numbers with a precision of up to 7 decimal digits.

```
#include <iostream>
using namespace std;
int main()              {
float my_float = 3.14159;
 point my_point = {1.0, 2.0};

        cout << "The value of my_float is: " << my_float << endl;
        cout << "The x-coordinate of my_point is: " << my_point.x << endl;
        cout << "The y-coordinate of my_point is: " << my_point.y << endl;

return 0;
}
```

This code will print the following output:
The value of my_float is: 3.14159
The x-coordinate of my_point is: 1.0
The y-coordinate of my_point is: 2.0

FOUNDAMENTAL TYPES

Fundamental types are the basic building blocks of data. They are predefined by the language and have a fixed size and format. The most common fundamental types are:

- int: Integers are whole numbers, without decimals.
- float: Floating-point numbers are numbers with a decimal point.
- double: Double-precision floating-point numbers are more precise than floats.
- char: Characters are single letters or symbols.
- bool: Boolean values can be either true or false.

STRUCTURE OF A C++ PROGRAM

C++

```
#include <iostream>

using namespace std;

int main() {
  // Your code goes here

  return 0;
}
```

Example 1.

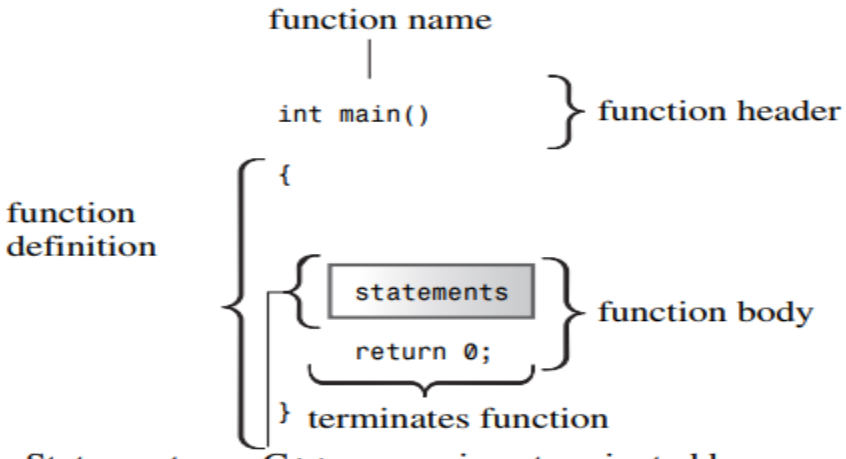
```
// myfirst.cpp--displays a message

#include <iostream>                          // a PREPROCESSOR directive
int main()                                   // function header
{                                            // start of function body
    using namespace std;                     // make definitions visible
    cout << "Come up and C++ me some time."; // message
    cout << endl;                            // start a new line
    cout << "You won't regret it!" << endl;  // more output
    return 0;                                // terminate main()
}                                            // end of function body
```

Note: if you're using an older compiler, you might need to use #include <iostream.h> instead of #include <iostream>; in that case, you also would omit the using namespace std; line. That is, you'd replace

The myfirst.cpp example has the following elements:

- Comments, indicated by the // prefix
- A preprocessor **#include** directive
- A function header: **int main()**
- A using namespace directive
- A function body, delimited by **{ and }**
- Statements that uses the C++ **cout** facility to display a message
- A return statement to terminate the **main() function**

The main() Function



The final statement in main(), called a return statement, terminates the function.

Statements and Semicolons

A statement represents a complete instruction to a computer. To understand your source code, a compiler needs to know when one statement ends and another begins. The practical upshot is that in C++ you should never omit the semicolon.

C++ Comments

The double slash (//) introduces a C++ comment. A comment is a remark from the programmer to the reader that usually identifies a section of a program or explains some aspect of the code. The compiler ignores comments.

This example uses a single-line comment before a line of code:

Example

```
// This is a comment
cout << "Hello World!";
```

This example uses a single-line comment at the end of a line of code:

Example

```
cout << "Hello World!"; // This is a comment
```

C++ Multi-line Comments

Multi-line comments start with /* and ends with */.

Any text between /* and */ will be ignored by the compiler:

Example

```
/* The code below will print the words Hello World!
to the screen, and it is amazing */
cout << "Hello World!";
```

Single or multi-line comments?

It is up to you which you want to use. Normally, we use // for short comments, and /* */ for longer.

Tokens and White Space

The indivisible elements in a line of code are called tokens. Generally, you must separate one token from the next with a space, tab, or carriage return, which collectively are termed white space. Some single characters, such as parentheses and commas, are tokens that need not be set off by white space. Here are some examples that illustrate when white

space can be used and when it can be omitted:

return0; // INVALID, must be return 0;

return(0); // VALID, white space omitted

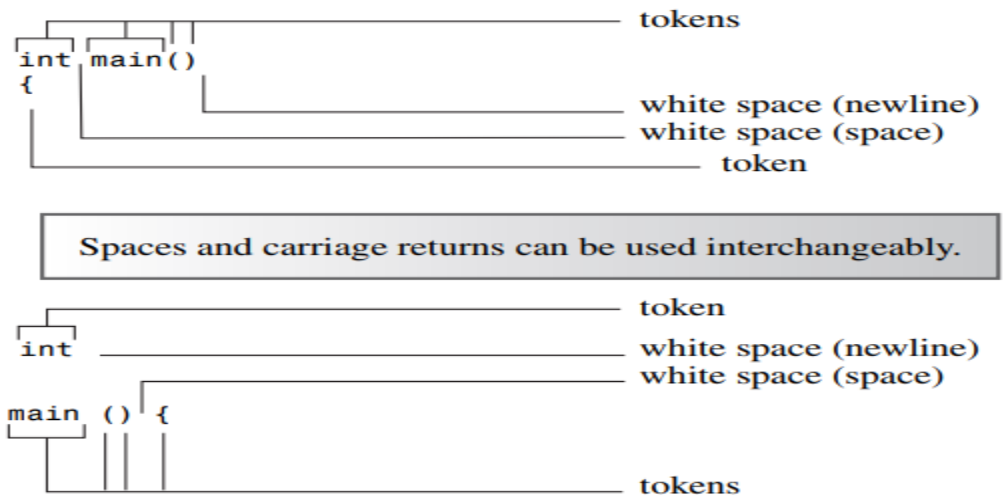
return (0); // VALID, white space used

intmain(); // INVALID, white space omitted

int main() // VALID, white space omitted in ()

int main ( ) // ALSO VALID, white space used in ( )

Tokens and White Space



C++ Statements

A C++ program is a collection of functions, and each function is a collection of statements. C++ has several kinds of statements, so let's look at some of the possibilities. From the example 2 provides two new kinds of statements. First, a declaration statement creates a variable. Second, an assignment statement provides a value for that variable. Also, the program shows a new capability for cout.

Example 2:

```
// carrots.cpp -- food processing program
// uses and displays a variable

#include <iostream>

int main()
{
    using namespace std;

    int carrots;            // declare an integer variable

    carrots = 25;            // assign a value to the variable
    cout << "I have ";
    cout << carrots;        // display the value of the variable
    cout << " carrots.";
    cout << endl;
    carrots = carrots - 1;  // modify the variable
    cout << "Crunch, crunch. Now I have " << carrots << " carrots." << endl;
    return 0;
}
```

A blank line separates the declaration from the rest of the program. This practice is the usual C convention, but it's somewhat less common in C++. Here is the program output

I have 25 carrots.

Crunch, crunch. Now I have 24 carrots.

The next few pages examine this program.

Declaration Statements and Variables

Computers are precise, orderly machines. To store an item of information in a computer, you must identify both the storage location and how much memory storage space the information requires. One relatively painless way to do this in C++ is to use a declaration statement to indicate the type of storage and to provide a label for the location.

- Variables are containers for storing data values.

For example, the program in example 2 has this declaration statement (note the semicolon):

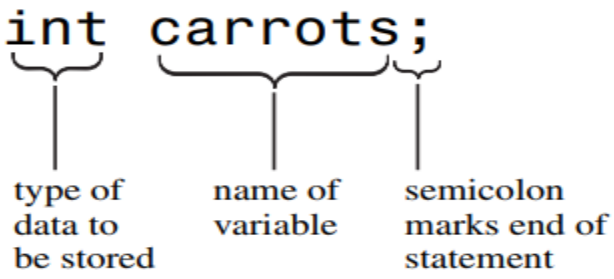
int carrots;

Why Must Variables Be Declared?

Some languages, notably BASIC, create a new variable whenever you use a new name, without the aid of explicit declarations.

In general, then, a declaration indicates the type of data to be stored and the name the program will use for the data that's stored there. In this particular case, the program creates a variable called carrots in which it can store an integer.

A Variable declaration



The declaration statement in the program is called a defining declaration statement, or definition, for short. This means that its presence causes the compiler to allocate memory space for the variable. In more complex situations, you can also have reference declarations. These tell the computer to use a variable that has already been defined elsewhere. In general, a declaration need not be a definition, but in this example it is.

Assignment Statements

An assignment statement assigns a value to a storage location. For example, the statement

carrots = 25;

assigns the integer 25 to the location represented by the variable carrots. The = symbol is called the assignment operator. One unusual feature of C++ (and C) is that you can use the assignment operator serially. For example, the following is valid code:

```
int steinway;
int baldwin;
int yamaha;
yamaha = baldwin = steinway = 88;
```

The assignment works from right to left. First, 88 is assigned to steinway; then the value of steinway, which is now 88, is assigned to baldwin; then baldwin's value of 88 is assigned to yamaha.

To create a variable, specify the type and assign it a value:

Syntax

type variableName = value;

Where type is one of C++ types (such as int), and variableName is the name of the variable (such as x or myName). The equal sign is used to assign values to the variable.

Example

Create a variable called **myNum** of type int and assign it the value **15**:

```
#include <iostream>
using namespace std;
int main() {
    int myNum = 15;
    cout << myNum;
return 0;
}
```

You can also declare a variable without assigning the value, and assign the value later:

```
#include <iostream>
using namespace std;
    int main() {
     int myNum;
     myNum = 15;
cout << myNum;
 return 0;
}
```

Note that if you assign a new value to an existing variable, it will overwrite the previous value:

Example

```
int myNum = 15;  // myNum is 15
myNum = 10;  // Now myNum is 10
cout << myNum;  // Outputs 10
```

A demonstration of other data types:

Example

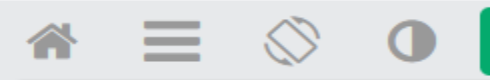
```
int myNum = 5;               // Integer (whole number without decimals)
double myFloatNum = 5.99;    // Floating point number (with decimals)
char myLetter = 'D';         // Character
string myText = "Hello";     // String (text)
bool myBoolean = true;       // Boolean (true or false)
```

C++ Declare Multiple Variables

To declare more than one variable of the **same type**, use a comma-separated list:

Example

```
int x = 5, y = 6, z = 50;
cout << x + y + z;
```



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

int main() {
  int x = 5, y = 6, z = 50;
  cout << x + y + z;
  return 0;
}
```

one Value to Multiple Variables

You can also assign the **same value** to multiple variables in one line:

Example

```
int x, y, z;
x = y = z = 50;
cout << x + y + z;
```

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

int main() {
  int x, y, z;
  x = y = z = 50;
  cout << x + y + z;
  return 0;
}
```

C++ Identifiers

All C++ **variables** must be **identified** with **unique names**.
These unique names are called **identifiers**. Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).
Note: It is recommended to use descriptive names in order to create understandable and maintainable code:

```
// Good
int minutesPerHour = 60;
// OK, but not so easy to understand what m actually is
int m = 60;
```

Run ›

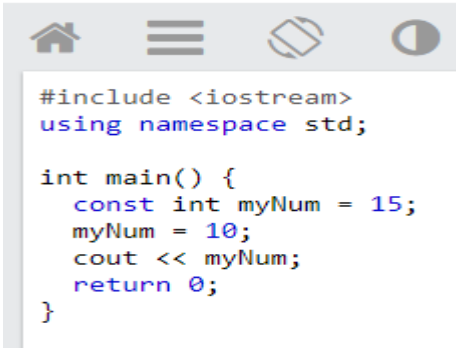
```
#include <iostream>
using namespace std;

int main() {
  // Good name
  int minutesPerHour = 60;

  // OK, but not so easy to understand what m actually is
  int m = 60;

  cout << minutesPerHour << "\n";
  cout << m;
  return 0;
}
```

Constants

When you do not want others (or yourself) to change existing variable values, use the const keyword (this will declare the variable as "constant", which means **unchangeable and read-only**):

Example

```
const int myNum = 15;  // myNum will always be 15
myNum = 10;  // error: assignment of read-only variable 'myNum'
```

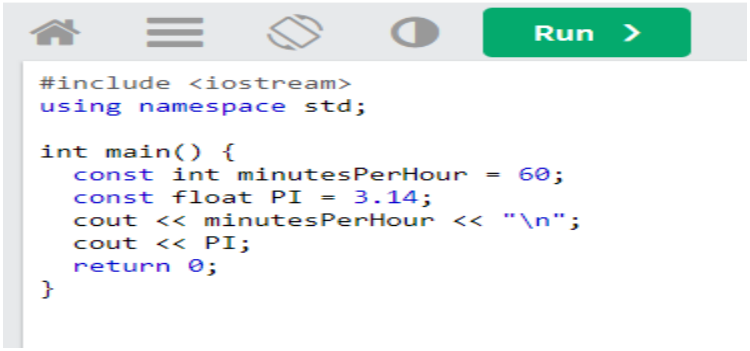


```
#include <iostream>
using namespace std;

int main() {
  const int myNum = 15;
  myNum = 10;
  cout << myNum;
  return 0;
}
```

You should always declare the variable as constant when you have values that are unlikely to change:

Example

```
const int minutesPerHour = 60;
const float PI = 3.14;
```




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

int main() {
  const int minutesPerHour = 60;
  const float PI = 3.14;
  cout << minutesPerHour << "\n";
  cout << PI;
  return 0;
}
```

Example 3:

```
// getinfo.cpp -- input and output
#include <iostream>

int main()
{
    using namespace std;

    int carrots;

    cout << "How many carrots do you have?" << endl;
    cin >> carrots;                  // C++ input
    cout << "Here are two more. ";
    carrots = carrots + 2;
// the next line concatenates output
    cout << "Now you have " << carrots << " carrots." << endl;
    return 0;
}
```

Here is an example of output from the program in Listing 2.3:

```
How many carrots do you have?
12
Here are two more. Now you have 14 carrots.
```

The program has two new features: using cin to read keyboard input and combining four output statements into one. Let's take a look.

Explain input/output operation in C++

Input/output (I/O) operations in C++ are used to read data from an input source and write data to an output sink. The most common I/O operators in C++ are the extraction operator (>>) and the insertion operator (<<). The extraction operator is used to read data from an input source and store it in a variable. For example, the following code reads an integer from the keyboard and stores it in the variable x:

```C++
int x;
cin >> x;
```

The insertion operator is used to write data to an output sink. For example, the following code writes the value of the variable x to the console:

```C++
cout << x;
```

In addition to the extraction and insertion operators, there are a number of other I/O operators in C++, such as:

- get(): Reads a single character from an input source.
- getline(): Reads a line of text from an input source.
- put(): Writes a single character to an output sink.
- write(): Writes a string of characters to an output sink.

I/O operations in C++ are handled by the iostream library. The iostream library provides a number of classes and functions that can be used to perform I/O operations. The most important classes in the iostream library are:

- istream: Represents an input stream.
- ostream: Represents an output stream.
- cin: The standard input stream.
- cout: The standard output stream.

The iostream library also provides a number of functions that can be used to perform I/O operations, such as:

- cin.get(): Reads a single character from the standard input stream.
- cout.put(): Writes a single character to the standard output stream.
- cout.write(): Writes a string of characters to the standard output stream.

I/O operations in C++ are a fundamental part of the language. They allow you to read data from input sources and write data to output sinks. This makes it possible to interact with the user, read data from files, and write data to files.

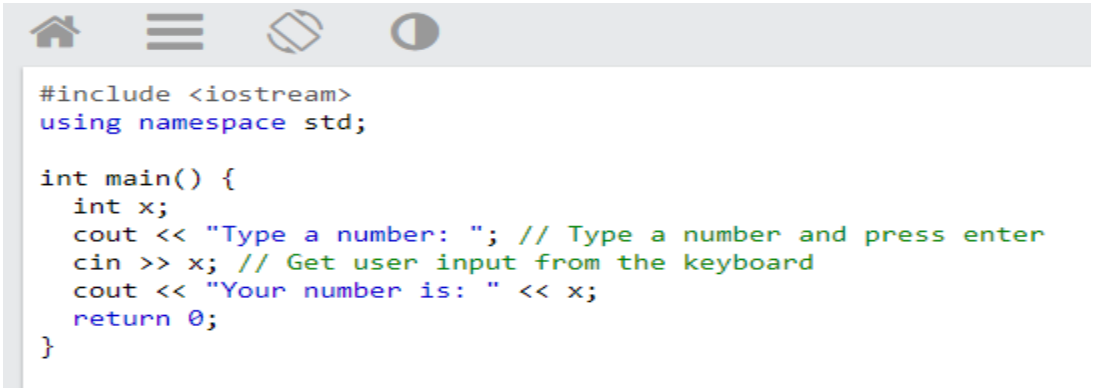
Cin and Cout Objects

Cin and cout are objects in the iostream library that are used to perform input and output operations in C++. cin is the standard input stream, and cout is the standard output stream.

- **cin** is used to read data from the standard input device, which is usually the keyboard. The data that is read from cin can be of any type, including integers, floating-point numbers, strings, and characters.
- **cout** is used to write data to the standard output device, which is usually the console. The data that is written to cout can be of any type, including integers, floating-point numbers, strings, and characters.

cin and cout are two of the most important objects in the iostream library. They are used in almost every C++ program that performs input and output operations.

Example



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

int main() {
  int x;
  cout << "Type a number: "; // Type a number and press enter
  cin >> x; // Get user input from the keyboard
  cout << "Your number is: " << x;
  return 0;
}
```

Example: A program that reads an integer from the user and prints it back out

```
#include <iostream>

using namespace std;

int main() {
    int number;
    cout << "Enter an integer: ";
    cin >> number;
    cout << "You entered: " << number << endl;
    return 0;
}
```

This program first prompts the user to enter an integer. It then uses the cin object to read the integer from the user. The cin object is a standard input stream that is used to read data from the keyboard.

Example: A program that reads a string from the user and prints it back out:

```cpp
#include <iostream>

using namespace std;

int main() {
    string name;
    cout << "Enter your name: ";
    cin >> name;
    cout << "Your name is: " << name << endl;
    return 0;
}
```

This program first prompts the user to enter their name. It then uses the cin object to read the name from the user. The cin object can also be used to read strings from the keyboard.

The program then prints the name that the user entered. It uses the cout object to print the name. Here are some additional points about cin and cout:

- They are objects of the istream and ostream classes, respectively.
- They are predefined in the iostream header file.
- They can be used to read and write data from and to a variety of input and output sources, including files, sockets, and pipes.
- They can be used to read and write data of any type, including integers, floating-point numbers, strings, and characters.

Using cin

As the output from above example demonstrates, the value typed from the keyboard (12) is eventually assigned to the variable carrots. The following statement performs that wonder:

cin >> carrots;

cout is pronounced "see-out". Used for **output**, and uses the insertion operator (<<) and cin is pronounced "see-in". Used for **input**, and uses the extraction operator (>>)

Creating a Simple Calculator

In this example, the user must input two numbers. Then we print the sum by calculating (adding) the two numbers:

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

int main() {
  int x, y;
  int sum;
  cout << "Type a number: ";
  cin >> x;
  cout << "Type another number: ";
  cin >> y;
  sum = x + y;
  cout << "Sum is: " << sum;
  return 0;
}
```

Concatenating with cout

The second new feature of getinfo.cpp is combining four output statements into one. The iostream file defines the << operator so that you can combine (that is, concatenate) output as follows:

cout << "Now you have " << carrots << " carrots." << endl;

This allows you to combine string output and integer output in a single statement. The resulting output is the same as what the following code produces:

cout << "Now you have ";

cout << carrots;

cout << " carrots";

cout << endl;

While you're still in the mood for cout advice, you can also rewrite the concatenated version this way, spreading the single statement over four lines:

cout << "Now you have "

<< carrots

<< " carrots."

<< endl;

That's because C++'s free format rules treat newlines and spaces between tokens interchangeably. This last technique is convenient when the line width cramps your style. Another point to note is that

Now you have 14 carrots.

appears on the same line as

Here are two more.

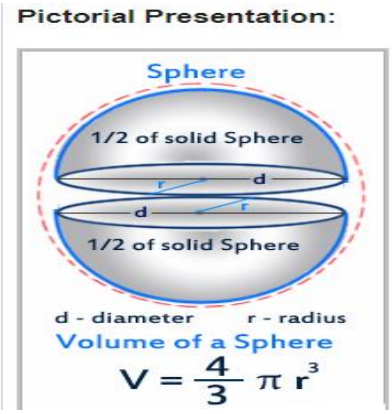
That's because, as noted before, the output of one cout statement immediately follows the output of the preceding cout statement. This is true even if there are other statements in between.

Example: Write a program in C++ to swap two numbers.

Sample Solution :-

```
#include <iostream>
using namespace std;
int main()
{
            cout << "\n\n Swap two numbers :\n";
            cout << "-----------------------\n";
            int num1, num2, temp;
            cout << " Input 1st number : ";
            cin >> num1 ;
            cout << " Input 2nd number : ";
            cin >> num2;
            temp=num2;
            num2=num1;
            num1=temp;
   cout << " After swapping the 1st number is : "<< num1 <<"\n" ;
   cout << " After swapping the 2nd number is : "<< num2 <<"\n\n" ;
}
```

Example: Write a program in C++ to calculate the volume of a sphere




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

   int main()
   {
            int rad1;
            float volsp;
      cout << "\n\n Calculate the volume of a sphere :\n";
      cout << "---------------------------------------\n";
      cout<<" Input the radius of a sphere : ";
            cin>>rad1;
            volsp=(4*3.14*rad1*rad1*rad1)/3;
      cout<<" The volume of a sphere is : "<< volsp << endl;
      cout << endl;
      return 0;
   }
```

Example: Write a program in C++ to calculate the volume of a cube.
Sample Solution:

```
#include <iostream>
```

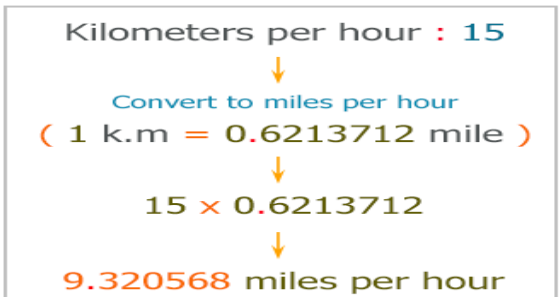
```
using namespace std;

  int main()
  {
          int sid1;
          float volcu;
     cout << "\n\n Calculate the volume of a cube :\n";
     cout << "---------------------------------------\n";
     cout<<" Input the side of a cube : ";
          cin>>sid1;
          volcu=(sid1*sid1*sid1);
     cout<<" The volume of a cube is : "<< volcu << endl;
     cout << endl;
     return 0;

  }
```

LAB EXERCISES:

i. Write a program in C++ to calculate the volume of a cylinder.
ii. Write a program in C++ to find the Area and Perimeter of a Rectangle.
iii. Write a program in C++ to find the area of any triangle using Heron's Formula.
iv. Write a program in C++ to find the area and circumference of a circle.
v. Write a program in C++ to convert temperature in Fahrenheit to Celsius.
vi. Write a program in C++ to find the third angle of a triangle.

Example Write a program in C++ that converts kilometres per hour to miles per hour.

Pictorial Presentation:




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

int main() {
 float kmph, miph;
cout << "\n\n Convert kilometers per hour to miles per hour :\n";
cout << "-------------------------------------------------\n";
cout << " Input the distance in kilometer : ";
cin >> kmph;
miph = (kmph * 0.6213712);
cout << " The "<< kmph <<" Km./hr. means "<< miph << " Miles/hr." << endl;
cout << endl;
```

```
    return 0;
}
```

Example: Write a program in C++ to convert temperature in Kelvin to Fahrenheit.

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

    int main()
    {
       float kel, frh;
                cout << "\n\n Convert temperature in Kelvin to Fahrenheit  :\n";
                cout << "-------------------------------------------------\n";
       cout << " Input the temperature in Kelvin : ";
       cin >> kel;
       frh = (9.0 / 5) * (kel - 273.15) + 32;
       cout << " The temperature in Kelvin    : " << kel << endl;
       cout << " The temperature in Fahrenheit : " << frh << endl;
      cout << endl;
       return 0;
    }
```

LAB EXERCISE:

i. Write a program in C++ to convert temperature in Kelvin to Celsius.
ii. Write a program in C++ to convert temperature in Fahrenheit to Kelvin.
iii. Write a program in C++ to convert temperature in Celsius to Kelvin.

Example: Write a program in C++ to input a single digit number and print a rectangular form of 4 columns and 6 rows

Sample Solution:

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

    int main()
    {
       int x;
cout << "\n\n Make a rectangular shape by a single digit number :\n";
cout << "-------------------------------------------------\n";
        cout<<" Input the number : ";
       cin>> x;
        cout<<" "<<x<<x<<x<<x<<endl;
        cout<<" "<<x<<" "<<" "<<x<<endl;
        cout<<" "<<x<<" "<<" "<<x<<endl;
        cout<<" "<<x<<" "<<" "<<x<<endl;
        cout<<" "<<x<<" "<<" "<<x<<endl;
        cout<<" "<<x<<x<<x<<x<<endl;
        cout << endl;
        return 0;
    }
```

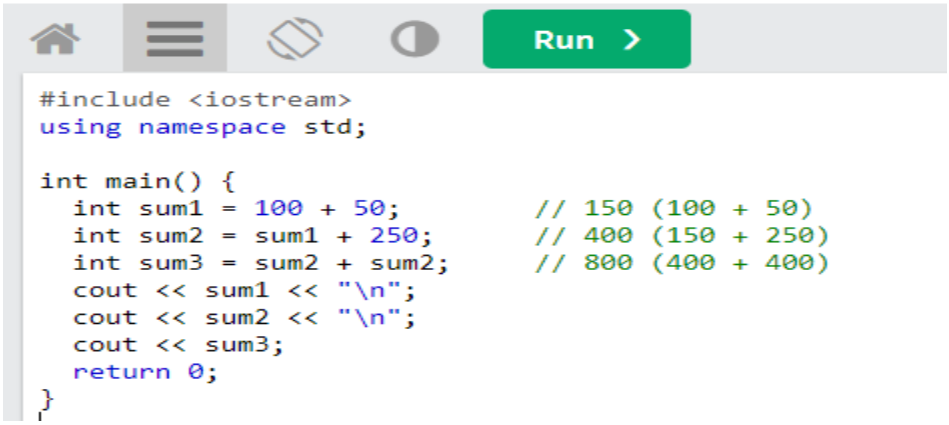
C++ Operators

- Operators are used to perform operations on variables and values.
- In the example below, we use the + **operator** to add together two values:

Example. Write a Program to add two numbers

```
#include <iostream>
using namespace std;
        int main() {
         int x = 100 + 50;
        cout << x;
 return 0;
}
```

Although the + operator is often used to add together two values, like in the example above, it can also be used to add together a variable and a value, or a variable and another variable:



C++ divides the operators into the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Bitwise operators

Arithmetic Operators

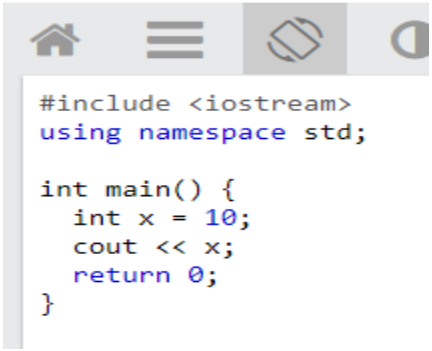
Arithmetic operators are used to perform common mathematical operations.

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

Assignment Operators

Assignment operators are used to assign values to variables. In the example below, we use the **assignment** operator (=) to assign the value **10** to a variable called **x**:

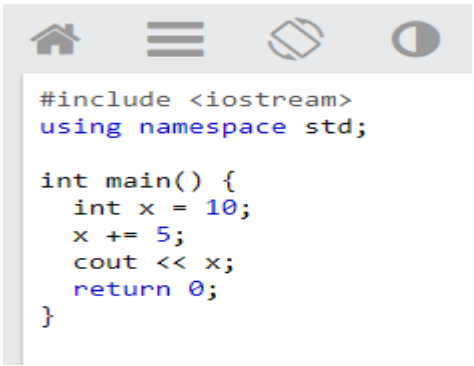
Example



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

int main() {
  int x = 10;
  cout << x;
  return 0;
}
```

The **addition assignment** operator (+=) adds a value to a variable:

Example



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

int main() {
  int x = 10;
  x += 5;
  cout << x;
  return 0;
}
```

A list of all assignment operators:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

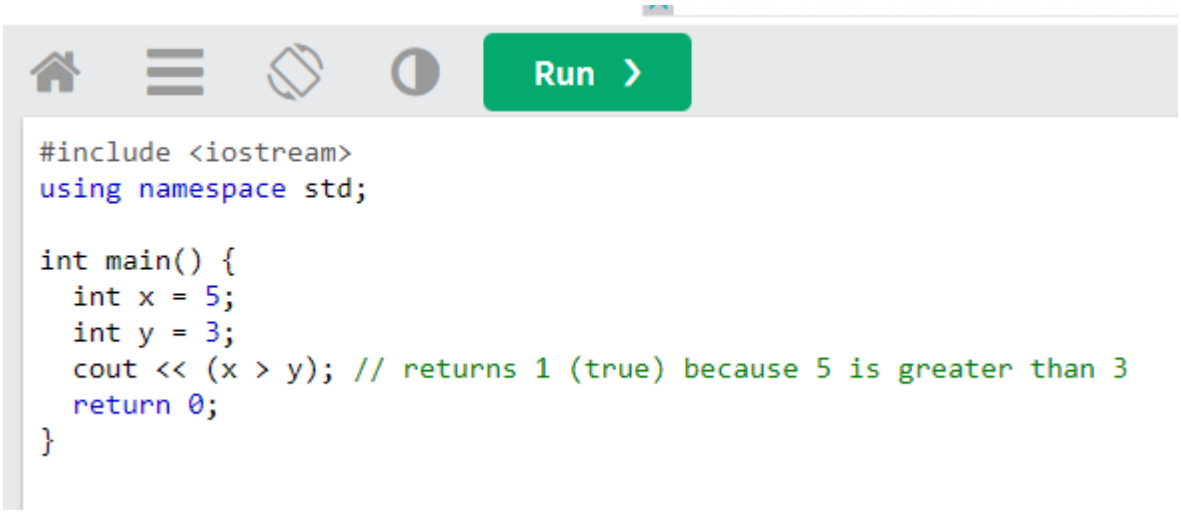
Comparison Operators

Comparison operators are used to compare two values (or variables). This is important in programming, because it helps us to find answers and make decisions.

The return value of a comparison is either 1 or 0, which means **true** (1) or **false** (0). These values are known as **Boolean values**, and you will learn more about them in the Booleans and If..Else chapter.

In the following example, we use the **greater than** operator (>) to find out if 5 is greater than 3:

Example



A list of all comparison operators:

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Logical Operators

As with comparison operators, you can also test for **true** (1) or **false** (0) values with **logical operators**. Logical operators are used to determine the logic between variables or values:

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

C++ Strings

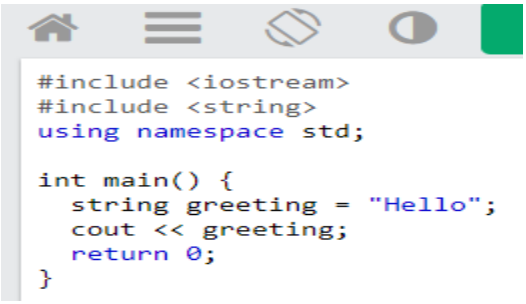
Strings are used for storing text. A string variable contains a collection of characters surrounded by double quotes:

Example
Create a variable of type string and assign it a value:

string greeting = "Hello";

To use strings, you must include an additional header file in the source code, the `<string>` library:

Example

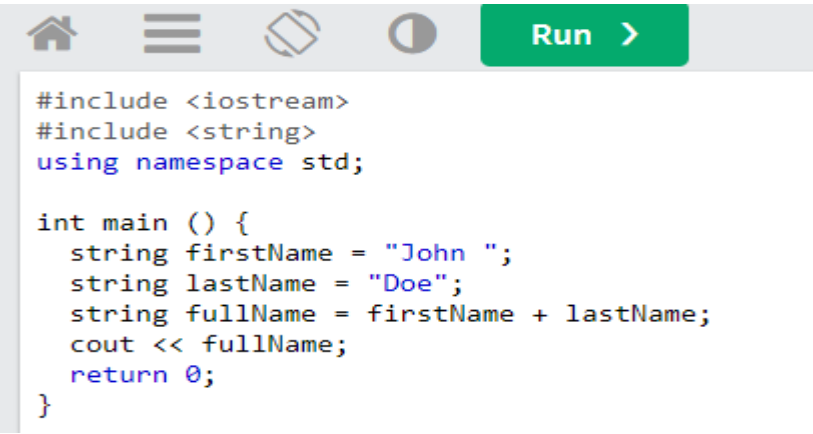


```
#include <iostream>
#include <string>
using namespace std;

int main() {
  string greeting = "Hello";
  cout << greeting;
  return 0;
}
```

String Concatenation

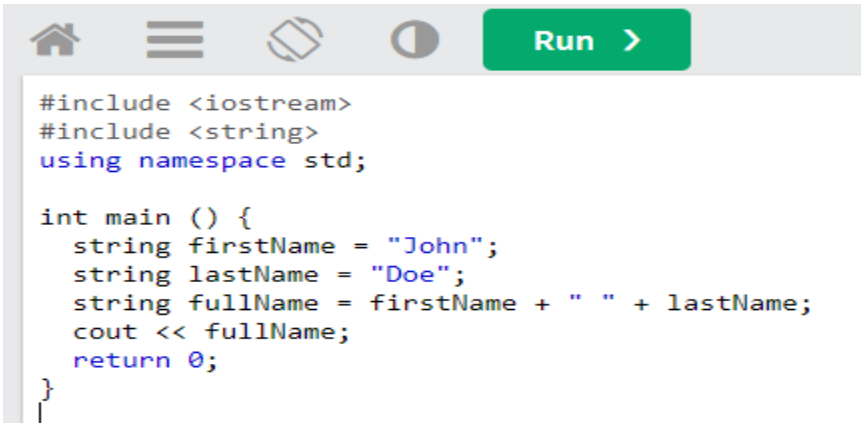
The + operator can be used between strings to add them together to make a new string. This is called **concatenation**:

Example



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

int main () {
  string firstName = "John ";
  string lastName = "Doe";
  string fullName = firstName + lastName;
  cout << fullName;
  return 0;
}
```

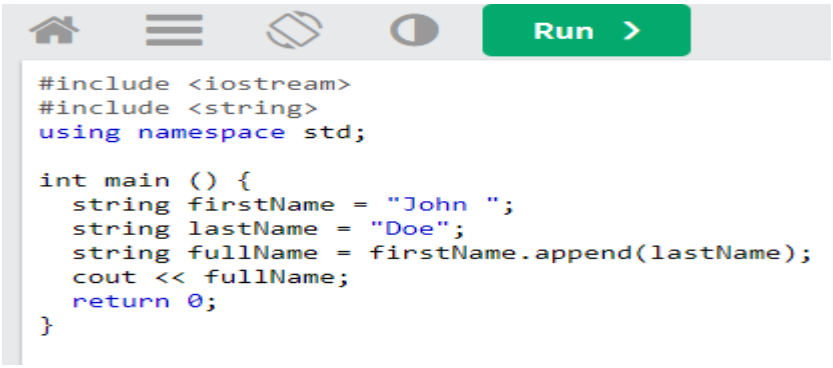
In the example above, we added a space after firstName to create a space between John and Doe on output. However, you could also add a space with quotes (" " or ' '):



Append

A string in C++ is actually an object, which contain functions that can perform certain operations on strings. For example, you can also concatenate strings with the append() function:

Example



User Input Strings

It is possible to use the extraction operator >> on cin to store a string entered by a user:

Example

```
string firstName;
cout << "Type your first name: ";
cin >> firstName; // get user input from the keyboard
cout << "Your name is: " << firstName;

// Type your first name: John
// Your name is: John
```

However, cin considers a space (whitespace, tabs, etc) as a terminating character, which means that it can only store a single word (even if you type many words):

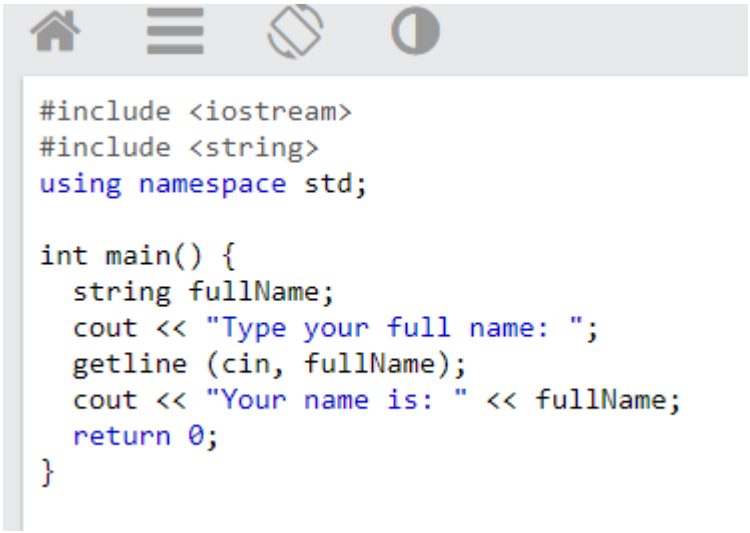
Example

```
string fullName;
cout << "Type your full name: ";
cin >> fullName;
cout << "Your name is: " << fullName;

// Type your full name: John Doe
// Your name is: John
```

From the example above, you would expect the program to print "John Doe", but it only prints "John".

That's why, when working with strings, we often use the getline() function to read a line of text. It takes cin as the first parameter, and the string variable as second:

Example:



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

int main() {
  string fullName;
  cout << "Type your full name: ";
  getline (cin, fullName);
  cout << "Your name is: " << fullName;
  return 0;
}
```

Omitting Namespace

You might see some C++ programs that runs without the standard namespace library. The using namespace std line can be omitted and replaced with the std keyword, followed by the :: operator for string (and cout) objects:

Example

```
#include <iostream>
#include <string>

int main() {
  std::string greeting = "Hello";
```

```
  std::cout << greeting;
  return 0;
}
```

Booleans

Very often, in programming, you will need a data type that can only have one of two values, like:

- YES / NO
- ON / OFF
- TRUE / FALSE

For this, C++ has a bool data type, which can take the values true (1) or false (0).

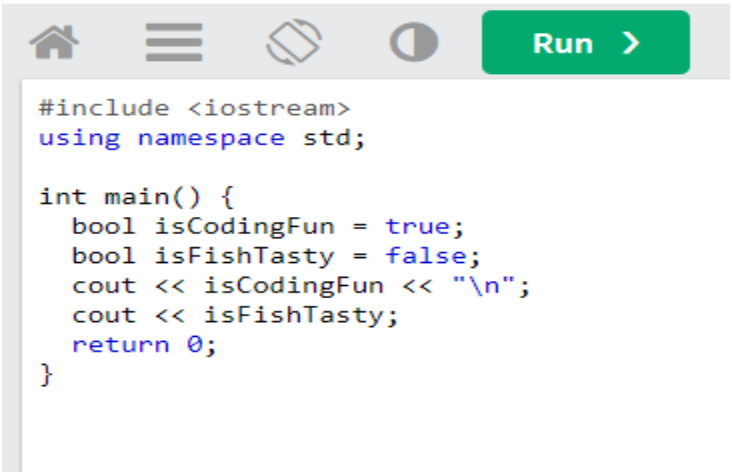
Boolean Values

A boolean variable is declared with the bool keyword and can only take the values true or false:

Example

```
bool isCodingFun = true;
bool isFishTasty = false;
cout << isCodingFun;  // Outputs 1 (true)
cout << isFishTasty;  // Outputs 0 (false)
```

From the example above, you can read that a true value returns 1, and false returns 0.
However, it is more common to return a boolean value by comparing values and variables



Boolean Expression

A **Boolean expression** returns a boolean value that is either 1 (true) or 0 (false). This is useful to build logic, and find answers. You can use a comparison operator, such as the **greater than** (>) operator, to find out if an expression (or variable) is true or false:

Example

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

int main() {
  int x = 10;
  int y = 9;
  cout << (x > y);
  return 0;
}
```

```
cout << (10 > 9); // returns 1 (true), because 10 is higher than 9
```

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

int main() {
  cout << (10 > 9);
  return 0;
}
```

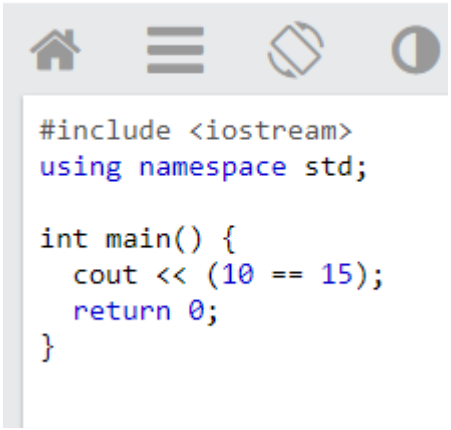
In the examples below, we use the **equal to** (==) operator to evaluate an expression:

Example

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

int main() {
  int x = 10;
  cout << (x == 10);
  return 0;
}
```

Example

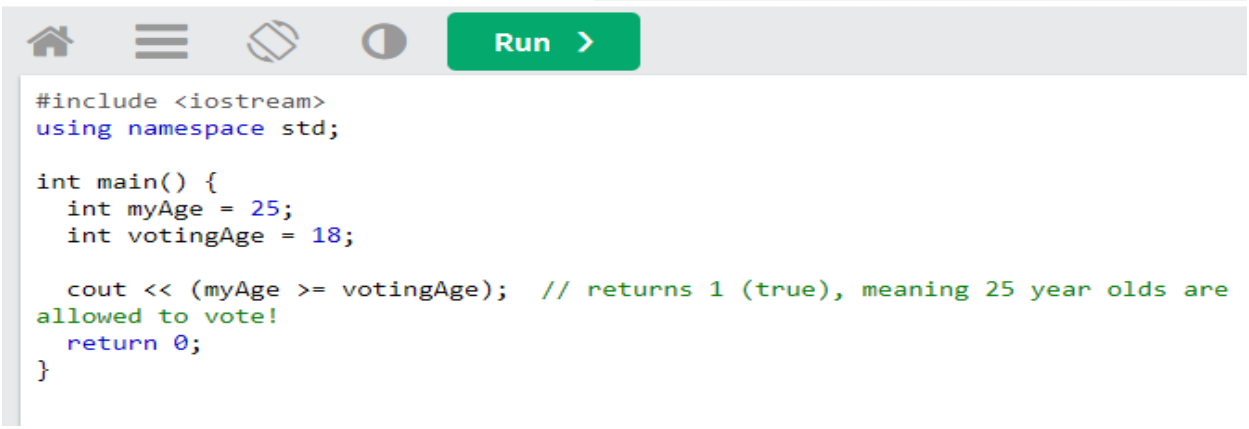
```
cout << (10 == 15);  // returns 0 (false), because 10 is not equal to 15
```



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

int main() {
  cout << (10 == 15);
  return 0;
}
```

Let's think of a "real life example" where we need to find out if a person is old enough to vote.

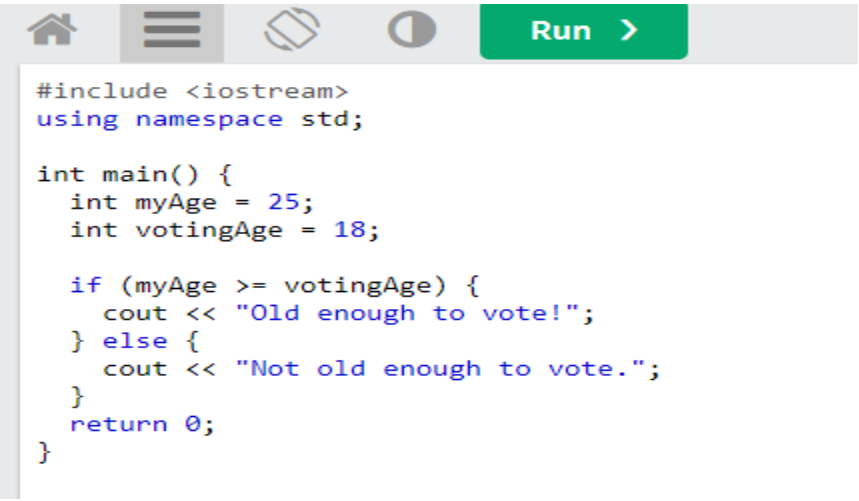
In the example below, we use the >= comparison operator to find out if the age (25) is **greater than OR equal to** the voting age limit, which is set to 18:




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

int main() {
  int myAge = 25;
  int votingAge = 18;

  cout << (myAge >= votingAge);  // returns 1 (true), meaning 25 year olds are
allowed to vote!
  return 0;
}
```

Cool, right? An even better approach (since we are on a roll now), would be to wrap the code above in an if...else statement, so we can perform different actions depending on the result:
Example

Output "Old enough to vote!" if myAge is **greater than or equal to** 18. Otherwise output "Not old enough to vote.":



Conditions

There come situations in real life when we need to make some decisions and based on these decisions, we decide what should we do next. Similar situations arise in programming also where we need to make some decisions and based on these decisions we will execute the next block of code. For example, in C if x occurs then execute y else execute z. There can also be multiple conditions like in C if x occurs then execute p, else if condition y occurs execute q, else execute r. This condition of C else-if is one of the many ways of importing multiple conditions.

C++ has only three kinds of control structures, which from this point forward we refer to as control statements: the sequence statement, selection statements (three types—if, if...else and switch) and repetition statements (three types—while, for and do...while).

You already know that C++ supports the usual logical conditions from mathematics:

- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b
- Equal to a == b
- Not Equal to: a != b

You can use these conditions to perform different actions for different decisions.

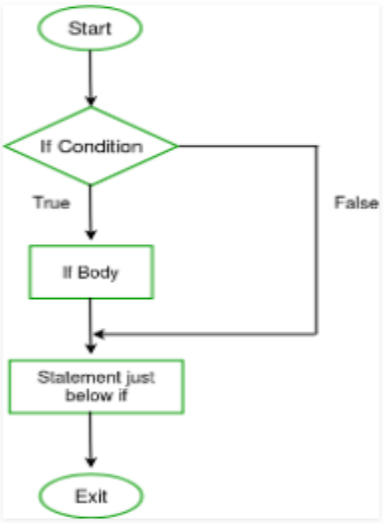
C++ has the following conditional statements:

- Use if to specify a block of code to be executed, if a specified condition is true
- Use else to specify a block of code to be executed, if the same condition is false
- Use else if to specify a new condition to test, if the first condition is false
- Use switch to specify many alternative blocks of code to be executed

The if Statement

Use the if statement to specify a block of C++ code to be executed if a condition is true.

Flowchart



Syntax

```
if (condition) {
  // block of code to be executed if the condition is true
}
```

Note that if is in lowercase letters. Uppercase letters (If or IF) will generate an error.

C++ program to illustrate If statement

```
// C++ program to illustrate If statement
#include<iostream>
using namespace std;

   int main()
   {
      int i = 10;

      if (i > 15)
      {
        cout<<"10 is less than 15";
      }
```

In the example below,we test two values to find out if 20 is greater than 18. If the condition is true, print some text:

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

```
        if (20 > 18) {
cout << "20 is greater than 18";
  }
 return 0;
}
```

We can also test variables:

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

int main() {
  int x = 20;
  int y = 18;
  if (x > y) {
    cout << "x is greater than y";
  }
  return 0;
}
```

Example explained

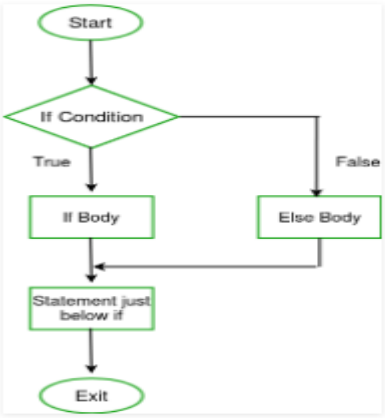
In the example above we use two variables, **x** and **y**, to test whether x is greater than y (using the > operator). As x is 20, and y is 18, and we know that 20 is greater than 18, we print to the screen that "x is greater than y".

The else Statement

Use the else statement to specify a block of code to be executed if the condition is false.

The IF-else statement also tests the condition. It executes IF block, if condition is true otherwise else block is executed. The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false.

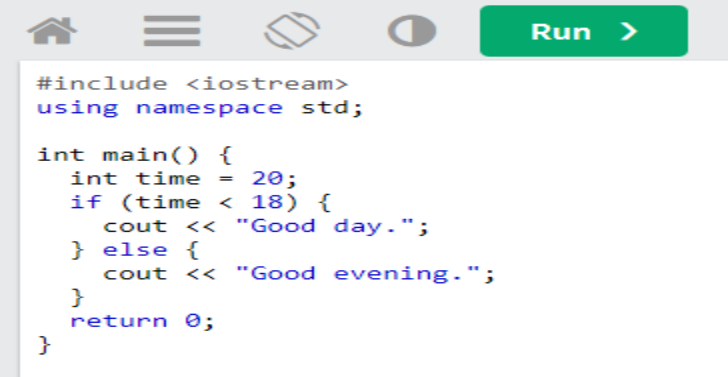
Flowchart:



Syntax

```
if (condition) {
  // block of code to be executed if the condition is true
} else {
  // block of code to be executed if the condition is false
}
```

Example



Example explained
In the example above, time (20) is greater than 18, so the condition is false. Because of this, we move on to the else condition and print to the screen "Good evening". If the time was less than 18, the program would print "Good day".

However, if the time was 14, our program would print "Good day."

Example:Write a program in C++ to check whether a number is EVEN or ODD.

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

  int main()
  {
          int num = 11;
if(num % 2==0){
cout << " it is even number\n";
}
Else
{
cout << " it is odd number\n";

     return 0;
  }
```

Example: Write a program in C++ to check whether a number is EVEN or ODD with input from user.

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

   int main()
   {
            int num;
cout<< “Enter a Number”;<<endl;
cin>>num;
if(num % 2==0){
cout << " it is even number";<<endl;
}
Else
{
cout << " it is odd number";<<endl;

      return 0;
   }
```

IF-ELSE-IF LADDER STATEMENT

The C++ if-else-if ladder statement executes one condition from multiple statements.

Write a program in C++ to check whether a number is positive, negative or zero

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

int main()
{
   signed long num1 = 0;
            cout << "\n\n Check whether a number is positive, negative or zero :\n";
            cout << "-------------------------------------------------------\n";
            cout << " Input a number : ";
   cin >> num1;
   if(num1 > 0)
   {
      cout << " The entered number is positive.\n\n";
   }
   else if(num1 < 0)
   {
      cout << " The entered number is negative.\n\n";
   }
   else
   {
      std::cout << " The number is zero.\n\n";
   }
   return 0;
}
```

Write a program in C++ to check GRADE

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

```
    int num;
    cout<<"Enter a number to check grade:";
    cin>>num;
        if (num <0 || num >100)
        {
           cout<<"wrong number";
        }
        else if(num >= 0 && num < 50){
           cout<<"Fail";
        }
        else if (num >= 50 && num < 60)
        {
           cout<<"D Grade";
        }
        else if (num >= 60 && num < 70)
        {
           cout<<"C Grade";
        }
        else if (num >= 70 && num < 80)
        {
           cout<<"B Grade";
        }
        else if (num >= 80 && num < 90)
        {
           cout<<"A Grade";
        }
        else if (num >= 90 && num <= 100)
        {
           cout<<"A+ Grade";
        }
  }
```

Switch Statements

Use the switch statement to select one of many code blocks to be executed.

Syntax

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

This is how it works:

- The switch expression is evaluated once
- The value of the expression is compared with the values of each case
- If there is a match, the associated block of code is executed
- The break and default keywords are optional, and will be described later in this chapter

The example below uses the weekday number to calculate the weekday name:

Example

```
int day = 4;
switch (day) {
  case 1:
    cout << "Monday";
    break;
  case 2:
    cout << "Tuesday";
    break;
  case 3:
    cout << "Wednesday";
    break;
  case 4:
    cout << "Thursday";
    break;
  case 5:
    cout << "Friday";
    break;
  case 6:
    cout << "Saturday";
    break;
  case 7:
    cout << "Sunday";
    break;
```

```
}
// Outputs "Thursday" (day 4)
```

Example: write a C++ program to accept a number from user and check the grade based on user's input.

```
#include <iostream>
using namespace std;
int main () {
      int num;
      cout<<"Enter a number to check grade:";
      cin>>num;
         switch (num)
         {
            case 10: cout<<"It is 10"; break;
            case 20: cout<<"It is 20"; break;
            case 30: cout<<"It is 30"; break;
            default: cout<<"Not 10, 20 or 30"; break;
         }
   }
```

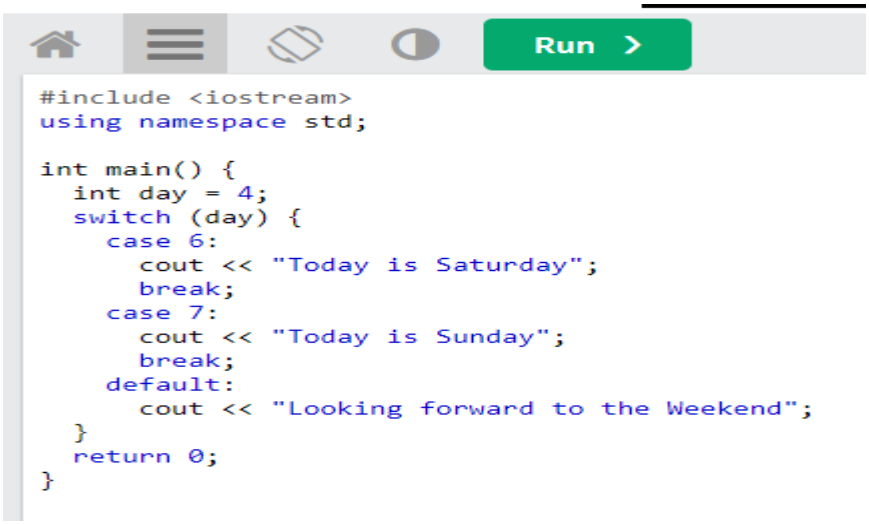
The break Keyword

- When C++ reaches a break keyword, it breaks out of the switch block.
- This will stop the execution of more code and case testing inside the block.
- When a match is found, and the job is done, it's time for a break. There is no need for more testing.

A break can save a lot of execution time because it "ignores" the execution of all the rest of the code in the switch block.

The default Keyword

The default keyword specifies some code to run if there is no case match:



Loops

Loops can execute a block of code as long as a specified condition is reached. Loops are handy because they save time, reduce errors, and they make code more readable. Loops in programming come into use when we need to repeatedly execute a block of statements. For example: Suppose we want to print “Hello World” 10 times. This can be done in two ways as shown below:

Iterative Method

An iterative method to do this is to write the cout<< “ ” statement 10 times.

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

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

Using Loops

In Loop, the statement needs to be written only once and the loop will be executed 10 times as shown below.

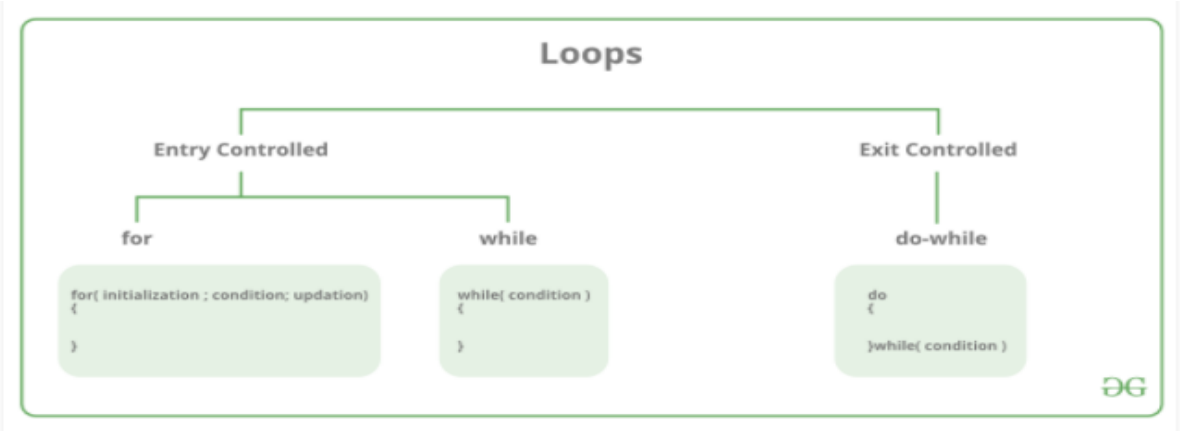
In computer programming, a loop is a sequence of instructions that is repeated until a certain condition is reached.

i. An operation is done, such as getting an item of data and changing it, and then some condition is checked such as whether a counter has reached a prescribed number.
ii. Counter not Reached: If the counter has not reached the desired number, the next instruction in the sequence returns to the first instruction in the sequence and repeat it.
iii. Counter reached: If the condition has been reached, the next instruction “falls through” to the next sequential instruction or branches outside the loop.

There are mainly two types of loops:

- Entry Controlled loops: In this type of loops the test condition is tested before entering the loop body. For Loop and While Loop are entry controlled loops.

- Exit Controlled Loops: In this type of loops the test condition is tested or evaluated at the end of loop body. Therefore, the loop body will execute at least once, irrespective of whether the test condition is true or false. do – while loop is exit controlled loop.



For Loop

C++ program to illustrate for loop

```
// C++ program to illustrate for loop
#include <iostream>
using namespace std;

int main()
{
   for (int i = 1; i <= 10; i++)
   {
      cout << "Hello World\n";
   }

   return 0;
}
```

Example

```
#include <iostream>
using namespace std;
int main() {
        for(int i=1;i<=10;i++){
          cout<<i <<"\n";
         }
   }
```

OUTPUT

1
2
3
4

5
6
7
8
9
10

```
#include <iostream>
using namespace std;
int main () {
      for(int i=1;i<=3;i++){
          for(int j=1;j<=3;j++){
         cout<<i<<" "<<j<<"\n";
        }
      }
   }
```

OUTPUT

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

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

int main () {
      for (; ;)
        {
              cout<<"Infinitive For Loop";
        }
   }
```

OUTPUT

```
Infinitive For Loop
Infinitive For Loop
```

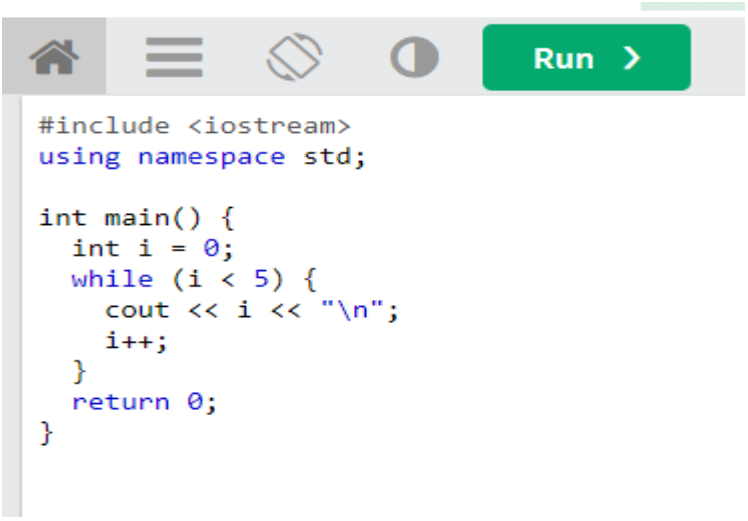
While Loop

The while loop loops through a block of code as long as a specified condition is true:

Syntax

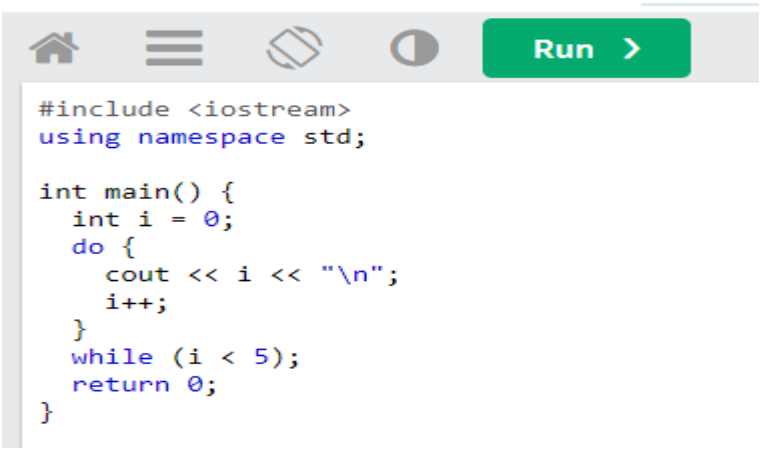
```
while (condition) {
  // code block to be executed
}
```

When you know exactly how many times you want to loop through a block of code, use the for loop instead of a while loop:



The Do/While Loop

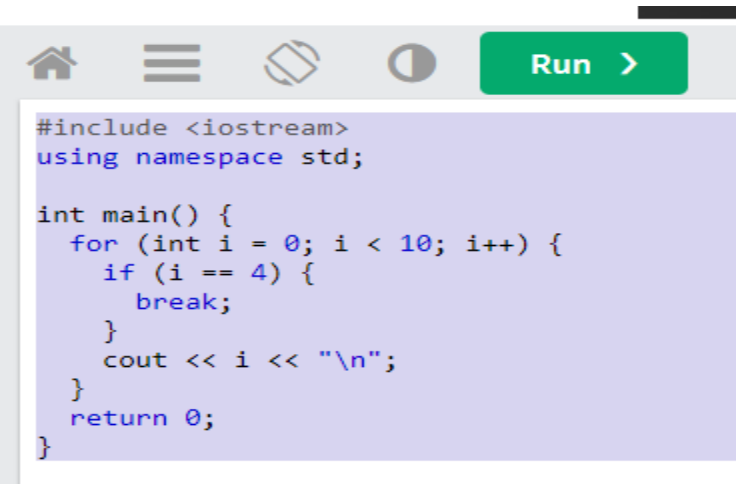
The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true. The example below uses a do/while loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:



Do not forget to increase the variable used in the condition, otherwise the loop will never end!

Break

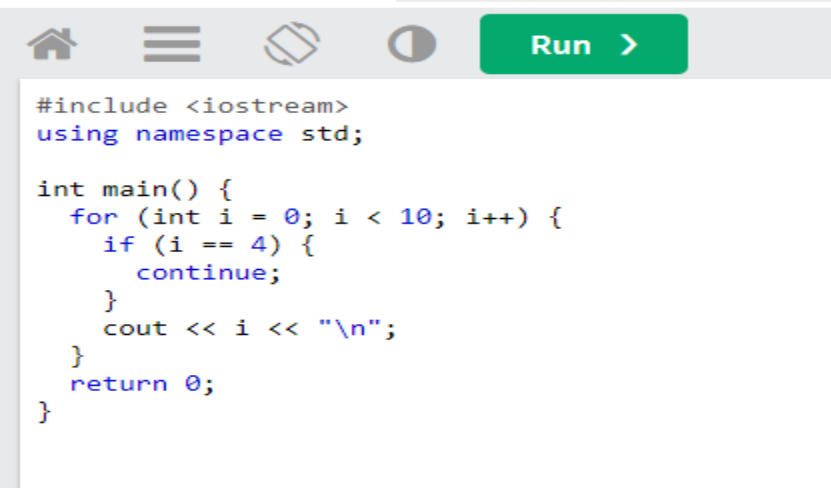
You have already seen the break statement used in an earlier chapter of this tutorial. It was used to "jump out" of a switch statement. The break statement can also be used to jump out of a **loop**. This example jumps out of the loop when i is equal to 4:



C++ Continue

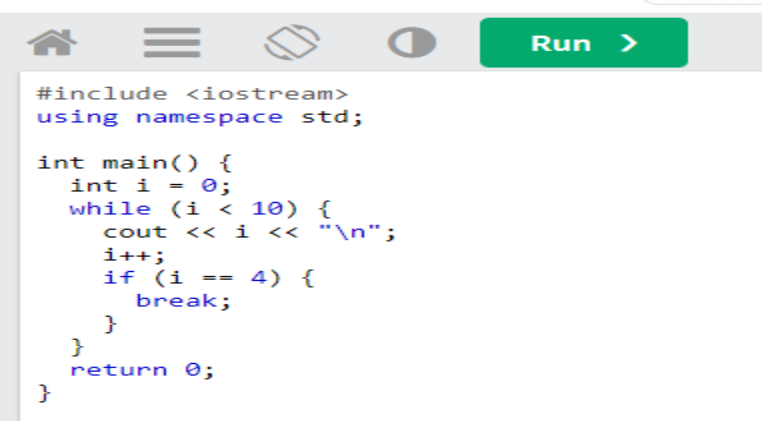
The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

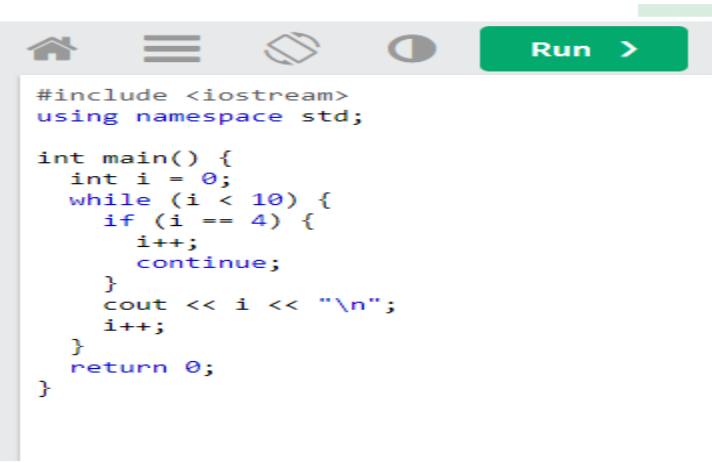
This example skips the value of 4:



Break and Continue in While Loop

You can also use break and continue in while loops:





FUNCTION

A function in C++ is a block of code that is executed when it is called. Functions can be used to perform a variety of tasks, such as calculating a value, printing a message, or reading data from a file. Because functions are the modules from which C++ programs are built and because they are essential to C++ OOP definitions, you should become thoroughly familiar with them.

A function definition in C++ has the following syntax:

return_type function_name(parameters) {

// Body of the function

}

C++ functions come in two varieties: those with return values and those without them. You can find examples of each kind in the standard C++ library of functions, and you can create your own functions of each type. Let's look at a library function that has a return value and then examine how you can write your own simple functions.

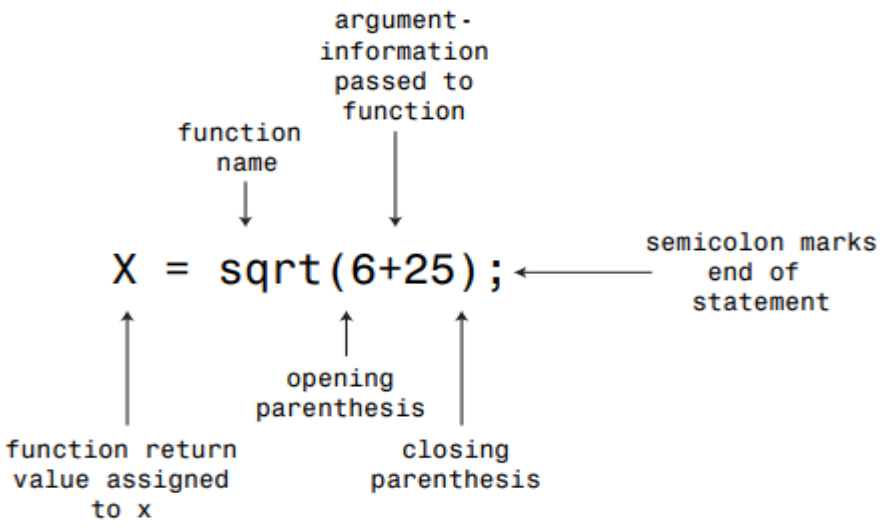
Using a Function That Has a Return Value

A function that has a return value produces a value that you can assign to a variable. For example, the standard C/C++ library includes a function called sqrt() that returns the square root of a number. Suppose you want to calculate the square root of 6.25 and assign it to the variable x. You can use the following statement in your program:

x = **sqrt**(6.25); // returns the value 2.5 and assigns it to x

The expression **sqrt**(6.25) invokes, or calls, the sqrt() function. The expression **sqrt**(6.25) is termed a function call, the invoked function is termed the *called function*, and the function containing the function call is termed the *calling function*

Function call syntax.



That's practically all there is to it, except that before the C++ compiler uses a function, it must know what kind of arguments the function uses and what kind of return value it has. That is, does the function return an integer? a character? a number with a decimal fraction? a guilty verdict? or something else? If it lacks this information, the compiler won't know how to interpret the return value. The C++ way to convey this information is to use a function prototype statement.

The return_type is the type of the value that the function returns. The function_name is the name of the function. The parameters are the arguments that are passed to the function. The body of the function is the code that is executed when the function is called.

Here is an example of a function definition

```
int factorial(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * factorial(n - 1);
```

```
    }
}
```

This function calculates the factorial of a number. The factorial of a number is the product of all the positive integers less than or equal to the number. For example, the factorial of 5 is 120.

The function factorial() has the following parameters:

- n: The number whose factorial is to be calculated.

The function factorial() has the following return value:

- int: The factorial of the number n.

The body of the function factorial() consists of two branches:

- If n is equal to 0, then the function returns 1.
- Otherwise, the function recursively calls itself, passing n - 1 as an argument. The recursive call will eventually reach the base case, where n is equal to 0, and the function will return 1.

The function factorial() can be called as follows:

```
int result = factorial(5);
```

This code will assign the value 120 to the variable result.

Functions are an essential part of C++ programming. They allow you to break down complex tasks into smaller, more manageable pieces. Functions also make your code more reusable and easier to understand.

Example

```
// sqrt.cpp -- using the sqrt() function

#include <iostream>
#include <cmath>    // or math.h

int main()
{
    using namespace std;

    double area;
    cout << "Enter the floor area, in square feet, of your home: ";
    cin >> area;
    double side;
    side = sqrt(area);
    cout << "That's the equivalent of a square " << side
         << " feet to the side." << endl;
    cout << "How fascinating!" << endl;
    return 0;
}
```

Note: If you're using an older compiler, you might have to use #include <math.h> instead of #include <cmath>

Here's a sample run of the program above

```
Enter the floor area, in square feet, of your home: 1536

That's the equivalent of a square 39.1918 feet to the side.
```

How fascinating!

Example

```
#include <iostream>
#include <cmath>

using namespace std;

int main() {
    double angle = 30;
    double sine = sin(angle);

    cout << "The sine of " << angle << " degrees is " << sine << endl;

    return 0;
}
```

This program first includes the iostream and cmath headers. The iostream header provides the cout object, which is used to print output to the console. The cmath header provides the sin() function, which is used to calculate the sine of an angle. The main function of the program defines a variable called angle and initializes it to 30 degrees. The program then uses the sin() function to calculate the sine of the angle and stores the result in the variable sine.

Finally, the program uses the cout object to print the value of sine to the console.

The output of the program should be:

The sine of 30 degrees is 0.5

Function Variations

Some functions require more than one item of information. These functions use multiple arguments separated by commas. For example, the math function pow() takes two arguments and returns a value equal to the first argument raised to the power given by the second argument. It has this prototype:

double pow(double, double); // prototype of a function with two arguments

If, say, you wanted to find 58 (5 to eighth power), you would use the function like this:

answer = pow(5.0, 8.0);

You could use the function this way:

myGuess = rand(); // function call with no arguments

User-Defined Functions

The standard C library provides more than 140 predefined functions. If one fits your needs, by all means use it. But often you have to write your own, particularly when you design classes.

Anyway, it's fun to design your own functions, so now let's examine that process. You've already used several user-defined functions, and they have all been named main().

```
// ourfunc.cpp -- defining your own function
        #include <iostream>
        void simon(int); // function prototype for simon()
        int main()
         {
       using namespace std;
       simon(3); // call the simon() function
            cout << "Pick an integer: ";
                 int count;
            cin >> count;
                 simon(count); // call it again
            cout << "Done!" << endl;
return 0;
}
                 void simon(int n) // define the simon() function
{
        using namespace std;
           cout << "Simon says touch your toes " << n << " times." << endl;
} // void functions don't need return statements
```

The main() function calls the simon() function twice, once with an argument of 3 and once with a variable argument count. In between, the user enters an integer that's used to set the value of count. The example doesn't use a newline character in the cout prompting message. This results in the user input appearing on the same line as the prompt. Here is a sample run of the program above.

```
Simon says touch your toes 3 times.
Pick an integer: 512
Simon says touch your toes 512 times.
Done!
```

Example : A program to find factorial of number using user define function

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

```
// This function calculates the factorial of a number.
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
int main() {
    int number;
    cout << "Enter a number: ";
    cin >> number;
    int factorialOfNumber = factorial(number);
    cout << "The factorial of " << number << " is " << factorialOfNumber << endl;
    return 0;
}
```

Sure, here is an example of a C++ program that uses a user-defined function:

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

// This function calculates the factorial of a number.
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main() {
    int number;
    cout << "Enter a number: ";
    cin >> number;
```

```
    int factorialOfNumber = factorial(number);
    cout << "The factorial of " << number << " is " << factorialOfNumber << endl;
    return 0;
}
```

This program first defines a function called factorial(). The factorial() function takes an integer as input and returns the factorial of that number. The factorial of a number is the product of all the positive integers less than or equal to the number. For example, the factorial of 5 is 120.

The factorial() function is recursive, which means that it calls itself. The base case of the recursion is when the input number is 0.

The output of the program should be:

Enter a number: 5

The factorial of 5 is 120

Inline, Overloaded and recursive functions, Passing and default Arguments

Inline functions are functions that are expanded in line when they are called. This means that the code for the function is copied into the place where the function is called, instead of the function being called and the code for the function being executed at a later time.

Inline functions can improve the performance of a program by reducing the overhead of function calls. However, inline functions can also make a program larger and more difficult to read.

```
#include <iostream>
using namespace std;
inline int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main() {
    int number;
    cout << "Enter a number: ";
    cin >> number;
    cout << "The factorial of " << number << " is " << factorial(number) << endl;
```

```
    return 0;
}
```

Overloaded functions are functions that have the same name but different parameters. Overloaded functions are used to allow a function to be called with different types of arguments. For example, the following code defines two overloaded functions called add():

```
int add(int x, int y) {
    return x + y;
}

double add(double x, double y) {
    return x + y;
}
```

The first add() function takes two integers as arguments and returns the sum of the two integers. The second add() function takes two doubles as arguments and returns the sum of the two doubles.

Overloaded functions can make a program more concise and easier to read. However, overloaded functions can also make a program more difficult to understand, especially if the functions have similar names.

Example

```
#include <iostream>
using namespace std;
int add(int x, int y) {
    return x + y;
}

double add(double x, double y) {
    return x + y;
}

int main() {
    int a = 10, b = 20;
    double c = 10.5, d = 20.5;

    cout << "The sum of a and b is " << add(a, b) << endl;
    cout << "The sum of c and d is " << add(c, d) << endl;

    return 0;
}
```

This program first defines two overloaded functions called add(). The first add() function takes two integers as arguments and returns the sum of the two integers. The second add() function takes two doubles as arguments and returns the sum of the two doubles.

The output of the program should be:
The sum of a and b is 30
The sum of c and d is 31

Recursive functions are functions that call themselves. Recursive functions are used to solve problems that can be broken down into smaller problems of the same type.

For example, the following code defines a recursive function called factorial() that calculates the factorial of a number:

```
int factorial(int n) {
   if (n == 0) {
      return 1;
   } else {
      return n * factorial(n - 1);
   }
}
```

The factorial() function is recursive because it calls itself. The base case of the recursion is when the input number is 0. In this case, the function simply returns 1. Otherwise, the function recursively calls itself, passing the input number minus 1 as an argument. The recursive call will eventually reach the base case, and the function will return 1. Recursive functions can be a powerful tool for solving problems. However, recursive functions can also be difficult to understand and debug.

Example

```cpp
#include <iostream>

using namespace std;

int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main() {
    int number;
    cout << "Enter a number: ";
    cin >> number;
    cout << "The factorial of " << number << " is " << factorial(number) << endl;
    return 0;
}
```

Passing arguments in C++ can be done by value or by reference. When an argument is passed by value, a copy of the argument is made and passed to the function. When an argument is passed by reference, the address of the argument is passed to the function.

Passing arguments by value is the default in C++. Passing arguments by reference can be used to improve the performance of a program by avoiding the need to make a copy of the argument. However, passing arguments by reference can also make a program more difficult to understand and debug.

Default arguments are arguments that have a default value that is used if the argument is not passed to the function. Default arguments can be used to make a function more concise and easier to use.

For example, the following code defines a function called print() that prints a message to the console:

```
void print(string message = "Default message") {
   cout << message << endl;
}
```

The print() function has a default argument for the message parameter. If the message parameter is not passed to the function, the default value "Default message" is used.

Default arguments can be a useful tool for making a function more concise and easier to use. However, default arguments can also make a function more difficult to understand and debug.

classes, methods, objects and member objects.

INTRODUCING ARRAYS

An array is a data structure that stores a collection of elements of the same data type. The elements of an array are stored in contiguous memory locations, which means that they are stored one after the other. For example, an array can hold 60 type int values that represent five years of game sales data, 12 short values that represent the number of days in each month, or 365 float values that indicate your food expenses for each day of the year. Each value is stored in a separate array element, and the computer stores all the elements of an array consecutively in memory.

Here are some of the advantages of using arrays:

- Arrays are efficient for storing and accessing data.
- Arrays are easy to use.
- Arrays are a versatile data structure.

Here are some of the disadvantages of using arrays:

- Arrays can be difficult to manage if they are large.
- Arrays can be inefficient if they are not used efficiently.
- Arrays can be difficult to debug if they are not used correctly.

Application of an array

- Storing the scores of a test: You can store the scores of a test in an array. The array can be used to calculate the average score, find the highest score, and find the lowest score.
- Storing the names of students in a class: You can store the names of students in a class in an array. The array can be used to print the names of the students, find the student with the highest GPA, and find the student with the lowest GPA.
- Storing the elements of a matrix: You can store the elements of a matrix in an array. The array can be used to perform matrix multiplication, matrix addition, and matrix subtraction.
- Storing the characters of a string: You can store the characters of a string in an array. The array can be used to find the length of the string, find the first occurrence of a character in the string, and find the last occurrence of a character in the string.

To create an array, you use a declaration statement. An array declaration should indicate three things:

- The type of value to be stored in each element
- The name of the array
- The number of elements in the array

You accomplish this in C++ by modifying the declaration for a simple variable and adding brackets that contain the number of elements.:

```
type arrayName[size];
```

For example, the following code declares an array of integers called myArray with a size of 10:

```
int myArray[10];
```

The elements of an array can be accessed using the following syntax:

```
arrayName[index];
```

For example, the following code accesses the element at index 5 of the myArray array:

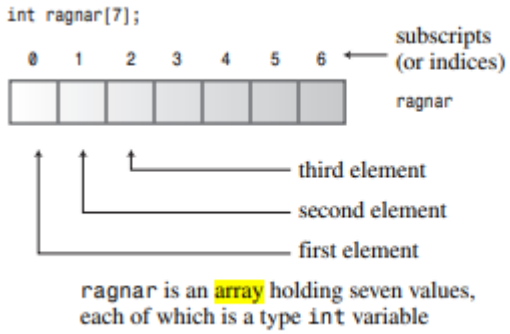
```
int element = myArray[5];
```

Arrays can be initialized when they are declared. The following code initializes the myArray array with the values 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10:

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

Arrays can be used to store a variety of data types, including integers, floats, strings, and objects. an array declaration enables you to create a lot of variables with a single declaration, and you can then use an index to identify and access individual elements

Creating an Array



Example: Write a program to print elements of an array

```
#include <iostream>
using namespace std;
int main() {
   int myArray[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

   for (int i = 0; i < 10; i++) {
      cout << myArray[i] << endl;
   }
   return 0;
}
```

This program declares an array of integers called myArray with a size of 10. The program then loops through the array and prints the value of each element to the console.

Example: Program to find the sum of the elements of an array

```
#include <iostream>
using namespace std;
   int main() {
   int myArray[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   int sum = 0;
   for (int i = 0; i < 10; i++) {
      sum += myArray[i];
   }
   cout << "The sum is " << sum << endl;
   return 0;
}
```

This program declares an array of integers called myArray with a size of 10. The program then loops through the array and adds the value of each element to the variable sum. The program then prints the value of sum to the console.

Example; Write Program to find the largest element of an array

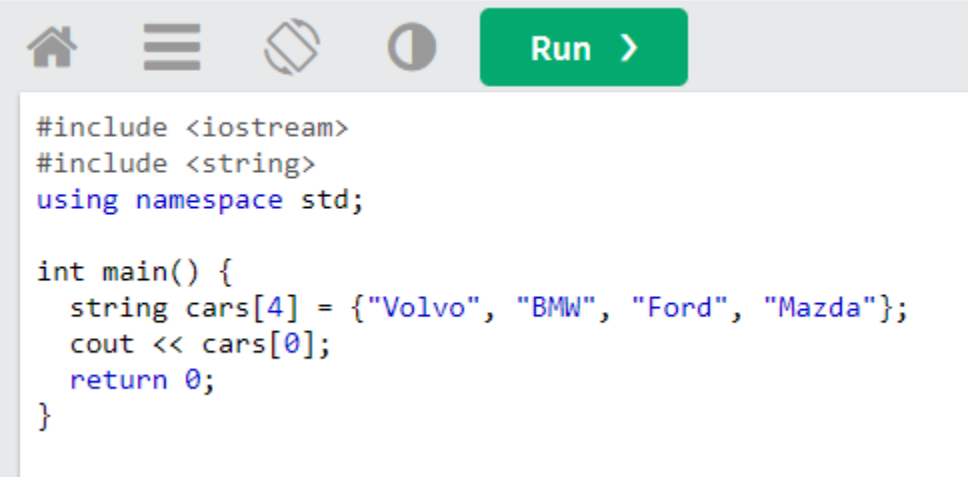
```
#include <iostream>
using namespace std;
int main() {
   int myArray[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   int largest = myArray[0];
   for (int i = 1; i < 10; i++) {
      if (myArray[i] > largest) {
         largest = myArray[i];
      }
   }
   cout << "The largest element is " << largest << endl;
   return 0;
}
```

This program declares an array of integers called myArray with a size of 10. The program then loops through the array and compares the value of each element to the variable largest. If the value of an element is greater than largest, then largest is updated to the value of the element. The program then prints the value of largest to the console.

Access the Elements of an Array

You access an array element by referring to the index number inside square brackets [].

This statement accesses the value of the **first element** in **cars**:



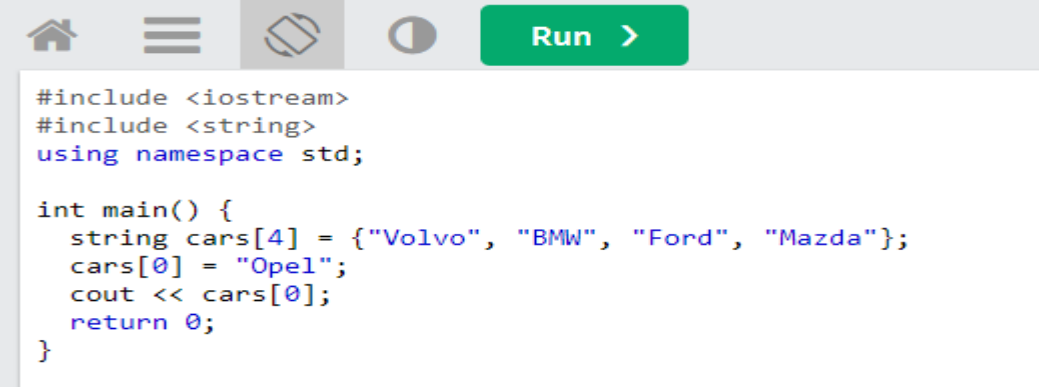
Note: Array indexes start with 0: [0] is the first element. [1] is the second element, etc

Change an Array Element

To change the value of a specific element, refer to the index number:

```
cars[0] = "Opel";
```

Example



Class and member arrays:

A class is a user-defined data type that can be used to create objects. A member array is an array that is declared as a member of a class. Member arrays can be used to store data that is related to the objects of a class. For example, the following code defines a class called Student with a member array called scores:

```
class Student {
public:
   int id;
   string name;
   int scores[10];
};
```

The scores member array can be used to store the scores of a student.

One-dimensional arrays:

A one-dimensional array is an array that has only one dimension. One-dimensional arrays are declared using the following syntax:

type arrayName[size];

For example, the following code declares a one-dimensional array of integers called myArray with a size of 10:

int myArray[10];

Multidimensional arrays:

A multidimensional array is an array that has two or more dimensions. Multidimensional arrays are declared using the following syntax:

type arrayName[size1][size2];

The myMatrix array can be used to store a matrix of integers.

Example: Write a Program to store scores of 10 students

```
#include <iostream>
using namespace std;
int main() {
   // Declare an array of integers called `scores` with a size of 10.
   int scores[10];

   // Initialize the elements of the array with the scores of a test.
   scores[0] = 90;
   scores[1] = 85;
   scores[2] = 75;
   scores[3] = 65;
   scores[4] = 55;
   scores[5] = 45;
   scores[6] = 35;
   scores[7] = 25;
   scores[8] = 15;
   scores[9] = 5;

   // Calculate the average score.
   int sum = 0;
   for (int i = 0; i < 10; i++) {
      sum += scores[i];
   }
   float average = sum / 10.0;

   // Print the average score.
   cout << "The average score is " << average << endl;

   // Find the highest score.
```

```
    int highestScore = scores[0];
    for (int i = 1; i < 10; i++) {
        if (scores[i] > highestScore) {
            highestScore = scores[i];
        }
    }

    // Print the highest score.
    cout << "The highest score is " << highestScore << endl;

    return 0;
}
```

This program declares an array of integers called scores with a size of 10. The program then initializes the elements of the array with the scores of a test. The program then calculates the average score, finds the highest score, and prints the results to the console.

Examples: Write a program that stores the names of students in a class:

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

int main() {
    // Declare an array of strings called `students` with a size of 10.
    string students[10];

    // Initialize the elements of the array with the names of students in a class.
    students[0] = "John Doe";
    students[1] = "Jane Doe";
    students[2] = "Mary Smith";
    students[3] = "Peter Jones";
    students[4] = "Susan Williams";
    students[5] = "David Brown";
    students[6] = "Elizabeth Green";
    students[7] = "Michael White";
    students[8] = "Sarah Black";
    students[9] = "James Blue";

    // Print the names of the students.
    for (int i = 0; i < 10; i++) {
        cout << students[i] << endl;
    }

    return 0;
}
```

Example: Write a program that stores the elements of a matrix:

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

```
    // Declare an array of integers called `matrix` with a size of 10x10.
    int matrix[10][10];
    // Initialize the elements of the array with the elements of a matrix.
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++) {
            matrix[i][j] = i * j;
        }
    }
    // Print the elements of the matrix.
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Example: Write a program that stores the characters of a string:

```
#include <iostream>
using namespace std;
int main() {
    // Declare an array of characters called `string` with a size of 10.
    char string[10];

    // Initialize the elements of the array with the characters of a string.
    string[0] = 'J';
    string[1] = 'o';
    string[2] = 'h';
    string[3] = 'n';
    string[4] = ' ';
    string[5] = 'D';
    string[6] = 'o';
    string[7] = 'e';
    string[8] = ' ';
    string[9] = '\0';

    // Print the characters of the string.
    for (int i = 0; string[i] != '\0'; i++) {
        cout << string[i];
    }
    return 0;
}
```

Arrays as arguments

arrays can be passed as arguments to functions. When an array is passed as an argument to a function, the entire array is passed to the function. This means that the function can access all of the elements of the array.

There are two ways to pass an array as an argument to a function in C++:

- Passing by reference: When an array is passed by reference, the address of the array is passed to the function. This means that the function can modify the elements of the array.
- Passing by value: When an array is passed by value, a copy of the array is passed to the function. This means that the function cannot modify the elements of the array.

Here is an example of how to pass an array as an argument to a function in C++:

```
void printArray(int *array, int size) {
   for (int i = 0; i < size; i++) {
      cout << array[i] << endl;
   }
}

int main() {
   int myArray[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   printArray(myArray, 10);

   return 0;
}
```

In this example, the printArray() function takes two arguments: an array of integers and the size of the array. The printArray() function prints the elements of the array to the console.

The printArray() function takes the address of the array as an argument. This means that the function can modify the elements of the array. The main() function creates an array of integers called myArray with a size of 10. The main() function then calls the printArray() function, passing the myArray array and the size of the array as arguments.

The printArray() function prints the elements of the myArray array to the console.

Example: Write a Program for Yam analysis

```
// arrayone.cpp -- small arrays of integers
#include <iostream>
int main()
{
using namespace std;
int yams[3]; // creates array with three elements
yams[0] = 7; // assign value to first element
yams[1] = 8;
yams[2] = 6;
int yamcosts[3] = {20, 30, 5}; // create, initialize array
// NOTE: If your C++ compiler or translator can’t initialize
// this array, use static int yamcosts[3] instead of
```

```
// int yamcosts[3]
cout << "Total yams = ";
cout << yams[0] + yams[1] + yams[2] << endl;
cout << "The package with " << yams[1] << " yams costs ";
cout << yamcosts[1] << " cents per yam.\n";
int total = yams[0] * yamcosts[0] + yams[1] * yamcosts[1];
total = total + yams[2] * yamcosts[2];
cout << "The total yam expense is " << total << " cents.\n";
cout << "\nSize of yams array = " << sizeof yams;
cout << " bytes.\n";
cout << "Size of one element = " << sizeof yams[0];
cout << " bytes.\n";
return 0;
}
```

Here is the output from the program

```
Total yams = 21
The package with 8 yams costs 30 cents per yam.
The total yam expense is 410 cents.
Size of yams array = 12 bytes.
Size of one element = 4 bytes.
```

The program in the yam analysis creates a three-element array called yams. Because yams have three elements, the elements are numbered from 0 through 2, and arrayone.cpp uses index values of 0 through 2 to assign values to the three individual elements. Each individual yam element is an int with all the rights and privileges of an int type, so arrayone.cpp can, and does, assign values to elements, add elements, multiply elements, and display elements.

Pointer, arithmetic pointer, array pointer

A pointer is a variable that stores the address of another variable. Pointers can be used to access the memory location of a variable, or to pass the address of a variable to a function.

For example, the following code declares a pointer to an integer:

```
int* p;
```

The variable p is a pointer to an integer. The asterisk (*) indicates that p is a pointer. The value of p is initially undefined.

We can assign the address of an integer variable to p using the & operator:

```
int x = 10;
p = &x;
```

The & operator returns the address of the variable x. We can then use p to access the value of x. For example, the following code prints the value of x:

```
cout << *p << endl;
```

The asterisk *() in front of p dereferences p and returns the value of the variable that p points to. In this case, the value of p is the address of x, so the asterisk ()* returns the value of x.

Arithmetic Pointer

Pointer arithmetic is the process of adding or subtracting integers to or from a pointer. This can be used to move the pointer to a different location in memory.

For example, the following code increments the value of p by 1:

```
p++;
```

This moves the pointer to the next integer in memory. The next integer in memory is the address of the variable x. So, the value of p is now the address of x.

We can also subtract integers from pointers. For example, the following code decrements the value of p by 1:

```
p--;
```

This moves the pointer to the previous integer in memory. The previous integer in memory is the address of the variable y. So, the value of p is now the address of y.

Array Pointer

An array pointer is a pointer that points to the first element of an array. The array name is also a pointer to the first element of the array. For example, the following code declares an array of integers and a pointer to the array:

```
int arr[10];
int* p = arr;
```

The variable arr is an array of 10 integers. The variable p is a pointer to the first element of the array. The value of p is the address of the variable arr[0]. We can use the array pointer to access the elements of the array. For example, the following code prints the value of the first element of the array:

```
cout << *p << endl;
```

The asterisk () in front of p dereferences p and returns the value of the variable that p points to. In this case, the value of p is the address of arr[0], so the asterisk () returns the value of arr[0].

Meta Class and Objects in C++

- Metaclass: A metaclass is a class whose instances are classes. In other words, a metaclass is a class that can be used to create other classes. This allows for a great deal of flexibility and control over the creation of classes.
- Object: An object is an instance of a class. An object has a state, which is represented by its member variables, and a behavior, which is represented by its member functions.

In C++, metaclasses are not supported natively. However, there are a number of ways to implement metaclasses using C++. One common way is to use the typeinfo class. The typeinfo class provides information about a class, including its name, its base classes, and its member functions. This information can be used to create a metaclass.

Another way to implement metaclasses in C++ is to use the reflection library. The reflection library provides a set of classes and functions that can be used to reflect on the structure of a class. This information can be used to create a metaclass.

Metaclasses can be used for a variety of purposes. For example, they can be used to:

- Create classes dynamically at runtime.
- Extend or modify the behavior of existing classes.
- Inspect the structure of classes.
- Generate code from classes.

Metaclasses can be a powerful tool for C++ programmers. However, they can also be complex and difficult to use. If you are considering using metaclasses, it is important to understand the implications and to use them carefully.

```
 #include <iostream>
#include <typeinfo>
using namespace std;
class Metaclass {
public:
   virtual void print() = 0;
};

class ClassA : public Metaclass {
public:
   void print() {
      cout << "This is a ClassA object." << endl;
   }
};

class ClassB : public Metaclass {
public:
   void print() {
      cout << "This is a ClassB object." << endl;
```

```
    }
};

int main() {
    Metaclass *metaclass;

    // Create a ClassA object.
    ClassA *classA = new ClassA();

    // Get the metaclass for the ClassA object.
    metaclass = typeid(classA).name();

    // Print the metaclass.
    cout << metaclass << endl;

    // Create a ClassB object.
    ClassB *classB = new ClassB();

    // Get the metaclass for the ClassB object.
    metaclass = typeid(classB).name();

    // Print the metaclass.
    cout << metaclass << endl;

    return 0;
}
```

This program first creates two classes, ClassA and ClassB. Both classes inherit from the Metaclass class, which has a single virtual function, print(). The print() function simply prints a message that identifies the type of the object.

The main function of the program creates two objects, one of type ClassA and one of type ClassB. It then gets the metaclass for each object and prints it out. The metaclass for a class is a pointer to the class's type information.

The output of the program is as follows:

```
Metaclass ClassA
Metaclass ClassB
```

As you can see, the metaclass for a class is a pointer to the class's type information. This information can be used to identify the class and to access its member functions and variables.

This is just a simple example of how metaclasses can be used in C++. Metaclasses can be used for a variety of purposes, such as creating classes dynamically at runtime, extending or modifying the behavior of existing classes, inspecting the structure of classes, and generating code from classes.

There are two main types of objects in C++:

- Class objects: These are objects that are created from a class. Class objects have a state, which is represented by their member variables, and a behavior, which is represented by their member functions.
- Built-in objects: These are objects that are built into the C++ language. Built-in objects include integers, floating-point numbers, strings, and characters.

In addition to these two main types of objects, there are also a number of other types of objects in C++, such as:

- Pointer objects: These are objects that point to other objects. Pointer objects can be used to access the data and functions of the object that they point to.
- Reference objects: These are objects that refer to other objects. Reference objects cannot be used to change the data of the object that they refer to.
- Null objects: These are objects that have no value. Null objects are often used to represent the absence of an object.

The type of object that is used in a C++ program depends on the specific needs of the program. For example, if a program needs to store data about a customer, then a class object would be used. If a program needs to perform mathematical calculations, then a built-in object would be used.

Type	Description
Class object	An object that is created from a class.
Built-in object	An object that is built into the C++ language.
Pointer object	An object that points to other objects.
Reference object	An object that refers to other objects.
Null object	An object that has no value.

Concept of object life time

The lifetime of an object in C++ is the period of time between its creation and its destruction. The lifetime of an object is determined by its scope. The scope of an object is the part of the program where the object is visible.

There are two main types of object lifetimes in C++:

- Automatic lifetime: Objects with automatic lifetime are created on the stack and destroyed when the function that created them returns.

- Dynamic lifetime: Objects with dynamic lifetime are created on the heap and destroyed when they are explicitly deleted.

  Here is a table that summarizes the two main types of object lifetimes in C++:

Type	Description
Automatic lifetime	Objects with automatic lifetime are created on the stack and destroyed when the function that created them returns.
Dynamic lifetime	Objects with dynamic lifetime are created on the heap and destroyed when they are explicitly deleted.

The lifetime of an object can also be affected by the following:

- Reference counting: Some objects in C++ use reference counting to manage their lifetime. When an object is created, its reference count is set to 1. When an object is no longer needed, its reference count is decremented. If the reference count reaches 0, the object is destroyed.
- Garbage collection: Some C++ compilers support garbage collection. Garbage collection is a process that automatically manages the lifetime of objects. When an object is no longer needed, it is automatically destroyed.

The lifetime of an object is an important concept in C++. It is important to understand the lifetime of objects in order to avoid memory leaks and other errors.

For example, the following code creates an object with automatic lifetime:

```
int main() {
   int x = 10; // Object with automatic lifetime
   return 0;
}
```

The object x is created on the stack when the main() function is called. The object x is destroyed when the main() function returns.

The following code creates an object with dynamic lifetime:

```
int main() {
   int *x = new int(10); // Object with dynamic lifetime
   delete x;
   return 0;
}
```

.

Here are some tips for avoiding memory leaks:

- Always delete objects that you no longer need.
- Use a memory management system that automatically manages the lifetime of objects.
- Use tools to detect memory leaks.

Object-Oriented Programming(OOP)

Procedural programming is about writing procedures or functions that perform operations on the data, while object-oriented programming is about creating objects that contain both data and functions.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the C++ code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

Tip: The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

Classes and Objects

Classes and objects are the two main aspects of object-oriented programming.

Look at the following illustration to see the difference between class and objects:

Another example:

So, a class is a template for objects, and an object is an instance of a class.

When the individual objects are created, they inherit all the variables and functions from the class.

- Classes are user-defined data types in C++. Classes can be used to represent real-world objects, such as cars, people, and houses. Classes contain data members, which store the data of the object, and member functions, which perform operations on the data of the object.
- Methods are functions that are defined within a class. Methods can be used to access and modify the data members of the class. Methods can also be used to perform operations on the data members of the class.
- Objects are instances of classes. Objects have the same data members and member functions as the class that they were created from. However, each object has its own unique set of data members, which are initialized when the object is created.
- Member objects are objects that are defined within a class. Member objects can be used to represent the parts of an object. For example, a car class might have a member object that represents the engine of the car.

Here is an example of a class in C++:

```
class Car {
public:
    int id;
    string make;
    string model;
    int year;

    void drive() {
        cout << "The car is driving." << endl;
    }
};
```

```
#include <iostream>

using namespace std;

class Car {
public:
   int id;
   string make;
   string model;
   int year;

   void drive() {
      cout << "The car is driving." << endl;
```

```
    }
};

int main() {
    Car myCar;
    myCar.id = 1;
    myCar.make = "Honda";
    myCar.model = "Accord";
    myCar.year = 2023;

    myCar.drive();

    return 0;
        }
```

This program first defines a class called Car. The Car class has four data members: id, make, model, and year. The class also has one member function: drive(). The drive() function prints a message to the console that says "The car is driving."

The main function of the program creates a car object called myCar. The myCar object is initialized with the values 1, "Honda", "Accord", and 2023 for the id, make, model, and year data members, respectively. The myCar object then calls the drive() function to print a message to the console that says "The car is driving."

The output of the program should be:

The car is driving.

This class defines a car object. The class has four data members: id, make, model, and year. The class also has one member function: drive(). The drive() function prints a message to the console that says "The car is driving."

Here is an example of how to create an object from the Car class:

```
Car myCar;
myCar.id = 1;
myCar.make = "Honda";
myCar.model = "Accord";
myCar.year = 2023;

myCar.drive();
```

This code creates a car object called myCar. The myCar object is initialized with the values 1, "Honda", "Accord", and 2023 for the id, make, model, and year data members, respectively. The myCar object then calls the drive() function to print a message to the console that says "The car is driving."

Constructor methods, constructor call methods and inline methods

- Constructor methods are special methods that are called when an object is created. Constructor methods can be used to initialize the data members of an object.
- Constructor call methods are methods that are used to call the constructor methods of other classes. Constructor call methods are useful when you need to create an object of a class that depends on another class.
- Inline methods are methods that are expanded in line when they are called. This means that the code for the method is copied into the place where the method is called, instead of the method being called and the code for the method being executed at a later time.

Here is an example of a constructor method:

```
class Car {
public:
    Car() {
        id = 0;
        make = "";
        model = "";
        year = 0;
    }

    int id;
    string make;
    string model;
    int year;
};
```

This code defines a constructor method for the Car class. The constructor method is called Car(). The Car() constructor method initializes the data members of the Car object to 0, "", "", and 0, respectively.

Here is an example of a constructor call method:

```
class Engine {
public:
    Engine() {}
};

class Car {
public:
    Car() {
        engine = new Engine();
    }

    Engine *engine;
};
```

This code defines a constructor call method for the Car class. The constructor call method calls the constructor method of the Engine class to create an Engine object. The Engine object is then stored in the engine data member of the Car object.

Here is an example of an inline

```
inline int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

This code defines an inline method called factorial(). The factorial() method calculates the factorial of a number. The factorial() method is inline, which means that the code for the method is copied into the place where the method is called, instead of the method being called and the code for the method being executed at a later time.

The concept of inheritance, Polymorphism and overloading in polymorphism

- Inheritance is a feature of object-oriented programming that allows a new class to inherit the properties and methods of an existing class. The new class is called the derived class, and the existing class is called the base class.
- Polymorphism is a feature of object-oriented programming that allows objects of different classes to respond to the same message in different ways. Polymorphism is achieved through inheritance and overloading.
- Overloading is a feature of C++ that allows a function to have multiple definitions with the same name, but with different parameters. Overloading is used to allow a function to be called with different types of arguments.

Here is an example of inheritance in C++:

```
class Animal {
public:
    void speak() {
        cout << "I am an animal." << endl;
    }
};

class Dog : public Animal {
public:
    void speak() {
        cout << "I am a dog." << endl;
    }
};
```

```
class Cat : public Animal {
public:
    void speak() {
        cout << "I am a cat." << endl;
    }
};

int main() {
    Animal *animal = new Dog();
    animal->speak();

    animal = new Cat();
    animal->speak();

    return 0;
}
```

This code defines three classes: Animal, Dog, and Cat. The Animal class is the base class. The Dog class and the Cat class are derived classes. The Dog class and the Cat class inherit the speak() method from the Animal class.

The main function of the program creates a Dog object and a Cat object. The Dog object and the Cat object are then used to call the speak() method. The speak() method of the Dog object prints "I am a dog." to the console. The speak() method of the Cat object prints "I am a cat." to the console.

In this example, polymorphism is achieved through inheritance. The speak() method of the Animal class is overridden in the Dog class and the Cat class. This allows the speak() method to behave differently depending on the type of object that it is called on.

Here is an example of overloading in C++:

```
int add(int x, int y) {
    return x + y;
}

int add(int x, int y, int z) {
    return x + y + z;
}

int main() {
    cout << add(1, 2) << endl; // 3
    cout << add(1, 2, 3) << endl; // 6

    return 0;
}
```

This code defines two functions called add(). The first add() function takes two integers as arguments and returns the sum of the two integers. The second add() function takes three integers as arguments and returns the sum of the three integers.

The main function of the program calls the add() function twice. The first time, the add() function is called with two arguments. The second time, the add() function is called with three arguments.
The output of the program should be:

3

6

In this example, overloading is achieved by having two functions with the same name, but with different parameters. The compiler can distinguish between the two functions by the number and type of the parameters.