**UNIT- I**

**C++ Programming Concepts: Review of C, input and output in C++, functions in C++- value parameters, reference parameters, Parameter passing, function overloading, function templates, arrays, pointers, new and delete operators, class and object, access specifiers, friend functions, constructors and destructor, Operator overloading, class templates.**

### **Procedural Programming VS Object-Oriented Programming**

Below are some of the differences between procedural and object-oriented programming:

| * **Procedural Oriented Programming** | * **Object-Oriented Programming** |
| --- | --- |
| * In procedural programming, the program is divided into small parts called *functions*. | * In object-oriented programming, the program is divided into small parts called *objects*. |
| * Procedural programming follows a *top-down approach*. | * Object-oriented programming follows a *bottom-up approach*. |
| * There is no access specifier in procedural programming. | * Object-oriented programming has access specifiers like private, public, protected, etc. |
| * Adding new data and functions is not easy. | * Adding new data and function is easy. |
| * Procedural programming does not have any proper way of hiding data so it is *less secure*. | * Object-oriented programming provides data hiding so it is *more secure*. |
| * In procedural programming, overloading is not possible. | * Overloading is possible in object-oriented programming. |
| * In procedural programming, there is no concept of data hiding and inheritance. | * In object-oriented programming, the concept of data hiding and inheritance is used. |
| * In procedural programming, the function is more important than the data. | * In object-oriented programming, data is more important than function. |
| * Procedural programming is based on the *unreal world*. | * Object-oriented programming is based on the *real world*. |
| * Procedural programming is used for designing medium-sized programs. | * Object-oriented programming is used for designing large and complex programs. |
| * Procedural programming uses the concept of procedure abstraction. | * Object-oriented programming uses the concept of data abstraction. |
| * Code reusability absent in procedural programming, | * Code reusability present in object-oriented programming. |
| * Examples: C, FORTRAN, Pascal, Basic, etc. | * Examples: C++, Java, Python, C#, etc. |

**C++ Introduction and Its Features**

**Introduction to C++**

C++ is a powerful, high-level, general-purpose programming language developed by Bjarne Stroustrup in 1983. It is an extension of the C programming language with added features for object-oriented programming (OOP). It supports procedural, object-oriented, and generic programming paradigms, making it a versatile language for software development.

**Key Features of C++**

1. **Object-Oriented Programming**
   * Supports key OOP concepts: Classes, Objects, Inheritance, Polymorphism, Encapsulation, and Abstraction.
2. **Platform Independent (Machine-Independent)**
   * C++ code is portable, and programs can run on different platforms with minimal or no modification.
3. **High Performance**
   * C++ is closer to hardware, enabling efficient memory and performance management.
4. **Rich Standard Library**
   * Provides a vast set of libraries (e.g., STL) that include algorithms, data structures, and more.
5. **Extensibility**
   * Enables the addition of new functionality by creating classes and objects.
6. **Static Type System**
   * Performs type checking at compile time to prevent runtime errors.
7. **Multi-Paradigm**
   * Combines procedural programming with object-oriented programming and generic programming.
8. **Low-Level Manipulation**
   * Provides direct access to hardware using pointers and memory management.

**Basic Structure of a C++ Program**

#include <iostream> // Include input/output stream

using namespace std; // Use the standard namespace

int main() {

cout<< "Hello, World!" <<endl; // Print output

return 0; // Return success status

}

**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.

<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.

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.

#include <iostream>

using namespace std;

int main( )

{

cout << "Welcome to CPP " << endl;

}

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.

#include <iostream>

using namespace std;

int main( ) {

int age;

cout << "Enter your age: ";

cin >> age;

cout << "Your age is: " << age << endl;

}

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.

#include <iostream>

using namespace std;

int main( ) {

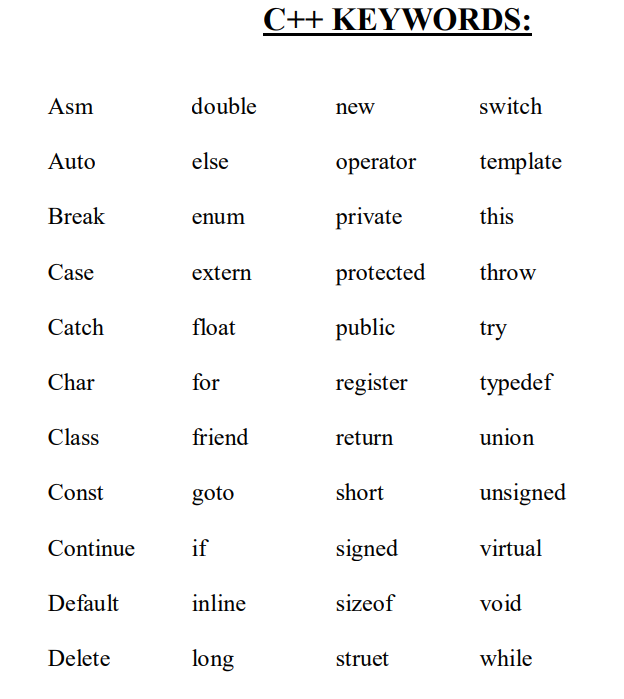
cout << "C++ Tutorial";

cout << " Javatpoint"<<endl;

cout << "End of line"<<endl;

}

**KEYWORDS:** The keywords implement specific C++ language feature. They are explicitly reserved identifiers and can’t be used as names for the program variables or other user defined program elements. The keywords not found in ANSI C are shown in red letter.



**IDENTIFIERS:**

Identifiers refers to the name of variable , functions, array, class etc. created by programmer. Each language has its own rule for naming the identifiers. The following rules are common for both C and C++.

1. Only alphabetic chars, digits and under score are permitted.

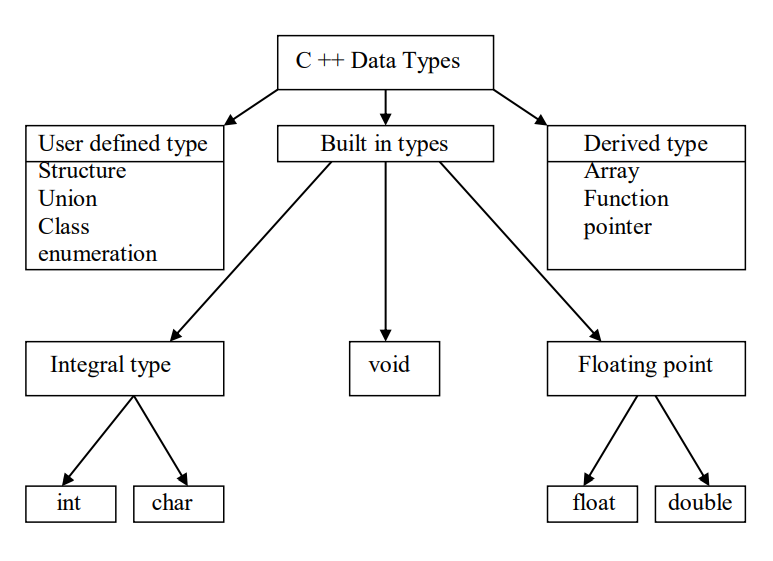
2. The name can’t start with a digit.

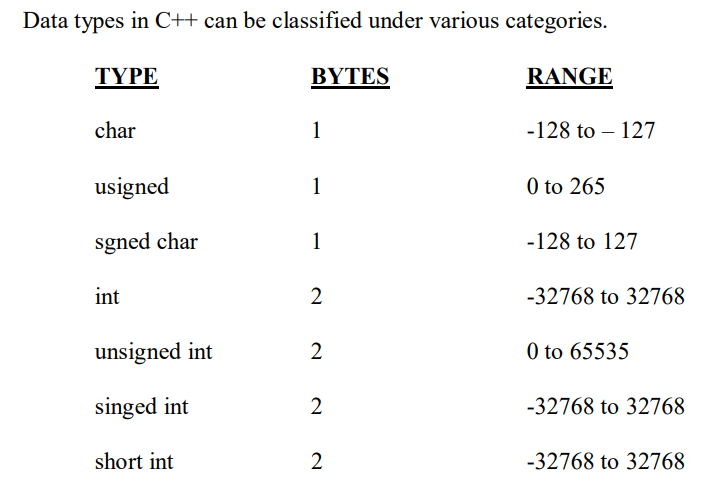
3. Upper case and lower case letters are distinct.

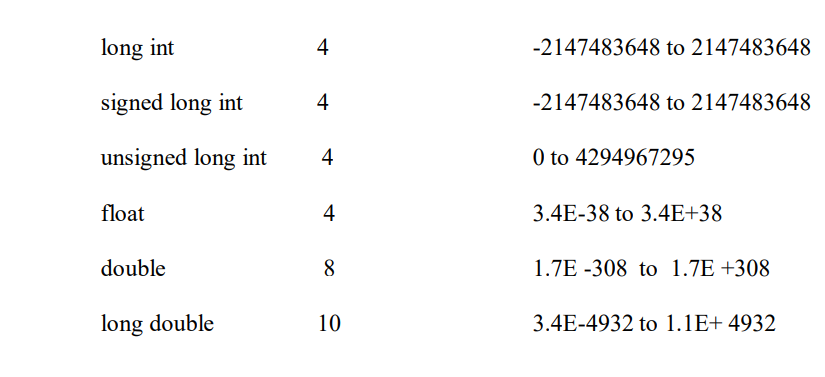
4. A declared keyword can’t be used as a variable name.

In ANSI C the maximum length of a variable is 32 chars but in c++ there is no bar.

**BASIC DATA TYPES IN C++**

****





**USER DEFINED DATA TYPES:**

**STRUCTERS AND CLASSES**

We have used user defined data types such as struct,and union in C. While these more features have been added to make them suitable for object oriented programming. C++ also permits us to define another user defined data type known as class which can be used just like any other basic data type to declare a variable. The class variables are known as objects, which are the central focus of oops.

**ENUMERATED DATA TYPE:**

An enumerated data type is another user defined type which provides a way for attaching names to number, these by increasing comprehensibility of the code.

The enum keyword automatically enumerates a list of words by assigning them values 0,1,2 and soon. This facility provides an alternative means for creating symbolic.

Example:

enum shape { circle,square,triangle}

enum colour{red,blue,green,yellow}

enum position {off,on}

The enumerated data types differ slightly in C++ when compared with ANSI C. In C++, the tag names shape, colour, and position become new type names. That means we can declare new variables using the tag names.

Example:

Shape ellipse;//ellipse is of type shape

colour background ; // back ground is of type colour

ANSI C defines the types of enums to be ints. In C++,each enumerated data type retains its own separate type. This means that C++ does not allow an int value to be automatically converted to an enum.

Example:

colour background =blue; //vaid

colour background =7; //error in c++

colour background =(colour) 7;//ok

How ever an enumerated value can be used in place of an int value.

Example: int c=red ;//valid, colour type promoted to int

By default,

the enumerators are assigned integer values starting with 0 for the first enumerator, 1 for the second and so on. We can also write

enum color {red, blue=4,green=8};

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

C++ also permits the creation of anonymous enums ( i.e, enums without tag names)

Example:

enum{off,on};

Here off is 0 and on is 1.these constants may be referenced in the same manner as regular constants.

Example:

int switch-1=off;

int switch-2=on;

ANSI C permits an enum defined with in a structure or a class, but the enum is globally visible. In C++ an enum defined with in a class is local to that class.Operators and expressions are fundamental building blocks in C++ programming. They are used to manipulate data and perform operations.

**DECLARATION OF VARIABLES:**

In ANSIC C all the variable which is to be used in programs must be declared at the beginning of the program .But in C++ we can declare the variables any whose in the program where it requires .This makes the program much easier to write and reduces the errors that may be caused by having to scan back and forth. It also makes the program easier to understand because the variables are declared in the context of their use.

**REFERENCE VARIABLES:**

C++interfaces a new kind of variable known as the reference variable. A references variable provides an alias.(alternative name) for a previously defined variable. For example ,if we make the variable sum a reference to the variable total, then sum and total can be used interchangeably to represent the variuble. A reference variable is created as follows:

**Synatx: Datatype & reference –name=variable name;**

Example:

**float total=1500;**

**float &sum=total;**

Here sum is the alternative name for variables total, both the variables refer to the same data object in the memory . A reference variable must be initialized at the time of declaration .

Note that C++ assigns additional meaning to the symbol & here & is not an address operator .The notation float & means reference to float.

**Example:**

**int n[10];**

**int &x=n[10];**

**char &a=’\n’;**

**Operators**

Operators in C++ are symbols that perform operations on variables and values. C++ operators can be categorized into the following types:

**1.1. Arithmetic Operators**

These operators are used to perform basic arithmetic operations like addition, subtraction, multiplication, division, and modulus (remainder).

* **+** : Addition
* **-** : Subtraction
* **\*** : Multiplication
* **/** : Division
* **%** : Modulus (remainder of division)

**Example:**

cpp

Copy

int a = 10, b = 3;

cout<< "a + b = " << a + b <<endl; // 13

cout<< "a - b = " << a - b <<endl; // 7

cout<< "a \* b = " << a \* b <<endl; // 30

cout<< "a / b = " << a / b <<endl; // 3 (integer division)

cout<< "a % b = " << a % b <<endl; // 1

**1.2. Relational Operators**

These operators are used to compare two values. They return a boolean value (true or false).

* **==** : Equal to
* **!=** : Not equal to
* **<** : Less than
* **>** : Greater than
* **<=** : Less than or equal to
* **>=** : Greater than or equal to

**Example:**

cpp

Copy

int x = 5, y = 8;

cout<< (x == y) <<endl; // 0 (false)

cout<< (x != y) <<endl; // 1 (true)

cout<< (x < y) <<endl; // 1 (true)

cout<< (x > y) <<endl; // 0 (false)

**1.3. Logical Operators**

These operators are used to perform logical operations on boolean values.

* **&&** : Logical AND
* **||** : Logical OR
* **!** : Logical NOT

**Example:**

cpp

Copy

bool p = true, q = false;

cout<< (p && q) <<endl; // 0 (false)

cout<< (p || q) <<endl; // 1 (true)

cout<< !p<<endl; // 0 (false)

**1.4. Assignment Operators**

These operators are used to assign values to variables.

* **=** : Simple assignment
* **+=** : Add and assign
* **-=** : Subtract and assign
* **\*=** : Multiply and assign
* **/=** : Divide and assign
* **%=** : Modulus and assign

**Example:**

cpp

Copy

int num = 10;

num += 5; // num = num + 5

cout<< num <<endl; // 15

num \*= 2; // num = num \* 2

cout<< num <<endl; // 30

**1.5. Unary Operators**

These operators operate on a single operand.

* **++** : Increment (adds 1 to the operand)
* **--** : Decrement (subtracts 1 from the operand)
* **+** : Unary plus (returns the value)
* **-** : Unary minus (negates the value)

**Example:**

cpp

Copy

int num = 10;

cout<< ++num <<endl; // 11 (pre-increment)

cout<< num++ <<endl; // 11 (post-increment)

cout<< num <<endl; // 12

cout<< --num <<endl; // 11 (pre-decrement)

cout<< num-- <<endl; // 11 (post-decrement)

cout<< num <<endl; // 10

**1.6. Bitwise Operators**

These operators are used to perform operations on binary numbers.

* **&** : Bitwise AND
* **|** : Bitwise OR
* **^** : Bitwise XOR
* **~** : Bitwise NOT
* **<<** : Left shift
* **>>** : Right shift

**Example:**

cpp

Copy

int a = 5, b = 3; // 5 = 0101, 3 = 0011 in binary

cout<< (a & b) <<endl; // 1 (AND)

cout<< (a | b) <<endl; // 7 (OR)

cout<< (a ^ b) <<endl; // 6 (XOR)

cout<< (~a) <<endl; // -6 (NOT)

cout<< (a << 1) <<endl; // 10 (Left shift)

cout<< (a >> 1) <<endl; // 2 (Right shift)

**1.7. Conditional (Ternary) Operator**

This operator evaluates a condition and returns one of two values depending on whether the condition is true or false.

* **condition ?value\_if\_true : value\_if\_false**

**Example:**

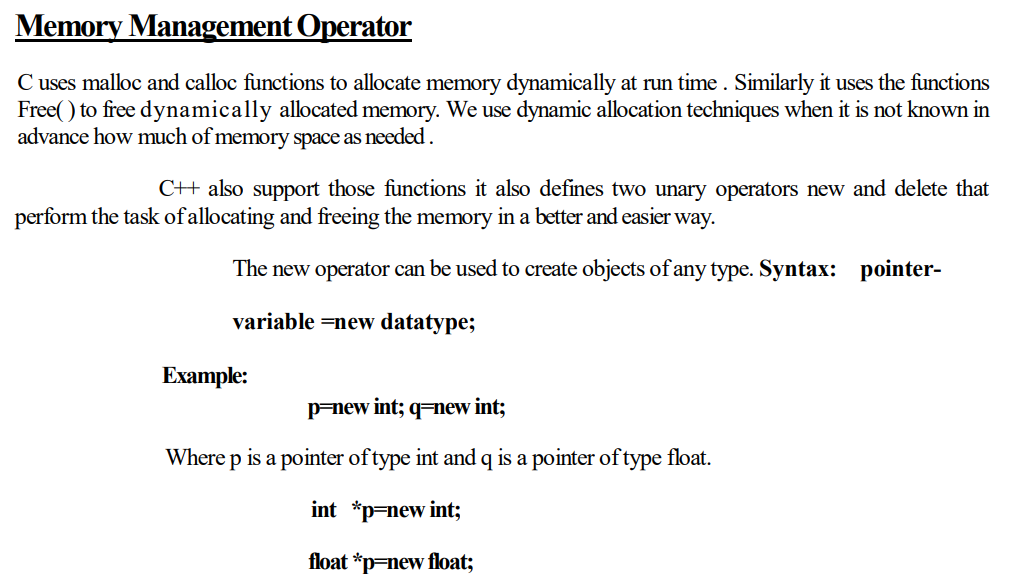
cpp

Copy

int a = 5, b = 3;

int result = (a > b) ? a : b;

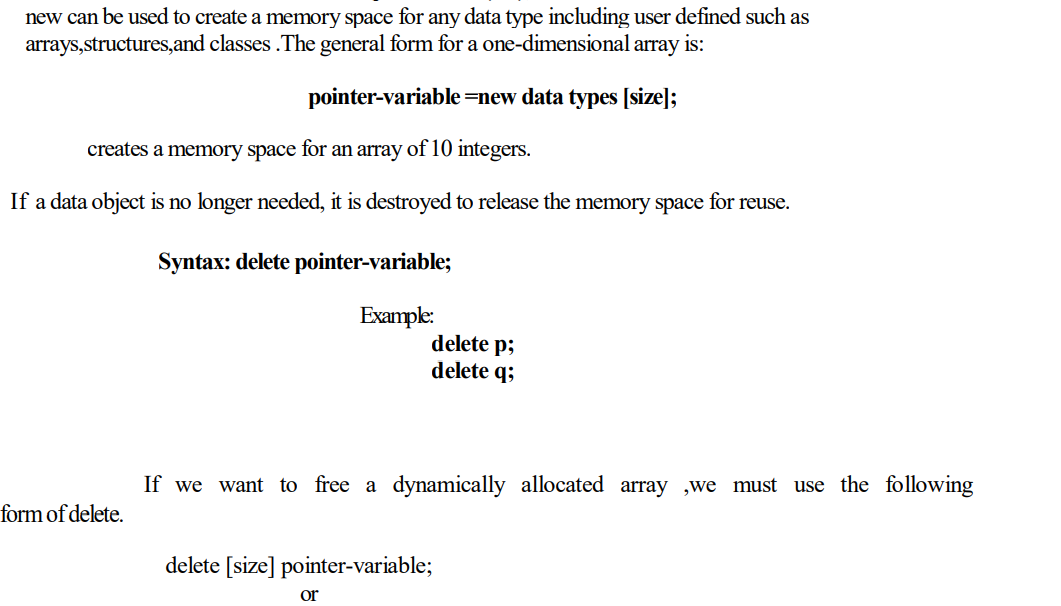
cout<< "Greater number: " << result <<endl; // 5

**Memory Management**

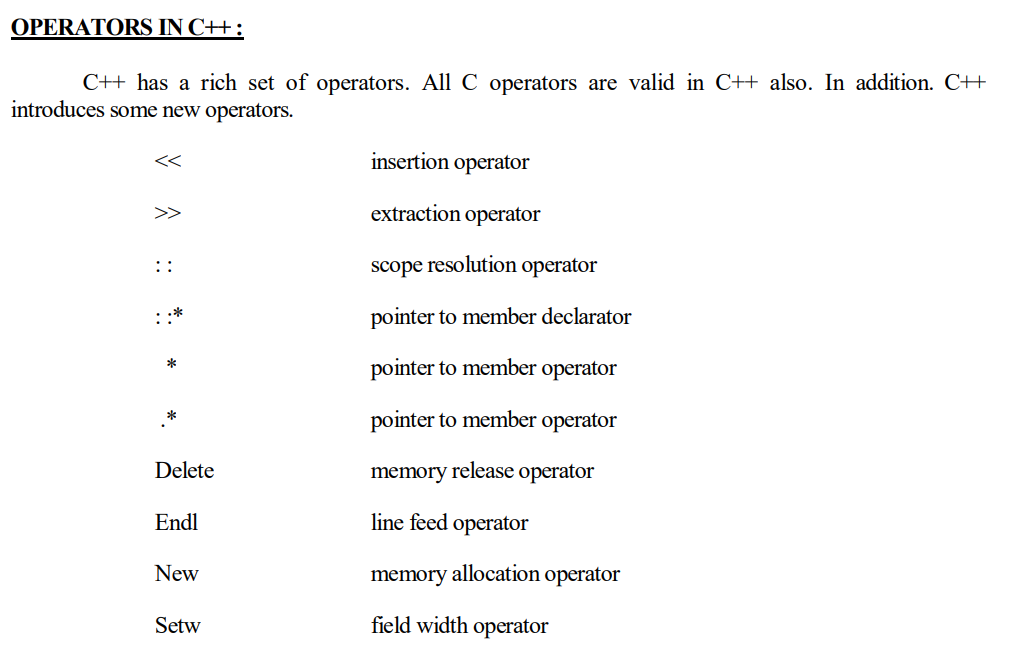
Subsequently, the statements

\*p=25;

\*q=7.5;

****

delete [ ] pointer variable;



**Expressions in C++**

An **expression** is a combination of variables, constants, operators, and function calls that are evaluated to produce a value.

**2.1. Types of Expressions**

* **Arithmetic Expressions**: Involve arithmetic operators to perform calculations.

cpp

Copy

int x = 10, y = 20;

int result = x + y \* 2; // 50 (since multiplication has higher precedence than addition)

* **Relational Expressions**: Return boolean values (true or false) by comparing variables.

cpp

Copy

bool isEqual = (x == y); // false

* **Logical Expressions**: Evaluate to a boolean value (true or false).

cpp

Copy

bool result = (x > 5 && y < 25); // true

* **Bitwise Expressions**: Perform bit-level operations.

cpp

Copy

int a = 5, b = 3;

int result = a &b; // 1 (AND)

* **Assignment Expressions**: Involve assigning a value to a variable.

cpp

Copy

int num = 5;

num += 3; // num = 8

* **Increment/Decrement Expressions**: Increase or decrease the value of a variable by 1.

cpp

Copy

int a = 10;

++a; // a = 11

a--; // a = 10

**2.2. Expression Evaluation**

C++ follows **operator precedence** and **associativity** rules to evaluate expressions.

* **Operator Precedence**: Determines the order in which operators are evaluated.
* **Associativity**: Determines the direction in which operators of the same precedence are evaluated.

For example:

cpp

Copy

int x = 5, y = 3, z = 2;

int result = x + y \* z; // 5 + (3 \* 2) = 11

**Operator Precedence:**

| **Operator** | **Precedence Order** |
| --- | --- |
| (), [], . | Highest |
| \*, /, % | Medium |
| +, - | Lower |

**3. Summary**

* **Operators** perform operations on operands (values or variables).
* **Expressions** are combinations of operators and operands that evaluate to a value.
* C++ supports various types of operators, such as **arithmetic**, **relational**, **logical**, **bitwise**, **assignment**, and **unary**.
* Understanding **operator precedence** and **associativity** is essential for writing correct and efficient expressions.

Operators and expressions are essential for manipulating data and controlling the flow of your program.

**Operator Precedence and Expression Evaluation in C++**

Operator precedence and expression evaluation are critical concepts in C++ programming. They determine the order in which parts of an expression are computed and ensure that results are predictable and accurate.

**1. Operator Precedence**

Operator precedence defines the rules for determining the sequence in which operators in an expression are evaluated. Operators with higher precedence are evaluated before those with lower precedence.**Operator Precedence Table**

Here is a list of C++ operators arranged from highest to lowest precedence:

|  |  |  |  |
| --- | --- | --- | --- |
| **Precedence** | **Operators** | **Description** | **Associativity** |
| 1 | :: | Scope resolution | Left-to-right |
| 2 | ++, -- (postfix), () (function call), [] (array subscript), -> (member access) | Postfix | Left-to-right |
| 3 | ++, -- (prefix), +, -, !, ~, \*, & (unary), sizeof, typeid | Unary | Right-to-left |
| 4 | \*, /, % | Multiplication, Division, Modulus | Left-to-right |
| 5 | +, - | Addition, Subtraction | Left-to-right |
| 6 | <<, >> | Bitwise Shift | Left-to-right |
| 7 | <, <=, >, >= | Relational Operators | Left-to-right |
| 8 | ==, != | Equality Operators | Left-to-right |
| 9 | & | Bitwise AND | Left-to-right |
| 10 | ^ | Bitwise XOR | Left-to-right |
| 11 | ` | ` | Bitwise OR | Left-to-right |
| 12 | && | Logical AND | Left-to-right |  |
| 13 | ` |  | ` | Logical OR | Left-to-right |
| 14 | ? : | Ternary | Right-to-left |  |  |
| 15 | =, +=, -=, \*=, /=, etc. | Assignment | Right-to-left |  |  |
| 16 | , | Comma | Left-to-right |  |  |

**2. Expression Evaluation**

Expression evaluation in C++ involves computing the value of expressions by following the precedence and associativity rules of operators.

**Key Points in Expression Evaluation**

1. **Order of Evaluation:**
   * Operators with higher precedence are evaluated first.
   * If two operators have the same precedence, associativity determines the evaluation order.
2. **Associativity:**
   * Left-to-right associativity means expressions are evaluated from left to right (e.g., a + b - c is evaluated as (a + b) - c).
   * Right-to-left associativity means expressions are evaluated from right to left (e.g., a = b = c is evaluated as a = (b = c)).
3. **Parentheses:**
   * Parentheses can override precedence and associativity. For example, in (a + b) \* c, the addition is performed first despite multiplication having higher precedence.
4. **Short-Circuit Evaluation:**
   * Logical operators && and || use short-circuit evaluation:
     + For &&, if the first operand is false, the second operand is not evaluated.
     + For ||, if the first operand is true, the second operand is not evaluated.

**3. Common Scenarios in Expression Evaluation**

**Example 1: Simple Arithmetic Expression**

int result = 5 + 3 \* 2;

**Explanation:**

* Multiplication (\*) has higher precedence than addition (+).
* The expression is evaluated as 5 + (3 \* 2) = 5 + 6 = 11.

**Example 2: Using Parentheses**

int result = (5 + 3) \* 2;

**Explanation:**

* Parentheses override the precedence.
* The expression is evaluated as (5 + 3) \* 2 = 8 \* 2 = 16.

**Example 3: Logical Operators**

bool result = (5 > 3) && (2 < 4);

**Explanation:**

* Relational operators (>, <) are evaluated first.
* Then the logical AND (&&) is evaluated.
* (5 > 3) = true, (2 < 4) = true, true && true = true.

**Example 4: Ternary Operator**

int result = (5 > 3) ?10 : 20;

**Explanation:**

* The condition (5 > 3) is evaluated first.
* Since it is true, the first operand (10) is chosen.
* Result: 10.

**4. Common Mistakes and Debugging Tips**

1. **Misinterpreting Precedence:**
   * Always refer to the precedence table for complex expressions.
   * Use parentheses to clarify the intended order of evaluation.
2. **Integer Division:**
   * Division of integers truncates the result to an integer. For example, 5 / 2 evaluates to 2.
3. **Short-Circuit Logic:**
   * Be mindful of short-circuiting when using logical operators (&&, ||). The second operand may not be evaluated.
4. **Uninitialized Variables:**
   * Using uninitialized variables in expressions leads to undefined behavior.
5. **Modulus Operator:**
   * The modulus operator (%) is only valid for integer operands. Using it with floating-point numbers results in a compilation error.

**5. Practical Examples**

**Example 1: Nested Expressions**

int result = 10 + 20 \* (5 / 2) - 3;

**Explanation:**

1. Parentheses are evaluated first: 5 / 2 = 2 (integer division).
2. Multiplication: 20 \* 2 = 40.
3. Addition and subtraction: 10 + 40 - 3 = 47.

**Example 2: Logical Operators**

bool result = (10 > 5) || (5 < 3) && (2 == 2);

**Explanation:**

1. && is evaluated before ||.
2. (5 < 3) && (2 == 2) = false && true = false.
3. (10 > 5) || false = true || false = true.

**6. Summary**

1. Operator precedence determines the order in which operators are evaluated.
2. Associativity resolves ambiguity when operators have the same precedence.
3. Parentheses can override default precedence and associativity.
4. Logical operators may use short-circuit evaluation.
5. Always test and debug expressions to ensure correctness, especially in complex scenarios.

By understanding these rules and practicing with examples, you can confidently evaluate and debug expressions in C++ programs.

**In C++,** **type conversion**

refers to the process of converting one data type into another. There are two main types of type conversions in C++:

1. **Implicit Type Conversion** (Automatic Conversion)
2. **Explicit Type Conversion** (Manual Conversion)

### 1. **Implicit Type Conversion (Automatic Conversion)**

Implicit type conversion happens automatically when you assign a value of one data type to a variable of another type. The C++ compiler performs the conversion automatically if it's safe, i.e., no data will be lost or corrupted during the conversion.

* **Example of Implicit Type Conversion:**
* int x = 5;
* double y = x; // int to double (automatic conversion)

In this example, the int variable x is automatically converted to a double when assigned to y. This is a **promotion**, where a smaller type (int) is converted to a larger type (double).

#### Common Implicit Conversions:

* **Integer to Floating Point**: int to float or double.
* **Char to Integer**: char to int (ASCII value).
* **Smaller Integer to Larger Integer**: short to int, int to long.

### 2. **Explicit Type Conversion (Type Casting)**

Explicit type conversion is performed manually by the programmer using **type casting**. It is necessary when the conversion is not automatically done by the compiler, or when you want to control how the conversion is done.

* **Syntax of Explicit Type Conversion (Type Casting)**:
* new\_type variable = (new\_type) expression;

#### Example of Explicit Type Conversion:

double x = 9.7;

int y = (int) x; // explicit conversion from double to int (drops decimal part)

In this case, the double value x is explicitly cast to int, which results in the value 9 (the decimal part is truncated).

### Types of Explicit Type Conversion

1. **C-Style Cast**: This is the traditional way of type casting in C/C++.
2. int x = (int) 3.5; // C-style cast from double to int
3. **Function-style Cast**: This is another form of casting, which is similar to C-style but uses function-like syntax.
4. double x = 3.5;
5. int y = int(x); // function-style cast
6. **static\_cast<>()**: The static\_cast operator is used for conversions between related types (e.g., int to float, base class to derived class). It checks for safety at compile time.
7. double x = 3.7;
8. int y = static\_cast<int>(x); // Converts double to int, truncates the decimal part
9. **dynamic\_cast<>()**: dynamic\_cast is primarily used for downcasting in polymorphic class hierarchies (classes with virtual functions). It performs runtime checks and returns nullptr for invalid casts.
10. Base\* basePtr = new Derived();
11. Derived\* derivedPtr = dynamic\_cast<Derived\*>(basePtr);
12. **const\_cast<>()**: const\_cast is used to add or remove the const qualifier from a variable.
13. const int x = 10;
14. int\* y = const\_cast<int\*>(&x); // Removing constness (dangerous if used improperly)
15. **reinterpret\_cast<>()**: reinterpret\_cast is used to convert one pointer type to another, even if they are completely unrelated. It performs a low-level conversion and is considered unsafe for regular usage.
16. int\* p = reinterpret\_cast<int\*>(0x1234); // Convert integer to pointer (unsafe)

### Example Code Demonstrating Type Conversions:

#include <iostream>

using namespace std;

int main() {

// Implicit conversion: int to double

int x = 5;

double y = x; // x is automatically converted to double

cout << "Implicit Conversion: " << y << endl;

// Explicit conversion (C-style cast): double to int

double a = 9.7;

int b = (int) a; // Manually convert double to int

cout << "Explicit Conversion (C-style cast): " << b << endl;

// Static cast

double c = 12.34;

int d = static\_cast<int>(c); // Convert double to int

cout << "Static cast: " << d << endl;

// Const cast (removing constness)

const int e = 10;

int\* f = const\_cast<int\*>(&e); // Removing const from a constant variable

\*f = 20; // Modifying the value (may lead to undefined behavior)

cout << "Const cast: " << \*f << endl;

// Reinterpret cast (dangerous and low-level)

int i = 123;

char\* ch = reinterpret\_cast<char\*>(&i); // Reinterpret int as char pointer

cout << "Reinterpret cast: " << \*ch << endl; // May not give meaningful result

return 0;

}

### Things to Keep in Mind:

* **Loss of Data**: When casting from a larger type to a smaller type (e.g., double to int), data loss may occur.
* **Safety**: Some casts, like reinterpret\_cast, should be used carefully because they can lead to unsafe memory access.
* **Runtime vs. Compile-Time Checks**: Some casts (like dynamic\_cast) are checked at runtime, whereas others (like static\_cast) are checked at compile time.

Type conversion allows flexibility in handling different data types, but it’s important to be aware of potential pitfalls, like data loss or undefined behavior, when performing type conversions in C++.

In C++, control statements are used to control the flow of the program based on certain conditions or loops. These statements allow you to make decisions, repeat actions, or alter the program's execution. The main types of control statements in C++ are:

**Conditional Statements (Decision Making)**

These are used to make decisions based on certain conditions.

**if Statement**: Executes a block of code if the specified condition is true.

if (condition) {

// code to be executed if condition is true

}

**if-else Statement**: Executes one block of code if the condition is true, and another block if it's false.

if (condition) {

// code to be executed if condition is true

} else {

// code to be executed if condition is false

}

**if-else if-else Statement**: Executes different blocks of code based on multiple conditions.

if (condition1) {

// code to be executed if condition1 is true

} else if (condition2) {

// code to be executed if condition2 is true

} else {

// code to be executed if none of the conditions are true

}

**switch Statement**: Used to select one of many code blocks to execute. It's generally used when you have multiple potential values for a variable.

switch (variable) {

case value1:

// code to be executed if variable == value1

break;

case value2:

// code to be executed if variable == value2

break;

default:

// code to be executed if no case matches

}

### **Looping Statements** (Repetition)

These are used to repeat a block of code multiple times.

**for Loop**: A loop that runs for a specified number of times.

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

// code to be executed repeatedly

}

**while Loop**: A loop that runs as long as the specified condition is true.

while (condition) {

// code to be executed repeatedly

}

**do-while Loop**: A loop that executes the block of code at least once, and then continues to execute as long as the condition is true.

do {

// code to be executed

} while (condition);

### **Jump Statements**

These are used to alter the control flow inside loops or conditional statements.

**break Statement**: Exits the current loop or switch statement and continues with the next statement after the loop.

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

if (i == 5) {

break; // Exit the loop when i is 5

}

}

**continue Statement**: Skips the current iteration of a loop and continues with the next iteration.

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

if (i == 5) {

continue; // Skip the iteration when i is 5

}

// code for other values of i

}

**return Statement**: Exits from a function and optionally returns a value.

int add(int a, int b) {

return a + b; // Return the sum of a and b

}

### **Exit Statement**

**exit() Function**: Ends the program immediately, regardless of where it is called. This is part of the cstdlib library.

#include <cstdlib>

if (some\_error\_condition) {

exit(1); // Exit the program with an error code

}

These control statements are fundamental in structuring your C++ programs, allowing you to make decisions, repeat tasks, and manage program flow efficiently.

**goto statement:**

-The goto allows to makes an absolute jump to another point in the program.

-This statement execution causes an unconditional jump or transfer of control from one statement to the other statement with in a program ignoring any type of nesting limitations.

-The destination is identified by a label, which is then used as an argument for the goto statement.

-A label is made of a valid identifier followed by a colon (:).

-The general form of goto statement is:

statement1;

statement2;

goto label\_name;

statement4;

label\_name: statement5;

statement6;

**Functions in CPP**

In programming, function refers to a segment that groups code to perform a specific task.

Depending on whether a function is predefined or created by programmer; there are two types of function:

1. Library Function
2. User-defined Function

Library Function:

Library functions are the built-in function in C++ programming. Programmer can use library function by invoking function directly; they don't need to write it themselves.

Example program on usage of library functions

#include <iostream>

#include <cmath>

using namespace std;

int main()

{

double number, squareRoot;

cout << "Enter a number: ";

cin >> number;

// sqrt() is a library function to calculate square root

squareRoot = sqrt(number);

cout << "Square root of " << number << " = " << squareRoot;

return 0;

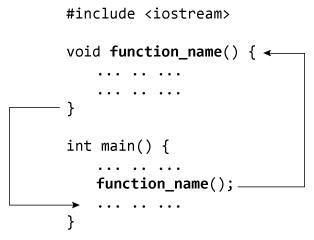
}

Enter a number: 26

Square root of 26 = 5.09902

## User-defined Function

C++ allows programmer to define their own function. A user-defined function groups code to perform a specific task and that group of code is given a name(identifier). When the function is invoked from any part of program, it all executes the codes defined in the body of function.



#include <iostream>

using namespace std;

// Function prototype (declaration)

int add(int, int);

int main()

{

int num1, num2, sum;

cout<<"Enters two numbers to add: ";

cin >> num1 >> num2;

// Function call

sum = add(num1, num2);

cout << "Sum = " << sum;

return 0;

}

// Function definition

int add(int a, int b)

{

int add;

add = a + b;

// Return statement

return add;

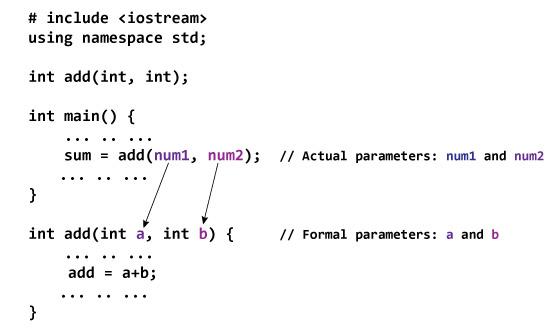
}

Passing Arguments to Function

In programming, argument (parameter) refers to the data which is passed to a function (function definition) while calling it.

In the above example, two variables, num1 and num2 are passed to function during function call. These arguments are known as actual arguments.

The value of num1 and num2 are initialized to variables a and b respectively. These arguments a and b are called formal arguments.



**Parameter passing**

There are different ways in which parameter data can be passed into and out of methods and functions. Let us assume that a function B() is called from another function A(). In this case A is called the “caller function” and B is called the “called function or callee function”. Also, the arguments which A sends to B are called actual arguments and the parameters of B are called formal arguments.

Important methods of Parameter Passing

Call By Value:

Changes made to formal parameter do not get transmitted back to the caller. Any modifications to the formal parameter variable inside the called function or method affect only the separate storage location and will not be reflected in the actual parameter in the calling environment. This method is also called as pass by value.

#include <iostream>

using namespace std;

void swap(int x, int y);

void swap(int x, int y) {

int temp;

temp = x;

x = y;

y = temp;

}

int main ()

{

int a = 100;

int b = 200;

cout << "Before swap, value of a :" << a << endl;

cout << "Before swap, value of b :" << b << endl;

swap(a, b);

cout << "After swap, value of a :" << a << endl;

cout << "After swap, value of b :" << b << endl;

return 0;

}

Call by reference

Changes made to formal parameter do get transmitted back to the caller through parameter passing. Any changes to the formal parameter are reflected in the actual parameter in the calling environment as formal parameter receives a reference (or pointer) to the actual data. This method is also called as pass by reference.

#include <iostream>

using namespace std;

void swap(int &x, int &y);

void swap(int &x, int &y) {

int temp;

temp = x;

x = y;

y = temp;

}

int main ()

{

int a = 100;

int b = 200;

cout << "Before swap, value of a :" << a << endl;

cout << "Before swap, value of b :" << b << endl;

swap(a, b);

cout << "After swap, value of a :" << a << endl;

cout << "After swap, value of b :" << b << endl;

return 0;

}

**Call by pointer**

The call by pointer method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

#include <iostream>

using namespace std;

void swap(int \*x, int \*y);

void swap(int \*x, int \*y) {

int temp;

temp = \*x;

\*x = \*y;

\*y = temp;

}

int main () {

int a = 100;

int b = 200;

cout << "Before swap, value of a :" << a << endl;

cout << "Before swap, value of b :" << b << endl;

swap(&a, &b);

cout << "After swap, value of a :" << a << endl;

cout << "After swap, value of b :" << b << endl;

return 0;

}

**Friend Function**

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

class className {

... .. ...

friend returnType functionName(arguments);

... .. ...

}

#include <iostream>

using namespace std;

class Distance {

private:

int meter;

// friend function

friend int addFive(Distance);

public:

Distance() : meter(0) {}

};

// friend function definition

int addFive(Distance d) {

//accessing private members from the friend function

d.meter += 5;

return d.meter;

}

int main() {

Distance D;

cout << "Distance: " << addFive(D);

return 0;

}

**C++ Function Overloading**

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

#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;

}

**Function templates**

Are **special functions that can operate with generic types**. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

template <typename T>

T functionName(T parameter1, T parameter2, ...)

{

// code

}

Example

int main() {

int result1;

double result2;

// calling with int parameters

result1 = add<int>(2, 3);

cout << result1 << endl;

// calling with double parameters

result2 = add<double>(2.2, 3.3);

cout << result2 << endl;

return 0;

}

**Pointers**

A **pointer** is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is −

type \*var-name;

Here, **type** is the pointer's base type; it must be a valid C++ type and **var-name** is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration –

int \*ip; // pointer to an integer

double \*dp; // pointer to a double

## Using Pointers in C++

There are few important operations, which we will do with the pointers very frequently. **(a)** We define a pointer variable. **(b)** Assign the address of a variable to a pointer. **(c)** Finally access the value at the address available in the pointer variable. This is done by using unary operator \* that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations −

#include <iostream>

using namespace std;

int main () {

int var = 20; // actual variable declaration.

int \*ip; // pointer variable

ip = &var; // store address of var in pointer variable

cout << "Value of var variable: ";

cout << var << endl;

// print the address stored in ip pointer variable

cout << "Address stored in ip variable: ";

cout << ip << endl;

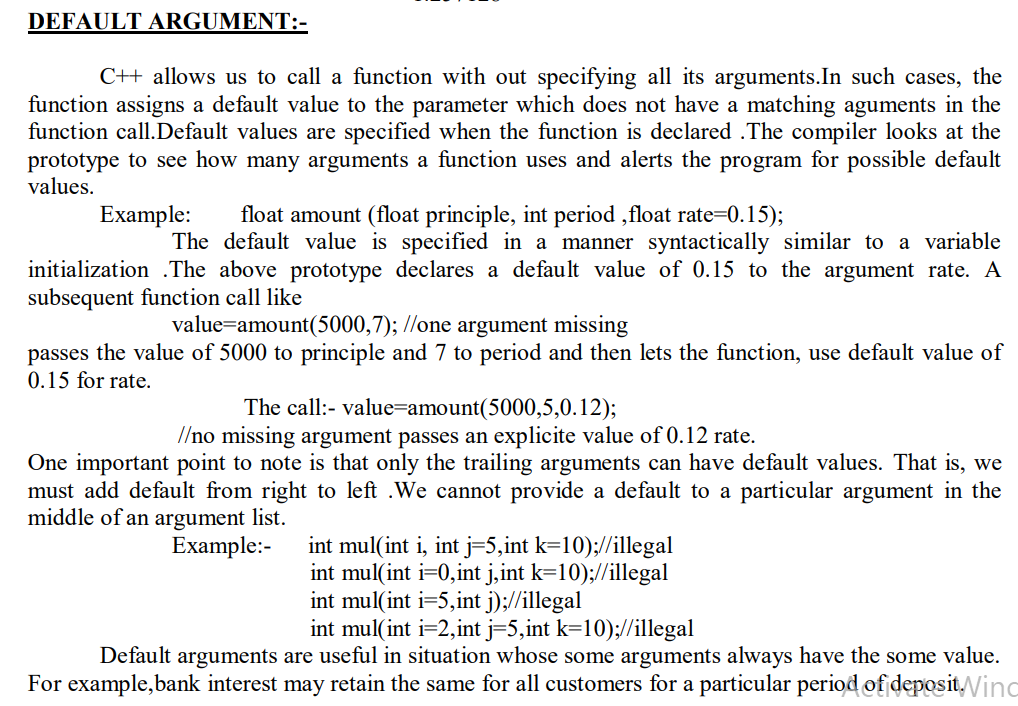
// access the value at the address available in pointer

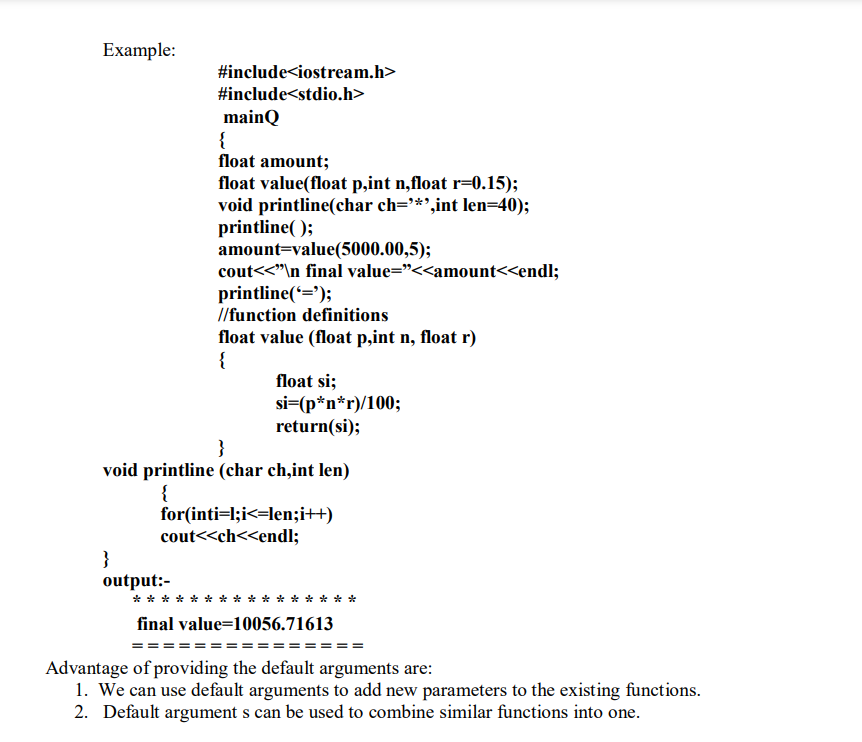
cout << "Value of \*ip variable: ";

cout << \*ip << endl;

return 0;

}



****

**Operator Overloading**

Using **operator overloading** in C++, you can specify more than one meaning for an operator in one scope. The purpose of operator overloading is to provide a special meaning of an operator for a user-defined data type.

With the help of operator overloading, you can redefine the majority of the C++ operators. You can also use operator overloading to perform different operations using one operator.

Syntax

To overload a C++ operator, you should define a special function inside the Class as follows:

class class\_name

{

... .. ...

public

return\_type operator symbol (argument(s))

{

... .. ...

}

... .. ...

};

## Can all C++ Operators be Overloaded?

No. There are C++ operators that can’t be overloaded.

They include:

* :: -Scope resolution operator
* ?: -ternary operator.
* . -member selector
* Sizeof operator
* \* -member pointer selector

Here are rules for Operator Overloading:

* For it to work, at least one operand must be a user-defined class object.
* You can only overload existing operators. You can’t overload new operators.
* Some operators cannot be overloaded using a friend function. However, such operators can be overloaded using member function.

// Overload ++ when used as prefix

#include <iostream>

using namespace std;

class Count {

private:

int value;

public:

// Constructor to initialize count to 5

Count() : value(5) {}

// Overload ++ when used as prefix

void operator ++ () {

value=value+100;

}

void display() {

cout << "Count: " << value << endl;

}

};

int main() {

Count count1;

// Call the "void operator ++ ()" function

++count1;

count1.display();

return 0;

}

**BINARY OPERATOR OVERLOADING BY CONCATENATING 2 STRINGS**

// C++ Program to concatenate two string using unary operator overloading

#include <iostream>

#include <string.h>

using namespace std;

// Class to implement operator overloading

// function for concatenating the strings

class AddString {

public:

// Classes object of string

char s1[25], s2[25];

// Parameterized Constructor

AddString(char str1[], char str2[])

{

// Initialize the string to class object

strcpy(this->s1, str1);

strcpy(this->s2, str2);

}

// Overload Operator+ to concat the string

void operator+()

{

cout << "\nConcatenation: " << strcat(s1, s2);

}

};

// Driver Code

int main()

{

// Declaring two strings

char str1[] = "Geeks";

char str2[] = "ForGeeks";

// Declaring and initializing the class

// with above two strings

AddString a1(str1, str2);

// Call operator function

+a1;

return 0;

}

´**Class Template** can also be defined similarly to the Function Template. When a class uses the concept of Template, then the class is known as generic class.

´Syntax

**template**<**class** Ttype>

**class** class\_name

{ }

´**Ttype** is a placeholder name which will be determined when the class is instantiated. We can define more than one generic data type using a comma-separated list. The Ttype can be used inside the class body.

´Now, we create an instance of a class

´class\_name<type> ob;

´**where class\_name**: It is the name of the class.

´**type**: It is the type of the data that the class is operating on.

´**ob**: It is the name of the object.

#include <iostream>

**using** **namespace** std;

**template**<**class** T>

**class** A

{

**public**:

T num1 = 5;

T num2 = 6;

**void** add()

{

std::cout << "Addition of num1 and num2 : " << num1+num2<<std::endl;

}

};

**int** main()

{

A<**int**> d;

d.add();

**return** 0;

}

**CLASS TEMPLATE WITH MULTIPLE PARAMETERS**

´We can use more than one generic data type in a class template, and each generic data type is separated by the comma.

Syntax

**template**<**class** T1, **class** T2, ......>

**class** class\_name

{

// Body of the class.

}

#include <iostream>

**using** **namespace** std;

**template**<**class** T1, **class** T2>

**class** A

{

T1 a;

T2 b;

**public**:

A(T1 x,T2 y)

{

a = x;

b = y;

}

**void** display()

{

std::cout << "Values of a and b are : " << a<<" ,"<<b<<std::endl;

}

};

**int** main()

{

A<**int**,**float**> d(5,6.5);

d.display();

**return** 0;

}

**Nontype Template Arguments**

The template can contain multiple arguments, and we can also use the non-type arguments In addition to the type T argument, we can also use other types of arguments such as strings, function names, constant expression and built-in types.

**template**<**class** T, **int** size>

**class** array

{

T arr[size]; // automatic array initialization.

};

´In the above case, the nontype template argument is size and therefore, template supplies the size of the array as an argument.

´Arguments are specified when the objects of a class are created:

´array<**int**, 15> t1; // array of 15 integers.

´array<**float**, 10> t2; // array of 10 floats.

´array<**char**, 4> t3; // array of 4 chars.

#include <iostream>

**using** **namespace** std;

**template**<**class** T, **int** size>

**class** A

{

**public**:

T arr[size];

**void** insert()

{

**int** i =1;

**for** (**int** j=0;j<size;j++)

{

arr[j] = i;

i++;

}

}

**void** display()

{

**for**(**int** i=0;i<size;i++)

{

std::cout << arr[i] << " ";

}

}

};

**int** main()

{

A<**int**,10> t1;

t1.insert();

t1.display();

**return** 0;

}

**Class**

We can think of class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object.

As, many houses can be made from the same description, we can create many objects from a class.

**How to define a class in C++?**

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.

class className

{

// some data

// some functions

};

**Example: Class in C++**

class Test

{

private:

int data1;

float data2;

public:

void function1()

{ data1 = 2; }

float function2()

{

data2 = 3.5;

return data2;

}

};

**Example: Class in C++**

class Test

{

private:

int data1;

float data2;

public:

void function1()

{ data1 = 2; }

float function2()

{

data2 = 3.5;

return data2;

}

};

## C++ Objects

To use the data and access functions defined in the class, you need to create objects.

### Syntax to Define Object in C++

className objectVariableName;

Example

class Test

{

private:

int data1;

float data2;

public:

void function1()

{ data1 = 2; }

float function2()

{

data2 = 3.5;

return data2;

}

};

int main()

{

Test o1, o2;

}

**How to access data member and member function in C++?**

You can access the data members and member functions by using a . (dot) operator. For example,

o2.function1();

This will call the function1() function inside the Test class for objects o2.

**Access Specifiers;**

In C++, there are three access specifiers:

public - members are accessible from outside the class

private - members cannot be accessed (or viewed) from outside the class

protected - members cannot be accessed from outside the class, however, they can be accessed in inherited classes.

1. Public Access Specifier

This keyword is used to declare the functions and variables public, and any part of the entire program can access it. The members and member methods declared public can be accessed by other classes and functions. The public members of a class can be accessed from anywhere in the program using the (.) with the object of that class.

2. Private Access Specifiers

The private keyword is used to create private variables or private functions. The private members can only be accessed from within the class. Only the member functions or the friend functions are allowed to access the private data of a class or the methods of a class.

Protected Access Specifiers

The protected keyword is used to create protected variables or protected functions. The protected members can be accessed within and from the derived/child class.

**C++ Classes and Objects**

**What is a Class?**

A class is a user-defined data type that acts as a blueprint for creating objects. It encapsulates data and methods into a single entity, providing a foundation for object-oriented programming.

**Class Definition Syntax**

class ClassName {

private: // Access specifier: Members are accessible only within the class

// Private data members

public: // Access specifier: Members are accessible outside the class

// Public data members and methods

// Member function

void display() {

cout<< "This is a class!" <<endl;

}

};

**Structure of a Class**

1. **Access Specifiers**
   * **private:** Members are accessible only within the class (default).
   * **public:** Members are accessible from outside the class.
   * **protected:** Members are accessible within the class and its derived classes.
2. **Data Members**
   * Variables that hold the data associated with the class.
3. **Member Functions**
   * Functions that operate on data members and define the behavior of the class.

**Creating and Using Objects**

**Example:**

#include <iostream>

using namespace std;

class Person {

private:

string name; int age;

public: // Method to set data

void setDetails(string personName, int personAge) {

name = personName;

age = personAge;

}

// Method to display data

void displayDetails() {

cout<< "Name: " << name << ", Age: " << age <<endl;

}

};

int main() {

Person person1; // Create an object of class Person

person1.setDetails("Alice", 25); // Set details for person1

person1.displayDetails(); // Display details for person1

return 0;

}

**Features of Classes and Objects in C++**

1. **Encapsulation**

Combines data (attributes) and functions (methods) into a single unit.

1. **Abstraction**

Hides the internal details and only exposes essential functionality.

1. **Inheritance**

Enables one class (derived class) to inherit properties and behaviors from another class (base class).

1. **Polymorphism**

Allows functions to behave differently based on the object that invokes them.

1. **Reusability**

Classes can be reused in multiple programs, improving modularity and maintainability.

**Structures in C++**

**1. Introduction to Structures**

A structure in C++ is a user-defined data type that allows grouping variables of different types under one name. Structures help in organizing data logically, making programs more modular and manageable.

**Syntax:**

struct StructureName {

data\_type member1;

data\_type member2;

// ... more members

};

**Example:**

struct Point {

int x;

int y;

};

int main() {

Point p1;

p1.x = 10;

p1.y = 20;

cout<< "Point: (" << p1.x << ", " << p1.y << ")";

return 0;

}

**2. Features of Structures**

1. **Grouping Variables:** Structures group variables of different types into a single unit.
2. **Access Using Dot Operator:** Structure members are accessed using the dot (.) operator.
3. **Memory Allocation:** Memory is allocated separately for each structure object.
4. **Structures in Functions:** Structures can be passed as arguments to functions.
5. **Nested Structures:** Structures can be nested within other structures.

**3. Declaration and Initialization**

**Structure Declaration:**

struct Person {

string name;

int age;

float height;

};

**Structure Initialization:**

Person p1 = {"Alice", 25, 5.6};

**Accessing Members:**

cout<< "Name: " << p1.name <<endl;

cout<< "Age: " << p1.age <<endl;

cout<< "Height: " << p1.height<<endl;

**4. Array of Structures**

You can define an array of structures to store multiple objects of the same type.

**Example:**

struct Student {

string name;

int rollNo;

float marks;

};

int main() {

Student students[3] = {

{"Alice", 1, 89.5},

{"Bob", 2, 76.3},

{"Carol", 3, 91.0}

};

for (int i = 0; i< 3; i++) {

cout<< students[i].name << " " << students[i].rollNo<< " " << students[i].marks <<endl;

}

return 0;

}

**5. Nested Structures**

Structures can contain other structures as members.

**Example:**

struct Date {

int day;

int month;

int year;

};

struct Employee {

string name;

int id;

Date joinDate;

};

int main() {

Employee e = {"Alice", 101, {15, 8, 2020}};

cout<< "Name: " << e.name <<endl;

cout<< "Join Date: " <<e.joinDate.day<< "/" <<e.joinDate.month<< "/" <<e.joinDate.year;

return 0;

}

**6. Passing Structures to Functions**

Structures can be passed to functions by value or by reference.

**By Value:**

void display(Point p) {

cout<< "Point: (" <<p.x<< ", " <<p.y<< ")";

}

int main() {

Point p = {10, 20};

display(p);

return 0;

}

**By Reference:**

void update(Point &p) {

p.x += 10;

p.y += 10;

}

int main() {

Point p = {10, 20};

update(p);

cout<< "Updated Point: (" <<p.x<< ", " <<p.y<< ")";

return 0;

}

**7. Structure with Pointers**

You can use pointers to structures to dynamically allocate memory or access structure members.

**Example:**

struct Circle {

float radius;

};

int main() {

Circle \*c = new Circle;

c->radius = 5.5;

cout<< "Radius: " << c->radius;

delete c;

return 0;

}

**8. Structures with Functions**

Structures can have member functions, though using a class is generally preferred.

**Example:**

struct Rectangle {

float length, breadth;

float area() {

return length \* breadth;

}

};

int main() {

Rectangle r = {5, 3};

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

return 0;

}

**9. Structure Comparison**

To compare two structures, you need to compare their members individually.

**Example:**

struct Point {

int x, y;

};

bool isEqual(Point p1, Point p2) {

return (p1.x == p2.x && p1.y == p2.y);

}

int main() {

Point p1 = {10, 20}, p2 = {10, 20};

if (isEqual(p1, p2))

cout<< "Points are equal";

else

cout<< "Points are not equal";

return 0;

}

**10. Difference Between Structure and Class**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Structure** | **Class** |
| **Default Access** | Members are public by default. | Members are private by default. |
| **Inheritance** | Cannot inherit from a structure. | Supports inheritance. |
| **Object-Oriented** | Limited OOP features. | Fully supports OOP. |

**11. Common Errors with Structures**

1. **Uninitialized Members:** Members not explicitly initialized may contain garbage values.
2. **Accessing Non-Existent Members:** Attempting to access a member not declared in the structure.
3. **Using Assignment Without Deep Copy:** For structures containing pointers, assignment may lead to shallow copies and dangling pointers.

**Example:**

struct Example {

int \*ptr;

};

int main() {

Example e1, e2;

e1.ptr = new int(5);

e2 = e1; // Shallow copy

delete e1.ptr;

cout<< \*(e2.ptr); // Dangling pointer access

return 0;}

**12. Applications of Structures**

1. Storing and managing related data (e.g., records in databases).
2. Representing geometric objects (e.g., points, rectangles).
3. Managing dynamic collections using pointers.
4. Building user-defined types for modular programming.

**this pointer**

is a special pointer that points to the current object of a class. It is an implicit pointer available in all non-static member functions. It is used to refer to the current instance of the class within the class’s member functions. The this pointer is particularly useful when you need to differentiate between member variables and function parameters that have the same name.

### **Key Points about this Pointer:**

* **Type of this Pointer**:  
  The type of the this pointer is pointer to the class type. For example, if the class is class MyClass, the type of this would be MyClass\* in a non-const member function.
* **Usage**:  
  It can be used to access the members of the current object and to distinguish between instance variables and local variables or parameters with the same name.
* **Only Available in Non-Static Member Functions**:  
  You cannot use the this pointer in static member functions because static member functions do not operate on a specific instance of the class.
* **this is a const pointer** in constant member functions:  
  In constant member functions, this is treated as a pointer to const (i.e., const MyClass\* this), so you cannot modify the object's non-mutable members.

### Syntax of this Pointer:

* this is implicitly passed to all non-static member functions.
* You can explicitly use this to refer to members of the current object.

### Example of Using this Pointer:

#include <iostream>

using namespace std;

class MyClass {

private:

int x;

public:

MyClass(int x) {

this->x = x; // Using `this` pointer to differentiate between member variable and constructor parameter

}

void show() {

cout << "Value of x: " << this->x << endl; // Accessing member variable using `this`

}

void setX(int x) {

this->x = x; // Using `this` pointer to set member variable

}

int getX() {

return this->x; // Accessing the member variable

}

};

int main() {

MyClass obj(10);

obj.show(); // Output: Value of x: 10

obj.setX(20);

obj.show(); // Output: Value of x: 20

return 0;

}

### Explanation of the Example:

* In the constructor MyClass(int x), there is a parameter x and a member variable x. The this pointer is used to differentiate between the member variable and the parameter.
  + this->x = x; refers to the member variable, while x refers to the parameter passed to the constructor.
* In the show() and getX() functions, the this pointer is used to access the member variable x.

### Example with const Member Function:

In constant member functions, this is a pointer to const, meaning you cannot modify the object:

class MyClass {

private:

int x;

public:

MyClass(int x) : x(x) {}

// Constant member function

void show() const {

// this->x = 20; // Error: can't modify a const object

cout << "Value of x: " << this->x << endl;

}

};

int main() {

MyClass obj(10);

obj.show(); // Output: Value of x: 10

return 0;

}

### Summary of Key Points:

* this is a pointer to the current object within a class's non-static member functions.
* It is used to distinguish between member variables and function parameters or local variables with the same name.
* this is a const pointer in constant member functions (const MyClass\* this).
* The this pointer is implicit, but you can explicitly use it if needed.

This pointer is a valuable feature for working with object-oriented code in C++ and allows you to write cleaner and more readable code when dealing with instance members.

**Constructors**

In C++, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C++ has the same name as class.

There can be three types of constructors in C++.

Default constructor

Parameterized constructor

Copy constructor

**C++ Default Constructor**

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

#include <iostream>

using namespace std;

class Employee

{

public:

Employee()

{

cout<<"Default Constructor"<<endl;

}

};

int main()

{

Employee e1; //creating an object of Employee

Employee e2;

return 0;

}

**C++ Parameterized Constructor**

A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

#include <iostream>

using namespace std;

class A {

private:

int num1, num2 ;

public:

A(int n1, int n2) {

num1 = n1;

num2 = n2;

}

void display() {

cout<<"num1 = "<< num1 <<endl;

cout<<"num2 = "<< num2 <<endl;

}

};

int main() {

A obj(3,8);

obj.display();

return 0;

}

**Copy Constructor**

In the C++ programming language, a copy constructor is a special constructor for creating a new object as a copy of an existing object. Copy constructors are the standard way of copying objects in C++,

Syntax of Copy Constructor

Classname (classname & objectname)

{

. . . .

}

#include<iostream>

using namespace std;

class Samplecopyconstructor

{

private:

int x, y; //data members

public:

Samplecopyconstructor(int x1, int y1)

{

x = x1;

y = y1;

}

/\* Copy constructor \*/

Samplecopyconstructor (Samplecopyconstructor &sam)

{

x = sam.x;

y = sam.y;

}

void display()

{

cout<<x<<" "<<y<<endl;

}

};

/\* main function \*/

int main()

{

Samplecopyconstructor obj1(10, 15); // Normal constructor

Samplecopyconstructor obj2 = obj1; // Copy constructor

cout<<"Normal constructor : ";

obj1.display();

cout<<"Copy constructor : ";

obj2.display();

return 0;

}

**Special characteristics of Constructors:**

* **They should be declared in the public section**
* **They do not have any return type, not even void**
* **They get automatically invoked when the objects are created**
* **They cannot be inherited through derived class can call the base class constructor**
* **Like other functions, they can have default arguments**
* **You cannot refer to their address**
* **Constructors cannot be virtual**

### **Constructor Overloading**

Just like other member functions, constructors can also be overloaded. Infact when you have both default and parameterized constructors defined in your class you are having Overloaded Constructors, one with no parameter and other with parameter.

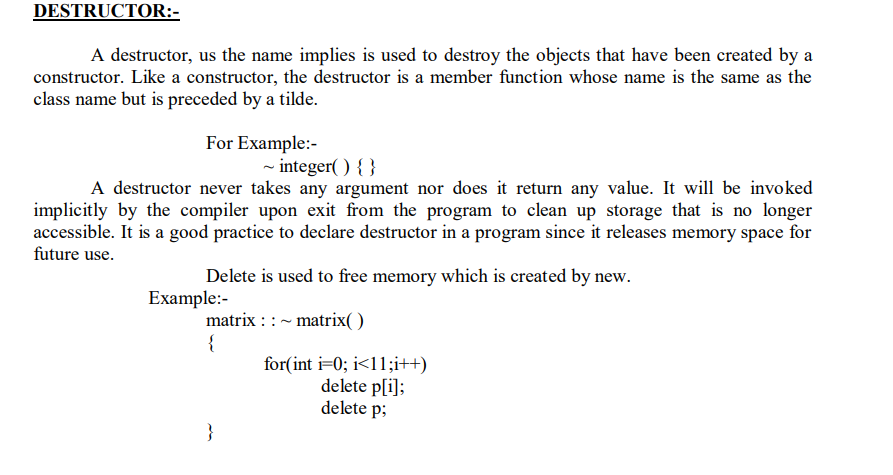
You can have any number of Constructors in a class that differ in parameter list.

**classStudent  
{  
 introllno;  
 stringname;  
 public:  
 Student(intx)  
 {  
 rollno=x;  
 name="None";  
 }  
 Student(intx,stringstr)  
 {  
 rollno=x;  
 name=str;  
 }  
};  
  
intmain()  
{  
 StudentA(10);  
 StudentB(11,"Ram");  
}**

In above case we have defined two constructors with different parameters, hence overloading the constructors.

One more important thing, if you define any constructor explicitly, then the compiler will not provide default constructor and you will have to define it yourself.

In the above case if we write Student S; in **main()**, it will lead to a compile time error, because we haven't defined default constructor, and compiler will not provide its default constructor because we have defined other parameterized constructors.



### **Destructors**

Destructor is a special class function which destroys the object as soon as the scope of object ends. The destructor is called automatically by the compiler when the object goes out of scope.

The syntax for destructor is same as that for the constructor, the class name is used for the name of destructor, with a **tilde** ~ sign as prefix to it.

**class A  
{  
 public:  
 ~A();  
};**

Destructors will never have any arguments.

#### Example to see how Constructor and Destructor is called

**class A  
{  
A()  
 {  
 cout << "Constructor called";  
 }  
  
~A()  
 {  
 cout << "Destructor called";  
 }  
};  
  
int main()  
{  
 A obj1; // Constructor Called  
 int x=1  
 if(x)  
 {  
 A obj2; // Constructor Called  
 } // Destructor Called for obj2  
} // Destructor called for obj1**

#### Single Definition for both Default and Parameterized Constructor

In this example we will use **default argument** to have a single definition for both defualt and parameterized constructor.

**class Dual  
{  
 int a;  
 public:  
 Dual(int x=0)  
 {  
 a=x;  
 }  
};  
  
int main()  
{  
 Dual obj1;  
 Dual obj2(10);  
}**

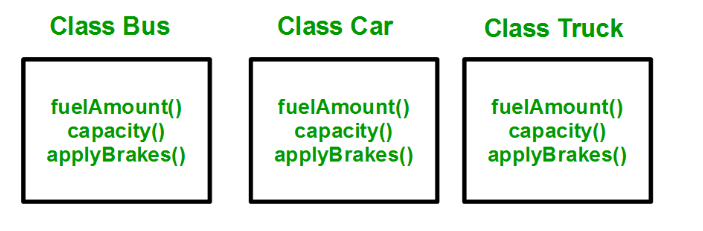
Here, in this program, a single Constructor definition will take care for both these object initializations. We don't need separate default and parameterized constructors.

Inheritance and Polymorphism, Exception Handling, throwing an exception, the try block, catching an exception, exception objects, exception specifications, catching all exceptions, Introduction to linked lists, stacks, queues and applications of stacks.

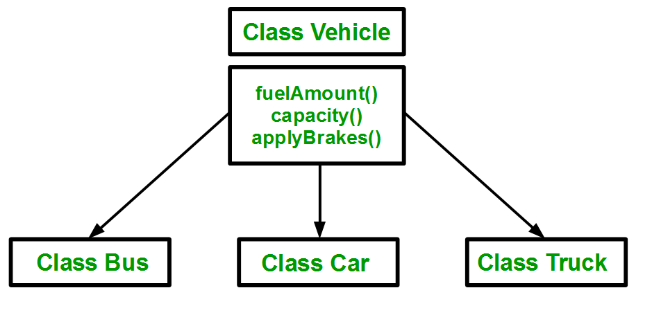
# Inheritance

C++ strongly supports the concept of reusability. The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adopted by another programmers. This is basically created by defining the new classes, reusing the properties of existing ones. The mechanism of deriving a new class from an old one is called 'INHERITENCE'. This is often referred to as IS-A' relationship because every object of the class being defined "is" also an object of inherited class. The old class is called 'BASE' class or ‘PARENT’ class and the new class is called ‘DERIEVED’ class or ‘CHILD’ class.

The capability of a [class](https://www.geeksforgeeks.org/c-classes-and-objects/)to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.  When we say derived class inherits the base class, it means, the derived class inherits all the properties of the base class, without changing the properties of base class and may add new features to its own. These new features in the derived class will not affect the base class. The derived class is the specialized class for the base class.



The above process results in duplication of the same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability.



Using inheritance, we have to write the functions only one time instead of three times as we have inherited the rest of the three classes from the base class (Vehicle).

**Implementing inheritance in C++**: For creating a **sub-class(child class/Derived class)** that is inherited from the base class we have to follow the below syntax.

**Derived Classes:**

A Derived class is defined as the class derived from the base class.

**Syntax**:

**class <derived\_class\_name> : <access-specifier> <base\_class\_name>**

{

**// members of derived class**

}

Where  
**class**  — keyword to create a new class  
**derived\_class\_name**   — name of the new class, which will inherit the base class  
**access-specifier**— either of private, public or protected. If neither is specified, **PRIVATE is taken as default**  
**base-class-name**  — name of the base class/arentclass/superclass

**Note**: **A derived class doesn’t inherit *access* to private data members.**

**However, it does inherit a full parent object, which contains any private members which that class declares.**

**The colon :** indicates that the a-class name is derived from the base class name. **The access specifier or the visibility mode is optional and, if present, may be public, private or protected. By default it is private.** Visibility mode describes the status of derived features e.g.

**Example:**

**class XYZ //base class**

**{**

**members of XYZ**

**};**

**class ABC : public XYZ //public derivation**

**{**

**members of ABC**

**};**

**class ABC : XYZ //private derivation (by default)**

**{**

**members of ABC**

**};**

**In the inheritance, some of the base class data elements and member functions are inherited into the derived class. We can add our own data and member functions and thus extend the functionality of the base class.**

Inheritance, when used to modify and extend the capabilities of the existing classes, becomes a very powerful tool for incremental program development.

**Following are the examples of Inheritance:**

**Example 1**

#include<iostream>

using namespace std;

**class Vehicle** **// Base class**

{  
**public:**  
    string brand = "Ford";  
    void honk()

{  
      cout << “honk honk ! \n" ;  
    }  
};

**class Car: public Vehicle**  **// Derived class**

{   
  **public:**  
    string model = "Mustang";  
};

int main()

{  
  Car myCar;  
  myCar.honk();  
  cout << myCar.brand + " " +myCar.model;  
  return 0;  
}

**Output:**

Honk honk!

Ford Mustang

**Example 2: Define member function without argument within the class**

 #include<iostream>

**using** **namespace** std;

**class Person //parentclass**

{

**int** id;

**char** name[100];

**public**:

**void set\_p()**

        {

            cout<<"Enter the Id:";

            cin>>id;

            cout<<"Enter the Name:";

            cin.get(name,100);

        }

**void** display\_p()

        {

   cout<<endl<<id<<"\t"<<name<<"\t";

        }

};

**class** Student: **private** Person

{

**//childclass**

**char** course[50];

**int** fee;

**public**:

**void** set\_s()

        {

            set\_p();

            cout<<"Enter the Course Name:";

            cin.getline(course,50);

            cout<<"Enter the Course Fee:";

            cin>>fee;

        }

**void** display\_s()

        {

            display\_p();

            cout<<course<<"\t"<<fee<<endl;

        }

};

int main()

{

**Student s;**

    s.set\_s();

    s.display\_s();

**return** 0;

}

**Output:**

Enter the Id: 101

Enter the Name: Dev

Enter the Course Name: GCS

Enter the Course Fee:70000

101 Dev GCS 70000

**Example 3: C++ program to demonstrate implementation of Inheritance**

  #include <iostream>

**using** **namespace** std;

 // Base class

**class** Parent

{

**public**:

**int** id\_p;

};

**// Sub class inheriting from Base Class(Parent)**

**class** Child : **public** Parent

{

**public**:

**int** id\_c;

};

// main function

**int** main()

{

    Child obj1;

**// An object of class child has all data members and member //functions of class parent**

    obj1.id\_c = 7;

    obj1.id\_p = 91;

    cout << "Child id is: " << obj1.id\_c << '\n';

    cout << "Parent id is: " << obj1.id\_p << '\n';

**return** 0;

}

**Output:**

Child id is : 7

Parent id is: 91

In the above program, the ‘Child’ class is publicly inherited from the ‘Parent’ class so the public data members of the class ‘Parent’ will also be inherited by the class ‘Child’.

# Modes of Inheritance:

There are 3 modes of inheritance.

**Public Mode**: If we derive a subclass from a public base class. **Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in the derived class.**

**Protected Mode:** If we derive a subclass from a Protected base class. **Then both public members and protected members of the base class will become protected in the derived class.**

**Private Mode**: If we derive a subclass from a Private base class. **Then both public members and protected members of the base class will become Private in the derived class.**

**An access declaration takes this general form:**

***base-class::member*;**

**Example: When a base class is inherited private:**

**class base**

**{**

**public:**

**int j; // public in base**

**};**

**// Inherit base as private.**

**class derived: private base**

**{**

**public:**

**// here is access declaration**

**base::j; // make j public again**

**.**

**.**

**.**

**};**

Because **base** is inherited as **private** by **derived**, the public member **j** is made a private member of **derived**. However, by including

**base::j;**

as the access declaration under **derived**'s **public** heading, **j** is restored to its public status.

**Example: C++ Implementation to show that a derived class doesn’t inherit access to private data members. However, it does inherit a full parent object.**

**class** A

{

**public**:

**int** x;

**protected**:

**int** y;

**private**:

**int** z;

};

**class** B : **public** A

{

    // x is public

    // y is protected

    // z is not accessible from B

};

**class** C : **protected** A

{

    // x is protected

    // y is protected

    // z is not accessible from C

};

**class** D : **private** A // 'private' is default for classes

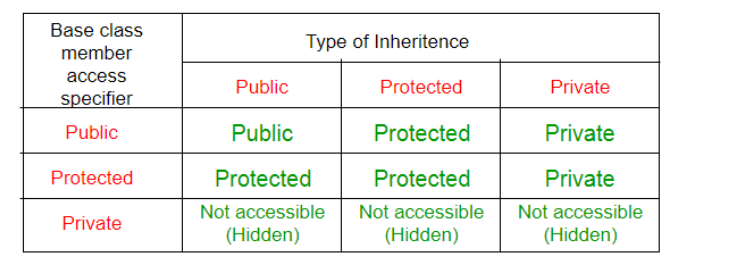
{

    // x is private

    // y is private

    // z is not accessible from D

};



**Making a Private Member Inheritable**

Basically we have visibility modes to specify that in which mode you are deriving the another class from the already existing base class. They are:

**Private:** when a base class is privately inherited by a derived class, 'public members' of the base class become private members of the derived class and therefore the public members of the base class can be accessed by its own objects using the dot operator. The result is that we have no member of base class that is accessible to the objects of the derived class.

b. **Public:** On the other hand, when the base class is publicly inherited, 'public members' of the base class become 'public members' of derived class and therefore they are accessible to the objects of the derived class.

c. **Protected:** C++ provides a third visibility modifier, protected, which serve a little purpose in the inheritance. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it. It cannot be accessed by functions outside these two classes.

**The private and protected members of a class can be accessed by:**

a. A function i.e. friend of a class.

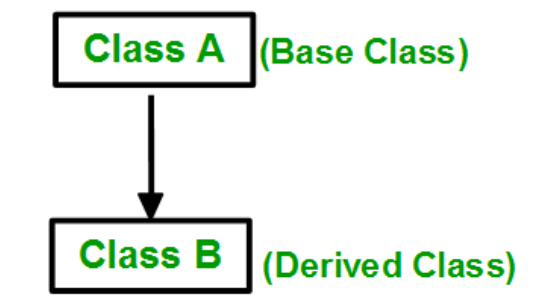
b. A member function of a class that is the friend of the class.

c. A member function of a derived class.

**Types of Inheritance in C++:-**

* Single inheritance
* Multilevel inheritance
* Multiple inheritance
* Hierarchical inheritance
* Hybrid inheritance

1. **Single Inheritance**: In single inheritance, a class is allowed to inherit from only one class. i.e. one subclass is inherited by one base class only.



**Syntax for Single Inheritance**:

**class subclass\_name : access\_mode base\_class**

{ // body of subclass

};

**Following are the examples to implement Single Inheritance:**

**Example 1:**

#include<iostream>

using namespace std;

**class Vehicle // base class**

{

public:

Vehicle()

{

cout << "This is a Vehicle\n";

}

};

**// sub class derived from a single base classes**

**class Car : public Vehicle**

{

};

**int main()** **// main function**

{

// Creating object of sub class will

// invoke the constructor of base classes

Car obj;

return 0;

}

**Output** This is a Vehicle

**Example 2:**

**#include<iostream.h>**

**#include<conio.h>**

**class worker**

**{**

**int age;**

**char name [10];**

**public:**

**void get ( );**

**void show( );**

**};**

**void worker : : get ( )**

**{**

**cout <<”yout name please”**

**cin >> name;**

**cout <<”your age please” ;**

**cin >> age;**

**}**

**void worker :: show ( )**

**{**

**cout <<“My name is :”<<name<< endl;**

**cout <<“My age is:”<< age<<endl;**

**}**

**class manager : public worker //derived class (publicly)**

**{**

**int n;**

**public:**

**void get ( ) ;**

**void show ( ) ;**

**};**

**void manager : : get ( )**

**{**

**worker : : get ( ) ; //the calling of base class input function**

**cout << “number of workers under you”;**

**cin >> n;**

**}**

**//( if they were public )**

**void manager :: show ( )**

**{**

**worker :: show ( ); //calling of base class o/p fn.**

**cout <<“in No. of workers under me are: “ << n;**

**}**

**int main ( )**

**{**

**clrscr ( ) ;**

**worker W1;**

**manager M1;**

**M1 .get ( );**

**M1.show ( ) ;**

**}**

**If you input the following to this program:**

**Your name please**

**Ravinder**

**Your age please**

**27**

**number of workers under you**

**30**

**Then the output will be as follows:**

**My name is : Ravinder**

**My age is : 27**

**No. of workers under me are : 30**

**Example 3:**

**The following program shows the single inheritance by private derivation.**

**#include<iostream.h>**

**#include<conio.h>**

**class worker //Base class declaration**

**{**

**int age;**

**char name [10] ;**

**public:**

**void get ( ) ;**

**void show ( ) ;**

**};**

**void worker : : get ( )**

**{**

**cout << “your name please” ;**

**cin >> name;**

**cout << “your age please”;**

**cin >>age;**

**}**

**void worker : show ( )**

**{**

**cout << “in my name is: “ <<name<< “in” << “my age is : “ <<age;**

**}**

**class manager : worker //Derived class (privately by default)**

**{**

**int now;**

**public:**

**void get ( ) ;**

**void show ( ) ;**

**};**

**void manager : : get ( )**

**{**

**worker : : get ( ); //calling the get function of base**

**cout << “number of worker under you”; class which is**

**cin >> now;**

**}**

**void manager : : show ( )**

**{**

**worker : : show ( ) ;**

**cout << “in no. of worker under me are : “ <<now;**

**}**

**main ( )**

**{**

**clrscr ( ) ;**

**worker wl ;**

**manager ml;**

**ml.get ( ) ;**

**ml.show ( );**

**}**

**Example 4:**

**The following program shows the single inheritance using protected derivation**

**#include<conio.h>**

**#include<iostream.h>**

**class worker //Base class declaration**

**{**

**protected:**

**int age; char name [20];**

**public:**

**void get ( );**

**void show ( );**

**};**

**void worker :: get ( )**

**{**

**cout >> “your name please”;**

**cin >> name;**

**cout << “your age please”;**

**cin >> age;**

**}**

**void worker :: show ( )**

**{**

**cout << “in my name is: “ << name << “in my age is “ <<age;**

**}**

**class manager:: protected worker // protected inheritance**

**{**

**int now;**

**public:**

**void get ( );**

**void show ( ) ;**

**};**

**void manager : : get ( )**

**{**

**cout << “please enter the name In”;**

**cin >> name;**

**cout<< “please enter the age In”; //Directly inputting the data**

**cin >> age; members of base class**

**cout << “ please enter the no. of workers under you:”;**

**cin >> now;**

**}**

**void manager : : show ( )**

**{**

**cout « "your name is : "«name«" and age is : "«age; cout «"In no. of workers under your are : "«now;**

**main ( )**

**{**

**clrscr ( ) ;**

**manager ml;**

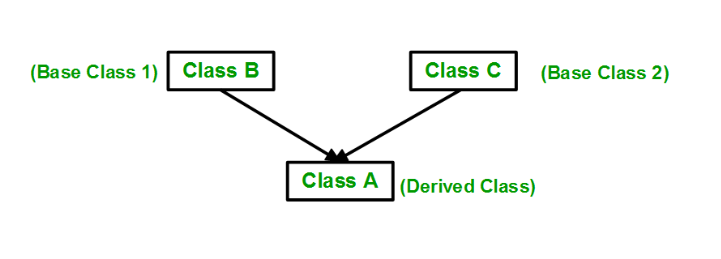
**ml.get ( ) ;**

**cout « "\n \n";**

**ml.show ( );**

**}**

1. **Multiple Inheritance:** Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e one **subclass** is inherited from more than one **base class**.



**Syntax**:

**class subclass\_name : access\_mode base\_class1, access\_mode base\_class2, ....**

{

// body of subclass

};

**Example:**

class B

{

... .. ...

};

class C

{

... .. ...

};

class A: public B, public C

{

... ... ...

};

Here, the number of base classes will be separated by a comma (‘, ‘) and the access mode for every base class must be specified.

**Example to demonstrate multiple inheritance:**

#include <iostream>

**using** **namespace** std;

**class Vehicle // first base class**

{

**public**:

**Vehicle()**

{

cout << "This is a Vehicle\n";

}

};

**class FourWheeler // second base class**

{

**public**:

    FourWheeler()

    {

 cout << "This is a 4 wheeler Vehicle\n";

    }

};

// sub class derived from two base classes

**class Car** : **public Vehicle,public FourWheeler**

{

};

**int main()** // main function

{

    // Creating object of sub class will

    // invoke the constructor of base classes.

**Car obj;**

**return** 0;

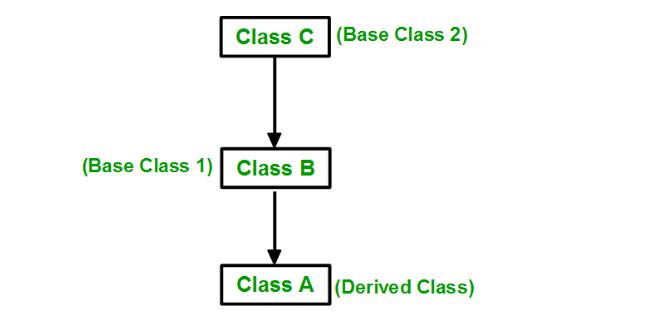
}

**Output**

This is a Vehicle

This is a 4 wheeler Vehicle

1. **Multilevel Inheritance**: In this type of inheritance, a derived class is created from another derived class.



**Syntax:**

class C

{

... .. ...

};

class B:public C

{

... .. ...

};

class A: public B

{

... ... ...

};

**Example1 to demonstrate Multilevel Inheritance:**

#include<iostream>

using namespace std;

**class A**

{

**protected:**

int a;

**public:**

**void set\_A()**

{

cout<<"Enter the Value of A=";

cin>>a;

}

**void disp\_A()**

{

cout<<endl<<"Value of A="<<a;

}

};

**class B: public A**

{

**protected:**

int b;

**public:**

**void set\_B( )**

{

cout<<"Enter the Value of B=";

cin>>b;

}

**void disp\_B( )**

{

cout<<endl<<"Value of B="<<b;

}

};

**class C: public B**

{

int c,p;

**public:**

**void set\_C()**

{

cout<<"Enter the Value of C=";

cin>>c;

}

**void disp\_C()**

{

cout<<endl<<"Value of C="<<c;

}

**void cal\_product()**

{

p=a\*b\*c;

**cout<<endl<<"Product of "<<a<<" \* "<<b**

**<<" \* "<<c<<" = “<<p;**

}

};

int main()

{

**C objc;**

**objc.set\_A();**

**objc.set\_B();**

**objc.set\_C();**

**objc.disp\_A();**

**objc.disp\_B();**

**objc.disp\_C();**

**objc.cal\_product();**

return 0;

}

**Example 2 to implement Multilevel Inheritance:**

#include <iostream>

using namespace std;

**class Vehicle**   **// base class**

{

**public:**

**Vehicle()**

{

cout << "This is a Vehicle\n";

}

};

// first sub\_class derived from class vehicle

**class fourWheeler : public Vehicle**

{

**public:**

**fourWheeler()**

{

cout << "Objects with 4 wheels are vehicles\n";

}

};

**// sub class derived from the derived base class fourWheeler**

**class Car : public fourWheeler**

{

**public:**

**Car()**

{

cout << "Car has 4 Wheels\n";

}

};

**int main()** **// main function**

{

**Car obj;** // Creating object of sub class will // invoke the constructor of //base classes first and then its own class.

return 0;

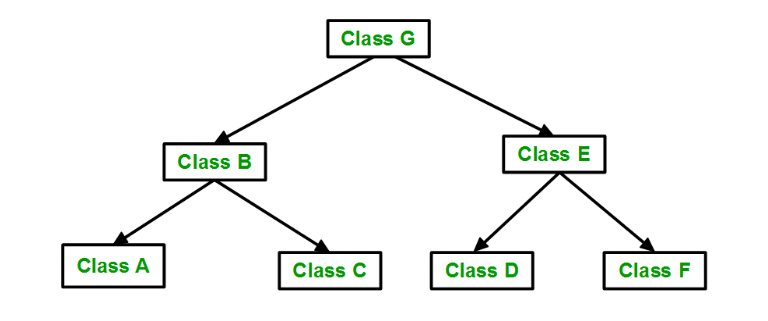
}

**Output: This is a Vehicle**

**Objects with 4 wheels are vehicles**

**Car has 4 Wheels**

1. **Hierarchical Inheritance**: In this type of inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.



**Syntax:**

**class A**

**{**

**// body of the class A.**

**}**

**class B : public A**

**{**

**// body of class B.**

**}**

**class C : public A**

**{**

**// body of class C.**

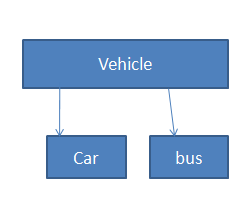
**}**

**class D : public A**

**{**

**// body of class D.**

**}**

**Example to implement Hierarchical Inheritance: **

#include <iostream>

using namespace std;

**class Vehicle // base class**

{

**public:**

**Vehicle()**

{

cout << "This is a Vehicle\n";

}

};

**class Car : public Vehicle // first sub class**

{

};

**class Bus : public Vehicle // second sub class**

{

};

**int main()** **// main function**

{

**// Creating object of sub class will**

**// invoke the constructor of base class**.

Car obj1;

Bus obj2;

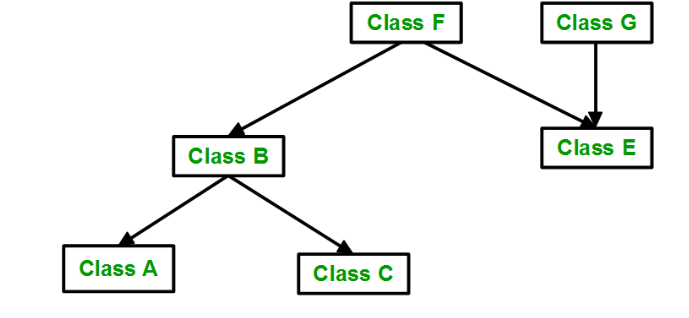
return 0;

}

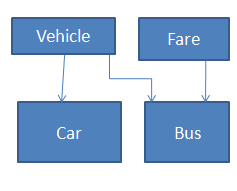
**Output: This is a Vehicle**

**This is a Vehicle**

1. **Hybrid (Virtual) Inheritance**: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.   
   Below image shows the combination of hierarchical and multiple inheritances:



**Example 1 to implement Hybrid Inheritance:**

****

#include <iostream>

using namespace std;

**class Vehicle // base class**

{

**public:**

Vehicle()

{

cout << "This is a Vehicle\n";

}

};

**class Fare // base class**

{

**public:**

Fare()

{

cout << "Fare of Vehicle\n";

}

};

**class Car : public Vehicle // first sub class**

{

};

**class Bus : public Vehicle, public Fare // second sub class**

{

};

**int main()** **// main function**

{

**// Creating object of sub class will**

**// invoke the constructor of base class.**

**Bus obj2;**

**return 0;**

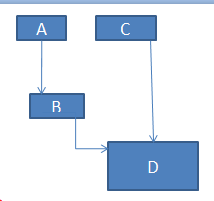
}

**Output**

This is a Vehicle

Fare of Vehicle

**Example 2 to demonstrate Hybrid Inheritance:**



#include <iostream>

using namespace std;

**class A**

{

**protected:**

int a;

**public:**

**void get\_a()**

{

cout << "Enter the value of 'a' : ";

cin>>a;

}

};

**class B : public A**

{

**protected:**

int b;

**public:**

**void get\_b()**

{

cout << "Enter the value of 'b' : ";

cin>>b;

}

};

**class C**

{

**protected:**

int c;

**public:**

**void get\_c()**

{

cout << "Enter the value of c is : ";

cin>>c;

}

};

**class D : public B, public C**

{

**protected:**

int d;

**public:**

**void mul()**

{

get\_a();

get\_b();

get\_c();

cout << "Multiplication of a,b,c is : " <<a\*b\*c;

}

};

**int main()**

{

D objd;

objd.mul();

return 0;

}

**Output:**

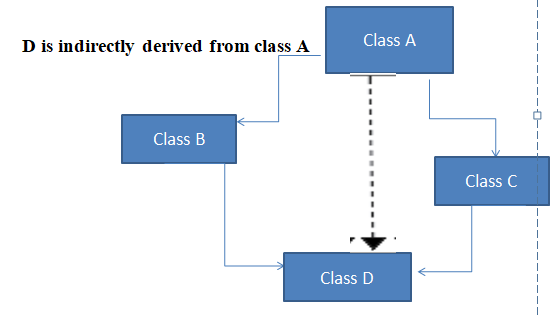
Enter the value of a: 10

Enter the value of b:5

Enter the value of c: 2

Multiplication of a,b,c is:100

1. **A special case of hybrid inheritance: Multipath inheritance**:   
   A derived class with two base classes and these two base classes have one common base class is called multipath inheritance. This type of inheritance involves other inheritance like multiple, multilevel, hierarchical etc. Ambiguity can arise in this type of inheritance.



**Example 1to implement Multipath Inheritance :**

#include <iostream>

using namespace std;

**class ClassA**

{

**public:**

int a;

};

**class ClassB : public ClassA**

{

**public:**

int b;

};

**class ClassC : public ClassA**

{

**public:**

int c;

};

**class ClassD : public ClassB, public ClassC**

{

**public:**

int d;

};

**int main()**

{

**ClassD obj;**

**// obj.a = 10; // Statement 1, Error**

**// obj.a = 100; // Statement 2, Error**

**obj.ClassB::a = 10; // Statement 3,valid**

**obj.ClassC::a = 100; // Statement 4, valid**

**obj.b = 20;**

**obj.c = 30;**

**obj.d = 40**;

**cout << " a from ClassB : " << obj.ClassB::a;**

**cout << "\n a from ClassC : " << obj.ClassC::a;**

cout << "\n b : " << obj.b;

cout << "\n c : " << obj.c;

cout << "\n d : " << obj.d << '\n';

}

**Output :**

a from ClassB : 10

a from ClassC : 100

b : 20

c : 30

d : 40

**There are 2 Ways to Avoid this Ambiguity:**

1. **Avoiding ambiguity using the scope resolution operator:** Using the scope resolution operator we can manually specify the path from which data member a will be accessed, as shown in statements 3 and 4, in the above example.

**obj.ClassB::a = 10; // Statement 3**

**obj.ClassC::a = 100; // Statement 4**

***Note:*** Still, there are two copies of Class A in Class-D.

1. **Avoiding ambiguity using the virtual base class:**

#include<iostream>

using namespace std;

**class ClassA**

{

**public:**

int a;

};

**class ClassB : virtual public ClassA**

{

**public:**

int b;

};

**class ClassC : virtual public ClassA**

{

**public:**

int c;

};

**class ClassD : public ClassB, public ClassC**

{

**public:**

int d;

};

**int main()**

{

ClassD obj;

**obj.a = 10; // Statement 3**

**obj.a = 100; // Statement 4**

obj.b = 20;

obj.c = 30;

obj.d = 40;

cout << "\n a : " << obj.a;

cout << "\n b : " << obj.b;

cout << "\n c : " << obj.c;

cout << "\n d : " << obj.d << '\n';

}

**Output:**

a : 100

b : 20

c : 30

d : 40

According to the above example, Class-D has only one copy of ClassA, therefore, statement 4 will overwrite the value of a, given in statement 3.

**Note:**

* When a base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class and therefore, the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class.
* On the other hand, when the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the derived class.

**Virtual Base Classes**

Consider a situation, where all the three kinds of inheritance, namely multi-level, multiple and hierarchical are involved.

Let us say the 'child' has two direct base classes ‘parent1’ and ‘parent2’ which themselves has a common base class ‘grandparent’. The child inherits the traits of ‘grandparent’ via two separate paths. It can also be inherit directly as shown by the broken line. The grandparent is sometimes referred to as ‘INDIRECT BASE CLASS’. Now, the inheritance by the child might cause some problems. All the public and protected members of ‘grandparent’ are inherited into ‘child’ twice, first via ‘parent1’ and again via ‘parent2’. So, there occurs a duplicacy which should be avoided.

**The duplication of the inherited members can be avoided by making common base class as the virtual base class:**

**Syntax to declare Virtual Base class:**

**class g\_parent**

{

//Body

};

**class parent1: virtual public g\_parent**

{

// Body

};

**class parent2**: **public virtual g\_parent**

{

// Body

};

**class child: public parent1, public parent2**

{

// body

};

When a class is virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exists between virtual base class and derived class.

**Note that keywords ‘virtual’ and ‘public’ can be used in either order .**

**Example to show the demonstration of virtual base class:**

#include<iostream.h>

#include<conio . h>

**class student // Base class declaration**

{

**protected:**

int r\_no;

**public:**

**void get\_n (int a)**

{

r\_no = a;

}

**void put\_n (void)**

{

cout << “Roll No. “ << r\_no<< “ln”;}

};

**class test : virtual public student** // Virtually declared common base class 1

{

**protected:**

int part1;

int part2;

**public:**

**void get\_m (int x, int y)**

{

part1= x;

part2=y;

}

**void putm (void)**

{

cout << “marks obtained: “ << “\n”;

cout << “part1 = “ << part1 << “\n”;

cout << “part2 = “<< part2 << “\n”;

}

};

**class sports : public virtual student**

// virtually declared common

{ //base class 2

**protected:**

int score;

**public:**

**void get\_s (int a)**

{

score = a ;

}

**void put\_s (void)**

{

cout << “sports wt.: “ <<score<< “\n”;

}

};

**class result: public test, public sports** //derived class

{

**private:** int total ;

**public:**

void show (void) ;

};

**void result : : show (void)**

{

total = part1 + part2 + score ;

put\_n ( );

put\_m ( );

put\_s ( ) ; cout << “\n total score= “ <<total<< “\n” ;

}

**int main ( )**

{

clrscr ( ) ;

result r ;

r.get\_n (345)

r.get\_m (30, 35) ;

r.get\_s(7) ;

r. show ( ) ;

return 0;

}

**Example to show hybrid inheritance using virtual base classes :**

#include<iostream>

using namespace std;

**Class A**

{

**protected:**

int x;

**public:**

**void get (int) ;**

**void show (void) ;**

};

**void A : : get (int a)**

{

x = a ;

}

**void A : : show (void)**

{

cout << X ;

}

**Class A1 : Virtual Public A**

{

**protected:**

int y ;

**public:**

**void get (int) ;**

**void show (void);**

};

**void A1 :: get (int a)**

{

y = a;

}

**void A1 :: show (void)**

{

cout <<y ;

}

**class A2 : Virtual public A**

{

**protected:**

int z ;

**public:**

**void get (int a)**

{

z =a;

}

**void show (void)**

{

cout << z;

}

};

**class A12 : public A1, public A2**

{

int r, t ;

**public:**

**void get (int a)**

{

r = a;

}

**void show (void)**

{

t = x + y + z + r ;

cout << “result =” << t ;

}

};

**int main ( )**

{

**A12 r ;**

**r.A : : get (3) ;**

**r.A1 : : get (4) ;**

**r.A2 : : get (5) ;**

**r.get (6) ;**

r . show ( ) ;

return 0;

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

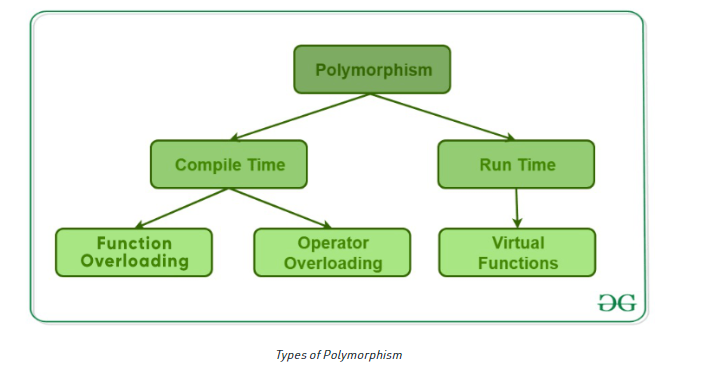
**Polymorphism**

**Introduction:**

The word “polymorphism” means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.  Polymorphism is considered one of the important features of Object-Oriented Programming.

## Types of Polymorphism:

* **Compile-time Polymorphism**
* **Runtime Polymorphism**



## 1. Compile-Time Polymorphism

This type of polymorphism is achieved by function overloading or operator overloading.

### A. Function Overloading

When there are multiple functions with the same name but different parameters, then the functions are said to be **overloaded,**hence this is known as Function Overloading. Functions can be overloaded by **changing the number of arguments** or/and **changing the type of arguments**. In simple terms, it is a feature of object-oriented programming providing many functions to have the same name but distinct parameters when numerous tasks are listed under one function name.

**Example to demonstrate function overloading or Compile-time Polymorphism**

#include <bits/stdc++.h>

using namespace std;

**class Poly**

{

public:

// Function with 1 int parameter

**void func(int x)**

{

cout << "value of x is " <<

x << endl;

}

// Function with same name but 1 double parameter

**void func(double x)**

{

cout << "value of x is " <<x << endl;

}

// Function with same name and 2 int parameters

**void func(int x, int y)**

{

cout << "value of x and y is " <<

x << ", " << y << endl;

}

};

// Driver code

**int main()**

{

**Poly obj1;**

// Function being called depends on the parameters passed

// func() is called with int value

**obj1.func(7);**  // func() is called with double value

**obj1.func(9.132);** // func() is called with 2 int values

**obj1.func(85, 64);**

return 0;

}

**Output**

value of x is 7

value of x is 9.132

value of x and y is 85, 64

.

### B. Operator Overloading

C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can make use of the addition operator (+) for string class to concatenate two strings. We know that the task of this operator is to add two operands. So a single operator ‘+’, when placed between integer operands, adds them and when placed between string operands, concatenates them.

**Example to demonstrate Operator Overloading or Compile-Time Polymorphism:**

#include <iostream>

using namespace std;

**class Complex**

{

private:

int real, imag;

public:

Complex(int r = 0,

int i = 0)

{

real = r;

imag = i;

}

// This is automatically called when '+' is used with between two //Complex objects

**Complex operator+(Complex const& obj)**

{

**Complex res;**

res.real = real + obj.real;

res.imag = imag + obj.imag;

return res;

}

void print()

{

cout << real << " + i" <<

imag << endl;

}

};

// Driver code

**int main()**

{

**Complex c1(10, 5), c2(2, 4);**

// An example call to "operator+"

**Complex c3 = c1 + c2;**

c3.print();

}

**Output**

12 + i9

## 2. Runtime Polymorphism

This type of polymorphism is achieved by **Function Overriding**. Late binding and dynamic polymorphism are other names for runtime polymorphism.The function call is resolved at runtime in [runtime polymorphism](https://www.geeksforgeeks.org/virtual-functions-and-runtime-polymorphism-in-cpp/). In contrast, with compile time polymorphism, the compiler determines which function call to bind to the object after deducing it at runtime.

### A. Function Overriding

Function Overriding occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

**Syntax to override a function:**

**class parent**

{

public:

void fun()

{

Statements;

}

**class child:public parent**

{

public:

void fun()

{

statements;

}

**int main()**

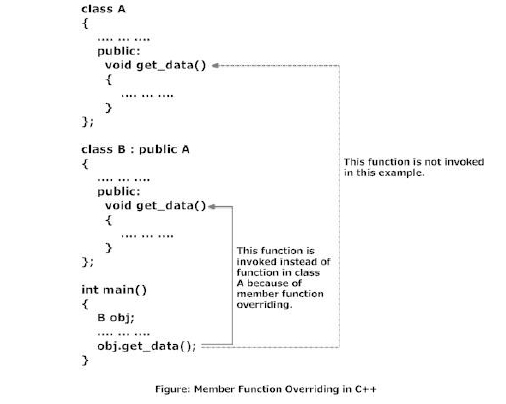
{

child obj;

obj.fun();

return 0;

}



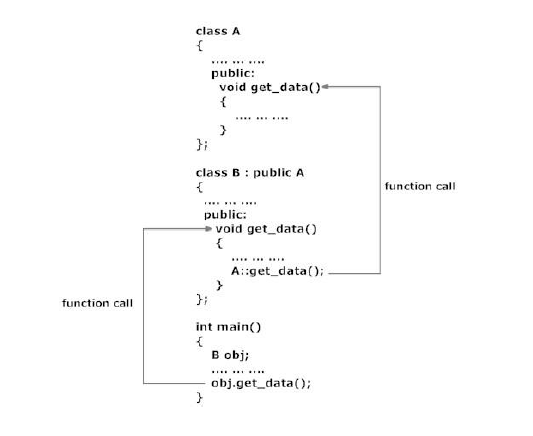
**Accessing the Overridden Function of Base Class From Derived Class**

To access the overridden function of base class from derived class, **scope resolution operator :: is used.**

**For example:** If you want to access get\_data() function of base class from derived class in above example then, the following statement is used in derived class**.**

**A*::get\_data; // Calling get\_data() of class A.***

**It is because, if the name of class is not specified, the compiler thinks get\_data() function is calling itself.**

****

**Example:**

**Example to demonstrate Function Overriding**

#include <bits/stdc++.h>

using namespace std;

**class Parent**

{

public:

**void fun()**

{

cout << "Base Function" << endl;

}

};

**class Child : public Parent**

{

public:

**void fun()**

{

cout << "Derived Function" << endl;

}

};

**int main()**

{

Child obj

obj.fun();

return 0;

}

**Output**

Derived Function

### B.Virtual Function

Virtual functions, one of advanced features of OOP is one that does not really exist but it« appears real in some parts of a program. A[virtual function](https://www.geeksforgeeks.org/virtual-function-cpp/) is a member function that is declared in the base class using the keyword virtual and is re-defined (Overridden) in the derived class.

#### Some Key Points About Virtual Functions:

* Virtual functions are Dynamic in nature.
* They are defined by inserting the keyword “**virtual**” inside a base class and are always declared with a base class and overridden in a child class
* A virtual function is called during Runtime

Polymorphism is also accomplished using pointers in C++. **It allows a pointer in a base class to point to either a base class object or to any derived class object.**

**class base**

{

//Data Members

//Member Functions

};

**class derived : public base**

{

//Data Members

//Member functions

};

**void main ( )**

{

**base \*ptr;** //pointer to class base

**derived obj ;**

**ptr = &obj ;** //indirect reference obj to the pointer

//Other Program statements

}

**The pointer ptr points to an object of the derived class obj.**

**But, a pointer to a derived class object may not point to a base class object without explicit casting.**

For example, the following assignment statements are not valid

**void main ( )**

{

**base obj;**

**derived \*ptr;**

**ptr = &obj; //invalid.... .explicit casting required**

//Other Program statements

}

**A derived class pointer cannot point to base class objects. But, it is possible by using explicit casting.**

**void main ( )**

{

**base obj ;**

**derived \*ptr; // pointer of the derived class**

**ptr = (derived \*) & obj; //correct reference**

//Other Program statements

}

**The general syntax of the virtual function declaration is:**

**class classname**

**{**

**private:**

**public:**

**virtual return\_type function\_name1 (arguments);**

**virtual return\_type function\_name2(arguments);**

**virtual return\_type function\_name3( arguments);**

**------------------**

**};**

**NOTE:**

* A destructor member function does not take any argument and no return type can be specified for it not even void.
* It is an error to redefine a virtual method with a change of return data type in the derived class with the same parameter types as those of a virtual method in the base class.
* Only a member function of a class can be declared as virtual. A non member function (nonmethod) of a class cannot be declared virtual.

**Example to demonstrate the Virtual Function:**

#include <iostream>

using namespace std;

// Declaring a Base class

**class A** {

public:

// virtual function

virtual void display()

{

cout << "Called virtual Base Class function" <<"\n\n";

}

void print()

{

cout << "Called class A print function" <<"\n\n";

}

};

// Declaring a Child Class

**class B : public A**

{

public:

void display()

{

cout << "Called B Display Function" <<"\n\n";

}

void print()

{

cout << "Called B print Function" <<"\n\n";

}

};

// Driver code

**int main()**

{

A\* base;

B child;

base = &child;

// This will call the virtual function

base->A::display();

// this will call the non-virtual function

base->print();

}

**Output**

Called virtual Base Class function

Called B print function

**Virtual destructors**

Just like declaring member functions as virtual, destructors can be declared as virtual, whereas constructors can not be virtual. Virtual Destructors are controlled in the same way as virtual functions. When a derived object pointed to by the base class pointer is deleted, destructor of the derived class as well as destructor of all its base classes are invoked. If destructor is made as non virtual destructor in the base class, only the base class’s destructor is invoked when the object is deleted.

**Example to demonstrate Virtual Destructors:**

**#include<iostream.h>**

**#include<string.h>**

**class father**

**{**

**protected:**

**char \*fname;**

**public:**

**father(char \*name)**

**{**

**fname=new char(strlen(name)+1);**

**strcpy(fname,name);**

**}**

**virtual ~father()**

**{**

**delete fname;**

**cout<<”~father is invoked…”;**

**}**

**virtual void show()**

**{**

**cout<<”father name…”<<fname;**

**}**

**};**

**class son: public father**

**{**

**protected:**

**char \*s\_name;**

**public:**

**son(char \*fname,char \*sname):father(fname)**

**{**

**sname=new char[strlen(sname)+1];**

**strcpy(s\_name,sname);**

**}**

**~son()**

**{**

**delete s\_name;**

**cout<<”~son() is invoked”<<endl;**

**}**

**void show()**

**{**

**cout<<”father’s name”<<fname;**

**cout<<”son’s name:”<<s\_name;**

**}**

**};**

**void main()**

**{**

**father \*basep;**

**basep =new father (“Ben”);**

**cout<<”basep points to base object…”**

**basep->show();**

**delete basep;**

**basep=new son(“David”,”Ben”);**

**cout<<”base points to derived object…”;**

**basep->show();**

**delete basep;**

**}**

**Pure Virtual Functions**

Generally a function is declared virtual inside a base class and we redefine it the derived classes. The function declared in the base class seldom performs any task. Pure virtual Functions are virtual functions with no definition. **They start with virtual keyword and ends with = 0**. **Here is the syntax for a pure virtual function,**

**virtual void f() = 0;**

**Example to demonstrate Pure Virtual Function :**

The class employee the functions are defined with empty body or no code inside the function. The code is written for the grade class. The methods of the derived class are invoked by the pointer to the base class.

**Example:**

#include<iostream>

**using namespace std;**

**class employee**

{

int code

char name [20] ;

**public:**

**virtual void getdata ( ) ;**

**virtual void display ( ) ;**

};

**class grade: public employee**

{

char grd [90] ;

float salary;

**public :**

**void getdata ( ) ;**

**void display ( );**

};

**void employee :: getdata ( )**

{

}

**void employee:: display ( )**

{

}

**void grade : : getdata ( )**

{

cout<< “ enter employee’s grade “;

cin> > grd ;

cout<< “\n enter the salary “ ;

cin>> salary;

}

**void grade : : display ( )**

{

cout«" Grade salary \n";

cout« grd« " "« salary« endl;

}

**int main ( )**

{

employee \*ptr ;

grade obj ;

ptr = &obj ;

ptr->getdata ( ) ;

ptr->display ( ) ;

return 0;

}

**Output**

enter employee’s grade A

enter the salary 250000

Grade salary

A 250000

**Abstract Class**

* **Abstract Class is a class which contains atleast one Pure Virtual function in it.**
* **Abstract classes are used to provide an Interface for its sub classes.**
* **Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.**

**Characteristics of Abstract Class**

Abstract class cannot be instantiated, but pointers and refrences of Abstract class type can be created.

2. Abstract class can have normal functions and variables along with a pure virtual function.

3. Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.

4. Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

**Example to demonstrate Abstract Class**

**class Base //Abstract base class**

{

**public:**

**virtual void show() = 0; //Pure Virtual Function**

};

**class Derived:public Base**

{

**public:**

**void show()**

{

cout << "Implementation of Virtual Function in Derived class"; }

};

**int main()**

{

// Base obj; //Compile Time Error

Base \*b;

Derived d;

b = &d;

b->show();

}

**Output** : Implementation of Virtual Function in Derived class