

What is C++

C++ is a general purpose, case-sensitive, free-form programming language that supports object-oriented, procedural and generic programming.

C++ is a middle-level language, as it encapsulates both high and low level language features.

Object-Oriented Programming (OOPs)

C++ supports the object-oriented programming, the four major pillar of object-oriented programming (OOPs) used in C++ are:

1. Inheritance
 2. Polymorphism
 3. Encapsulation
 4. Abstraction
-

C++ Standard Libraries

Standard C++ programming is divided into three important parts:

- The core library includes the data types, variables and literals, etc.
- The standard library includes the set of functions manipulating strings, files, etc.
- The Standard Template Library (STL) includes the set of methods manipulating a data structure.

Usage of C++

By the help of C++ programming language, we can develop different types of secured and robust applications:

- Window application

- Client-Server application
- Device drivers
- Embedded firmware etc

C++ Program

In this tutorial, all C++ programs are given with C++ compiler so that you can easily change the C++ program code.

File: main.cpp

```
1. #include <iostream> // for turbo # include<iostream.h>
// #include<conio.h>
2. using namespace std;
3. int main() {
4.     cout << "Hello C++ Programming";
5.     return 0;
6. }
```

C++ history

C++ programming language was developed in 1980 by Bjarne Stroustrup at bell laboratories of AT&T (American Telephone & Telegraph), located in U.S.A.

Bjarne Stroustrup is known as the **founder of C++ language**.

It was developed for adding a feature of **OOP (Object Oriented Programming)** in C without significantly changing the C component.

C++ programming is "relative" (called a superset) of C, it means any valid C program is also a valid C++ program.

C++ Features

It provides a lot of features that are given below.

1. Simple
2. Abstract Data types
3. Machine Independent or Portable
4. Mid-level programming language
5. Structured programming language
6. Rich Library
7. Memory Management
8. Quicker Compilation
9. Pointers
10. Recursion
11. Extensible
12. Object-Oriented
13. Compiler based
14. Reusability

Turbo C++ - Download & Installation

There are many compilers available for C++. You need to download any one. Here, we are going to use **Turbo C++**. It will work for both C and C++. To install the Turbo C++ software, you need to follow following steps.

1. Download Turbo C++
2. Create turbo c directory inside c drive and extract the tc3.zip inside c:\turbo c
3. Double click on install.exe file
4. Click on the tc application file located inside c:\TC\BIN to write the c program

To write the first C++ program, open the C++ console and write the following code:

1. `#include <iostream.h>`
2. `#include<conio.h>`

```
3. void main() {  
4.     clrscr();  
5.     cout << "Welcome to C++ Programming."  
6.     getch();  
7. }
```

#include<iostream.h> includes the **standard input output** library functions. It provides **cin** and **cout** methods for reading from input and writing to output respectively.

#include <conio.h> includes the **console input output** library functions. The **getch()** function is defined in **conio.h** file.

void main() The **main()** function is the entry point of every program in C++ language. The **void** keyword specifies that it returns no value.

cout << "Welcome to C++ Programming." is used to print the data **"Welcome to C++ Programming."** on the console.

getch() The **getch()** function asks for a single character. Until you press any key, it blocks the screen.

Then **click on the run menu then run sub menu** to run the c++ program.

By shortcut

Or, press ctrl+f9 keys compile and run the program directly.

You can view the user screen any time by pressing the **alt+f5** keys.

Now **press Esc** to return to the turbo c++ console.

C++ Basic Input/Output

C++ I/O operation is using the stream concept. Stream is the sequence of bytes or flow of data. It makes the performance fast.

If bytes flow from main memory to device like printer, display screen, or a network connection, etc, this is called as **output operation**.

If bytes flow from device like printer, display screen, or a network connection, etc to main memory, this is called as **input operation**.

I/O Library Header Files

Header File	Function and Description
<iostream>	It is used to define the cout , cin and cerr objects, which correspond to standard output stream, standard input stream and standard error stream, respectively.
<iomanip>	It is used to declare services useful for performing formatted I/O, such as setprecision and setw .
<fstream>	It is used to declare services for user-controlled file processing.

Standard output stream (cout)

The **cout** is a predefined object of **ostream** class. It is connected with the standard output device, which is usually a display screen. The **cout** is used in conjunction with stream insertion operator (<<) to display the output on a console

Let's see the simple example of standard output stream (cout):

1. `#include <iostream>`
2. `using namespace std;`
3. `int main() {`
4. `char ary[] = "Welcome to C++ tutorial";`
5. `cout << "Value of ary is: " << ary << endl;`
6. `}`

Standard input stream (cin)

The **cin** is a predefined object of **istream** class. It is connected with the standard input device, which is usually a keyboard. The cin is used in conjunction with stream extraction operator (>>) to read the input from a console.

Let's see the simple example of standard input stream (cin):

```
1. #include <iostream>
2. using namespace std;
3. int main( ) {
4.     int age;
5.     cout << "Enter your age: ";
6.     cin >> age;
7.     cout << "Your age is: " << age << endl;
8. }
```

Standard end line (endl)

The **endl** is a predefined object of **ostream** class. It is used to insert a new line characters and flushes the stream.

Let's see the simple example of standard end line (endl):

```
1. #include <iostream>
2. using namespace std;
3. int main( ) {
4.     cout << "C++ Tutorial";
5.     cout << " Hello"<<endl;
6.     cout << "End of line"<<endl;
7. }
```

C++ Variable

A variable is a name of memory location. It is used to store data. Its value can be changed and it can be reused many times.

It is a way to represent memory location through symbol so that it can be easily identified.

1. **int** x;
2. **float** y;
3. **char** z;

Rules for defining variables

A variable can have alphabets, digits and underscore.

A variable name can start with alphabet and underscore only. It can't start with digit.

No white space is allowed within variable name.

A variable name must not be any reserved word or keyword e.g. char, float etc.

Valid variable names:

1. **int** a;
2. **int** _ab;
3. **int** a30;

Invalid variable names:

1. **int** 4;
2. **int** x y;
3. **int** double;

C++ Data Types

A data type specifies the type of data that a variable can store such as integer, floating, character etc.

There are 4 types of data types in C++ language.

Types	Data Types
Basic Data Type	int, char, float, double, etc
Derived Data Type	array, pointer, etc
Enumeration Data Type	enum
User Defined Data Type	structure

Basic Data Types

The basic data types are integer-based and floating-point based. C++ language supports both signed and unsigned literals.

The memory size of basic data types may change according to 32 or 64 bit operating system.

What is a token?

A C++ program is composed of tokens which are the smallest individual unit. Tokens can be one of several things, including keywords, identifiers, constants, operators, or punctuation marks.

There are 6 types of tokens in c++:

1. Keyword
2. Identifiers
3. Constants
4. Strings

5. Special symbols

6. Operators

C++ Keywords

A keyword is a reserved word. You cannot use it as a variable name, constant name etc. **A list of 32 Keywords in C++ Language which are also available in C language are given below.**

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

A list of 30 Keywords in C++ Language which are not available in C language are given below.

asm	dynamic_cast	namespace	reinterpret_cast	bool
explicit	new	static_cast	false	catch
operator	template	friend	private	class
this	inline	public	throw	const_cast
delete	mutable	protected	true	try

typeid	typename	using	virtual	wchar_t
--------	----------	-------	---------	---------

2. Identifiers

In C++, an identifier is a name given to a variable, function, or another object in the code. Identifiers are used to refer to these entities in the program, and they can consist of letters, digits, and underscores. However, some rules must be followed when choosing an identifier:

- The first character must be a letter or an underscore.
- Identifiers cannot be the same as a keyword.
- Identifiers cannot contain any spaces or special characters except for the underscore.
- Identifiers are case-sensitive, meaning that a variable named "myVariable" is different from a variable named "myvariable".

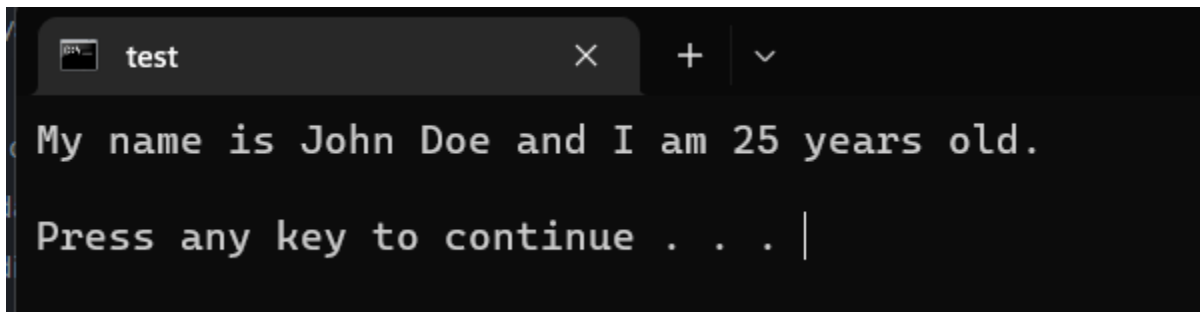
Here are some examples of valid and invalid identifiers in C++:

- **Valid:**
 - my_variable
 - student_name
 - balance_due
- **Invalid:**
 - my variable (contains a space)
 - student# (contains a special character)
 - int (same as a keyword)

It's important to choose meaningful and descriptive names for your identifiers to make your code easier to read and understand.

Example:

```
1. #include <iostream>
2.
3. using namespace std;
4.
5. int main()
6. {
7.     // Define a variable named "age" of type int and initialize it to 25
8.     int age = 25;
9.
10.    // Define a variable named "name" of type string and initialize it to "John Doe"
11.    string name = "John Doe";
12.
13.    // Print the value of the "age" and "name" variables
14.    cout << "My name is " << name << " and I am " << age << " years old." << endl;
15.
16.    return 0;
17. }
```

Output:A screenshot of a terminal window with a dark background. The title bar at the top shows a window icon, the text 'test', and standard window controls (close, maximize, minimize). The terminal displays the output of the C++ program: 'My name is John Doe and I am 25 years old.' followed by a new line and the prompt 'Press any key to continue . . . |' with a cursor at the end.**Explanation:**

In this program, the variable's age and name are identifiers used to store and access the values of the variables. The identifier age is used to store a person's age, and the identifier name is used to store the person's name. Notice how the identifiers are chosen to be descriptive and meaningful, which makes the code easier to read and understand. You can use any valid identifier in your C++ programs if you follow the rules for naming identifiers discussed earlier.

- The first character must be a letter or an underscore.
- Identifiers cannot be the same as a keyword.
- Identifiers cannot contain any spaces or special characters except for the underscore.
- Identifiers are case-sensitive, meaning that a variable named "myVariable" is different from a variable named "myvariable".

Here are some examples of valid and invalid identifiers in C++:

- **Valid:**
 - my_variable
 - student_name
 - balance_due
- **Invalid:**
 - my variable (contains a space)
 - student# (contains a special character)
 - int (same as a keyword)

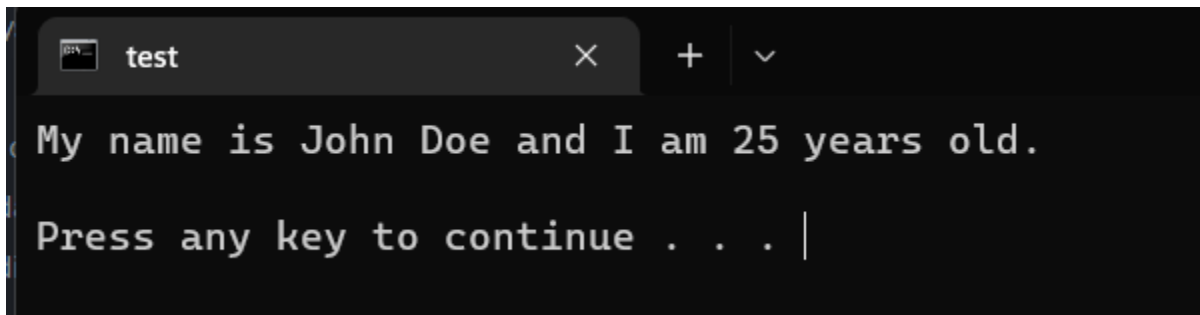
It's important to choose meaningful and descriptive names for your identifiers to make your code easier to read and understand.

Example:

1. `#include <iostream>`
- 2.
3. `using namespace std;`
- 4.

```
5. int main()
6. {
7.     // Define a variable named "age" of type int and initialize it to 25
8.     int age = 25;
9.
10.    // Define a variable named "name" of type string and initialize it to "John Doe"
11.    string name = "John Doe";
12.
13.    // Print the value of the "age" and "name" variables
14.    cout << "My name is " << name << " and I am " << age << " years old." << endl;
15.
16.    return 0;
17. }
```

Output:

A screenshot of a terminal window with a dark background. The window has a title bar with a single tab labeled 'test'. The terminal displays the output of the C++ program: 'My name is John Doe and I am 25 years old.' followed by a prompt 'Press any key to continue . . . |' with a cursor at the end.

Explanation:

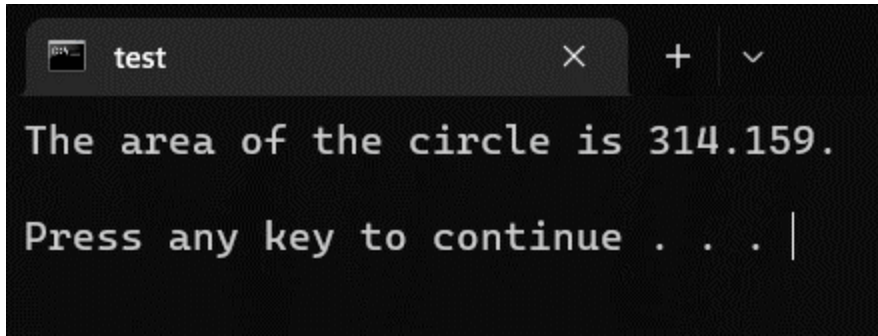
In this program, the variable's age and name are identifiers used to store and access the values of the variables. The identifier age is used to store a person's age, and the identifier name is used to store the person's name. Notice how the identifiers are chosen to be descriptive and meaningful, which makes the code easier to read and understand. You can use any valid identifier in your C++ programs if you follow the rules for naming identifiers discussed earlier.

Using constants instead of regular variables is important when you need to represent fixed values in your code. This can make your code more readable and maintainable and prevent accidental changes to the value of the constant.

Example:

```
1. #include <iostream>
2.
3. using namespace std;
4.
5. int main()
6. {
7.     // Define a constant named "PI" of type double and initialize it to 3.14159
8.     const double PI = 3.14159;
9.
10.    // Define a variable named "radius" of type double and initialize it to 10.0
11.    double radius = 10.0;
12.
13.    // Calculate the area of a circle with radius 10.0
14.    double area = PI * radius * radius;
15.
16.    // Print the area of the circle
17.    cout << "The area of the circle is " << area << "." << endl;
18.
19.    return 0;
20. }
```

Output:



```
test
The area of the circle is 314.159.
Press any key to continue . . . |
```

Explanation:

In this program, the constant PI is used to represent the value of pi. The value of PI is defined using the `const` keyword and is initialized with the value of pi. The constant PI is then used in calculating the area of a circle, which is multiplied by the radius of the circle squared to determine the area.

Notice how the constant PI is used the same way as a regular variable, but it cannot be changed once it has been defined. This makes it a perfect choice for representing fixed values used frequently in the code.

You can use constants in your C++ programs to represent fixed values that cannot be changed. This can make your code more readable and maintainable and prevent accidental changes to the value of the constant.

4. String:

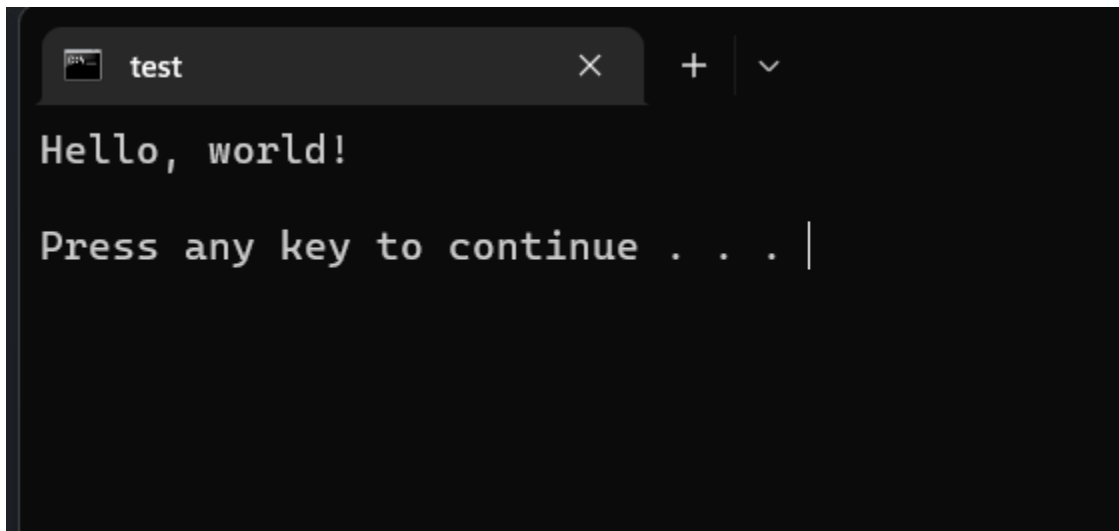
In C++, a string is a sequence of characters that represents text. ***Strings are commonly used in C++ programs to store and manipulate text data.***

To use strings in C++, you must include the `<string>` header file at the beginning of your program. This will provide access to the string class, which is used to create and manipulate strings in C++. Here is an example:

1. `#include <iostream>`
2. `#include <string>`
- 3.
4. `using namespace std;`
- 5.
6. `int main()`

```
7. {  
8.    // Define a string variable named "message" and initialize it with a string value  
9.    string message = "Hello, world!";  
10.  
11.    // Print the value of the "message" variable  
12.    cout << message << endl;  
13.  
14.    return 0;  
15. }
```

Output:

A screenshot of a terminal window with a dark background. The window has a title bar with the text 'test' and standard window control buttons (close, maximize, and a dropdown arrow). The terminal displays the text 'Hello, world!' on the first line. On the second line, it shows the prompt 'Press any key to continue . . . |' with a vertical cursor line at the end.

Explanation:

In this example, the string class creates a string variable named message. The message variable is initialized with the string "Hello, world!" and then printed to the screen using the cout object.

You can use the string class to perform various operations on strings, such as concatenating strings, comparing strings, searching for substrings, and more. For more information, you can consult the documentation for the string class.

5. Special symbols:

In C++, several special symbols are used for various purposes. The ampersand (&) is used to represent the address of a variable, while the tilde (~) is used to represent the bitwise NOT operator. Some of the most common special symbols include the asterisk (*), used as the multiplication and pointer operators.

The pound sign (#) is used to represent the preprocessor directive, a special type of instruction processed by the preprocessor before the code is compiled. The percent sign (%) is used as the modulus operator, which is used to find the remainder when one number is divided by another.

The vertical bar (|) is used to represent the bitwise OR operator, while the caret (^) is used to represent the bitwise XOR operator. The exclamation point (!) is used to represent the logical NOT operator, which is used to negate a Boolean value.

In addition to these symbols, C++ also has a number of other special characters that are used for a variety of purposes. For example, the backslash (\) is used to escape special characters, the double quotes (") are used to enclose string literals, and the single quotes (') are used to enclose character literals.

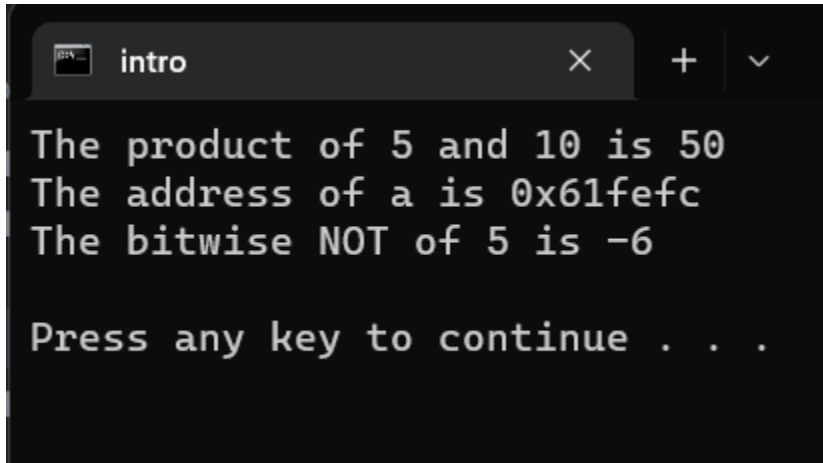
Overall, the special symbols in C++ are an important part of the language and are used in a variety of different contexts to perform a wide range of operations.

Example:

1. `#include <iostream>`
- 2.
3. `using namespace std;`
- 4.
5. `int main() {`
6. `int a = 5;`
7. `int b = 10;`
8. `int c = a * b; // Use of the asterisk (*) as the multiplication operator`
9. `cout << "The product of " << a << " and " << b << " is " << c << endl;`
- 10.
11. `int *ptr = &a; // Use of the ampersand (&) to get the address of a variable`
12. `cout << "The address of a is " << ptr << endl;`
- 13.
14. `int d = ~a; // Use of the tilde (~) as the bitwise NOT operator`

```
15. cout << "The bitwise NOT of " << a << " is " << d << endl;
16.
17. return 0;
18. }
```

Output:



```
intro
The product of 5 and 10 is 50
The address of a is 0x61fefc
The bitwise NOT of 5 is -6

Press any key to continue . . .
```

In this program, the asterisk (*) is used as the multiplication operator to calculate the product of two variables, a and b. The ampersand (&) is used to get the address of the a variable, which is then printed to the console. Finally, the tilde (~) is used as the bitwise NOT operator to negate the value of a.

6. Operators:

In C++, operators are special symbols that are used to perform operations on one or more operands. An operand is a value on which an operator acts. C++ has a wide range of operators, *including arithmetic operators, relational operators, logical operators, and others.*

Arithmetic operators are used to perform mathematical operations such as addition, subtraction, multiplication, and division. Some examples of arithmetic operators include + (used for addition), - (used for subtraction), * (used for multiplication), and / (used for division).

Relational operators are used to compare two values and determine their relationship. Some examples of relational operators include == (used to check for equality), != (used to check for inequality), > (used to check if a value is greater than another), and < (used to check if a value is less than another).

Logical operators are used to combine two or more relational expressions and determine their logical relationship. Some examples of logical operators include `&&` (used for the logical AND operation), `||` (used for the logical OR operation), and `!` (used for the logical NOT operation).

In addition to these operators, C++ also has a number of other operators that are used for different purposes. For example, the bitwise operators are used to perform bitwise operations on individual bits of a value, and the assignment operators are used to assign a value to a variable.

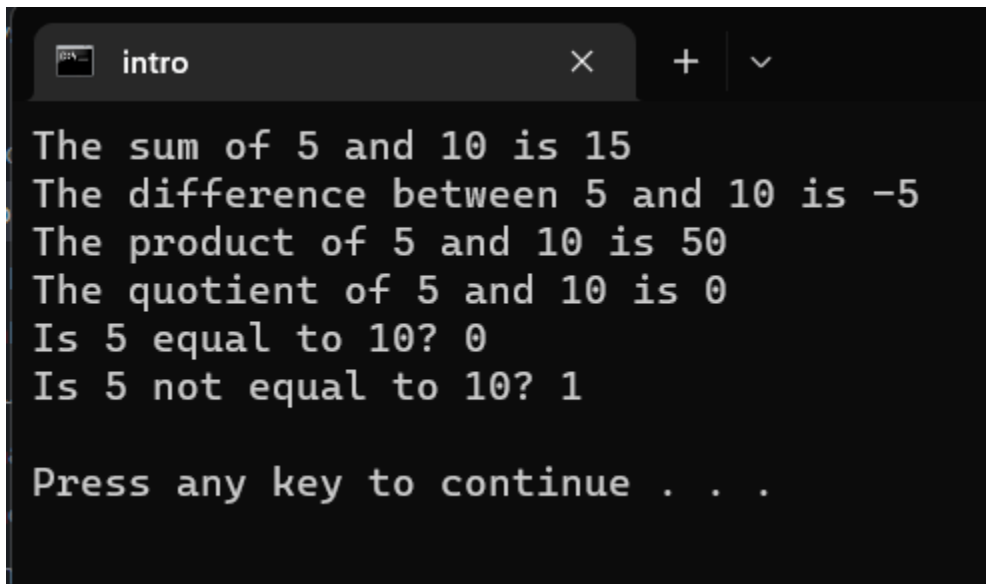
Overall, operators are an essential part of C++ and are used to perform a wide range of operations in a program.

Example:

```
1. #include <iostream>
2.
3. using namespace std;
4.
5. int main() {
6.     int a = 5;
7.     int b = 10;
8.     int c = a + b; // Use of the plus (+) operator for addition
9.     cout << "The sum of " << a << " and " << b << " is " << c << endl;
10.
11.    c = a - b; // Use of the minus (-) operator for subtraction
12.    cout << "The difference between " << a << " and " << b << " is " << c << endl;
13.
14.    c = a * b; // Use of the asterisk (*) operator for multiplication
15.    cout << "The product of " << a << " and " << b << " is " << c << endl;
16.
17.    c = a / b; // Use of the forward slash (/) operator for division
18.    cout << "The quotient of " << a << " and " << b << " is " << c << endl;
19.
20.    bool d = (a == b); // Use of the double equals (==) operator to check for equality
21.    cout << "Is " << a << " equal to " << b << "? " << d << endl;
```

```
22.  
23. d = (a != b); // Use of the exclamation point and equals (!=) operator to check for  
    inequality  
24. cout << "Is " << a << " not equal to " << b << "? " << d << endl;  
25.  
26. return 0;  
27. }
```

Output:



```
intro × + ∨  
The sum of 5 and 10 is 15  
The difference between 5 and 10 is -5  
The product of 5 and 10 is 50  
The quotient of 5 and 10 is 0  
Is 5 equal to 10? 0  
Is 5 not equal to 10? 1  
  
Press any key to continue . . .
```

Explanation:

In this program, various operators are used to perform different operations. The + operator is used for addition, the - operator is used for subtraction, the * operator is used for multiplication, and the / operator is used for division. The == and != operators are used to check for equality and inequality, respectively.

Type Compatibility

A type cast is basically a conversion from one type to another. There are two types of type conversion:

Implicit Type Conversion Also known as 'automatic type conversion'.

- Done by the compiler on its own, without any external trigger from the user.
- Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.

All the data types of the variables are upgraded to the data type of the variable with largest data type.

bool -> char -> short int -> int ->

unsigned int -> long -> unsigned ->

long long -> float -> double -> long double

- It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

Example of Type Implicit Conversion:

```
//Working of Implicit type-conversion

#include <iostream>

using namespace std;

int main() {

    int num_int;

    double num_double = 9.99;

    // implicit conversion

    // assigning a double value to an int variable

    num_int = num_double;

    cout << "num_int = " << num_int << endl;

    cout << "num_double = " << num_double << endl;

    return 0;

}
```

Explicit Type Conversion: This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type.

In C++, it can be done by two ways:

- **Converting by assignment:** This is done by explicitly defining the required type in front of the expression in parenthesis. This can be also considered as forceful casting.

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    // initializing a double variable
```

```
    double num_double = 3.56;
```

```
    cout << "num_double = " << num_double << endl;
```

```
    // C-style conversion from double to int
```

```
    int num_int1 = (int)num_double;
```

```
    cout << "num_int1  = " << num_int1 << endl;
```

```
    // function-style conversion from double to int
```

```
    int num_int2 = int(num_double);
```

```
    cout << "num_int2  = " << num_int2 << endl;
```

```
    return 0;    }
```

where *type* indicates the data type to which the final result is converted.

Conversion using Cast operator: A Cast operator is an unary operator which forces one data type to be converted into another data type.

C++ supports four types of casting:

- Static Cast
- Dynamic Cast
- Const Cast
- Reinterpret Cast

Static Cast

This is the simplest type of cast that can be used. It is a compile-time cast. It does things like implicit conversions between types (such as int to float, or pointer to void*), and it can also call explicit conversion functions.

Syntax of static_cast

static_cast <dest_type> (source);

```
// C++ Program to demonstrate
// static_cast
#include <iostream>
using namespace std;
```

```
// Driver code
```

```
int main()
```

```
{
```

```
    float f = 3.5;
```

```
    // Implicit type case
```

```
    // float to int
```

```
    int a = f;
```

```
    cout << "The Value of a: " << a;
```



```

// using static_cast for float to int
int b = static_cast<int>(f);
cout << "\nThe Value of b: " << b;
}

```

Dynamic_Cast in C++

Dynamic Cast: A cast is an operator that converts data from one type to another type. In C++, dynamic casting is mainly used for safe downcasting at run time. To work on dynamic_cast there must be one virtual function in the base class. A dynamic_cast works only polymorphic base class because it uses this information to decide safe downcasting.

Syntax:

dynamic_cast <new_type>(Expression)

- **Downcasting:** Casting a base class pointer (or reference) to a derived class pointer (or reference) is known as downcasting. In figure 1 casting from the Base class pointer/reference to the “derived class 1” pointer/reference showing downcasting (Base ->Derived class).
- **Upcasting:** Casting a derived class pointer (or reference) to a base class pointer (or reference) is known as upcasting. In figure 1 Casting from Derived class 2 pointer/reference to the “Base class” pointer/reference showing Upcasting (Derived class 2 -> Base Class).

```

#include <iostream>
using namespace std;

```

// Base class declaration

```

class Base {
    void print()
    {
        cout << "Base" << endl;
    }
};

```

// Derived Class 1 declaration

```

class Derived1 : public Base {
    void print()

```

```

{
    cout << "Derived1" << endl;
}
};

```

// Derived class 2 declaration

```

class Derived2 : public Base {
    void print()
    {
        cout << "Derived2" << endl;
    }
};

```

// Driver Code

```

int main()
{
    Derived1 d1;

    // Base class pointer hold Derived1
    // class object
    Base* bp = dynamic_cast<Base*>(&d1);

    // Dynamic casting
    Derived2* dp2 = dynamic_cast<Derived2*>(bp);
    if (dp2 == nullptr)
        cout << "null" << endl;

    return 0;
}

```

const_cast

const_cast is used to cast away the constness of variables. Following are some interesting facts about const_cast.

1) const_cast can be used to change non-const class members inside a const member function. Consider the following code snippet. Inside const member function fun(), 'this' is treated by the compiler as 'const student* const this', i.e. 'this' is a constant pointer to a constant object, thus compiler doesn't allow to change the data members through 'this' pointer. const_cast changes the type of 'this' pointer to 'student* const this'.

```

#include <iostream>
using namespace std;

class student
{
private:
    int roll;
public:
    // constructor
    student(int r):roll(r) {}

    // A const function that changes roll with the help of const_cast
    void fun() const
    {
        ( const_cast <student*> (this) )->roll = 5;
    }

    int getRoll() { return roll; }
};

int main(void)
{
    student s(3);
    cout << "Old roll number: " << s.getRoll() << endl;

    s.fun();

    cout << "New roll number: " << s.getRoll() << endl;

    return 0;
}

```

Reinterpret Cast

reinterpret_cast is a type of casting operator used in C++.

- *It is used to convert a pointer of some data type into a pointer of another data type, even if the data types before and after conversion are different.*
- *It does not check if the pointer type and data pointed by the pointer is same or not.*

*Syntax : data_type *var_name =
reinterpret_cast <data_type *>(pointer_variable);*

Return Type

- *It doesn't have any return type. It simply converts the pointer type.*

Parameters

- *It takes only one parameter i.e., the source pointer variable (p in above example).*

*// CPP program to demonstrate working of
// reinterpret_cast
#include <iostream>
using namespace std;*

```
int main()
{
    int* p = new int(65);
    char* ch = reinterpret_cast<char*>(p);
    cout << *p << endl;
    cout << *ch << endl;
    cout << p << endl;
    cout << ch << endl;
    return 0;
}
```

Purpose for using reinterpret_cast

- 1. reinterpret_cast is a very special and dangerous type of casting operator. And is suggested to use it using proper data type i.e., (pointer data type should be same as original data type).*
- 2. It can typecast any pointer to any other data type.*
- 3. It is used when we want to work with bits.*
- 4. If we use this type of cast then it becomes a non-portable product. So, it is suggested not to use this concept unless required.*
- 5. It is only used to typecast any pointer to its original type.*
- 6. Boolean value will be converted into integer value i.e., 0 for false and 1 for true.*

Advantages of Type Conversion:

- **This is done to take advantage of certain features of type hierarchies or type representations.**
- **It helps to compute expressions containing variables of different data types.**

What is Operator Precedence

Operator Precedence in C++ programming is a rule that describe which operator is solved first in an expression. For example: * and / have same precedence and their associativity is Left to Right, so the expression $18 / 2 * 5$ is treated as $(18 / 2) * 5$.

Let's take another example say $x = 8 + 4 * 2$; here value of x will be 16 and not 24 why, because * operator has a higher precedence than + operator. So $4*2$ gets multiplied first and then adds into 8.

Operator Associativity means how operators of the same precedence are evaluated in an expression. Associativity can be either from left to right or right to left.

The table below shows all the operators in C++ with precedence and associativity.

Precedence and Associativity of C++ Operators

OPERATOR	DESCRIPTION	ASSOCIATIVITY
R		
()	Parentheses	left to right
[]	Array Subscript	
. ->	Member Selector	
++ --	Post-Increment / Decrement	
++ --	Pre-Increment / Decrement	right to left
+ -	Unary plus / minus	
! ~	not operator and bitwise complement	
(type)	type casting	
*	Dereference operator	
&	Address of operator	
sizeof	Determine size in bytes	
* / %	Multiplication, Division and Modulus	left to right
+ -	Addition and Subtraction	left to right

<< >>	Bitwise leftshift and right shift	left to right
< <=	relational less than / less than or	left to right
> >=	equal to	
	relational greater than / greater than	
	or equal to	
== !=	Relational equal to / not equal to	left to right
&	Bitwise AND	left to right
^	Bitwise exclusive OR	left to right
 	Bitwise inclusive OR	left to right
&&	Logical AND	left to right
 	Logical OR	left to right
? :	Ternary operator	right to left
=	Assignment operator	right to left
+= -=	Addition / subtraction assignment	
*= /=	Multiplication / division assignment	
%= &=	Modulus and bitwise assignment	
^= =	Bitwise exclusive / inclusive OR	
<<= >>=	assignment	

**Bitwise leftshift / rightshift
assignment**

comma operator

right to left

Please don't get confused after seeing the above table. They all are used in a different situation but not all at the same time. For solving basic equation we will consider the following operator precedence only.

() Brackets will be solved first.

*** / % Which ever come first from left to right in your equation.**

+ - Which ever come first from left to right in your equation.

Now let's see some examples for more understanding.

Example 1

$$45 \% 2 + 3 * 2$$

45 % 2 + 3 * 2 will be solved as per operator precedence rule

1 + 3 * 2 will be solved as per operator precedence rule

1 + 6 will be solved as per operator precedence rule

7 (Answer)

Function

A function is a set of statements that takes input, does some specific computation, and produces output. The idea is to put some commonly or repeatedly done tasks together to make a function so that instead of writing the same code again and again for different inputs, we can call this function.

In simple terms, a function is a block of code that runs only when it is called.

Syntax:



Syntax of Function

// C++ Program to demonstrate working of a function

#include <iostream>

using namespace std;

// Following function that takes two parameters 'x' and 'y'

// as input and returns max of two input numbers

```
int max(int x, int y)
```

```
{
```

```
    if (x > y)
```

```
        return x;
```

```
    else
```

```
        return y;
```

```
}
```

```
// main function that doesn't receive any parameter and
```

```
// returns integer
```

```
int main()
```

```
{
```

```
    int a = 10, b = 20;
```

```
    // Calling above function to find max of 'a' and 'b'
```

```
    int m = max(a, b);
```

```
    cout << "m is " << m;
```

```
    return 0;
```

```
}
```

Time complexity: $O(1)$

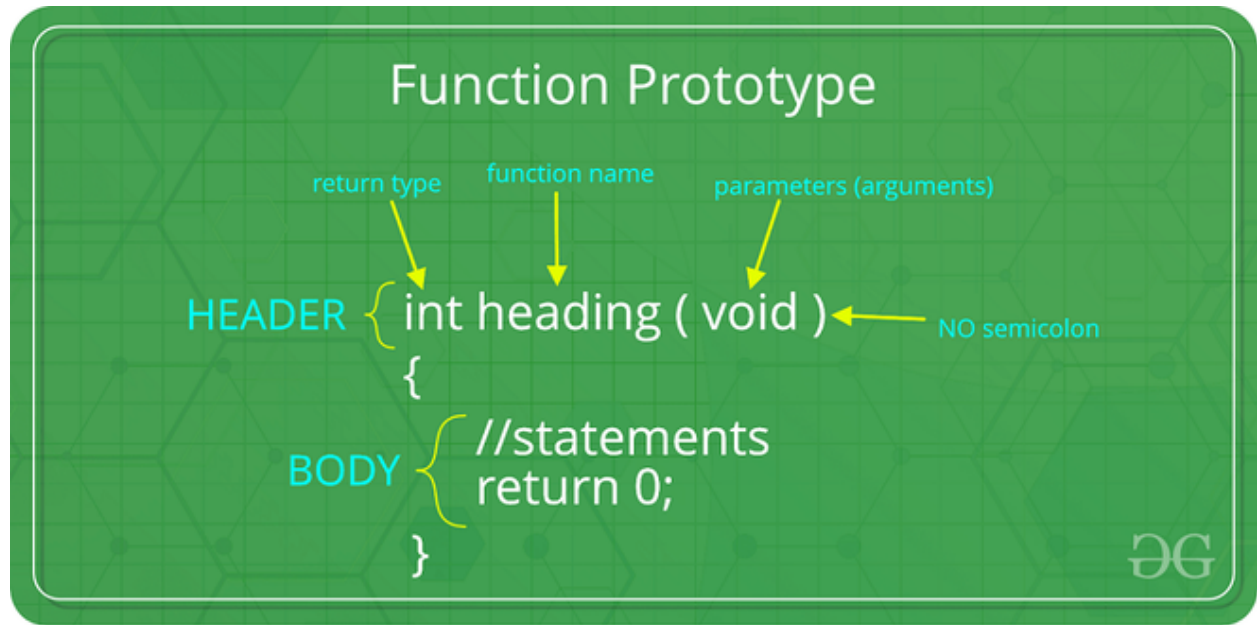
Space complexity: $O(1)$

Why Do We Need Functions?

- Functions help us in *reducing code redundancy*. If functionality is performed at multiple places in software, then rather than writing the same code, again and again, we create a function and call it everywhere. This also helps in maintenance as we have to make changes in only one place if we make changes to the functionality in future.
- Functions make code *modular*. Consider a big file having many lines of code. It becomes really simple to read and use the code, if the code is divided into functions.
- Functions provide *abstraction*. For example, we can use library functions without worrying about their internal work.

Function Declaration

A function declaration tells the compiler about the number of parameters, data types of parameters, and returns type of function. Writing parameter names in the function declaration is optional but it is necessary to put them in the definition. Below is an example of function declarations. (parameter names are not present in the below declarations)



Function Declaration

```
// C++ Program to show function that takes  
// two integers as parameters and returns  
// an integer  
int max(int, int);
```

```
// A function that takes an int  
// pointer and an int variable  
// as parameters and returns  
// a pointer of type int  
int* swap(int*, int);
```

```
// A function that takes  
// a char as parameter and
```

// returns a reference variable

char* call(char b);

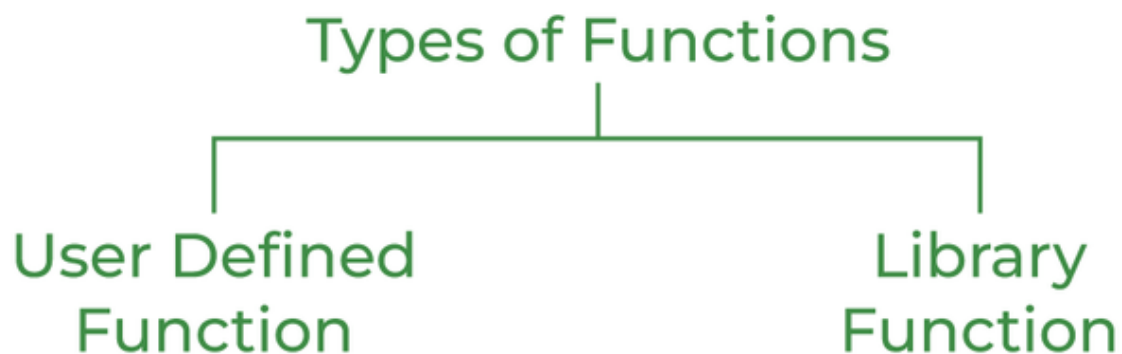
// A function that takes a

// char and an int as parameters

// and returns an integer

int fun(char, int);

Types of Functions



Types of Function in C++

User Defined Function

User-defined functions are user/customer-defined blocks of code specially customized to reduce the complexity of big programs. They are also commonly known as “*tailor-made functions*” which are built only to satisfy the condition in

which the user is facing issues meanwhile reducing the complexity of the whole program.

Library Function

Library functions are also called “*built-in Functions*“. These functions are part of a compiler package that is already defined and consists of a special function with special and different meanings. Built-in Function gives us an edge as we can directly use them without defining them whereas in the user-defined function we have to declare and define a function before using them.

For Example: `sqrt()`, `setw()`, `strcat()`, etc.

Parameter Passing to Functions

The parameters passed to the function are called *actual parameters*. For example, in the program below, 5 and 10 are actual parameters.

The parameters received by the function are called *formal parameters*. For example, in the above program x and y are formal parameters.

```
class Multiplication {  
    int multiply(int x, int y) { return x * y; }  
public  
    static void main()  
    {  
        Multiplication M = new Multiplication();  
        int gfg = 5, gfg2 = 10;  
        int gfg3 = multiply(gfg, gfg2);  
        cout << "Result is " << gfg3;  
    }  
}
```

Formal Parameter

Actual Parameter

There are two most popular ways to pass parameters:

1. ***Pass by Value:*** In this parameter passing method, values of actual parameters are copied to the function's formal parameters. The actual and formal parameters are stored in different memory locations so any changes made in the functions are not reflected in the actual parameters of the caller.
2. ***Pass by Reference:*** Both actual and formal parameters refer to the same locations, so any changes made inside the function are reflected in the actual parameters of the caller.

// C++ Program to demonstrate function definition

#include <iostream>

using namespace std;

void fun(int x)

{

// definition of

// function

x = 30;

}

```
int main()
```

```
{
```

```
    int x = 20;
```

```
    fun(x);
```

```
    cout << "x = " << x;
```

```
    return 0;
```

```
}
```

```
// C++ Program to demonstrate working of
```

```
// function using pointers
```

```
#include <iostream>
```

```
using namespace std;
```

```
void fun(int* ptr) { *ptr = 30; }
```

```
int main()
```

```
{
```

```
    int x = 20;
```



```
    fun(&x);

    cout << "x = " << x;


    return 0;

}
```

Function Definition

Pass by value is used where the value of x is not modified using the function fun().

Output

x = 20

Time complexity: O(1)

Space complexity: O(1)

Functions Using Pointers

The function fun() expects a pointer ptr to an integer (or an address of an integer). It modifies the value at the address ptr. The dereference operator * is used to access the value at an address. In the statement ‘*ptr = 30’, the value at address ptr is changed to 30. The address operator & is used to get the address of a variable of any

data type. In the function call statement ‘fun(&x)’, the address of x is passed so that x can be modified using its address.

Time complexity: O(1)

Space complexity: O(1)

Difference between call by value and call by reference in C++

Call by value	Call by reference
A copy of the value is passed to the function	An address of value is passed to the function
Changes made inside the function are not reflected on other functions	Changes made inside the function are reflected outside the function as well
Actual and formal arguments will be created at	Actual and formal arguments will be created at

different memory location	same memory location.
----------------------------------	------------------------------

Points to Remember About Functions in C++

- 1. Most C++ program has a function called main() that is called by the operating system when a user runs the program.**
- 2. Every function has a return type. If a function doesn't return any value, then void is used as a return type. Moreover, if the return type of the function is void, we still can use the return statement in the body of the function definition by not specifying any constant, variable, etc. with it, by only mentioning the 'return;' statement which would symbolize the termination of the function as shown below:**
- 3. To declare a function that can only be called without any parameter, we should use "void fun(void)". As a side note, in C++, an empty list means a function can only be called without any parameter. In C++, both void fun() and void fun(void) are same.**

Main Function

The main function is a special function. Every C++ program must contain a function named main. It serves as the entry point for the program. The computer will start running the code from the beginning of the main function.

Types of Main Functions

- 1. Without parameters:**
- 2. With parameters:**

The reason for having the parameter option for the main function is to allow input from the command line. When you use the main function with parameters, it saves every group of characters (separated by a space) after the program name as elements in an array named argv.

Since the main function has the return type of int, the programmer must always have a return statement in the code. The number that is returned is used to inform the calling program what the result of the program's execution was. Returning 0 signals that there were no problems.

C++ Recursion

When function is called within the same function, it is known as recursion in C++.

The function which calls the same function, is known as recursive function.

A function that calls itself, and doesn't perform any task after function call, is known as tail recursion. In tail recursion, we generally call the same function with return statement.

Syntax:

To know more see [this article](#).

C++ Passing Array to Function

In C++, to reuse the array logic, we can create a function. To pass an array to a function in C++, we need to provide only the array name.

```
function_name(array_name[]); //passing array to function
```

```
#include <iostream>
```

```
using namespace std;
```

```
void printMin(int arr[5]);
```

```
int main()

{

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

    printMin(ar); // passing array to function

}

void printMin(int arr[5])

{

    int min = arr[0];

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

        if (min > arr[i]) {

            min = arr[i];

        }

    }

    cout << "Minimum element is: " << min << "\n";

}
```

// Code submitted by Susobhan Akhuli

Time complexity: $O(n)$ where n is the size of the array

Space complexity: $O(n)$ where n is the size of the array.

C++ Overloading (Function)

If we create two or more members having the same name but different in number or type of parameters, it is known as C++ overloading. In C++, we can overload:

- *methods,*
- *constructors and*
- *indexed properties*

Types of overloading in C++ are:

- *Function overloading*
- *Operator overloading*

C++ Function Overloading

Function Overloading is defined as the process of having two or more functions with the same name, but different parameters. In function overloading, the function is redefined by using either different types or number of arguments. It is only through these differences a compiler can differentiate between the functions.

The advantage of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

Example: changing number of arguments of add() method

// program of function overloading when number of arguments

```
// vary

#include <iostream>

using namespace std;

class Cal {

public:

    static int add(int a, int b) { return a + b; }

    static int add(int a, int b, int c)

    {

        return a + b + c;

    }

};

int main(void)

{

    Cal C; // class object declaration.

    cout << C.add(10, 20) << endl;

    cout << C.add(12, 20, 23);

    return 0;

}
```

// Code Submitted By Susobhan Akhuli

Time complexity: $O(1)$

Space complexity: $O(1)$

Example: when the type of the arguments vary.

// Program of function overloading with different types of

// arguments.

#include <iostream>

using namespace std;

int mul(int, int);

float mul(float, int);

int mul(int a, int b) { return a * b; }

float mul(double x, int y) { return x * y; }

int main()

{

int r1 = mul(6, 7);

float r2 = mul(0.2, 3);

cout << "r1 is : " << r1 << endl;

cout << "r2 is : " << r2 << endl;

return 0;

}

// Code Submitted By Susobhan Akhuli

Time Complexity: $O(1)$

Space Complexity: $O(1)$

Function Overloading and Ambiguity

When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as *function overloading ambiguity*.

When the compiler shows the ambiguity error, the compiler does not run the program.

Causes of Ambiguity:

- *Type Conversion.*
- *Function with default arguments.*
- *Function with pass-by-reference.*

```
#include <iostream>
```

```
using namespace std;
```

```
void fun(int);
```

```
void fun(float);
```

```
void fun(int i) { cout << "Value of i is : " << i << endl; }
```

```
void fun(float j)
```

```
{
```

```
    cout << "Value of j is : " << j << endl;
```

```

}

int main()

{

    fun(12);

    fun(1.2);

    return 0;

}

```

// Code Submitted By Susobhan Akhuli

The above example shows an error “*call of overloaded ‘fun(double)’ is ambiguous*“. The fun(10) will call the first function. The fun(1.2) calls the second function according to our prediction. But, this does not refer to any function as in C++, all the floating point constants are treated as double not as a float. If we replace float to double, the program works. Therefore, this is a type conversion from float to double.

Function with Default Arguments:-

The above example shows an error “*call of overloaded ‘fun(int)’ is ambiguous*“. The fun(int a, int b=9) can be called in two ways: first is by calling the function with one argument, i.e., fun(12) and another way is calling the function with two arguments, i.e., fun(4,5). The fun(int i) function is invoked with one argument. Therefore, the compiler could not be able to select among fun(int i) and fun(int a,int b=9).

Function with Pass By Reference:-

The above example shows an error “*call of overloaded ‘fun(int&)’ is ambiguous*“.

The first function takes one integer argument and the second function takes a reference parameter as an argument. In this case, the compiler does not know which function is needed by the user as there is no syntactical difference between the fun(int) and fun(int &).

```
#include <iostream>

using namespace std;

void fun(int);

void fun(int, int);

void fun(int i) { cout << "Value of i is : " << i << endl; }

void fun(int a, int b = 9)
{
    cout << "Value of a is : " << a << endl;
    cout << "Value of b is : " << b << endl;
}

int main()
{
    fun(12);

    return 0;
}
```

// Code Submitted By Susobhan Akhuli

```
#include <iostream>  
  
using namespace std;  
  
void fun(int);  
  
void fun(int&);  
  
int main()  
  
{  
  
    int a = 10;  
  
    fun(a); // error, which fun()?  
  
    return 0;  
  
}  
  
void fun(int x) { cout << "Value of x is : " << x << endl; }  
  
void fun(int& b)  
  
{  
  
    cout << "Value of b is : " << b << endl;  
  
}
```

// Code Submitted By Susobhan Akhuli

Friend Function

- **A friend function is a special function in C++ which in spite of not being a member function of a class has the privilege to access private and protected data of a class.**
- **A friend function is a non-member function or an ordinary function of a class, which is declared by using the keyword “friend” inside the class. By declaring a function as a friend, all the access permissions are given to the function.**
- **The keyword “friend” is placed only in the function declaration but not in the function definition.**
- **When the friend function is called neither the name of the object nor the dot operator is used. However, it may accept the object as an argument whose value it wants to access.**
- **A friend function can be declared in any section of the class i.e. public, private, or protected.**

Declaration of friend function in C++

Syntax:

```
class <class_name> {  
  
    friend <return_type> <function_name>(argument/s);  
  
};
```

Example_1: Find the largest of two numbers using Friend FunctionOutput

```
#include <iostream>

using namespace std;

class Largest {
    int a, b, m;

public:
    void set_data();

    friend void find_max(Largest);
};

void Largest::set_data()
{
    cout << "Enter the first number : ";

    cin >> a;

    cout << "\nEnter the second number : ";

    cin >> b;
}

void find_max(Largest t)
{
    if (t.a > t.b)
```

```
        t.m = t.a;

    else

        t.m = t.b;

    cout << "\nLargest number is " << t.m;
}

int main()
{
    Largest l;

    l.set_data();

    find_max(l);

    return 0;
}
```

Enter the first number : 789

Enter the second number : 982

Largest number is 982

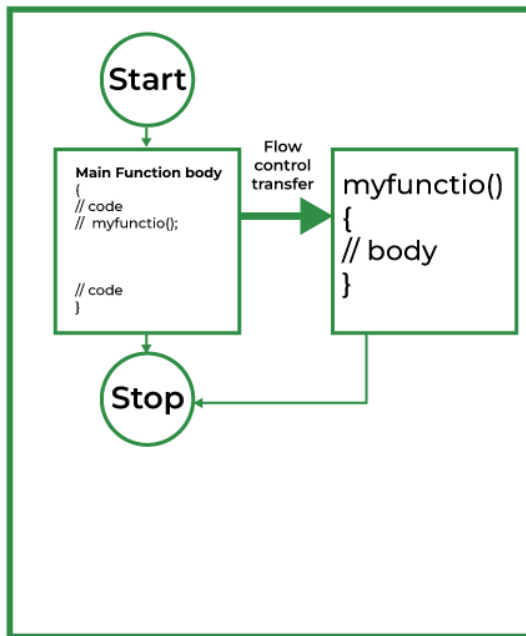
Inline Functions in C++

C++ provides inline functions to reduce the function call overhead. An inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of the inline function call. This substitution is performed by the C++ compiler at compile time. An inline function may increase efficiency if it is small.

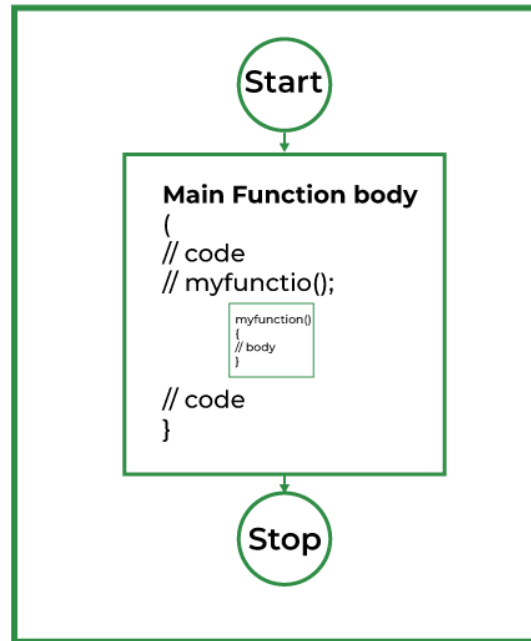
Syntax:

```
inline return-type function-name(parameters)  
  
{  
  
    // function code  
  
}
```


Normal Function



Inline Function



Remember, inlining is only a request to the compiler, not a command. The compiler can ignore the request for inlining.

// define an inline function

// that prints the sum of 2 integers

```
inline void printSum(int num1,int num2) {
```

```
    cout << num1 + num2 << "\n";
```

```
}
```

```
int main() {
```

```
// call the inline function

// first call

printSum(10, 20);

// second call

printSum(2, 5);

// third call

printSum(100, 400);

return 0;

}
```

C++ Classes and Objects

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 brands 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, and mileage are their properties.

- A Class is a user-defined data type that has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables together, these data members and member functions define 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 *applying brakes*, *increasing 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 the keyword `class` followed by the name of the class. The body of the 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 the class can be accessed using the dot('.') operator with the object. For example, if the name of the 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++](#). There are three access modifiers: public, private, and protected.

```
// C++ program to demonstrate accessing of data members
```

```
#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

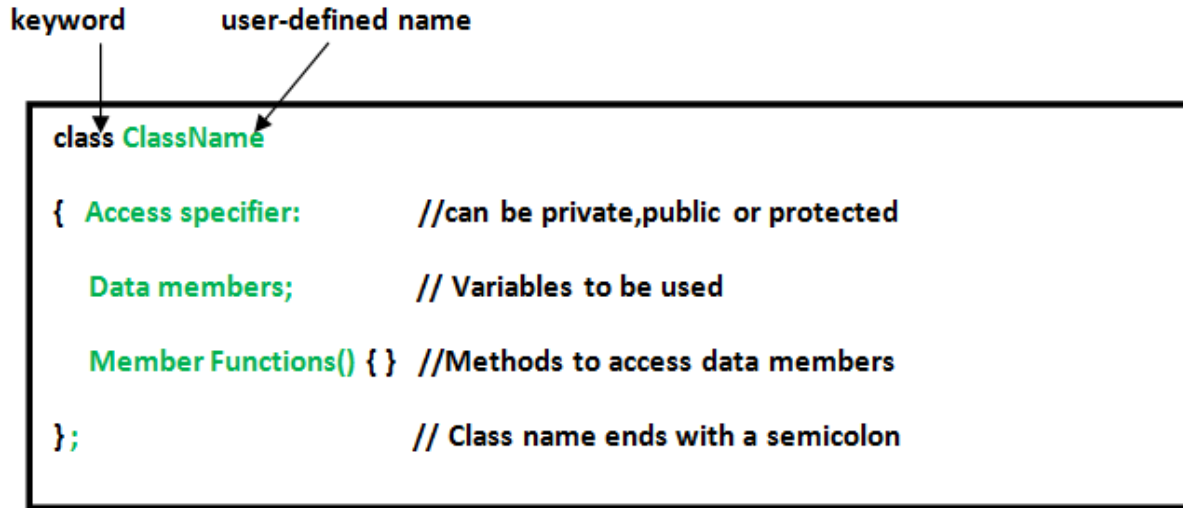
    obj1.geekname = "Abhi";

    // accessing member function

    obj1.printname();

    return 0;

}
```



Output

Geekname is:Abhi

Member Functions in Classes

There are 2 ways to define a member function:

- Inside class definition
- Outside class definition

To define a member function outside the class definition we have to use the scope resolution:: operator along with the class name and function name.

// C++ program to demonstrate function

// declaration outside class

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
class Geeks
```

```
{
```

```
    public:
```

```
    string geekname;
```

```
    int id;
```

```
    // printname is not defined inside class definition
```

```
    void printname();
```

```
    // printid is defined inside class definition
```

```
    void printid()
```

```
{

    cout <<"Geek id is: "<<id;

}

};

// Definition of printname using scope resolution operator ::

void Geeks::printname()

{

    cout <<"Geekname is: "<<geekname;

}

int main() {

    Geeks obj1;
```



```
obj1.geekname = "xyz";

obj1.id=15;


// call printname()

obj1.printname();

cout << endl;


// call printid()

obj1.printid();

return 0;

}
```

Output

```
Geekname is: xyz
```

```
Geek id is: 15
```

Note that all the member functions defined inside the class definition are by default inline, but you can also make any non-class function inline by using the keyword inline with them. Inline functions are actual functions, which are copied everywhere during compilation, like pre-processor macro, so the overhead of function calls is reduced.

Note: Declaring a [friend function](#) is a way to give private access to a non-member function.

Constructors

[Constructors](#) are special class members which are called by the compiler every time an object of that class is instantiated. Constructors have the same name as the class and may be defined inside or outside the class definition.

There are 3 types of constructors:

- [Default Constructors](#)
- Parameterized Constructors
- [Copy Constructors](#)

```
// C++ program to demonstrate constructors
```

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
class Geeks
```

```
{
```

```
    public:
```

```
    int id;
```

```
    //Default Constructor
```

```
    Geeks()
```

```
{
```

```
    cout << "Default Constructor called" << endl;
```

```
    id=-1;
```

```
}
```

```
//Parameterized Constructor
```

```
Geeks(int x)
```

```
{
```

```
    cout <<"Parameterized Constructor called "<< endl;
```

```
    id=x;
```

```
}
```

```
};
```

```
int main() {
```

```
    // obj1 will call Default Constructor
```

```
    Geeks obj1;
```

```
    cout <<"Geek id is: "<<obj1.id << endl;
```

```
// obj2 will call Parameterized Constructor

Geeks obj2(21);

cout <<"Geek id is: " <<obj2.id << endl;

return 0;

}
```

Output

```
Default Constructor called
```

```
Geek id is: -1
```

```
Parameterized Constructor called
```

```
Geek id is: 21
```

A Copy Constructor creates a new object, which is an exact copy of the existing object. The compiler provides a default Copy Constructor to all the classes.

Syntax:

```
class-name (class-name &) {}
```

Destructors

[Destructor](#) is another special member function that is called by the compiler when the scope of the object ends.

// C++ program to explain destructors

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
class Geeks
```

```
{
```

```
    public:
```

```
    int id;
```

```
    //Definition for Destructor
```

```
    ~Geeks()
```

```
    {
```

```
        cout << "Destructor called for id: " << id << endl;
```

```
    }
```

```
};
```

```
int main()
```

```
{
```

```
    Geeks obj1;
```

```
    obj1.id=7;
```

```
    int i = 0;
```

```
    while ( i < 5 )
```

```
    {
```

```
        Geeks obj2;
```

```
        obj2.id=i;
```

```
        i++;
```

```
    } // Scope for obj2 ends here
```

```
    return 0;
```

```
} // Scope for obj1 ends here
```

Output

```
Destructor called for id: 0
Destructor called for id: 1
Destructor called for id: 2
Destructor called for id: 3
Destructor called for id: 4
Destructor called for id: 7
```

Interesting Fact (Rare Known Concept)

Why do we give semicolons at the end of class?

Many people might say that it's a basic syntax and we should give a semicolon at the end of the class as its rule defines in cpp. But the main reason why semi-colons are there at the end of the class is compiler checks if the user is trying to create an instance of the class at the end of it.

Yes just like structure and union, we can also create the instance of a class at the end just before the semicolon. As a result, once execution reaches at that line, it creates a class and allocates memory to your instance.

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

```
class Demo{
int a, b;
    public:
    Demo() // default constructor
    {
        cout << "Default Constructor" << endl;
    }
    Demo(int a, int b):a(a),b(b) //parameterised constructor
    {
```

```

        cout << "parameterized constructor -values" << a << " " << b <<
endl;
    }

```

```

}instance;

```

```

int main() {

```

```

    return 0;
}

```

Output

Default Constructor

We can see that we have created a class instance of Demo with the name “instance”, as a result, the output we can see is Default Constructor is called.

Similarly, we can also call the parameterized constructor just by passing values here

```

#include <iostream>

using namespace std;

```

```

class Demo{
    public:
    int a, b;
    Demo()
    {
        cout << "Default Constructor" << endl;
    }
}

```



```

    }
    Demo(int a, int b):a(a),b(b)
    {
        cout << "parameterized Constructor values-" << a << " " << b <<
endl;
    }

```

```

}instance(100,200);

```

```

int main() {

```

```

    return 0;

```

```

}

```

Output

```

parameterized Constructor values-100 200

```

So by creating an instance just before the semicolon, we can create the Instance of class.

C++ Class Member Functions

A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

Let us take previously defined class to access the members of the class using a member function instead of directly accessing them –

```
class Box {  
  
    public:  
  
        double length;    // Length of a box  
  
        double breadth;    // Breadth of a box  
  
        double height;    // Height of a box  
  
        double getVolume(void); // Returns box volume  
  
};
```

Member functions can be defined within the class definition or separately using **scope resolution operator, : –**. Defining a member function within the class definition declares the function **inline**, even if you do not use the inline specifier. So either you can define **Volume()** function as below –

```
class Box {  
  
    public:  
  
        double length;    // Length of a box  
  
        double breadth;    // Breadth of a box  
  
        double height;    // Height of a box  
  
  
        double getVolume(void) {  
  
            return length * breadth * height;  
  
        }  
  
};
```

```
    }  
};
```

If you like, you can define the same function outside the class using the **scope resolution operator** (::) as follows –

```
double Box::getVolume(void) {  
    return length * breadth * height;  
}
```

Here, only important point is that you would have to use class name just before :: operator. A member function will be called using a dot operator (.) on a object where it will manipulate data related to that object only as follows –

```
Box myBox;    // Create an object
```

```
myBox.getVolume(); // Call member function for the object
```

```
#include <iostream>
```

```
using namespace std;
```

```
class Box {
```

```
public:
```

```
    double length;    // Length of a box
```

```
    double breadth;    // Breadth of a box
```

```
double height;          // Height of a box
```

```
// Member functions declaration
```

```
double getVolume(void);
```

```
void setLength( double len );
```

```
void setBreadth( double bre );
```

```
void setHeight( double hei );
```

```
};
```

```
// Member functions definitions
```

```
double Box::getVolume(void) {
```

```
    return length * breadth * height;
```

```
}
```

```
void Box::setLength( double len ) {
```

```
    length = len;
```

```
}
```

```
void Box::setBreadth( double bre ) {
```

```
    breadth = bre;
```

```
}
```

```
void Box::setHeight( double hei ) {
```

```
    height = hei;
```

```
}
```

```
// Main function for the program
```

```
int main() {
```

```

Box Box1;           // Declare Box1 of type Box
Box Box2;           // Declare Box2 of type Box
double volume = 0.0; // Store the volume of a box here

// box 1 specification
Box1.setLength(6.0);
Box1.setBreadth(7.0);
Box1.setHeight(5.0);

// box 2 specification
Box2.setLength(12.0);
Box2.setBreadth(13.0);
Box2.setHeight(10.0);

// volume of box 1
volume = Box1.getVolume();
cout << "Volume of Box1 : " << volume <<endl;

// volume of box 2
volume = Box2.getVolume();
cout << "Volume of Box2 : " << volume <<endl;
return 0;
}

```

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”.

Two Important property of Encapsulation

1. **Data Protection:** Encapsulation protects the internal state of an object by keeping its data members private. Access to and modification of these data members is restricted to the class's public methods, ensuring controlled and secure data manipulation.
2. **Information Hiding:** Encapsulation hides the internal implementation details of a class from external code. Only the public interface of the

class is accessible, providing abstraction and simplifying the usage of the class while allowing the internal implementation to be modified without impacting external code.

```
#include <iostream>

using namespace std;

class temp{
    int a;

int b;

public:
int solve(int input){
    a=input;
    b=a/2;
    return b;
}
};
```

```
int main() {
int n;

cin>>n;

temp half;

int ans=half.solve(n);

cout<<ans<<endl;
```

}

Features of Encapsulation

Below are the features of encapsulation:

1. 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.
2. The function which we are making inside the class must use only member variables, only then it is called *encapsulation*.
3. 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.
4. Encapsulation improves readability, maintainability, and security by grouping data and methods together.
5. It helps to control the modification of our data members.
- 6.
7. It helps to control the modification of our data members.

Encapsulation in C++



Encapsulation also leads to [data abstraction](#). 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.

```
#include <iostream>
```

```
#include <string>
```

```
using namespace std;
```

```
class Person {
```

```
private:
```

```
    string name;

    int age;

public:

    Person(string name, int age) {

        this->name = name;

        this->age = age;

    }

    void setName(string name) {

        this->name = name;

    }

    string getName() {

        return name;

    }

    void setAge(int age) {

        this->age = age;

    }

    int getAge() {

        return age;

    }

};
```

```
int main() {

    Person person("John Doe", 30);
```

```
cout << "Name: " << person.getName() << endl;
```

```
cout << "Age: " << person.getAge() << endl;
```

```
person.setName("Jane Doe");
```

```
person.setAge(32);
```

```
cout << "Name: " << person.getName() << endl;
```

```
cout << "Age: " << person.getAge() << endl;
```

```
return 0;
```

```
}
```

```
// 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;
```

```
}
```

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.

```
#include <iostream>
```

```
using namespace std;

// declaring class
class Circle {
    // access modifier
private:
    // Data Member
    float area;
    float radius;

public:
    void getRadius()
    {
        cout << "Enter radius\n";
        cin >> radius;
    }
    void findArea()
    {
        area = 3.14 * radius * radius;
        cout << "Area of circle=" << area;
    }
};

int main()
```

```
{  
  
    // creating instance(object) of class  
  
    Circle cir;  
  
    cir.getRadius(); // calling function  
  
    cir.findArea(); // calling function  
  
}
```

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:** Private access specifier means that the member function or data member can only be accessed by other member functions of the same class.
- **Protected:** A **protected** access specifier means that the member function or data member can be accessed by other member functions of the same class or by derived classes.
- **Public:** Public access specifier means that the member function or data member can be accessed by any code.

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

Points to Consider

As we have seen in the above example, access specifiers play an important role in implementing encapsulation in C++. The process of implementing encapsulation can be sub-divided into two steps:

1. Creating a class to encapsulate all the data and methods into a single unit.
2. Hiding relevant data using access specifiers.

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](https://www.geeksforgeeks.org/write.geeksforgeeks.org). See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

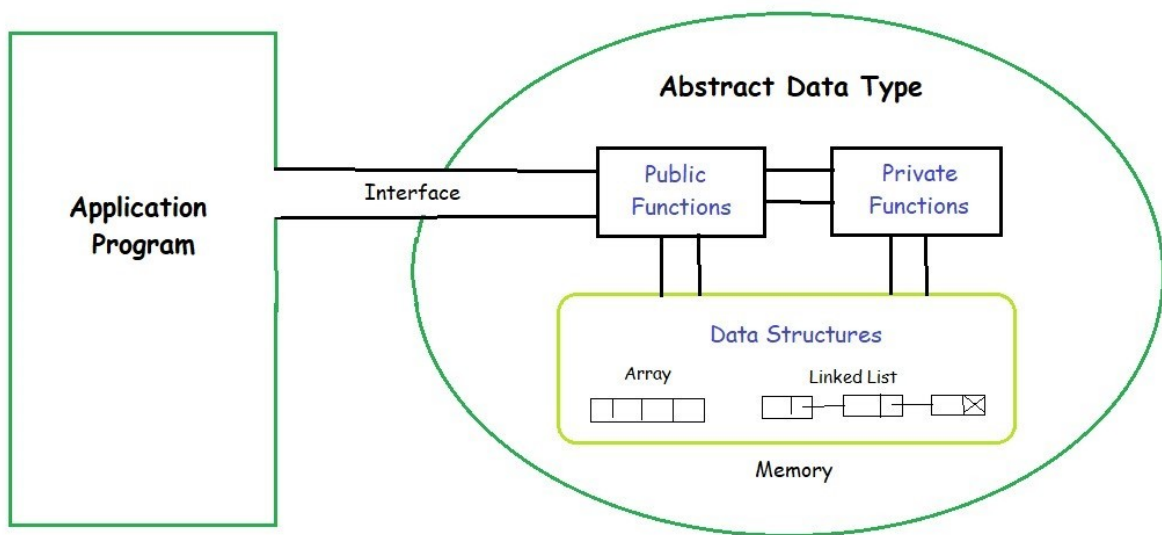
Encapsulation in C++



Abstract Data Types

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of values and a set of operations. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation-independent view.

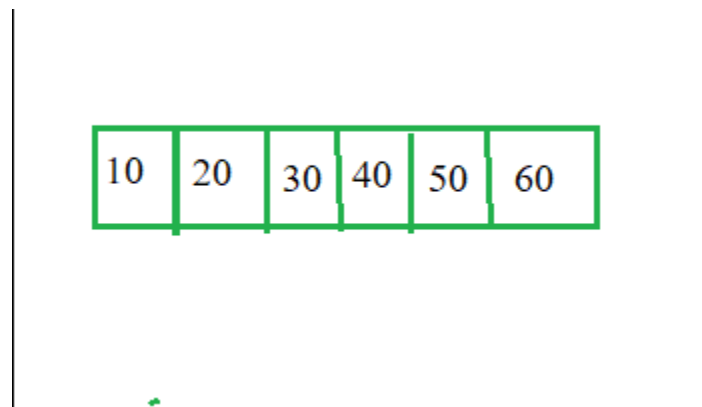
The process of providing only the essentials and hiding the details is known as abstraction.



The user of [data type](#) does not need to know how that data type is implemented, for example, we have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented.

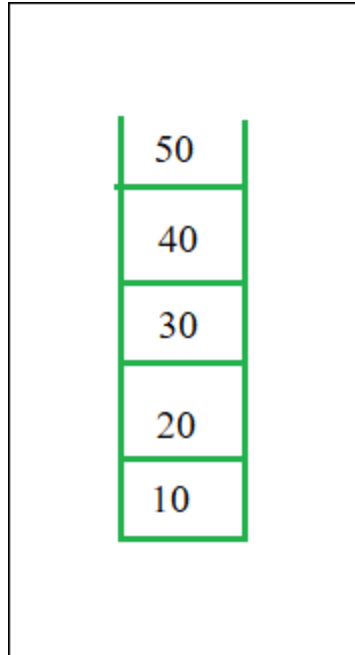
So a user only needs to know what a data type can do, but not how it will be implemented. Think of ADT as a black box which hides the inner structure and design of the data type. Now we'll define three ADTs namely [List](#) ADT, [Stack](#) ADT, [Queue](#) ADT.

List ADT



- The data is generally stored in key sequence in a list which has a head structure consisting of *count*, *pointers* and *address of compare function* needed to compare the data in the list.
- The data node contains the *pointer* to a data structure and a *self-referential pointer* which points to the next node in the list.
- The **List ADT Functions** is given below:
 - `get()` – Return an element from the list at any given position.
 - `insert()` – Insert an element at any position of the list.
 - `remove()` – Remove the first occurrence of any element from a non-empty list.
 - `removeAt()` – Remove the element at a specified location from a non-empty list.
 - `replace()` – Replace an element at any position by another element.
 - `size()` – Return the number of elements in the list.
 - `isEmpty()` – Return true if the list is empty, otherwise return false.
 - `isFull()` – Return true if the list is full, otherwise return false.

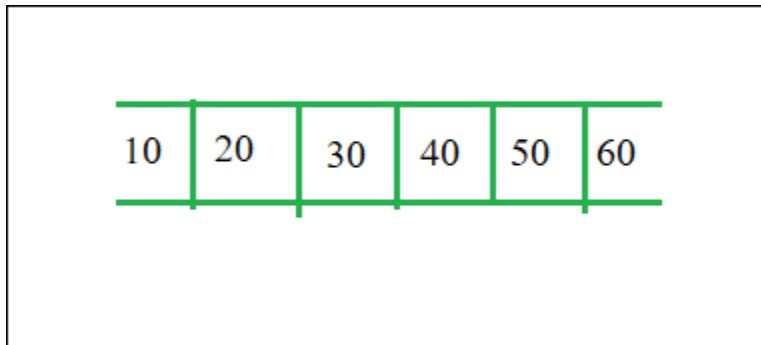
Stack ADT



- In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored.
- The program allocates memory for the *data* and *address* is passed to the stack ADT.
- The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack.
- The stack head structure also contains a pointer to *top* and *count* of number of entries currently in stack.
- `push()` – Insert an element at one end of the stack called top.
- `pop()` – Remove and return the element at the top of the stack, if it is not empty.
- `peek()` – Return the element at the top of the stack without removing it, if the stack is not empty.

- `size()` – Return the number of elements in the stack.
- `isEmpty()` – Return true if the stack is empty, otherwise return false.
- `isFull()` – Return true if the stack is full, otherwise return false.

3. Queue ADT



- The queue abstract data type (ADT) follows the basic design of the stack abstract data type.
- Each node contains a void pointer to the *data* and the *link pointer* to the next element in the queue. The program's responsibility is to allocate memory for storing the data.
- `enqueue()` – Insert an element at the end of the queue.
- `dequeue()` – Remove and return the first element of the queue, if the queue is not empty.
- `peek()` – Return the element of the queue without removing it, if the queue is not empty.
- `size()` – Return the number of elements in the queue.
- `isEmpty()` – Return true if the queue is empty, otherwise return false.
- `isFull()` – Return true if the queue is full, otherwise return false.

Features of ADT:

Abstract data types (ADTs) are

Abstract data types (ADTs) are a way of encapsulating data and operations on that data into a single unit. Some of the key features of ADTs include:

- **Abstraction:** The user does not need to know the implementation of the data structure only essentials are provided.
- **Better Conceptualization:** ADT gives us a better conceptualization of the real world.
- **Robust:** The program is robust and has the ability to catch errors.
- **Encapsulation:** ADTs hide the internal details of the data and provide a public interface for users to interact with the data. This allows for easier maintenance and modification of the data structure.
- **Data Abstraction:** ADTs provide a level of abstraction from the implementation details of the data. Users only need to know the operations that can be performed on the data, not how those operations are implemented.
- **Data Structure Independence:** ADTs can be implemented using different data structures, such as arrays or linked lists, without affecting the functionality of the ADT.
- **Information Hiding:** ADTs can protect the integrity of the data by allowing access only to authorized users and operations. This helps prevent errors and misuse of the data.

- **Modularity:** ADTs can be combined with other ADTs to form larger, more complex data structures. This allows for greater flexibility and modularity in programming.

Overall, ADTs provide a powerful tool for organizing and manipulating data in a structured and efficient manner.

Abstract data types (ADTs) have several advantages and disadvantages that should be considered when deciding to use them in software development. Here are some of the main advantages and disadvantages of using ADTs:

Advantages:

- **Encapsulation:** ADTs provide a way to encapsulate data and operations into a single unit, making it easier to manage and modify the data structure.
- **Abstraction:** ADTs allow users to work with data structures without having to know the implementation details, which can simplify programming and reduce errors.
- **Data Structure Independence:** ADTs can be implemented using different data structures, which can make it easier to adapt to changing needs and requirements.
- **Information Hiding:** ADTs can protect the integrity of data by controlling access and preventing unauthorized modifications.

- **Modularity:** ADTs can be combined with other ADTs to form more complex data structures, which can increase flexibility and modularity in programming.

Disadvantages:

- **Overhead:** Implementing ADTs can add overhead in terms of memory and processing, which can affect performance.
- **Complexity:** ADTs can be complex to implement, especially for large and complex data structures.
- **Learning Curve:** Using ADTs requires knowledge of their implementation and usage, which can take time and effort to learn.
- **Limited Flexibility:** Some ADTs may be limited in their functionality or may not be suitable for all types of data structures.
- **Cost:** Implementing ADTs may require additional resources and investment, which can increase the cost of development.

Overall, the advantages of ADTs often outweigh the disadvantages, and they are widely used in software development to manage and manipulate data in a structured and efficient way. However, it is important to consider the specific needs and requirements of a project when deciding whether to use ADTs.

Static Member Function in C++

The static keyword is used with a variable to make the memory of the variable static once a static variable is declared its memory can't be changed. To know more about static keywords refer to the article [static Keyword in C++](#).

Static Member in C++

Static members of a class are not associated with the objects of the class. Just like a static variable once declared is allocated with memory that can't be changed every object points to the same memory. To know more about the topic refer to a [static Member in C++](#).

Example:

```
class Person{  
    static int index_number;  
};
```

// C++ Program to demonstrate

// Static member in a class

```
#include <iostream>
```

```
using namespace std;
```

```
class Student {
```

```
public:
```

```
    // static member
```

```
    static int total;
```

```
    // Constructor called
```

```
    Student() { total += 1; }
```

```
};
```

```
int Student::total = 0;
```



```
int main()
{
    // Student 1 declared
    Student s1;
    cout << "Number of students:" << s1.total << endl;

    // Student 2 declared
    Student s2;
    cout << "Number of students:" << s2.total << endl;

    // Student 3 declared
    Student s3;
    cout << "Number of students:" << s3.total << endl;
    return 0;
}
```

Static Member Function in C++

Static Member Function in a class is the function that is declared as static because of which function attains certain properties as defined below:

- A static member function is independent of any object of the class.
- A static member function can be called even if no objects of the class exist.

- A static member function can also be accessed using the class name through the scope resolution operator.
- A static member function can access static data members and static member functions inside or outside of the class.
- Static member functions have a scope inside the class and cannot access the current object pointer.
- You can also use a static member function to determine how many objects of the class have been created.

The reason we need Static member function:

- Static members are frequently used to store information that is shared by all objects in a class.
- For instance, you may keep track of the quantity of newly generated objects of a specific class type using a static data member as a counter. This static data member can be increased each time an object is generated to keep track of the overall number of objects.

// C++ Program to show the working of

// static member functions

#include <iostream>

using namespace std;

class Box

{

```
private:

static int length;

static int breadth;

static int height;


public:


static void print()
{
    cout << "The value of the length is: " << length << endl;
    cout << "The value of the breadth is: " << breadth << endl;
    cout << "The value of the height is: " << height << endl;
}

};
```

```
// initialize the static data members
```

```
int Box :: length = 10;
int Box :: breadth = 20;
int Box :: height = 30;
```

```
// Driver Code
```

```

int main()
{

    Box b;

    cout << "Static member function is called through Object name: \n" <<
endl;

    b.print();

    cout << "\nStatic member function is called through Class name: \n" <<
endl;

    Box::print();

    return 0;
}

```

Array of object

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.

Example:

Let's consider an example of taking random integers from the user.

34	67	78	32	78	98	89
----	----	----	----	----	----	----

0	1	2	3	4	5	6
---	---	---	---	---	---	---

← Array Indices →

Array of 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[number of objects];
```

The Array of Objects stores *objects*. An array of a class type is also known as an array of objects.

Example#1:

Storing more than one Employee data. Let's assume there is an array of objects for storing employee data emp[50].

Objects	Employee Id	Employee name
<u>emp[0]</u> →		
<u>emp[1]</u> →		
<u>emp[2]</u> →		
<u>emp[3]</u> →		

Below is

// C++ program to implement

// the above approach

```
#include<iostream>
```

```
using namespace std;
```

```
class Employee
```

```
{
```

```
int id;
```

```
char name[30];
```

```
public:
```

```
void getdata();//Declaration of function
```

```
void putdata();//Declaration of function
```

```
};
```

```

void Employee::getdata(){//Defining of function
cout<<"Enter Id : ";
cin>>id;
cout<<"Enter Name : ";
cin>>name;
}

void Employee::putdata(){//Defining of function
cout<<id<<" ";
cout<<name<<" ";
cout<<endl;
}

int main(){
Employee emp; //One member
emp.getdata();//Accessing the function
emp.putdata();//Accessing the function
return 0;

}

```

Advantages of Array of Objects:

1. The array of objects represent storing multiple objects in a single name.

2. In an array of objects, the data can be accessed randomly by using the index number.
3. Reduce the time and memory by storing the data in a single variable.

What is an object?

In [object-oriented programming \(OOP\)](#), objects are the things you think about first in designing a program and they are also the units of code that are eventually derived from the process. In between, each object is made into a generic [class](#) of object, and even more generic classes are defined so that objects can share models and reuse the class definitions in their code. Each object is an instance of a particular class or subclass with the class's own methods or procedures and data [variables](#). An object is what actually runs in the computer.

Characteristics of an object

All individual objects possess three basic characteristics -- identity, state and behavior. Understanding these characteristics is crucial to knowing how objects and object-oriented logic work.

- **Identity** means that each object has its own [object identifier](#) and can be differentiated from all other objects. Each object's name, or identity, is unique and distinct from other objects.

- **State** refers to the properties of an object. For example, values of variables in the object contain data that can be added, changed or deleted.
- **Behavior** refers to actions that the object can take. For example, one object can respond to another object to carry out software functions.

Some of the things in programming that can be defined as objects include the following:

- **variables**, which hold values that can be changed;
- **data structures**, which are specialized formats used to organize and process data;
- **functions**, which are named procedures that perform a defined task; and
- **methods**, which are programmed procedures that are defined as components of a parent class and are included in any instance of that class.

What is C++ Garbage Collection?

Garbage collection is a memory management technique. It is a separate automatic memory management method which is **used in programming languages** where manual memory management is not preferred or done.

In the manual memory management method, the user is required to mention the memory which is in use and which can be deallocated, whereas the garbage collector collects the memory which is occupied by variables or objects which are no more in use in the program. Only memory will be managed by garbage collectors, other resources such as destructors, user interaction window or files will not be handled by the garbage collector.

ew languages need garbage collectors as part of the language for good efficiency. These languages are called as garbage-collected languages.

For example, Java, C# and most of the scripting languages needs garbage collection as part of their functioning. Whereas languages such as **C and C++ support** manual memory management which works similar to the garbage collector. There are few languages that support both garbage collection and manually managed memory

allocation/deallocation and in such cases, a separate heap of memory will be allocated to the garbage collector and manual memory.

Some of the bugs can be prevented when the garbage collection method is used. Such as:

- dangling pointer problem in which the memory pointed is already deallocated whereas the pointer still remains and points to different reassigned data or already deleted memory
- the problem which occurs when we try to delete or deallocate a memory second time which has already been deleted or reallocated to some other object
- removes problems or bugs associated with data structures and does the memory and data handling efficiently
- memory leaks or memory exhaustion problem can be avoided

Advantages and Disadvantages of Manual Memory Management

Advantages of manual memory management are that the user would have complete control over both allocating and deallocating operations and also know when a new memory is allocated and when it is deallocated or released. But in the case of garbage collection, at the exact same instance after the usage the memory will not be released it will get released when it encounters it during the periodic operation.

Also in the case of manual memory management, the destructor will be called at the same moment when we call the 'delete' command. But in case of garbage collector that is not implemented.

There are a few issues associated with using manual memory management. Sometimes we might tend to double delete the memory occupied. When we delete the already deleted pointer or memory, there are chances that the pointer might be referencing some other data and can be in use.

Advantages and Disadvantages of Garbage Collector

One major disadvantage of garbage collection is the time involved or CPU cycles involved to find the unused memory and deleting it, even if the user knows which pointer memory can be released and not in use. Another disadvantage is, we will not know the time when it is deleted nor when the destructor is called.