**\*Object-oriented programming** (**OOP**) is a [programming paradigm](https://en.wikipedia.org/wiki/Programming_paradigm) based on the concept of "[objects](https://en.wikipedia.org/wiki/Object_(computer_science))", which can contain [data](https://en.wikipedia.org/wiki/Data) and [code](https://en.wikipedia.org/wiki/Computer_program). The data is in the form of [fields](https://en.wikipedia.org/wiki/Field_(computer_science)) (often known as [attributes](https://en.wikipedia.org/wiki/Attribute_(computing)) or *properties*), and the code is in the form of procedures (often known as [*methods*](https://en.wikipedia.org/wiki/Method_(computer_science))).

Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behavior.

**StructuredProgramming:**  
Structured Programming, as name suggests, is a technique that is considered as precursor to OOP and usually consists of well-structured and separated modules. In this programming, user can create its own user-defined functions as well as this methodology tries to resolve issues that are associated with unconditional transfers to allow programmers follow logic of programs. It also requires more discipline at the design and logical structuring stage.

**Example :** Pascal, ALGOL, C, Modula-2, etc.

[**Object-OrientedProgramming**](https://www.geeksforgeeks.org/object-oriented-programming-in-cpp/)**:**  
Object-Oriented Programming, as name suggests, is a different approach to programming that brings together data and functions that execute on them. It basically supports encapsulation, abstraction, inheritance, polymorphism, etc. It also includes data hiding feature therefore it is more secure. This model is based on real life an entity that focuses on by whom task is to be done rather than focusing on what to do.

|  |  |
| --- | --- |
| **Structured Programming** | **Object-Oriented Programming** |
| It is a subset of procedural programming. | It relies on concept of objects that contain data and code. |
| Programs are divided into small programs or functions. | Programs are divided into objects or entities. |
| It is all about facilitating creation of programs with readable code and reusable components. | It is all about creating objects that usually contain both functions and data. |
| Its main aim is to improve and increase quality, clarity, and development time of computer program. | Its main aim is to improve and increase both quality and productivity of system analysis and design. |
| It simply focuses on functions and processes that usually work on data. | It simply focuses on representing both structure and behavior of information system into tiny or small modules that generally combines data and process both. |
| It is a method of organizing, managing and coding programs that can give or provide much easier modification and understanding. | It is a method in which set of objects can vary dynamically and can execute just by acting and reading to each other. |
| In this, methods are written globally and code lines are processed one by one i.e., Run sequentially. | In this, method works dynamically, make calls as per need of code for certain time. |
| It generally follows “Top-Down Approach”. | It generally follows “Bottom-Up Approach”. |
| It provides less flexibility and abstraction as compared to object-oriented programming. | It provides more flexibility and abstraction as compared to structured programming. |
| It is more difficult to modify structured program and reuse code as compared to object-oriented programs. | It is less difficult to modify object-oriented programs and reuse code as compared to structured programs. |
| It gives more importance of code. | It gives more importance to data. |

## \*Top C++ Features

1. **Simple**

We expect to understand a new programming language thoroughly when we start using it. However, C++ is one of the most-simple programming languages. As long as you are familiar with the C language, moving to C++ will be very easy. Unlike other programming languages, C++ provides a structured approach (programming paradigm that facilitates the creation of programs with readable code and reusable components.) and allows you to design a modular solution by dividing the problem into parts. In addition, you have access to a variety of library functions and data types that can be used when implementing the solution. For any type of project development, C++ uses the bottom-up approach, which helps break large tasks down into small pieces.

1. **Object Oriented**

C++ is an Object-Oriented Programming Language (OOP). OOPs are the most important feature of C++ since they enable easier development and maintenance. Objects (instances of a class) can be created and destroyed while programming. C++ follows the following concept of OOPs to make development and maintenance of the software system easier:

* **Polymorphism:** Poly means “many” and morphism means “form”. Thus, polymorphism enhances flexible programming by allowing data to be displayed in more than one form.
* **Inheritance:** Inheritance is the process of allowing a class to inherit the features or properties (fields and methods) of another class. As a result, the program is less redundant, and it is easy to move properties from one class to another.
* **Data Encapsulation:** Essentially, encapsulation means packaging together relevant data and its associated methods and functions into a single unit/block.
* **Data Abstraction:** It refers to the process of displaying only the key details/information of data to users without including the underlying/background details or the method used to gather it. Units that are trivial or not essential are not displayed.
* **Data Hiding:**It is the process of protecting data from unauthorized access. Essentially, it secures the data.

1. **Machine Independent and Platform-Dependent**

C++ is machine-independent, which means programs can be executed on many machines with little or no changes. C++, however, is not platform-independent, so programs can only be executed on the platform on which they were developed & compiled. The compiler produces an .exe file that is OS-dependent, and cannot be run on different operating systems. Suppose you wrote a program or piece of code that runs on Linux, Windows, and Mac OSX, making it Machine Independent, but the executable file of the C++ program cannot run on different Operating systems/platforms.

1. **Popular**

With its support for object-oriented programming, C++ can be a good basis for a number of other programming languages. C++ is a favorite today for games development and complex business applications because of its incredibly fast speed and precise memory management. The language is applied to a variety of different fields requiring high-performance applications. Adobe products, Unreal Engine, and popular browsers such as Chrome and Firefox are all built using C++.

1. **Case-Sensitive**

C++ is case-sensitive, which means that it treats uppercase and lowercase characters differently. For instance, when we write cout (used to display the output) as ‘Cout’ or “COUT”, the meaning of the term changes. Some programming languages, such as HTML and MySQL, do not care about case.

1. **Mid-level programming language**

Having the features of both a low-level (machine-level) language and a high-level language (user-oriented language that is easily understandable and close to human language), C++ is termed a mil-level or intermediate programming language. C++ is used to develop low-level system applications, such as kernels, drivers, etc., and high-level applications, such as GUIs, desktop applications, etc.

1. **Structured Programming Language**

C++ is a structured programming language. Programming in C++ is modular i.e. we are able to break the complex code into smaller pieces (sudividing a computer program into separate sub-programs) and combine them using functions, classes, and objects. This results in cleaner code and makes maintenance much easier.

1. **Rich Library**

C++ provides developers with many built-in functions or libraries that save developers time and make development faster. Some of the header files include:

* **<iostream>:**Include C++ standard input and output functions
* **<cmath>:** Include math library functions
* **<cstdlib>:** Includes functions that convert numbers to text and vice versa, allocate memory, create random numbers, and perform various other functions.
* **<fstream>:** Include functions to create files, write information to files, and read information from files.
* **<iterator>:** Include classes for accessing data in the C++ Standard Library containers
* **<algorithm>:** Include functions for manipulating data in C++ Standard Library containers

1. **Memory Management**

The C++ language allows dynamic memory allocation, which means that we can allocate memory for a variable, object, or array at run time. In C++, memory is divided into stack and heap storage. Stacks refer to the memory allocated for variables inside functions. The heap, on the other hand, refers to unused memory that can be dynamically allocated. If you don’t know the amount of memory required to store information in a defined variable, the size can be determined manually during runtime. New and delete operators are used to allocate and deallocate memory, respectively.

1. **Powerful and Fast**

C++ execution time and compiling time are faster than any other programming language. In addition, it supports a wide range of data types, functions, and operators. As a result, operating systems, browsers, games, and so forth, can be developed. C++ is an advanced language that supports a wide range of programming techniques, like procedural, object-oriented, functional, etc. Due to this, C++ is very powerful and flexible.

**Pointer**

A pointer is a variable in C++ that stores the address of another variable. They point to a variable’s location in memory. Pointers are supported by C++ and can solve a large number of problems requiring memory access. A pointer can be used for memory, structures, functions, arrays, etc. Pointers are declared as:

int \*pointVar;

Or

int\* pointVar;

In the above example, var is a variable holding the value ‘5’ and 0x61ff08 is the address of the var (). pointVar is the pointer whose value 0x61ff08 points to the address of the variable var.

**Recursion**

When a function calls itself, it is called a recursive function, and this technique is known as recursion. In this way, the function can be repeated multiple times since it calls itself during its execution. We can call the functions, which conserves memory by not having to repeatedly write the same code. It reduces the length of our code and makes it cleaner.

1. **Integration and Extensibility**

Object-oriented support enables C++ programs to be maintainable and extensible. This means that large scale applications can be created. C++ has the potential to integrate and apply newer features easily. It has been used effectively for many diverse applications, including mobile app and game development, software development (e.g., Image Ready, Adobe Priemere, etc), and web browser development (e.g., Google Chrome, Mozilla Firefox, etc.), as well as bank applications (e.g., Infosys Finacle, etc.).

1. **Compiler based**

It is not possible to run C++ code without compilation, since it is a compiler based programming language. To execute our program, we must first compile it using a compiler that translate the program into machine language which can be understood directly by the system. The resulting program is therefore highly efficient and faster than most interpreter-based languages, such as Python and Java.

1. **Multi-threading**

As a specialized form of multitasking, multithreading allows your system to run several parts/threads of programs simultaneously. The multithreaded program consists of two or more threads (define a separate path of execution) that run simultaneously. Since C++ does not have built-in support for multithreaded applications, it relies entirely on the external libraries to provide this functionality.

# \*I/O operations in C++

Every program takes some data as input and generates processed data as an output following the familiar input process output cycle. It is essential to know how to provide the input data and present the results in the desired form. The [use of the **cin** and **cout**](https://www.geeksforgeeks.org/cincout-vs-scanfprintf/)is already known with the [operator **>>** and **<<**](https://www.geeksforgeeks.org/overloading-stream-insertion-operators-c/) for the [input and output operations](https://www.geeksforgeeks.org/unformatted-input-output-operations-in-cpp/).

In C++, input and output (I/O) operators are used to take input and display output. The operator used for taking the input is known as the **extraction**or get **from operator** (>>), while the operator used for displaying the output is known as the **insertion**or **put to operator**(<<).

* [Input Operator](https://ecomputernotes.com/cpp/introduction-to-oop/input-output-operator-in-cpp#Input_Operator)
* [Output Operator](https://ecomputernotes.com/cpp/introduction-to-oop/input-output-operator-in-cpp#Output_Operator)
* [Cascading of Input/Output Operators](https://ecomputernotes.com/cpp/introduction-to-oop/input-output-operator-in-cpp#Cascading_of_InputOutput_Operators)

## Input Operator

The input operator, commonly known as the extraction operator (>>), is used with the standard input stream, cin. As stated earlier, cin treats data as a stream of characters. These characters flow from cin to the program through the input operator. The input operator works on two operands, namely, the c in stream on its left and a variable on its right. Thus, the input operator takes (extracts) the value through cin and stores it in the variable.

To understand the concept of an input operator, consider this example.

A program to demonstrate the working of an input operator.

#include

using namespace, std;

int main () {

int a;

cin>>a;

a = a+1;

return 0;

}

In this example, the statement cin>> a takes an input from the user and stores it in the variable a.

## Output Operator

The output operator, commonly known as the insertion operator (<<), is used. The standard output stream cout Like cin, cout also treats data as a stream of characters. These characters flow from the program to cout through the output operator. The output operator works on two operands, namely, the cout stream on its left and the expression to be displayed on its right. The output operator directs (inserts) the value to cout.

To understand the concept of output operator, consider this example.

A program to demonstrate the working of an output operator.

#include

using namespace std;

int main () {

int a;

cin>>a;

a=a+1;

cout<<a;

return 0;

}

This example is similar to Example 1. The only difference is that the value of the variable a is displayed through the instruction cout << a .

## Cascading of Input/Output Operators

The cascading of the input and output operators refers to the consecutive occurrence of input or output operators in a single statement.

To understand the concept of cascading of the input/output operator, consider these examples.

A program without cascading of the input/output operator.

#include

using namespace std;

int main () {

int a, b;

cin>>a;

cin>>b;

cout<<"The value of a is";

cout<<a;

cout<<"The value of b is";

cout<<b;

return 0;

}

In this example, all cin and cout statements use separate input and output operators respectively However, these statements can be combined by cascading the input and output operators accordingly as shown in this example.

A program with cascading of the input/output operator

#include

using namespace std;

int main () {

int a, b;

cin>>a>>b;

cout<<"The value of b is : "<<b;

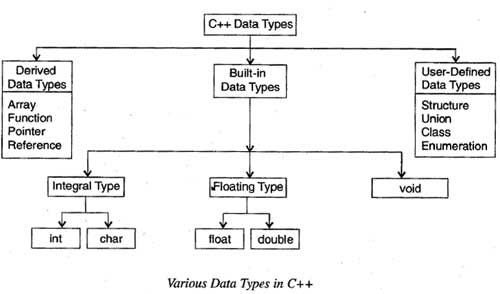
cout<<"The value of a is "<<a;

return 0;

}

In this example, the cascaded input operators wait for the user to input two values and the cascaded output operator first displays the message The value of a is: and then displays the value stored in a. Similar is the case for the next statement.

# \*Data Types

A [data type](https://ecomputernotes.com/java/data-type-variable-and-array/explain-data-types-in-java) determines the type and the operations that can be performed on the data. C++ provides various data types and each [data type](https://ecomputernotes.com/java/data-type-variable-and-array/explain-data-types-in-java) is represented differently within the [computer](https://ecomputernotes.com/fundamental/introduction-to-computer/what-is-computer)’s [memory](https://ecomputernotes.com/fundamental/input-output-and-memory/memory). The various data types provided by C++ are *built-in data types,* *derived data types*and *user-defined data types*as shown in Figure. 

* [Built-In Data Types](https://ecomputernotes.com/cpp/introduction-to-oop/data-types-in-cpp#Built-In_Data_Types)
* [User-Defined Data Types](https://ecomputernotes.com/cpp/introduction-to-oop/data-types-in-cpp#User-Defined_Data_Types)

## Built-InDataTypes

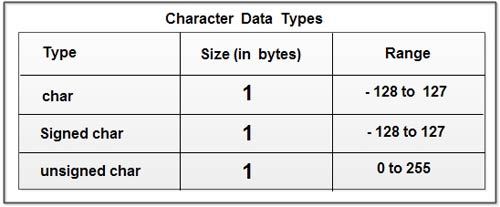
  The basic (fundamental) data types provided by c++ are *integral, floating point*and *void*data type. Among these data types, theintegral and floating-point data types can be preceded by several typemodifiers. These modifiers (also known as type qualifiers) are thekeywords that alter either size or range or both of the data types. Thevarious modifiers are short, long, signed and unsigned. Bydefault the modifier is signed.

In addition to these basic data types, ANSI C++ has introduced two more data types namely, bool and wchar\_t.

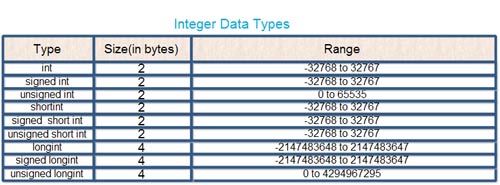
**Integral Data Type**: The integral data type is used to store integers and includes char (character) and int (integer) data types.

***Char:***Characters refer to the alphabet, numbers and other characters (such as {, @, #, etc.) defined in the ASCII character set. In C++, the char data type is also treated as an integer data type as the characters are internally stored as integers that range in value from -128 to 127. The char data type occupies 1 byte of [memory](https://ecomputernotes.com/fundamental/input-output-and-memory/memory) (that is, it holds only one character at a time).

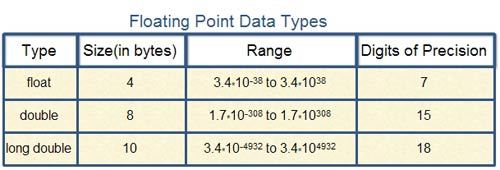
The modifiers that can precede char are signed and unsigned. The various character data types with their size and range are listed in Table



***Int****:*Numbers without the fractional part represent integer data. In C++, the int data type is used to store integers such as 4, 42, 5233, -32, -745. Thus, it cannot store numbers such as 4.28, -62.533. The various integer data types with their size and range are listed in Table



**Floating-point Data Type:**A floating-point data type is used to store real numbers such as 3 .28, 64. 755765, 8.01, -24.53. This data type includes float and double’ data types. The various floating -point data types with their size and range are listed in Table



**Void**: The void data type is used for specifying an empty parameter list to a function and return type for a function. When void is used to specify an empty parameter list, it indicates that a function does not take any arguments and when it is used as a return type for a function, it indicates that a function does not return any value. For void, no memory is allocated and hence, it cannot store anything. As a result, void cannot be used to declare simple variables, however, it can be used to declare generic pointers.

**Bool :** The boo1data type can hold only Boolean values, that is; either true or false, where true represents 1 and false represents O. It requires only one bit of storage, however, it is stored as an integer in the memory. Thus, it is also considered as an integral data type. The bool data type is most commonly used for expressing the results of logical operations performed on the data. It is also used as a return type of a function indicating the success or the failure of the function.

n addition to char data type, C++ provides another data type wchar\_t which is used to store 16- bit wide characters. Wide characters are used to hold large character sets associated with some non-English languages.

#include <iostream>

using namespace std;

int main() {

cout << "Size of char : " << sizeof(char) << endl;

cout << "Size of int : " << sizeof(int) << endl;

cout << "Size of short int : " << sizeof(short int) << endl;

cout << "Size of long int : " << sizeof(long int) << endl;

cout << "Size of float : " << sizeof(float) << endl;

cout << "Size of double : " << sizeof(double) << endl;

cout << "Size of wchar\_t : " << sizeof(wchar\_t) << endl;

return 0;

}

Size of char : 1

Size of int : 4

Size of short int : 2

Size of long int : 4

Size of float : 4

Size of double : 8

Size of wchar\_t : 4

**Derived Data Types**: Data types that are derived from the built-in data types are known as derived data types. The various derived data types provided by C++ are *arrays, junctions, references*and *pointers.*

***Array***An array is a set of elements of the same data type that are referred to by the same name. All the elements in an array are stored at contiguous (one after another) memory locations and each element is accessed by a unique index or subscript value. The subscript value indicates the position of an element in an array.

***Function***A function is a self-contained program segment that carries out a specific well-defined task. In C++, every program contains one or more functions which can be invoked from other parts of a program, if required.

***Reference***A reference is an alternative name for a variable. That is, a reference is an alias for a variable in a program. A variable and its reference can be used interchangeably in a program as both refer to the same memory location. Hence, changes made to any of them (say, a variable) are reflected in the other (on a reference).

***Pointer***A pointer is a variable that can store the memory address of another variable. Pointers allow to use the memory dynamically. That is, with the help of pointers, memory can be allocated or de-allocated to the variables at run-time, thus, making a program more efficient.

## User-DefinedDataTypes

Various user-defined data types provided by C++ are *structures, unions, enumerations*and *classes.*

**Structure, Union andClass:**Structure and union are the significant features of C language. Structure and union provide a way to group similar or dissimilar data types referred to by a single name. However, C++ has extended the concept of structure and union by incorporating some new features in these data types to support object -oriented programming.

C++ offers a new user-defined data type known as class, which forms the basis of object-oriented programming. Class acts as a template which defines the data and functions that are included in an object of a class. Classes are declared using the keyword class. Once a class has been declared, its object can be easily created.

Though the enumerations are treated as integers internally in C++, the compiler issues a warning, if an int value is assigned to an enum type. For example, consider these statements.

## Enumerated Types

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

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

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

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

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

enum color { red, green, blue } c;

c = blue;

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

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

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

## typedef Declarations

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

typedef type newname;

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

typedef int feet;

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

feet distance;

## C++ Variables

Variables are containers for storing data values.

## Declaring (Creating) Variables

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.

To create a variable that should store a number, look at the following example:

### Example

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

int myNum = 15;  
cout << myNum;

**Example**

#include <iostream>

using namespace std;

int main() {

int myNum = 15; // Now myNum is 15

myNum = 10; // Now myNum is 10

cout << myNum;

return 0;

}

## Other Types

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)

# Static Keyword in C++

* **Static variables in a Function**: When a variable is declared as static, space for **it gets allocated for the lifetime of the program**. Even if the function is called multiple times, space for the static variable is allocated only once and the value of variable in the previous call gets carried through the next function call. This is useful application where previous state of function needs to be stored.

#include <iostream>

#include <string>

using namespace std;

void demo()

{

    // static variable

    static int count = 0;

    cout << count << " ";

    // value is updated and

    // will be carried to next

    // function calls

    count++;

}

int main()

{

    for (int i=0; i<5; i++)

        demo();

    return 0;

}

**Output**

0 1 2 3 4

# C++ 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'

### Example

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

## What are Pointers?

In C++, a pointer refers to a variable that holds the address of another variable. Like regular variables, pointers have a data type. For example, a pointer of type integer can hold the address of a variable of type integer. A pointer of character type can hold the address of a variable of character type.

String food = "Pizza"; // A food variable of type string  
  
cout<< food;  // Outputs the value of food (Pizza)  
cout << &food; // Outputs the memory address of food (**0x6dfed4**)

### Example

#include <iostream>

#include <string>

using namespace std;

int main() {

string food = "Pizza"; // A string variable

string\* ptr = &food; // A pointer variable that stores the address of food

// Output the value of food

cout << food << "\n";

// Output the memory address of food

cout << &food << "\n";

// Output the memory address of food with the pointer

cout << ptr << "\n";

return 0;

}

**OUTPUT**

Pizza  
0x6dfed4

**EXAMPLE:**

#include <iostream>

#include <string>

using namespace std;

int main() {

string food = "Pizza"; // Variable declaration

string\* ptr = &food; // Pointer declaration

// Reference: Output the memory address of food with the pointer

cout << ptr << "\n";

// Dereference: Output the value of food with the pointer

cout << \*ptr << "\n";

return 0;

}

OUTPUT

0x6dfed4  
Pizza

## Modify the Pointer Value

**EXAMPLE:**

#include <iostream>

#include <string>

using namespace std;

int main() {

string food = "Pizza";

string\* ptr = &food;

// Output the value of food

cout << food << "\n";

// Output the memory address of food

cout << &food << "\n";

// Access the memory address of food and output its value

cout << \*ptr << "\n";

// Change the value of the pointer

\*ptr = "Hamburger";

// Output the new value of the pointer

cout << \*ptr << "\n";

// Output the new value of the food variable

cout << food << "\n";

return 0;

}

OUTPUT

Pizza  
0x6dfed4  
Pizza  
Hamburger  
Hamburger

# Type Conversion in C++

The conversion of a variable from one data type to another is called Type Conversion in C++. Type conversion in C++ is most commonly used to perform mathematical and logical operations on two variables with different data types. Using type conversion, we change the data type of one variable to become compatible with the other variable. For example, while calculating the sum of two numbers - where one number is a floating-point number, and the other is an integer, we convert the integer to a floating-point number to operate correctly.

## Type Conversion and Its Types

Type conversion is the method of converting one data type to another. There are two types of Type Conversions in C++:

Implicit type conversion, and

Explicit type conversion

Implicit type conversion

#include <iostream>

using namespace std;

int main() {

int int\_var;

float float\_var = 20.5;

int\_var = float\_var;

// Trying to store the value of float\_var in int\_var.

cout << "The value of int\_var is: " << int\_var << endl;

cout << "The value of float\_var is: " << float\_var << endl;

return 0;

}

**Output:**

The value of int\_var is: 20

The value of float\_var is: 20.5

**Explicit type conversion**

Explicit Type Conversions are those conversions that are done by the programmer manually. In other words, explicit conversion allows the programmer to typecast (change) the data type of a variable to another type. Hence, it is also called typecasting. Generally, we use the explicit type conversion if we do not want to follow the implicit type conversion rules.

Explicit type conversion in C++ can be done in two ways:

Conversion using the Assignment Operator

Conversion using the Cast Operator

Conversion using the Assignment Operator

#include <iostream>

using namespace std;

int main() {

char char\_var = 'a';

int int\_var;

// Explicitly converting a character variable to an integer variable.

int\_var = (int) char\_var; // Using cast notation.

cout << "The value of char\_var is: " << char\_var << endl;

cout << "The value of int\_var is: " << int\_var << endl;

return 0;

}

The value of char\_var is: a

The value of int\_var is: 97

**C++ Conditions and If Statements**

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

**EXAMPLE:**

#include <iostream>

using namespace std;

int main() {

int time = 22;

if (time < 10) {

cout << "Good morning.";

} else if (time < 20) {

cout << "Good day.";

} else {

cout << "Good evening.";

}

return 0;

}

**OUTPUT:**

Good evening.

**C++ Switch Statements**

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

Syntax

switch(expression)

{  
  case x:  
    //codeblock  
    break;  
  case y:  
    //codeblock  
    break;  
  default:  
    //codeblock  
}

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:

#include <iostream>

using namespace std;

int main() {

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;

}

return 0;

}

**OUTPUT:**

Thursday

**C++ Loops**

Loops can execute a block of code as long as a specified condition is reached.

Loops are ha

ndy because they save time, reduce errors, and they make code more readable.

**C++ 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  
}

**EXAMPLE**

#include <iostream>

using namespace std;

int main() {

int i = 0;

while (i < 5) {

cout << i << "\n";

i++;

}

return 0;

}

**OUTPUT:**

0  
1  
2  
3  
4

**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.

Syntax

do {  
  // code block to be executed  
}  
while (*condition*);

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:

#include <iostream>

using namespace std;

int main() {

int i = 0;

do {

cout << i << "\n";

i++;

}

while (i < 5);

return 0;

}

**OUTPUT:**

0  
1  
2  
3  
4

**C++ For Loop**

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:

Syntax

for (statement 1; statement 2; statement 3) {  
  // code block to be executed  
}

**Statement 1** is executed (one time) before the execution of the code block.

**Statement 2** defines the condition for executing the code block.

**Statement 3** is executed (every time) after the code block has been executed.

The example below will print the numbers 0 to 4:

#include <iostream>

using namespace std;

int main() {

for (int i = 0; i < 5; i++) {

cout << i << "\n";

}

return 0;

}

**OUTPUT**

0  
1  
2  
3  
4

**Nested Loops**

It is also possible to place a loop inside another loop. This is called a **nested loop**.

The "inner loop" will be executed one time for each iteration of the "outer loop":

#include <iostream>

using namespace std;

int main() {

// Outer loop

for (int i = 1; i <= 2; ++i) {

cout << "Outer: " << i << "\n"; // Executes 2 times

// Inner loop

for (int j = 1; j <= 3; ++j) {

cout << " Inner: " << j << "\n"; // Executes 6 times (2 \* 3)

}

}

return 0;

}

**OUTPUT:**

Outer: 1  
 Inner: 1  
 Inner: 2  
 Inner: 3  
Outer: 2  
 Inner: 1  
 Inner: 2  
 Inner: 3

**C++ Break and Continue**

**C++ 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](https://www.w3schools.com/cpp/cpp_switch.asp) 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:

#include <iostream>

using namespace std;

int main() {

for (int i = 0; i < 10; i++) {

if (i == 4) {

break;

}

cout << i << "\n";

}

return 0;

}

**OUTPUT**

0  
1  
2  
3

**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:

#include <iostream>

using namespace std;

int main() {

for (int i = 0; i < 10; i++) {

if (i == 4) {

continue;

}

cout << i << "\n";

}

return 0;

}

**OUPUT**

0  
1  
2  
3  
5  
6  
7  
8  
9

**C++ Arrays**

**Array**

It is a group of variables of similar data types referred to by a single element.

Its elements are stored in a contiguous memory location.

The size of the array should be mentioned while declaring it.

Array elements are always counted from zero (0) onward.

Array elements can be accessed using the position of the element in the array.

The array can have one or more dimensions.

An array in C/C++ or be it in any programming language is a collection of similar data items stored at contiguous memory locations and elements can be accessed randomly using indices of an array.  They can be used to store the collection of primitive data types such as int, float, double, char, etc of any particular type. To add to it, an array in C/C++ can store derived data types such as structures, pointers etc. Given below is the picture representation of an array.



Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type, specify the name of the array followed by square brackets and specify the number of elements it should store:

string cars[4];

string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};

**To create an array of three integers, you could write:**

int myNum[3] = {10, 20, 30};

Example:

#include <iostream>

using namespace std;

int main() {

int myNumbers[5] = {10, 20, 30, 40, 50};

for (int i = 0; i < 5; i++) {

cout << myNumbers[i] << "\n";

}

return 0;

}

OUTPUT

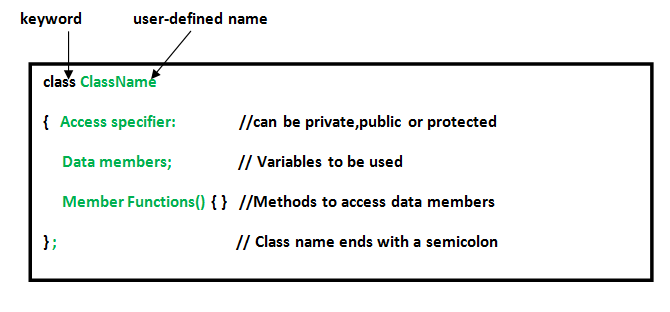
10  
20  
30  
40  
50

**Class:** A class in C++ is the building block that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object. For Example: Consider the Class of Cars. There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

* A Class is a user defined data-type which has data members and member functions.
* Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.
* In the above example of class Car, the data member will be speed limit, mileage etc and member functions can be apply brakes, increase speed etc.

An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

**Defining Class and Declaring Objects**

A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.

**Declaring Objects:**

When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects. **Syntax:**

**ClassName ObjectName;**

**Accessing data members and member functions**: The data members and member functions of class can be accessed using the dot(‘.’) operator with the object. For example if the name of object is obj and you want to access the member function with the name printName() then you will have to write obj.printName() .

**Accessing Data Members**

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member. This access control is given by [Access modifiers in C++](https://www.geeksforgeeks.org/access-modifiers-in-c/). There are three access modifiers : **public, private and protected**.

#include <bits/stdc++.h>

using namespace std;

class Geeks {

    // Access specifier

public:

    // Data  Members

    string geekname;

    // Member Functions()

    void printname() { cout << "Geekname is:" << geekname; }

};

int main()

{

    // Declare an object of class geeks

    Geeks obj1;

    // accessing data member

    obj1.geekname = "Abhi";

    // accessing member function

    obj1.printname();

    return 0;

}

Output

Geekname is:Abhi

**METHODS:**

Class is a blueprint of an object, which has data members and member functions also known as methods. A method is a procedure or function in the oops concept. A method is a function that belongs to a class.

There are two ways to define a procedure or function that belongs to a class:

Inside Class Definition

Outside Class Definition

1. Inside Class Definition

The member function is defined inside the class definition it can be defined directly. Similar to accessing a data member in the class we can also access the public member functions through the class object using the [dot operator (.)](https://www.geeksforgeeks.org/dot-operator-in-c-c/).

Syntax:

class class\_name{

public:

return\_type Method\_name() // method inside class definition

{

// body of member function

}

};

// C++ program for Inside Class Definition

#include <iostream>

using namespace std;

class rectangle {

private:

int length;

int breadth;

public:

void getdata(int l, int b)

{

length = l; // this pointer

breadth = b;

}

// area() function inside class

int area() { return (length \* breadth); }

// perimeter() function outside class

int perimeter() { return 2 \* (length + breadth); }

};

int main()

{

// Creating object

rectangle r;

r.getdata(2,3);

cout << "perimeter: " << r.perimeter() << endl;

cout << "area: " << r.area() << endl;

return 0;

}

**OUTPUT**

perimeter: 10

area: 6

**2. Outside Class Definition**

The member function is defined outside the class definition it can be defined using the [scope resolution operator](https://www.geeksforgeeks.org/scope-resolution-operator-in-c/). Similar to accessing a data member in the class we can also access the public member functions through the class object using the dot operator (.).

**Syntax:**

class Class\_name{

public:

return\_type Method\_name(); // method outside class definition

};

// Outside the Class using scope resolution operator

return\_type Class\_name :: Method\_name() {

// body of member function

}

#include <iostream>

using namespace std;

class rectangle {

private:

int length;

int breadth;

public:

void getdata(int l, int b);

int area();

int perimeter();

};

void rectangle::getdata(int l, int b)

{

length=l;

breadth=b;

}

int rectangle::area() { return (length \* breadth); }

 int rectangle::perimeter()

{    return 2 \* (length + breadth);

}

int main()

{

// Creating object

rectangle r;

r.getdata(2,3);

cout << "perimeter: " << r.perimeter() << endl;

cout << "area: " << r.area() << endl;

return 0;

}

**ABSTRACTION**

Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

**Types of Abstraction:**

**Data abstraction –**This type only shows the required information about the data and hides the unnecessary data.

**Control Abstraction –**This type only shows the required information about the implementation and hides unnecessary information.

**Abstraction using Classes**

We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.

**Abstraction in Header files**

One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

**Abstraction using Access Specifiers**

* Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members. For example:
* Members declared as **public**in a class can be accessed from anywhere in the program.
* Members declared as **private**in a class, can be accessed only from within the class. They are not allowed to be accessed from any part of the code outside the class.

We can easily implement abstraction using the above two features provided by access specifiers. Say, the members that define the internal implementation can be marked as private in a class. And the important information needed to be given to the outside world can be marked as public. And these public members can access the private members as they are inside the class.

**Example**

#include <iostream>

using namespace std;

class implementAbstraction {

private:

    int a, b;

public:

    // method to set values of

    // private members

    void set(int x, int y)

    {

        a = x;

        b = y;

    }

    void display()

    {

        cout << "a = " << a << endl;

        cout << "b = " << b << endl;

    }

};

int main()

{

    implementAbstraction obj;

    obj.set(10, 20);

    obj.display();

    return 0;

}

**Output**

a = 10

b = 20

**Advantages of Data Abstraction**

* Helps the user to avoid writing the low-level code
* Avoids code duplication and increases reusability.
* Can change the internal implementation of the class independently without affecting the user.
* Helps to increase the security of an application or program as only important details are provided to the user.
* It reduces the complexity as well as the redundancy of the code, therefore increasing the readability.

**Encapsulation in C++**

Encapsulation in C++ is defined as the wrapping up of data and information in a single unit. In Object Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them.

Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section, etc. Now,

The finance section handles all the financial transactions and keeps records of all the data related to finance.

Similarly, the sales section handles all the sales-related activities and keeps records of all the sales.

Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month.

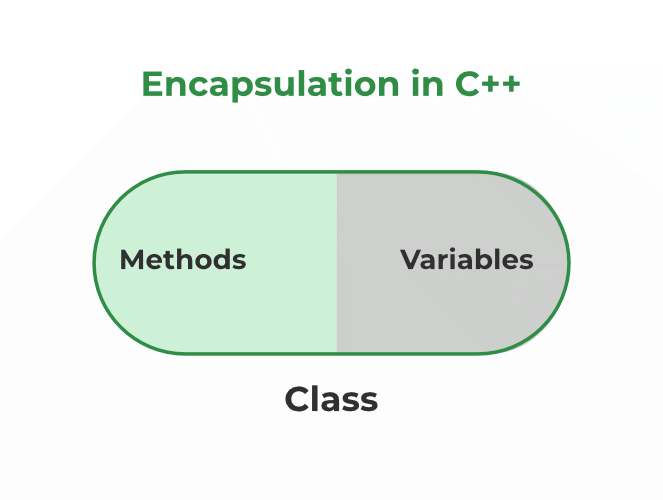
In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data.

This is what **Encapsulation**is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name “sales section”.

**Features of Encapsulation**

Below are the features of encapsulation:

* We can not access any function from the class directly. We need an object to access that function that is using the member variables of that class.
* The function which we are making inside the class must use only member variables, only then it is called encapsulation.
* If we don’t make a function inside the class which is using the member variable of the class then we don’t call it encapsulation.
* Increase in the security of data
* It helps to control the modification of our data members.



Encapsulation also leads to [data abstraction](https://www.geeksforgeeks.org/abstraction-in-cpp/). Using encapsulation also hides the data, as in the above example, the data of the sections like sales, finance, or accounts are hidden from any other section.

Example:

// C++ program to demonstrate

// Encapsulation

#include <iostream>

using namespace std;

class Encapsulation {

private:

    // Data hidden from outside world

    int x;

public:

    // Function to set value of

    // variable x

    void set(int a) { x = a; }

    // Function to return value of

    // variable x

    int get() { return x; }

};

// Driver code

int main()

{

    Encapsulation obj;

    obj.set(5);

    cout << obj.get();

    return 0;

}

OUTPUT

5

**Explanation:**In the above program, the variable **x** is made private. This variable can be accessed and manipulated only using the functions get() and set() which are present inside the class. Thus we can say that here, the variable x and the functions get() and set() are bound together which is nothing but encapsulation.

**Role of Access Specifiers in Encapsulation**

Access specifiers facilitate Data Hiding in C++ programs by restricting access to the class member functions and data members. There are three types of access specifiers in C++:

* **Private**
* **Protected**
* **Public**

By **default**, all data members and member functions of a class are made**private**by the compiler.

**ACCESS SPECIFIERS**

Access modifiers are used to implement an important aspect of Object-Oriented Programming known as [**Data Hiding**](https://practice.geeksforgeeks.org/problems/what-is-data-hiding).

Access Modifiers or Access Specifiers in a [class](https://www.geeksforgeeks.org/c-classes-and-objects/) are used to assign the accessibility to the class members, i.e., they set some restrictions on the class members so that they can’t be directly accessed by the outside functions.  
There are 3 types of access modifiers available in C++:

* **Public**
* **Private**
* **Protected**

 If we do not specify any access modifiers for the members inside the class, then by default the access modifier for the members will be **Private**.

**1. Public**: All the class members declared under the public specifier will be available to everyone. The data members and member functions declared as public can be accessed by other classes and functions too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

**Example:**

|  |
| --- |
| #include<iostream>  using namespace std;    // class definition  class Circle  {      public:          double radius;            double  compute\_area()          {              return 3.14\*radius\*radius;          }    };    // main function  int main()  {      Circle obj;        // accessing public datamember outside class      obj.radius = 5.5;        cout << "Radius is: " << obj.radius << "\n";      cout << "Area is: " << obj.compute\_area();      return 0;  } |

Output:

Radius is: 5.5

Area is: 94.985

In the above program, the data member radius is declared as public so it could be accessed outside the class and thus was allowed access from inside main().

**2. Private**: The class members declared as private can be accessed only by the member functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the [friend functions](https://www.geeksforgeeks.org/friend-class-function-cpp/) are allowed to access the private data members of the class.

**Example:** (accesing private member outside class which leads to error)

|  |
| --- |
| #include<iostream>  using namespace std;    class Circle  {      // private data member      private:          double radius;        // public member function      public:          double  compute\_area()          {   // member function can access private              // data member radius              return 3.14\*radius\*radius;          }    };    // main function  int main()  {      // creating object of the class      Circle obj;        // trying to access private data member      // directly outside the class      obj.radius = 1.5;        cout << "Area is:" << obj.compute\_area();      return 0;  } |

**Output:**

In function 'int main()':

11:16: error: 'double Circle::radius' is private

double radius;

^

31:9: error: within this context

obj.radius = 1.5;

^

The output of the above program is a compile time error because we are not allowed to access the private data members of a class directly from outside the class. Yet an access to obj.radius is attempted, but radius being a private data member, we obtained the above compilation error.

However, we can access the private data members of a class indirectly using the public member functions of the class.

**Example:**

#include<iostream>

using namespace std;

class Circle

{

    // private data member

    private:

        double radius;

    // public member function

    public:

        void compute\_area(double r)

        {   // member function can access private

            // data member radius

            radius = r;

            double area = 3.14\*radius\*radius;

            cout << "Radius is: " << radius << endl;

            cout << "Area is: " << area;

        }

};

// main function

int main()

{

    // creating object of the class

    Circle obj;

    // trying to access private data member

    // directly outside the class

    obj.compute\_area(1.5);

    return 0;

}

**Output**:

Radius is: 1.5

Area is: 7.065

**3. Protected**: The protected access modifier is similar to the private access modifier in the sense that it can’t be accessed outside of its class unless with the help of a friend class. The difference is that the class members declared as Protected can be accessed by any subclass (derived class) of that class as well.

**UML**

**UML (Unified Modeling Language)** is a general-purpose, graphical modeling language in the field of Software Engineering. UML is used to specify, visualize, construct, and document the artifacts (major elements) of the software system. It was initially developed by Grady Booch, Ivar Jacobson, and James Rumbaugh in 1994-95 at Rational software, and its further development was carried out through 1996. In 1997, it got adopted as a standard by the Object Management Group.

UML is not a programming language, it is rather a visual language. We use UML diagrams to portray the behavior and structure of a system. UML helps software engineers, businessmen and system architects with modelling, design and analysis.

The Object Management Group (OMG) adopted Unified Modelling Language as a standard in 1997. Its been managed by OMG ever since. International Organization for Standardization (ISO) published UML as an approved standard in 2005.

UML is linked with object oriented design and analysis. UML makes the use of elements and forms associations between them to form diagrams. Diagrams in UML can be broadly classified as:

Structural Diagrams – Capture static aspects or structure of a system.

Behavior Diagrams – Capture dynamic aspects or behavior of the system.

**Structure Diagrams**

* [Class Diagram](https://creately.com/blog/diagrams/uml-diagram-types-examples/#ClassDiagram)
* [Component Diagram](https://creately.com/blog/diagrams/uml-diagram-types-examples/#ComponentDiagram)
* [Deployment Diagram](https://creately.com/blog/diagrams/uml-diagram-types-examples/#DeploymentDiagram)
* [Object Diagram](https://creately.com/blog/diagrams/uml-diagram-types-examples/#ObjectDiagram)
* [Package Diagram](https://creately.com/blog/diagrams/uml-diagram-types-examples/#PackageDiagram)

**Behavioral Diagrams**

* [Use Case Diagram](https://creately.com/blog/diagrams/uml-diagram-types-examples/#UseCaseDiagram)
* [Activity Diagram](https://creately.com/blog/diagrams/uml-diagram-types-examples/#ActivityDiagram)
* [State Machine Diagram](https://creately.com/blog/diagrams/uml-diagram-types-examples/#StateMachDiagram)
* [Sequence Diagram](https://creately.com/blog/diagrams/uml-diagram-types-examples/#SequenceDiagram)
* [Communication Diagram](https://creately.com/blog/diagrams/uml-diagram-types-examples/#CommDiagram)

**Object Oriented Concepts Used in UML –**

**Class** – A class defines the blue print i.e. structure and functions of an object.

**Objects** – Objects help us to decompose large systems and help us to modularize our system. Modularity helps to divide our system into understandable components so that we can build our system piece by piece. An object is the fundamental unit (building block) of a system which is used to depict an entity.

**Inheritance** – Inheritance is a mechanism by which child classes inherit the properties of their parent classes.

**Abstraction** – Mechanism by which implementation details are hidden from user.

**Encapsulation** – Binding data together and protecting it from the outer world is referred to as encapsulation.

**Polymorphism** – Mechanism by which functions or entities are able to exist in different forms.

**Use-case diagrams**

In UML, use-case diagrams model the behavior of a system and help to capture the requirements of the system.

Use-case diagrams describe the high-level functions and scope of a system. These diagrams also identify the interactions between the system and its actors. The use cases and actors in use-case diagrams describe what the system does and how the actors use it, but not how the system operates internally.

Use-case diagrams illustrate and define the context and requirements of either an entire system or the important parts of the system. You can model a complex system with a single use-case diagram, or create many use-case diagrams to model the components of the system. You would typically develop use-case diagrams in the early phases of a project and refer to them throughout the development process.

Use-case diagrams are helpful in the following situations:

Before starting a project, you can create use-case diagrams to model a business so that all participants in the project share an understanding of the workers, customers, and activities of the business.

While gathering requirements, you can create use-case diagrams to capture the system requirements and to present to others what the system should do.

During the analysis and design phases, you can use the use cases and actors from your use-case diagrams to identify the classes that the system requires.

During the testing phase, you can use use-case diagrams to identify tests for the system.

The following topics describe model elements in use-case diagrams:

[**Use cases**](https://www.ibm.com/docs/en/SS8PJ7_9.6.1/com.ibm.xtools.modeler.doc/topics/cuc.html)  
A use case describes a function that a system performs to achieve the user’s goal. A use case must yield an observable result that is of value to the user of the system.

[**Actors**](https://www.ibm.com/docs/en/SS8PJ7_9.6.1/com.ibm.xtools.modeler.doc/topics/cactor.html)  
An actor represents a role of a user that interacts with the system that you are modeling. The user can be a human user, an organization, a machine, or another external system.

[**Relationships in use-case diagrams**](https://www.ibm.com/docs/en/SS8PJ7_9.6.1/com.ibm.xtools.modeler.doc/topics/crelsme_ucd.html)  
In UML, a relationship is a connection between model elements. A UML relationship is a type of model element that adds semantics to a model by defining the structure and behavior between the model elements.

**Notation DescriptionVisual Representation**

**Actor**

Someone interacts with use case (system function).

Named by noun.

Actor plays a role in the business

Similar to the concept of user, but a user can play different roles

**For example:**

A prof. can be instructor and also researcher

plays 2 roles with two systems

Actor triggers use case(s).

Actor has a responsibility toward the system (inputs), and Actor has expectations from the system (outputs).

Use Case Diagram Notation - Actor

**Use Case**

System function (process - automated or manual)

Named by verb + Noun (or Noun Phrase).

i.e. Do something

Each Actor must be linked to a use case, while some use cases may not be linked to actors.

Use Case Diagram Notation - Use Case

**Communication Link**

The participation of an actor in a use case is shown by connecting an actor to a use case by a solid link.

Actors may be connected to use cases by associations, indicating that the actor and the use case communicate with one another using messages.

Use Case Diagram Notation - Communication Link

**Boundary of system**

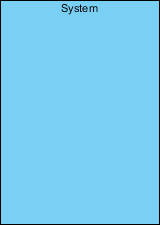
The system boundary is potentially the entire system as defined in the requirements document.

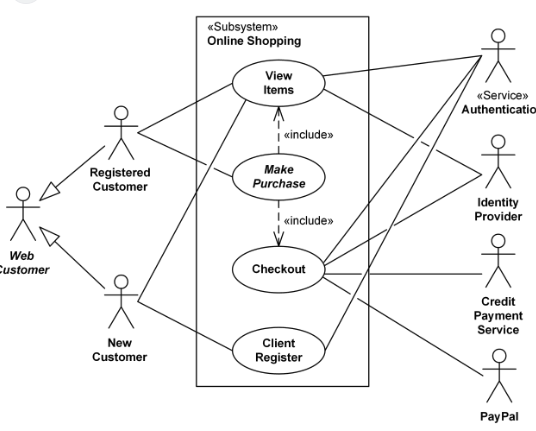
For large and complex systems, each module may be the system boundary.

For example, for an ERP system for an organization, each of the modules such as personnel, payroll, accounting, etc.

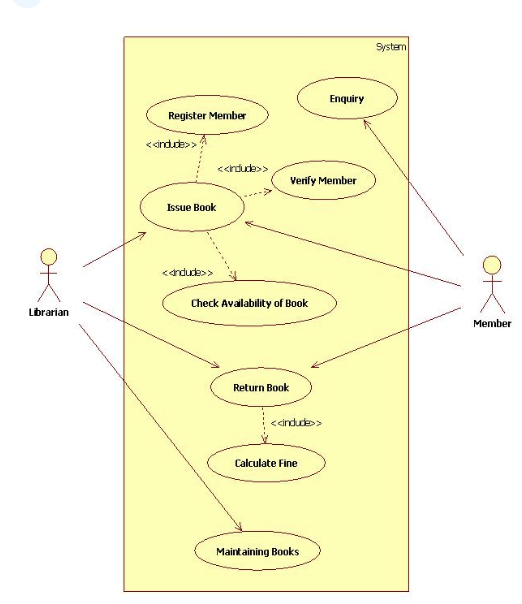
can form a system boundary for use cases specific to each of these business functions.

The entire system can span all of these modules depicting the overall system boundary

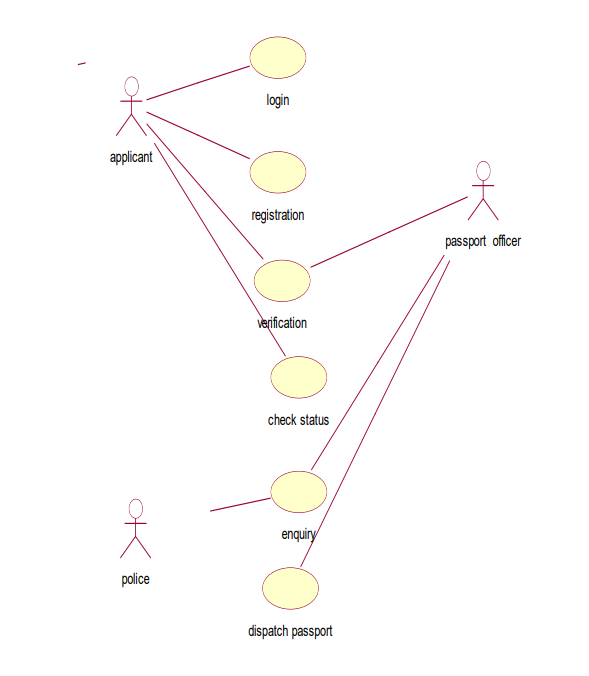




Use-case Diagram for Order Processing System



Use-case Diagram for Library Management System



Use-case Diagram for Passport Automation System

**CLASS DIAGRAM:**

Class diagram is a static diagram. It represents the static view of an application. Class diagram is not only used for visualizing, describing, and documenting different aspects of a system but also for constructing executable code of the software application.

The class diagrams are widely used in the modeling of objectoriented systems because they are the only UML diagrams, which can be mapped directly with object-oriented languages.

The [UML](https://en.wikipedia.org/wiki/Unified_Modeling_Language) Class diagram is a graphical notation used to construct and visualize object oriented systems. A class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's:

classes,

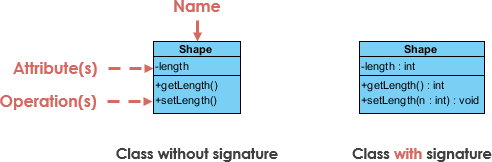
their attributes,

operations (or methods),

and the relationships among objects.

**UML Class Notation**

A class represent a concept which encapsulates state (attributes) and behavior (operations). Each attribute has a type. Each operation has a signature. The class name is the only mandatory information.



**Class Name:**

The name of the class appears in the first partition.

**Class Attributes:**

Attributes are shown in the second partition.

The attribute type is shown after the colon.

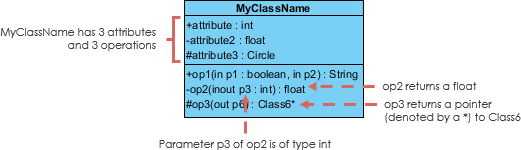
Attributes map onto member variables (data members) in code.

**Class Operations (Methods):**

Operations are shown in the third partition. They are services the class provides.

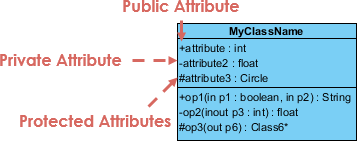
The return type of a method is shown after the colon at the end of the method signature.

The return type of method parameters are shown after the colon following the parameter name. Operations map onto class methods in code



**Class Visibility**

The +, - and # symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.



+ denotes public attributes or operations

- denotes private attributes or operations

# denotes protected attributes or operations

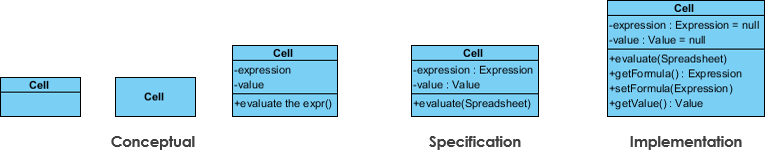
A diagram can be interpreted from various perspectives:

**Conceptual:** represents the concepts in the domain

**Specification:** focus is on the interfaces of Abstract Data Type (ADTs) in the software

**Implementation:** describes how classes will implement their interfaces

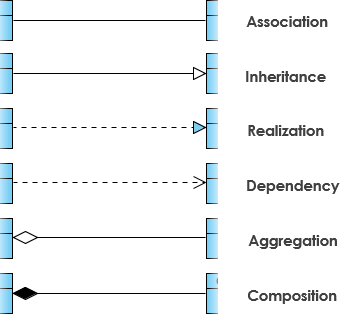
The perspective affects the amount of detail to be supplied and the kinds of relationships worth presenting. As we mentioned above, the class name is the only mandatory information.



**Relationships between classes**

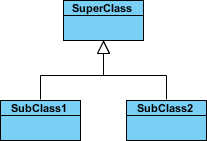
UML is not just about pretty pictures. If used correctly, UML precisely conveys how code should be implemented from diagrams. If precisely interpreted, the implemented code will correctly reflect the intent of the designer. Can you describe what each of the relationships mean relative to your target programming language shown in the Figure below?

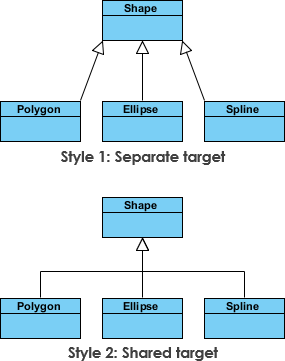
If you can't yet recognize them, no problem this section is meant to help you to understand UML class relationships. A class may be involved in one or more relationships with other classes. A relationship can be one of the following types:



**Inheritance (or Generalization):**

Generalization uses a “is-a” relationship from a specialization to the generalization class. Common structure and behaviour are used from the specializtion to the generalized class. At a very broader level you can understand this as inheritance. Why I take the term inheritance is, you can relate this term very well. Generalization is also called a “Is-a” relationship.





**Association**

* Associations are relationships between classes in a UML Class Diagram. They are represented by a solid line between classes. Associations are typically named using a verb or verb phrase which reflects the real world problem domain.
* Association is a relationship between two objects. In other words, association defines the multiplicity between objects. You may be aware of one-to-one, one-to-many, many-to-one, many-to-many all these words define an association between objects. Aggregation is a special form of association. Composition is a special form of aggregation.

https://javapapers.com/wp-content/uploads/2010/06/association.jpg

**Example:**A Student and a Faculty are having an association.

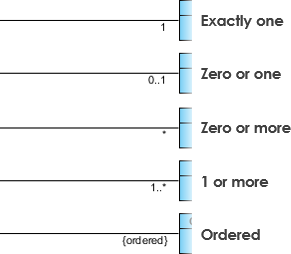
**Cardinality**

Cardinality is expressed in terms of:

one to one

one to many

many to many



**Aggregation**

Aggregation is a special case of association. A directional association between objects. When an object ‘has-a’ another object, then you have got an aggregation between them. Direction between them specified which object contains the other object. Aggregation is also called a “Has-a” relationship.

* A special type of association.
* It represents a "part of" relationship.
* Class2 is part of Class1.
* Objects of Class1 and Class2 have separate lifetimes.

Aggregation

**Composition**

Composition is a special case of aggregation. In a more specific manner, a restricted aggregation is called composition. When an object contains the other object, if the contained object cannot exist without the existence of container object, then it is called composition

* A special type of aggregation where parts are destroyed when the whole is destroyed.
* Objects of Class2 live and die with Class1.
* Class2 cannot stand by itself.

Composition

**Difference between aggregation and composition**

Composition is more restrictive. When there is a composition between two objects, the composed object cannot exist without the other object. This restriction is not there in aggregation. Though one object can contain the other object, there is no condition that the composed object must exist. The existence of the composed object is entirely optional. In both aggregation and composition, direction is must. The direction specifies, which object contains the other object.

Example: A Library contains students and books. Relationship between library and student is aggregation. Relationship between library and book is composition. A student can exist without a library and therefore it is aggregation. A book cannot exist without a library and therefore its a composition. For easy understanding I am picking this example. Don’t go deeper into example and justify relationships!

**Dependency**

Change in structure or behaviour of a class affects the other related class, then there is a dependency between those two classes. It need not be the same vice-versa. When one class contains the other class it this happens.

https://javapapers.com/wp-content/uploads/2010/06/dependency.jpg

* An object of one class might use an object of another class in the code of a method.
* A special type of association.
* Exists between two classes if changes to the definition of one may cause changes to the other (but not the other way around).

Class1 depends on Class2

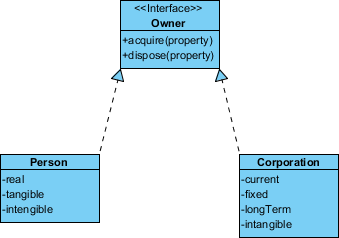
Dependency

Dependency

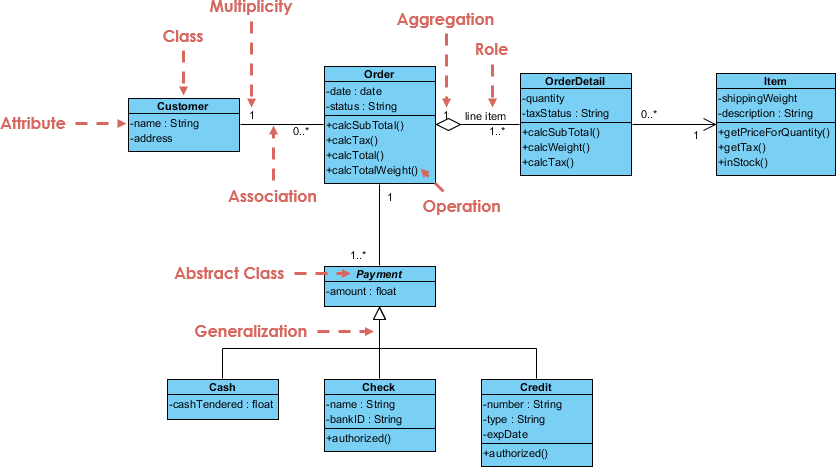
**Realization**

Realization is a relationship between the blueprint class and the object containing its respective implementation level details. This object is said to realize the blueprint class. In other words, you can understand this as the relationship between the interface and the implementing class.

For example, the Owner interface might specify methods for acquiring property and disposing of property. The Person and Corporation classes need to implement these methods, possibly in very different ways.



**Class Diagram Example: Order System**



**Purpose of Class Diagrams**

The purpose of class diagram is to model the static view of an application. Class diagrams are the only diagrams which can be directly mapped with object-oriented languages and thus widely used at the time of construction.

UML diagrams like activity diagram, sequence diagram can only give the sequence flow of the application, however class diagram is a bit different. It is the most popular UML diagram in the coder community.

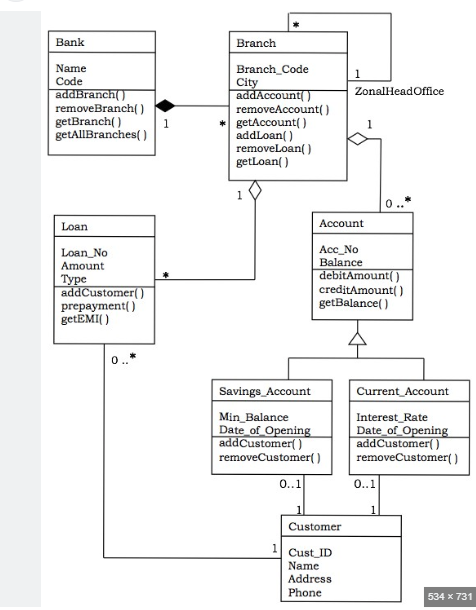
**The purpose of the class diagram can be summarized as −**

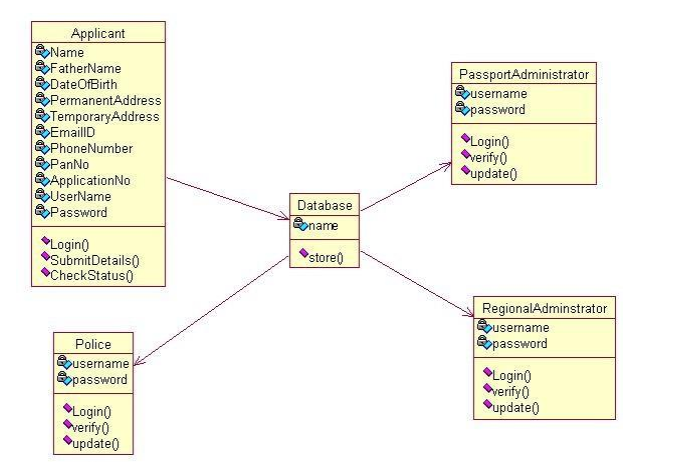
* Analysis and design of the static view of an application.
* Describe responsibilities of a system.
* Base for component and deployment diagrams.
* Forward and reverse engineering.
* A collection of class diagrams represent the whole system.

**The following points should be remembered while drawing a class diagram −**

* The name of the class diagram should be meaningful to describe the aspect of the system.
* Each element and their relationships should be identified in advance.
* Responsibility (attributes and methods) of each class should be clearly identified
* For each class, minimum number of properties should be specified, as unnecessary properties will make the diagram complicated.
* Use notes whenever required to describe some aspect of the diagram. At the end of the drawing it should be understandable to the developer/coder.
* Finally, before making the final version, the diagram should be drawn on plain paper and reworked as many times as possible to make it correct.
* Class diagram clearly shows the mapping with object-oriented languages such as Java, C++, etc. From practical experience, class diagram is generally used for construction purpose.

.





CLASS DIAGRAM FOR PASSPORT AUTOMATION SYSTEM

