Contents

[Contents 1](#_Toc515475328)

[1 C++ 6](#_Toc515475329)

[1.1 Abstract Class 6](#_Toc515475330)

[1.1.1 Definition 6](#_Toc515475331)

[1.1.2 Syntax 6](#_Toc515475332)

[1.1.3 Example 6](#_Toc515475333)

[1.2 Abstraction 6](#_Toc515475334)

[1.2.1 Definition 6](#_Toc515475335)

[1.3 Assignment Operator 7](#_Toc515475336)

[1.4 auto\_ptr 8](#_Toc515475337)

[1.4.1 Definition 8](#_Toc515475338)

[1.4.2 Syntax 8](#_Toc515475339)

[1.4.3 Example 8](#_Toc515475340)

[1.5 Casting 8](#_Toc515475341)

[1.5.1 static\_cast 8](#_Toc515475342)

[1.6 Constructor 10](#_Toc515475343)

[1.6.1 Definition 10](#_Toc515475344)

[1.6.2 Syntax 11](#_Toc515475345)

[1.6.3 Example 11](#_Toc515475346)

[1.7 Copy Constructor 11](#_Toc515475347)

[1.7.1 Definition 11](#_Toc515475348)

[1.7.2 Syntax 12](#_Toc515475349)

[1.7.3 Example 12](#_Toc515475350)

[1.7.4 Copy Constructor Called, When ? 12](#_Toc515475351)

[1.8 Dangling Pointer 13](#_Toc515475352)

[1.8.1 Definition 13](#_Toc515475353)

[1.8.2 Example 13](#_Toc515475354)

[1.9 Deep Copy 13](#_Toc515475355)

[1.9.1 Definition 13](#_Toc515475356)

[1.9.2 Syntax 13](#_Toc515475357)

[1.9.3 Example 13](#_Toc515475358)

[1.10 Diamond Problem 13](#_Toc515475359)

[1.10.1 Definition 13](#_Toc515475360)

[1.10.2 Example 14](#_Toc515475361)

[1.11 Exception 14](#_Toc515475362)

[1.11.1 Definition 14](#_Toc515475363)

[1.11.2 Syntax 15](#_Toc515475364)

[1.11.3 Example 15](#_Toc515475365)

[1.12 Explicit 15](#_Toc515475366)

[1.12.1 Definition 15](#_Toc515475367)

[1.12.2 Syntax 15](#_Toc515475368)

[1.12.3 Example 15](#_Toc515475369)

[1.13 Friend Function 15](#_Toc515475370)

[1.13.1 Definition 15](#_Toc515475371)

[1.13.2 Syntax 16](#_Toc515475372)

[1.13.3 Example 16](#_Toc515475373)

[1.14 Function Overloading 16](#_Toc515475374)

[1.14.1 Definition 16](#_Toc515475375)

[1.14.2 Syntax 16](#_Toc515475376)

[1.14.3 Example 16](#_Toc515475377)

[1.15 Function Overriding 17](#_Toc515475378)

[1.15.1 Definition 17](#_Toc515475379)

[1.15.2 Example 17](#_Toc515475380)

[1.16 Function Pointer 18](#_Toc515475381)

[1.16.1 Definition 18](#_Toc515475382)

[1.16.2 Syntax 18](#_Toc515475383)

[1.16.3 Example 18](#_Toc515475384)

[1.17 Inheritance 20](#_Toc515475385)

[1.17.1 Definition 20](#_Toc515475386)

[1.17.2 Syntax 23](#_Toc515475387)

[1.17.3 Example 23](#_Toc515475388)

[1.18 Inline Function 24](#_Toc515475389)

[1.18.1 Definition 24](#_Toc515475390)

[1.18.2 Syntax 24](#_Toc515475391)

[1.18.3 Example 24](#_Toc515475392)

[1.19 Mutable 25](#_Toc515475393)

[1.19.1 Definition 25](#_Toc515475394)

[1.19.2 Syntax 25](#_Toc515475395)

[1.19.3 Example 25](#_Toc515475396)

[1.20 Memory Leak 25](#_Toc515475397)

[1.20.1 Definition 25](#_Toc515475398)

[1.20.2 Example 25](#_Toc515475399)

[1.21 Name Hiding 26](#_Toc515475400)

[1.21.1 Definition 26](#_Toc515475401)

[1.21.2 Example 26](#_Toc515475402)

[1.22 Namespace 26](#_Toc515475403)

[1.22.1 Creating a Namespace 26](#_Toc515475404)

[1.22.2 Example 26](#_Toc515475405)

[1.22.3 The using directive 27](#_Toc515475406)

[1.23 OOPS Concept 28](#_Toc515475407)

[1.23.1 Definition 28](#_Toc515475408)

[1.24 Operator Overloading 29](#_Toc515475409)

[1.24.1 Definition 29](#_Toc515475410)

[1.24.2 Syntax 29](#_Toc515475411)

[1.24.3 Example 30](#_Toc515475412)

[1.24.4 Example 30](#_Toc515475413)

[1.25 Overloading New and Delete operator 30](#_Toc515475414)

[1.26 Object Slicing 33](#_Toc515475415)

[1.26.1 Definition 33](#_Toc515475416)

[1.26.2 Syntax 33](#_Toc515475417)

[1.26.3 Example 33](#_Toc515475418)

[1.27 Polymerphisim 33](#_Toc515475419)

[1.27.1 Definition 33](#_Toc515475420)

[1.27.2 Syntax 33](#_Toc515475421)

[1.27.3 Example 33](#_Toc515475422)

[1.28 Pointers 33](#_Toc515475423)

[1.28.1 Definition 33](#_Toc515475424)

[1.28.2 Pointer to constant 33](#_Toc515475425)

[1.28.3 Constant pointer to variable. 34](#_Toc515475426)

[1.28.4 constant pointer to constant 34](#_Toc515475427)

[1.29 Pure Virtual Function 35](#_Toc515475428)

[1.29.1 Definition 35](#_Toc515475429)

[1.29.2 Syntax 35](#_Toc515475430)

[1.29.3 Example 35](#_Toc515475431)

[1.30 RTTI 35](#_Toc515475432)

[1.30.1 Definition 35](#_Toc515475433)

[1.30.2 Example 36](#_Toc515475434)

[1.31 Templates 36](#_Toc515475435)

[1.31.1 Function Templates 37](#_Toc515475436)

[1.31.2 Class Templates 38](#_Toc515475437)

[1.32 this pointer (this\*) 39](#_Toc515475438)

[1.32.1 Definition 39](#_Toc515475439)

[1.32.2 Syntax 39](#_Toc515475440)

[1.32.3 Example 39](#_Toc515475441)

[1.33 Virtual Destructor 39](#_Toc515475442)

[1.33.1 Definition 39](#_Toc515475443)

[1.34 Virtual Function 39](#_Toc515475444)

[1.34.1 Definition 39](#_Toc515475445)

[1.34.2 Syntax 39](#_Toc515475446)

[1.34.3 Example 39](#_Toc515475447)

[1.35 Virtual Table(vtable) and \_vptr 39](#_Toc515475448)

[1.35.1 Definition 39](#_Toc515475449)

[1.35.2 Syntax / Example 39](#_Toc515475450)

[1.36 Volatile 39](#_Toc515475451)

[1.36.1 Definition 39](#_Toc515475452)

[1.36.2 Syntax / Example 39](#_Toc515475453)

[2 STL(Standard Template Library) 39](#_Toc515475454)

[2.1 STL Container 39](#_Toc515475455)

[2.1.1 Array 39](#_Toc515475456)

[2.1.1.1 at ( ) 39](#_Toc515475457)

[2.1.1.2 [ ] Operator 39](#_Toc515475458)

[2.1.1.3 front() 39](#_Toc515475459)

[2.1.1.4 back() 39](#_Toc515475460)

[2.1.1.5 fill() 39](#_Toc515475461)

[2.1.1.6 swap() 39](#_Toc515475462)

[2.1.1.7 operators ( == , != , > , < , >= , <= ) 39](#_Toc515475463)

[2.1.1.8 Empty() 39](#_Toc515475464)

[2.1.1.9 Size() 39](#_Toc515475465)

[2.1.1.10 max\_size() 39](#_Toc515475466)

[2.1.1.11 begin() 39](#_Toc515475467)

[2.1.1.12 End 39](#_Toc515475468)

[3 Vs (Differences) 39](#_Toc515475469)

[3.1 Inline Function Vs Macros 39](#_Toc515475470)

[3.2 Pointer Vs Reference 39](#_Toc515475471)

[3.3 new Vs malloc() 39](#_Toc515475472)

[3.4 Copy Constructor Vs Assignment Operator 39](#_Toc515475473)

[3.5 Semaphore Vs Mutex 39](#_Toc515475474)

[4 Summary in Tables 39](#_Toc515475475)

[4.1 Storage Classes 39](#_Toc515475476)

[4.2 Visibility After Inheritance 39](#_Toc515475477)

[4.3 Summary of Function Behavior 39](#_Toc515475478)

[4.4 Association, Aggregation, and Composition 39](#_Toc515475479)

[4.5 Sequence Containers (Vector, List, Deque) 39](#_Toc515475480)

[5 Networking 39](#_Toc515475481)

[5.1 Client-Server Communication 39](#_Toc515475482)

[5.2 TCP 3-Way HandShake 39](#_Toc515475483)

[6 Memory Layout of C Programs 39](#_Toc515475484)

[7 The Build Process - C/C++ 39](#_Toc515475485)

[8 Important C Programs 39](#_Toc515475486)

[8.1 String Functions 39](#_Toc515475487)

[8.1.1 strlen() 39](#_Toc515475488)

[8.1.2 strcpy() 39](#_Toc515475489)

[8.1.3 strcat () 39](#_Toc515475490)

[8.1.4 strcmp () 39](#_Toc515475491)

[8.1.5 atoi () 39](#_Toc515475492)

[8.2 count how many tokens are there in given string 39](#_Toc515475493)

[8.3 Swap two variables using macro 39](#_Toc515475494)

[8.4 Multiply by 5 with given number using bitwise operator 39](#_Toc515475495)

[8.5 Toggle the k- th bit of a given number using bitwise operator 39](#_Toc515475496)

[8.6 REVERSE A NUMBER USING ARITHMATIC OPERATOR 39](#_Toc515475497)

[8.7 PRINT THE GIVEN COMMAND LINE ARGUMENTS 39](#_Toc515475498)

[8.8 Write a line backward 39](#_Toc515475499)

[8.9 Print 12345678910987654321 39](#_Toc515475500)

[8.10 Variable Length Arguments 39](#_Toc515475501)

[8.11 Variable Length Arguments 39](#_Toc515475502)

[8.12 swap two numbers without using temp variable 39](#_Toc515475503)

[8.13 function to dynamically allocate 2-dimensional array using malloc. 39](#_Toc515475504)

[8.14 Palindrome Check 39](#_Toc515475505)

[8.15 Count no of ‘1’ in a given integer 39](#_Toc515475506)

[8.16 Without Using “sizeof” operator print size of given structure 39](#_Toc515475507)

[8.17 Arrenge 0’s and 1’s in given array. 39](#_Toc515475508)

[8.18 Factorila Of a Number Using Recursion 39](#_Toc515475509)

[8.19 fibonacci Series 39](#_Toc515475510)

[8.20 Arrays: Left Rotation 39](#_Toc515475511)

[8.21 string concatenation (strcat) in c using macro 39](#_Toc515475512)

[8.22 What is the sizeof() empty main() or any empty function() ? 39](#_Toc515475513)

[8.23 Implement Your Own sizeof() 39](#_Toc515475514)

[8.24 Implement Your Own toupper() and tolower() 39](#_Toc515475515)

[8.25 How to find length of a string without string.h and loop in C? 39](#_Toc515475516)

[8.26 Write a C program to find the GCD of two numbers 39](#_Toc515475517)

[9 Important DataStructure Programs 39](#_Toc515475518)

[9.1 To get the middle of the linked list 39](#_Toc515475519)

[9.2 Program to check loop in single linked list 39](#_Toc515475520)

[9.3 Bubble Sort 39](#_Toc515475521)

[10 Multi-Threading 39](#_Toc515475522)

[10.1 What Is a Thread? 39](#_Toc515475523)

[10.2 pthread\_create() 39](#_Toc515475524)

[10.2.1 SYNOPSIS 39](#_Toc515475525)

[10.2.2 PARAMETERS 39](#_Toc515475526)

[10.2.3 Program 39](#_Toc515475527)

[10.2.4 Run The Program 39](#_Toc515475528)

[10.2.5 Output 39](#_Toc515475529)

[11 SHELL Scripting 39](#_Toc515475530)

[11.1 Environment Variable 39](#_Toc515475531)

[11.1.1 Example 39](#_Toc515475532)

[11.2 " if " conditional commands 39](#_Toc515475533)

[11.2.1 Syntax 39](#_Toc515475534)

[11.2.2 Example 39](#_Toc515475535)

[12 Design Patterns 39](#_Toc515475536)

[12.1 Singleton Design Pattern 39](#_Toc515475537)

[12.1.1 Definition 39](#_Toc515475538)

[13 git 39](#_Toc515475539)

[13.1 Commands 39](#_Toc515475540)

[13.1.1 git init 39](#_Toc515475541)

[13.1.2 git remote add origin 39](#_Toc515475542)

[13.1.3 git pull <branch\_name> <remote\_URL/remote\_name> 39](#_Toc515475543)

[13.1.4 git status 39](#_Toc515475544)

[13.1.5 git add 39](#_Toc515475545)

[13.1.6 git commit 39](#_Toc515475546)

# C++

## Abstract Class

### Definition

An *abstract class* is a class that is designed to be specifically used as a base class. An abstract class contains at least one ***pure virtual function***. You declare a pure virtual function by using a *pure specifier* (= 0) in the declaration of a virtual member function in the class declaration.

### Syntax

class class\_name {

public:

virtual void fun\_name() = 0;

};

### Example

class A {

public:

virtualvoid action() = 0;

};

class B : public A {

public:

B() {}

void action() {

// Do stuff

}

};

class MyClass {

public:

void setInstance(A newInstance) {

instance = newInstance;

}

void doSomething() {

instance.action();

}

private:

A instance;

};

int main(int argc, char\*\* argv) {

MyClass c;

B myInstance;

c.setInstance(myInstance);

c.doSomething();

return 0;

}

## Abstraction

### Definition

* Abstraction is the concept of exposing only the required essential characteristics and behavior with respect to a context.
* Hiding of data is known as data abstraction. In object, oriented programming language this is implemented automatically while writing the code in the form of class and object.

## Assignment Operator

The compiler created copy constructor and assignment operator may not be sufficient when we have pointers or any run time allocation of resource like file handle, a network connection..etc. For example, consider the following program.

#include<iostream>

using namespace std;

class Test

{

int \*ptr;

public:

Test(int i = 0) { ptr = new int(i); }

void setValue(int i) { \*ptr = i; }

void print() { cout << \*ptr << endl; }

Test & operator = (const Test &t);

};

Test & Test::operator = (const Test &t)

{

// Check for self assignment

if (this != &t)

\*ptr = \*(t.ptr);

return \*this;

}

int main()

{

Test t1(5);

Test t2;

t2 = t1;

t1.setValue(10);

t2.print();

return 0;

}

We should also add a copy constructor to the above class, so that the statements like “Test t3 = t4;” also don’t cause any problem.

Note the if condition in assignment operator. While overloading assignment operator, we must check for **self assignment**. Otherwise assigning an object to itself may lead to unexpected results (See [this](https://www.geeksforgeeks.org/archives/8673)). **Self assignment** check is not necessary for the above ‘Test’ class, because ‘ptr’ always points to one integer and we may reuse the same memory. But in general, it is a recommended practice to do self-assignment check.

## auto\_ptr

### Definition

### Syntax

### Example

## Casting

A cast is a special operator that forces one data type to be converted into another. As an operator, a cast is unary and has the same precedence as any other unary operator.

C++ provides a variety of ways to cast between types:

* static\_cast
* reinterpret\_cast
* const\_cast
* dynamic\_cast
* C-style casts

### static\_cast

**static\_cast** is the main workhorse in our C++ casting world. **static\_cast** handles implicit conversions between types (e.g. integral type conversion, any pointer type to void\*).

Intensions are more clear in C++ style cast (express your intent better and make code review easier). And error found at compile-time.

**Reason-1**

C-style casting is hard to find in code, but you can search **static\_cast** keyword.

#include <iostream>

using namespace std;

int main() {

float f = 3.5;

int a;

//a = f; //C-Style Casting

a = static\_cast <int> (f); // Value of a is 3

return 0;

}

**Reason-2**

Use **static\_cast** when conversion between types is provided through conversion operator or conversion constructor.

#include <iostream>

#include <string>

using namespace std;

class Int{

private:

int x;

public:

Int(int x = 0) :x{ x }{

cout << "conversion constructor" << endl;

}

operator string(){

cout << "conversion operator" << endl;

return std::to\_string(x);

}

};

int main(){

Int obj(3);

string str1 = obj;

obj = 20;

string str2 = static\_cast<string>(obj);

obj = static\_cast<Int>(30);

return 0;

}

**Output:**

conversion constructor

conversion operator

conversion constructor

conversion operator

conversion constructor

**Reason-3**

static\_cast is more restrictive than C-Style casting.

EXAMPLE: char\* to int\* is allowed in C-Style but not with static\_cast

#include <iostream>

using namespace std;

int main(){

char c; //1 byte data

int\* p = (int\*)&c; //4 byte data

\*p = 5; // Pass at compile-time but FAIL at run time.

//that's why it is dangerous

int\* ip = static\_cast<int\*>(&c); //FAIL//Compile-time error.

//Because not compatible pointer type

return 0;

}

**Reason-4**

static\_cast avoid cast from derived to private base pointer.

#include <iostream>

class Base{};

class Derived : private Base{};

int main(){

Derived d1;

Base \*bp1 = (Base\*)&d1; //Allowed at compile time.

Base \*bp2 = static\_cast<Base\*>(&d1); //FAIL//Compile-time error. //static\_cast : conversion from //'Derived \*' to 'Base \*' exists,

//but is inaccessible ,it is //**privately derived**

// No problem in public

}

**Reason-5**

Use for all up-casting, but never use for confused down cast because there are no runtime checks performed for static\_cast conversion.dynamic\_cast should be used here.

#include <iostream>

**Base**

using namespace std;

class Base{};

**d2**1

**d1**1

class Derived1 : public Base{};

class Derived2 : public Base{};

int main(){

Derived1 d1;

**d1p**1

**d2p**

Derived2 d2;

Base \*bp1 = static\_cast<Base\*>(&d1);

Base \*bp2 = static\_cast<Base\*>(&d2);

Derived1 \*dp1 = static\_cast<Derived1\*>(bp2); // dynamic\_cast should be used

Derived2 \*dp2 = static\_cast<Derived2\*>(bp1); // dynamic\_cast should be used

return 0;

}

**Reason-6**

static\_cast should be preferred when converting **to void\*** OR **from void\*.**

#include <iostream>

using namespace std;

int main(){

int i = 10;

void \*v = static\_cast<void\*>(&i);

int \*ip = static\_cast<int\*>(v);

return 0;

}

## Constructor

### Definition

A constructor is a function that initializes the members of an object. A constructor only knows how to build an object of its own class.

Constructors aren't automatically inherited between base and derived classes. If you don't supply one in the derived class, a default will be provided but this may not do what you want.

If no constructor is supplied then a default one is created by the compiler without anyparameters. There must always be a constructor, even if it is the default and empty. If you supply a constructor with parameters then a default will NOT be created. If the class is using virtual function it is used to initialize the pointer to the virtual table, and, in class hierarchies, it calls the constructors of the base classes. Constructors in the base class and derived class use initialization lists to initialize the members of the class.

* Constructors are just functions with the same name as the class.
* Constructors are intended to initialize the members of the class when an instance of that class is created.
* Constructors are not called directly (except through initialization lists)
* Constructors are never virtual.

1. A constructor cannot be virtual because at the time when constructor is invoked the virtual table would not be available in the memory. hence, we cannot have virtual constructor.
2. Declaring something virtual in C++ means that it can be overridden by a sub-class of the current class, however the constructor is called when the objected is created, at that time you cannot be creating a sub-class of the class you must be creating the class so there would never be any need to declare a constructor virtual.

* Multiple constructors for the same class can be defined. They must have different parameters to distinguish them.

### Syntax

classclass\_name {

…

…

public:

class\_name(void); // constuctor,

};

### Example

class rectangle { // A simple class

int height;

int width;

public:

rectangle(void); // with a constuctor,

~rectangle(void); // and a destructor

};

rectangle::rectangle(void) // constuctor

{

height = 6;

width = 6;

}

## Copy Constructor

### Definition

A copy constructor is a special [constructor](#_Constructor) in the C++ programming language creating a new object as a copy of an existing object. The first argument of such a constructor is a ***reference to an object*** of the same type as is being constructed (const or non-const), which might be followed by parameters of any type (all having default values).

Normally the compiler automatically creates a copy constructor for each class (known as a ***default copy constructor***) but for special cases the programmer creates the copy constructor, known as a ***user-defined copy constructor***. In such cases, the compiler does not create one.

### Syntax

class\_name(const class\_name& obj\_name);

### Example

#include<iostream>

using namespace std;

class Sample\_copyconstructor

{

private:

int x, y; // data members

public:

Sample\_copyconstructor (int x1, int y1)

{

x = x1;

y = y1;

}

// Copy constructor

Sample\_copyconstructor (const Sample\_copyconstructor &sam)

{

x = sam.x;

y = sam.y;

}

void display()

{

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

}

};

int main()

{

Sample\_copyconstructor obj1(10, 15); // Normal constructor

Sample\_copyconstructor obj2 = obj1; // Copy constructor

cout << "Normal constructor : ";

obj1.display();

cout << "Copy constructor : ";

obj2.display();

return 0;

}

Output:

Normal constructor : 10 15

Copy constructor : 10 15

### Copy Constructor Called, When ?

* When an object of the class is returned by value.
* When an object of the class is passed (to a function) by value as an argument.
* When an object is constructed based on another object of the same class.
* When compiler generates a temporary object.

## Dangling Pointer

### Definition

Suppose we allocated a chunk of memory and store its address in a pointer. If the chunk of memory is freed and if the pointer continuous to point to that loction, the pointer is said dangling pointer.

### Example

class Sample

{

public:

int \*ptr;

Sample(inti)

{

ptr = new int(i);

}

~Sample()

{

delete ptr;

}

void PrintVal()

{

cout<< "The value is " << \*ptr;

}

};

void SomeFunc(Sample x)

{

cout<< "Say i am in someFunc " <<endl;

}

intmain()

{

Sample s1 = 10;

SomeFunc(s1);

s1.PrintVal();

}

## Deep Copy

### Definition

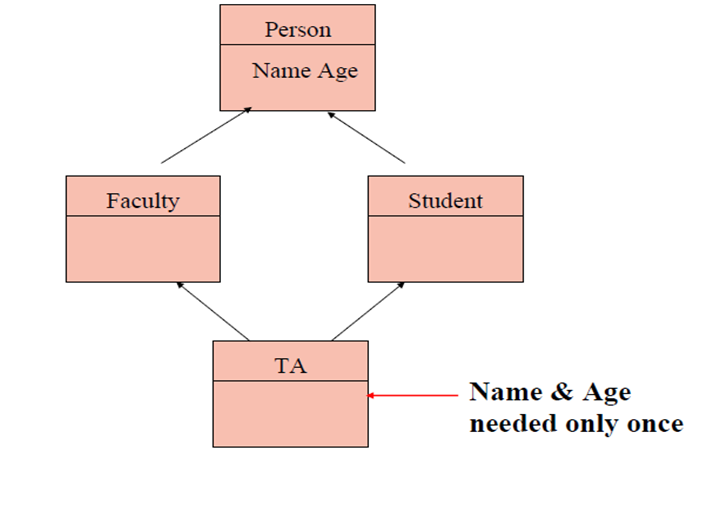
### Syntax

### Example

## Diamond Problem

### Definition

The diamond problem occurs when two superclasses of a class have a common base class. For example, in the following diagram, the TA class gets two copies of all attributes of Person class, this causes ambiguities.



### Example

#include<iostream>

using namespace std;

class Animal{

public:

int age;

void walk(){

cout << "animal walks" << endl;

}

};

class Tiger :virtual public Animal {

};

class Lion : virtual public Animal {

};

class Liger : public Tiger, public Lion {

};

int main(){

Liger Li1;

Li1.walk();

return 0;

}

## Exception

### Definition

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: try, catch, and throw.

* **throw** − A program throws an exception when a problem shows up. This is done using a throw keyword.
* **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
* **try** − A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.
* A better way is to use catch(...) as a default statement along with other catch statement so that it can catch all those exception which are not handled by other catch statements.

### Syntax

try

{

statements;

... ... ...

throw exception;

}

catch (type argument)

{

statements;

... ... ...

}

### Example

#include <iostream>

using namespace std;

double division(int a, int b) {

if( b == 0 ) {

throw "Division by zero condition!";

}return (a/b);

}

int main () {

int x = 50;

int y = 0;

double z = 0;

try

{

z = division(x, y);

cout << z << endl;

}

catch (const char\* msg)

{

cerr << msg << endl;

}

return 0;

}

## Explicit

### Definition

### Syntax

### Example

## Friend Function

### Definition

**Following are some important points about friend functions and classes:**

* Friendship is not mutual. If a class A is friend of B, then B doesn’t become friend of A automatically.
* Friendship is not inherited
* Friend of the class can be member of some other class.
* Friend of one class can be friend of another class or all the classes in one program, such a friend is known as GLOBAL FRIEND.
* Friend can access the private or protected members of the class in which they are declared to be friend, but they can use the members for a specific object.
* Friends are non-members hence do not get “this” pointer.
* Friends, can be friend of more than one class, hence they can be used for message passing between the classes.
* Friend can be declared anywhere (in public, protected or private section) in the class.

### Syntax

### Example

## Function Overloading

### Definition

These functions having different number or type (or both) or order of parameters are known as overloaded functions.

### Syntax

int overloaded() { }

int overloaded (int g){ }

int overloaded (float g){ }

int overloaded (int g, double s){ }

int overloaded (float s, int g, double k){ }

### Example

#include <iostream>

using namespace std;

long add(long, long);

float add(float, float);

int main()

{

long a, b, c;

float e, f, g;

cout << "Enter two integers\n";

cin >> a >> b;

c = add(a, b);

cout << "Sum of integers: " << c << endl;

cout << "Enter two floating point numbers\n";

cin >> e >> f;

g = add(e, f);

cout << "Sum of floats: " << g << endl;

}

long add(long c, long g)

{

long sum;

sum = c + g;

return sum;

}

float add(float c, float g)

{

float sum;

sum = c + g;

return sum;

}

## Function Overriding

### Definition

If we inherit a class into the derived class and provide a definition for one of the base class's function again inside the derived class, then that function is said to be overridden, and this mechanism is called Function Overriding

Requirements for Overriding

* Inheritance should be there. Function overriding cannot be done within a class. For this we require a derived class and a base class.
* Function that is redefined must have exactly the same declaration in both base and derived class, that means same name, same return type and same parameter list.

### Example

class Base

{

public:

void shaow()

{

cout << "Base class\t";

}

};

class Derived:public Base

{

public:

void show()

{

cout << "Derived Class";

}

}

int main()

{

Base b; //Base class object

Derived d; //Derived class object

b.show(); //Early Binding Ocuurs

d.show();

}

Output : Base class Derived class

## Function Pointer

### Definition

Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.

Unlike normal pointers, we do not allocate de-allocate memory using function pointers.

A function’s name can also be used to get functions’ address. For example, in the below program, we have removed address operator ‘&’ in assignment. We have also changed function call by removing \*, the program still works.

### Syntax

**void (\*fun\_ptr)(int)**

### Example

#include <stdio.h>

// A normal function with an int parameter

// and void return type

void fun(int a)

{

printf("Value of a is %d\n", a);

}

int main()

{

//void(\*fun\_ptr)(int) = &fun;

void(\*fun\_ptr)(int) = fun; // & removed

//(\*fun\_ptr)(10);

fun\_ptr(10); // \* removed

return 0;

}

Like normal pointers, we can have an array of function pointers. Below example in point 5 shows syntax for array of pointers.

Function pointer can be used in place of switch case. For example, in below program, user is asked for a choice between 0 and 2 to do different tasks.

#include <stdio.h>

void add(int a, int b)

{

printf("Addition is %d\n", a + b);

}

void subtract(int a, int b)

{

printf("Subtraction is %d\n", a - b);

}

void multiply(int a, int b)

{

printf("Multiplication is %d\n", a\*b);

}

int main()

{

// fun\_ptr\_arr is an array of function pointers

void(\*fun\_ptr\_arr[])(int, int) = { add, subtract, multiply };

unsigned int ch, a = 15, b = 10;

printf("Enter Choice: 0 for add, 1 for subtract and 2 "

"for multiply\n");

scanf\_s("%u", &ch);

if (ch > 2) return 0;

(\*fun\_ptr\_arr[ch])(a, b);

return 0;

}

Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function.

For example, consider the following C program where wrapper() receives a void fun() as parameter and calls the passed function.

#include <stdio.h>

// Two simple functions

void fun1() { printf("Fun1\n"); }

void fun2() { printf("Fun2\n"); }

// A function that receives a simple function

// as parameter and calls the function

void wrapper(void(\*fun)())

{

fun();

}

int main()

{

wrapper(fun1);

wrapper(fun2);

return 0;

}

Output:

Fun1

Fun2

## Inheritance

### Definition

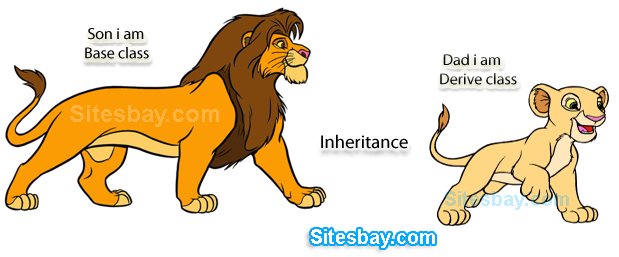
The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important feature of Object Oriented Programming.

Important points

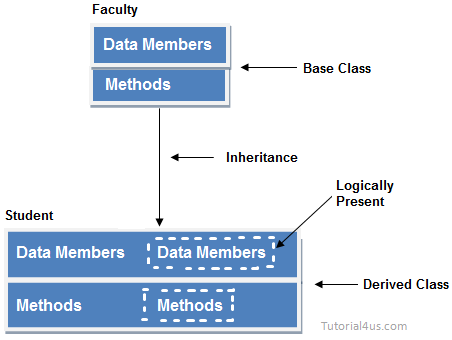
* In the inheritance the class which is give data members and methods is known as base or super or parent class.
* The class which is taking the data members and methods is known as sub or derived or child class.

Real Life Example of Inheritance in C++

The real life example of inheritance is child and parents, all the properties of father are inherited by his son.



**Diagram**



In the above diagram data members and methods are represented in broken line are inherited from faculty class and they are visible in student class logically.

Advantage of inheritance

If we develop any application using this concept than that application have following advantages,

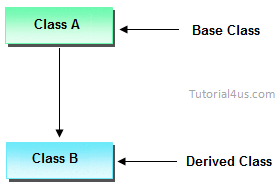
* Application development time is less.
* Application take less memory.
* Application execution time is less.
* Application performance is enhance (improved).
* Redundancy (repetition) of the code is reduced or minimized so that we get consistence results and less storage cost.

Types of Inheritance

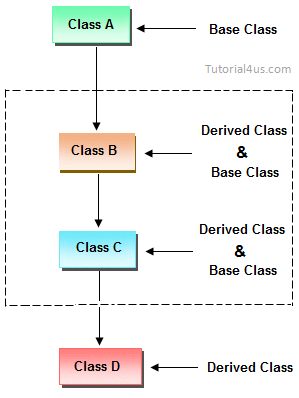
Based on number of ways inheriting the feature of base class into derived class it have five types they are:

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

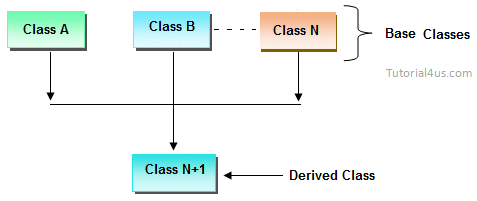
**Single inheritance**

* In single inheritance there exists single base class and single derived class.
* 

**Multilevel** **inheritances**

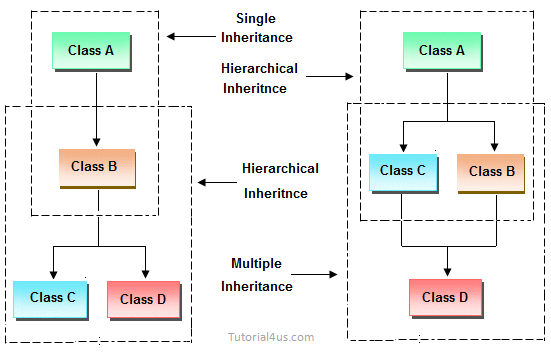
* In Multilevel inheritances there exists single base class, single derived class and Multilevel intermediate base classes.
* Single base class + single derived class + Multilevel intermediate base classes.
* Intermediate base classes
* An intermediate base class is one in one context with access derived class and in another context same class access base class.
* 
* Hence all the above three inheritance types are supported by both classes and interfaces.

**Multiple inheritance**

* In multiple inheritance there exist multiple classes and singel derived class.
* 

**Hybrid inheritance**

* Combination of any inheritance type



### Syntax

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

{

//body of subclass

};

### Example

#include<iostream.h>

#include<conio.h>

class employee

{

public:

int salary;

};

class developer : public employee

{

employee e;

public:

void salary()

{

cout<<"Enter employee salary: ";

cin>>e.salary; // access base class data member

cout<<"Employee salary: "<<e.salary;

}

};

void main()

{

clrscr();

developer obj;

obj.salary();

getch();

}

## Inline Function

### Definition

* The inline functions are a C++ enhancement feature to increase the execution time of a program.
* Functions can be instructed to compiler to make them inline so that compiler can replace those function definition wherever those are being called.
* Compiler replaces the definition of inline functions at compile time instead of referring function definition at runtime.

NOTE- This is just a suggestion to compiler to make the function inline, if function is big (in term of executable instruction etc) then, compiler can ignore the “inline” request and treat the function as normal function.

Remember, inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances like:

* If a function contains a loop. (for, while, do-while)
* If a function contains static variables.
* If a function is recursive.
* If a function return type is other than void, and the return statement doesn’t exist in function body.
* If a function contains switch or goto statement.

**Inline functions provide following advantages:**

* Function call overhead doesn’t occur.
* It also saves the overhead of push/pop variables on the stack when function is called.
* It also saves overhead of a return call from a function.
* When you inline a function, you may enable compiler to perform context specific optimization on the body of function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of calling context and the called context.
* Inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function call preamble and return.

**Inline function disadvantages:**

* The added variables from the inlined function consumes additional registers, After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization. This means that when inline function body is substituted at the point of function call, total number of variables used by the function also gets inserted. So the number of register going to be used for the variables will also get increased. So if after function inlining variable numbers increase drastically then it would surely cause an overhead on register utilization.
* If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of same code.
* Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.
* Inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled because compiler would require to replace all the code once again to reflect the changes, otherwise it will continue with old functionality.
* Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.
* Inline functions might cause thrashing because inlining might increase size of the binary executable file. Thrashing in memory causes performance of computer to degrade.

### Syntax

inline return-type function - name(parameters)

{

// function code

}

### Example

#include <iostream>

using namespace std;

inline int Max(int x, int y) {

return (x > y) ? x : y;

}

int main() {

cout << "Max (20,10): " << Max(20, 10) << endl;

cout << "Max (0,200): " << Max(0, 200) << endl;

cout << "Max (100,1010): " << Max(100, 1010) << endl;

return 0;

}

**Output:**

Max (20,10): 20

Max (0,200): 200

Max (100,1010): 1010

## Lambda expression

### Definition

In C++11, a lambda expression—often called a lambda—is a convenient way of defining an anonymous function object right at the location where it is invoked or passed as an argument to a function. Typically lambdas are used to encapsulate a few lines of code that are passed to algorithms or asynchronous methods.

### Syntax

[capture clause](parameters) -> return-type

{

//definition of method

}

### Example

**Regular Function**

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

bool is\_greater\_than\_5(int value)

{

return (value > 5);

}

int main()

{

vector<int> numbers{ 1, 2, 3, 4, 5, 10, 15, 20, 25, 35, 45, 50 };

auto greater\_than\_5\_count = count\_if(numbers.begin(), numbers.end(), ***is\_greater\_than\_5***);

cout << "The number of elements greater than 5 is: "

<< greater\_than\_5\_count << "." << endl;

}

**Lambda expression**

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

int main(){

vector<int> numbers{ 1, 2, 3, 4, 5, 10, 15, 20, 25, 35, 45, 50 };

auto great\_than\_5\_count =

count\_if(numbers.begin(), numbers.end(), ***[](int x) { return (x > 5); }***);

cout << "The number of elements greater than 5 is: "

<< greater\_than\_5\_count << "." << endl;

}

## Mutable

### Definition

Mutable data member is that member which can always be changed; even if the object is const type. It is just opposite to “const”. Sometime we required to use only one or two data member as a variable and other as a constant. In that situation, mutable is very helpful to manage classes.

### Syntax

mutable member-variable-declaration;

### Example

#include <iostream>

usingstd::cout;

classTest{

public:

int x;

mutable int y;

Test() { x = 4; y = 10; }

};

intmain(){

constTest t1;

t1.y = 20;

cout << t1.y;

return0;

}

## Memory Leak

### Definition

Memory leak occurs when programmers create a memory in heap and forget to delete it.

Memory leaks are particularly serious issues for programs like daemons and servers which by definition never terminates.

### Example

/\* Function with memory leak \*/

#include <stdlib.h>

void f()

{

int \*ptr = (int \*)malloc(sizeof(int));

/\* Do some work \*/

return; /\* Return without freeing ptr\*/

}

//To avoid memory leaks, memory allocated on heap should always be freed when no longer needed.

void f()

{

int \*ptr = (int \*)malloc(sizeof(int));

/\* Do some work \*/

free(ptr);

return;

}

## Name Hiding

### Definition

Name hiding happens when an overloaded member function is declared in the scope of a derived class.

### Example

class Base{

public:

virtual ~Base() {}

virtual void someFunction() const;

virtual void someFunction(int x) const;

};

void Base::someFunction() const {}

void Base::someFunction(int x) const {}

class Derived : public Base

{

public:

void someFunction() const;

};

void Derived::someFunction() const {}

int main(int argc, char \*\*argv)

{

Derived d;

d.someFunction(123);

return 0;

}

Compilation Error: no matching function for call to 'Derived::someFunction(int)'

## Namespace

Namespace is a container for identifiers. It puts the names of its members in a distinct space so that they don't conflict with the names in other namespaces or global namespace.

### Creating a Namespace

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows

namespace namespace\_name {

// code declarations

}

To call the namespace-enabled version of either function or variable, prepend (::) the namespace name as follows

namespace\_name::code; // code could be variable or function.

### Example

Let us see how namespace scope the entities including variable and functions

#include <iostream>

using namespace std;

// first name space

namespace first\_space {

void func() {

cout << "Inside first\_space" << endl;

}

}

// second name space

namespace second\_space {

void func() {

cout << "Inside second\_space" << endl;

}

}

int main() {

first\_space::func();// Calls function from first name space.

second\_space::func();// Calls function from second name space.

return 0;

}

OUTPUT::

Inside first\_space

Inside second\_space

### The using directive

You can also avoid prepending of namespaces with the **using namespace** directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace. The namespace is thus implied for the following code

#include <iostream>

using namespace std;

// first name space

namespace first\_space {

void func() {

cout << "Inside first\_space" << endl;

}

}

// second name space

namespace second\_space {

void func() {

cout << "Inside second\_space" << endl;

}

}

using namespace first\_space;

int main() {

// This calls function from first name space.

func();

return 0;

}

OUTPUT::

Inside first\_space

## OOPS Concept

### Definition

* Objects
* Classes
* Abstraction
* Encapsulation
* Inheritance
* Overloading
* Exception Handling

**Objects**

Objects are the basic unit of OOP. They are instances of class, which have data members and use various member functions to perform tasks.

**Class**

It is similar to structures in C language. Class can also be defined as user defined data type but it also contains functions in it. So, class is basically a blueprint for object. It declares & defines what data variables the object will have and what operations can be performed on the class's object

**Abstraction**

Abstraction refers to showing only the essential features of the application and hiding the details. In C++, classes provide methods to the outside world to access & use the data variables, but the variables are hidden from direct access. This can be done access specifiers.

**Encapsulation**

It can also be said data binding. Encapsulation is all about binding the data variables and functions together in class.

**Inheritance**

Inheritance is a way to reuse once written code again and again. The class which is inherited is called base calls & the class which inherits is called derived class. So when, a derived class inherits a base class, the derived class can use all the functions which are defined in base class, hence making code reusable.

**Polymorphism**

It is a feature, which lets us create functions with same name but different arguments, which will perform differently. That is function with same name, functioning in different way. Or, it also allows us to redefine a function to provide its new definition. You will learn how to do this in details soon in coming lessons.

**Exception Handling**

Exception handling is a feature of OOP, to handle unresolved exceptions or errors produced at runtime.

## Operator Overloading

### Definition

* Operator Overloading is a static type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type.
* Operator those are not overloaded are follows

1. Scope Resolution Operator (::)
2. Pointer-to-member Operator (.\*)
3. Member Access or Dot operator (.)
4. Ternary or Conditional Operator (?:)
5. Object size Operator (sizeof)
6. Object type Operator (typeid)

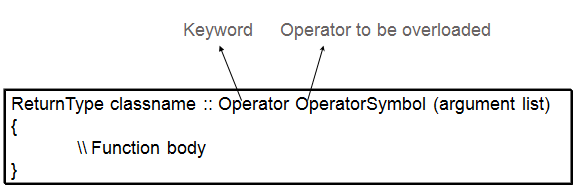
* Operator overloading can be done by **implementing a function** which can be :

1. Member Function
2. Non-Member Function
3. Friend Function

* Following are some **restrictions** to be kept in mind while implementing operator overloading.

1. Precedence and Associativity of an operator cannot be changed.
2. Arity (numbers of Operands) cannot be changed.
3. Unary operator remains unary, binary remains binary etc.
4. No new operators can be created, only existing operators can be overloaded.
5. Cannot redefine the meaning of a procedure.
6. You cannot change how integers are added.

### Syntax



### Example

#include<iostream>

usingnamespace std;

class Test

{

private:

int count;

public:

Test(): count(5){}

void operator ++()

{

count = count+1;

}

void Display() { cout<<"Count: "<<count; }

};

int main()

{

Test t;

// this calls "function void operator ++()" function

++t;

t.Display();

return 0;

}

### Example

## Overloading New and Delete operator

The new and delete operators can also be overloaded like other operators in C++. New and Delete operators can be overloaded globally or they can be overloaded for specific classes.

* If these operators are overloaded using member function for a class, it means that these operators are overloaded **only for that specific class**.
* If overloading is done outside a class (i.e. it is not a member function of a class), the overloaded ‘new’ and ‘delete’ will be called anytime you make use of these operators (within classes or outside classes). This is **global overloading**.

**Syntax for overloading the new operator :**

*void\* operator new(size\_t size);*

The overloaded new operator receives size of type size\_t, which specifies the number of bytes of memory to be allocated. The return type of the overloaded new must be void\*.The overloaded function returns a pointer to the beginning of the block of memory allocated.

**Syntax for overloading the delete operator :**

*void operator delete(void\*);*

The function receives a parameter of type void\* which has to be deleted. Function should not return anything.  
**NOTE:** Both overloaded new and delete operator functions are **static members**by default. Therefore, they don’t have access to **this pointer.**

***Overloading new and delete operator for a specific class***

// CPP program to demonstrate Overloading new and delete operator for a specific class

#include<iostream>

#include<stdlib.h>

using namespace std;

class student

{

string name;

int age;

public:

student()

{

cout << "Constructor is called\n";

}

student(string name, int age)

{

this->name = name;

this->age = age;

}

void display()

{

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

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

}

void \* operator new(size\_t size)

{

cout << "Overloading new operator with size: " << size << endl;

void \* p = ::new student();

//void \* p = malloc(size); will also work fine

return p;

}

void operator delete(void \* p)

{

cout << "Overloading delete operator " << endl;

free(p);

}

};

int main()

{

student \* p = new student("Yash", 24);

p->display();

delete p;

}

**OUTPUT** :

Overloading new operator with size: 16

Constructor is called

Name:Yash

Age:24

Overloading delete operator

**NOTE:** In the above new overloaded function, we have allocated dynamic memory through new operator, but it should be global new operator otherwise it will go in recursion  
void \*p = new student(); // this will go in recursion as**new** will be overloaded again and again  
void \*p = ::new student(); // this is correct

// CPP program to demonstrate

***Global overloading of new and delete operator***

// Global overloading of new and delete operator

#include<iostream>

#include<stdlib.h>

using namespace std;

void \* operator new(size\_t size){

cout << "New operator overloading " << endl;

void \* p = malloc(size);

return p;

}

void operator delete(void \* p){

cout << "Delete operator overloading " << endl;

free(p);

}

int main(){

int n = 5, i;

int \* p = new int[3];

for (i = 0; i<n; i++)

p[i] = i;

cout << "Array: ";

for (i = 0; i<n; i++)

cout << p[i] << " ";

cout << endl;

delete p;

}

**OUTPUT** :

New operator overloading

Array: 0 1 2 3 4

Delete operator overloading

**NOTE:** In the code above, in new overloaded function we cannot allocate memory using **::new int[5]** as it will go in recursion. We need to allocate memory using malloc only.

***Why to overload new and delete?***

1. The overloaded new operator function can accept arguments; therefore, a class can have multiple overloaded new operator functions. This gives the programmer more flexibility in customizing memory allocation for objects.
2. Overloaded new or delete operators also provide **Garbage Collection** for class’s object.
3. **Exception handling routine** can be added in overloaded new operator function.
4. Sometimes you want operators new and delete to do something **customized that the compiler-provided versions don’t offer**. For example, You might write a custom operator delete that overwrites deallocated memory with zeros in order to increase the security of application data.
5. We can use **realloc() function** in new function to re-allocate memory dynamically.
6. Overloaded new operator also enables programmers to squeeze some **extra performance** out of their programs. For example, In a class, to speed up the allocation of new nodes, a list of deleted nodes is maintained so that their memory can be reused when new nodes are allocated.In this case, the overloaded delete operator will add nodes to the list of deleted nodes and the overloaded new operator will allocate memory from this list rather than from the heap to speedup memory allocation. Memory from the heap can be used when the list of deleted nodes is empty.

## Object Slicing

### Definition

### Syntax

### Example

## Polymerphisim

### Definition

### Syntax

### Example

## Pointers

### Definition

* Pointers store address of variables or a memory location.
* To access address of a variable to a pointer, we use the unary operator & (ampersand) that returns the address of that variable. For example &x gives us address of variable x.
* One more operator is unary \* (Astrik) which is used for two things :
* To declare a pointer variable: When a pointer variable is declared in C/C++, there must a (\*) before its name.

#include <stdio.h>

int main(void)

{

int i = 10;

int j = 20;

int \*ptr = &i; /\* pointer to integer \*/

printf("\*ptr: %d\n", \*ptr);

/\* pointer is pointing to another variable \*/

ptr = &j;

printf("\*ptr: %d\n", \*ptr);

/\* we can change value stored by pointer \*/

\*ptr = 100;

printf("\*ptr: %d\n", \*ptr);

return 0;

}

### Pointer to constant

* Pointer to constant can be declared in following two ways.

Syntax

const int \*ptr;

or

int const \*ptr;

Example

#include <stdio.h>

int main(void)

{

int i = 10;

int j = 20;

const int \*ptr = &i; /\* ptr is pointer to constant \*/

printf("ptr: %d\n", \*ptr);

\*ptr = 100; /\* error: object pointed cannot be modified using the pointer ptr \*/

ptr = &j; /\* valid \*/

printf("ptr: %d\n", \*ptr);

return 0;

}

### Constant pointer to variable.

Syntax

int \*const ptr;

* Above declaration is constant pointer to integer variable, means we can change value of object pointed by pointer, but cannot change the pointer to point another variable.

Example

#include <stdio.h>

int main(void)

{

int i = 10;

int j = 20;

int \*const ptr = &i; /\* constant pointer to integer \*/

printf("ptr: %d\n", \*ptr);

\*ptr = 100; /\* valid \*/

printf("ptr: %d\n", \*ptr);

ptr = &j; /\* error \*/

return 0;

}

### constant pointer to constant

Syntax

const int \*const ptr;

* Above declaration is constant pointer to constant variable which means we cannot change value pointed by pointer as well as we cannot point the pointer to other variable. Let us see with example.

Example

#include <stdio.h>

int main(void)

{

int i = 10;

int j = 20;

const int \*const ptr = &i; /\* constant pointer to constant integer \*/

printf("ptr: %d\n", \*ptr);

ptr = &j; /\* error \*/

\*ptr = 100; /\* error \*/

return 0;

}

## Pure Virtual Function

### Definition

### Syntax

### Example

## RTTI

### Definition

In C++, RTTI (Run-time type information) is a mechanism that exposes information about an object’s data type at runtime and is available only for the classes which have at least one virtual function. It allows the type of an object to be determined during program execution.

### Example

For example, dynamic\_cast uses RTTI and following program fails with error “cannot dynamic\_cast `b’ (of type `class B\*’) to type `class D\*’ (source type is not polymorphic) ” because there is no virtual function in the base class B.

// CPP program to illustrate

// Run Time Type Identification

#include<iostream>

using namespace std;

class B { };

class D : public B {};

int main()

{

B \*b = new D;

D \*d = dynamic\_cast<D\*>(b);

if (d != NULL)

cout << "works";

else

cout << "cannot cast B\* to D\*";

getchar();

return 0;

}

Adding a virtual function to the base class B makes it working.

|  |
| --- |
| // CPP program to illustrate  // Run Time Type Identification  #include<iostream>  using namespace std;  class B { virtual void fun() {} };  class D : public B { };  int main()  {  B \*b = new D;  D \*d = dynamic\_cast<D\*>(b);  if (d != NULL)  cout << "works";  else  cout << "cannot cast B\* to D\*";  getchar();  return 0;  } |

**Output**::

works

## Smart Pointers

### Definition

* Using smart pointers, we can make pointers to work in way that we don’t need to explicitly call delete.
* Smart pointer is a wrapper class over a pointer with operator like \* and -> overloaded. The objects of smart pointer class look like pointer, but can do many things that a normal pointer can’t like automatic destruction (yes, we don’t have to explicitly use delete), reference counting and more.
* The idea is to make a class with a pointer, destructor and overloaded operators like \* and ->. Since destructor is automatically called when an object goes out of scope, the dynamically allocated memory would automatically delete (or reference count can be decremented).

### Example

#include<iostream>

using namespace std;

class SmartPtr

{

int \*ptr; // Actual pointer

public:

explicit SmartPtr(int \*p = NULL) { ptr = p; }

// Destructor

~SmartPtr() { delete(ptr); }

// Overloading dereferencing operator

int &operator \*() { return \*ptr; }

};

int main()

{

SmartPtr ptr(new int());

\*ptr = 20;

cout << \*ptr;

// We don't need to call delete ptr: when the object

// ptr goes out of scope, destructor for it is automatically

// called and destructor does delete ptr.

return 0;

}

Output:

20

we can use templates to write a generic smart pointer class.

#include<iostream>

using namespace std;

// A generic smart pointer class

template <class T>

class SmartPtr

{

T \*ptr; // Actual pointer

public:

// Constructor

explicit SmartPtr(T \*p = NULL) { ptr = p; }

// Destructor

~SmartPtr() { delete(ptr); }

// Overloading dereferncing operator

T & operator \* () { return \*ptr; }

// Overloding arrow operator so that members of T can be accessed

// like a pointer (useful if T represents a class or struct or

// union type)

T \* operator -> () { return ptr; }

};

int main()

{

SmartPtr<int> ptr(new int());

\*ptr = 20;

cout << \*ptr;

return 0;

}

Output:

20

## Templates

* Templates are powerful features of C++ which allows you to write **generic** programs.
* In simple terms, you can create a single function or a class to work with different data types using templates.
* Templates are often used in larger codebase for the purpose of code **reusability** and **flexibility** of the programs.
* The concept of templates can be used in two different ways:
* Function Templates
* Class Templates

### Function Templates

* A function template works in a similar to a normal [function](https://www.programiz.com/cpp-programming/function), with one key difference.
* A single function template can work with different data types at once but, a single normal function can only work with one set of data types.
* Normally, if you need to perform identical operations on two or more types of data, you use function overloading to create two functions with the required function declaration.
* However, a better approach would be to use function templates because you can perform the same task writing less and maintainable code.

**Syntax**

A function template starts with the keyword **template** followed by template parameter/s

inside **< >** which is followed by function declaration.

template <class/typename T>

T someFunction(T arg)

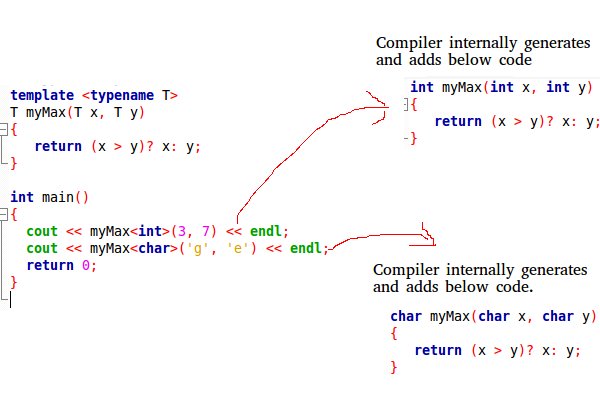
{

... .. ...

}

In the above code, T is a template argument that accepts different data types (int, float), and **class (Or typename)** is a keyword.

When, an argument of a data type is passed to someFunction( ), compiler generates a new version of someFunction() for the given data type.



### Class Templates

* Like function templates, you can also create class templates for generic class operations.
* Sometimes, you need a class implementation that is same for all classes, only the data types used are different.
* Normally, you would need to create a different class for each data type OR create different member variables and functions within a single class.
* This will unnecessarily bloat your code base and will be hard to maintain, as a change is one class/function should be performed on all classes/functions.
* However, class templates make it easy to reuse the same code for all data types.

**Syntax**

template <class T>

class className

{

... .. ...

public:

T var;

T someOperation(T arg);

... .. ...

};

* In the above declaration, **T** is the template argument which is a placeholder for the data type used.
* Inside the class body, a member variable **var** and a member function **someOperation()** are both of type T.

**How to create a class template object?**

To create a class template object, you need to define the data type inside a < > when creation.

className<dataType> classObject;

**For example:**

className<int> classObject;

className<float> classObject;

className<string> classObject;

**Simple calculator using <Class template>**

#include <iostream>

using namespace std;

template <class T>

class Calculator

{

private:

T num1, num2;

public:

Calculator(T n1, T n2)

{

num1 = n1;

num2 = n2;

}

void displayResult()

{

cout << "Numbers are: " << num1 << " and " << num2 << "." << endl;

cout << "Addition is: " << add() << endl;

cout << "Subtraction is: " << subtract() << endl;

cout << "Product is: " << multiply() << endl;

cout << "Division is: " << divide() << endl;

}

T add() { return num1 + num2; }

T subtract() { return num1 - num2; }

T multiply() { return num1 \* num2; }

T divide() { return num1 / num2; }

};

int main()

{

Calculator<int> intCalc(2, 1);

Calculator<float> floatCalc(2.4, 1.2);

cout << "Int results:" << endl;

intCalc.displayResult();

cout << endl << "Float results:" << endl;

floatCalc.displayResult();

return 0;

}

## this pointer (this\*)

### Definition

* The ‘this’ pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions.
* ‘this’ pointer is a constant pointer that holds the memory address of the current object.
* ‘this’ pointer is not available in static member functions as static member functions can be called without any object (with class name).

### Syntax

class ClassName {

private:

int dataMember;

public:

method(int a) {

// this pointer stores the address of object obj and access dataMember

this->dataMember = a;

... .. ...

}

}

int main() {

ClassName obj;

obj.method(5);

... .. ...

}

### Example

#include<iostream>

usingnamespace std;

class Box {

public:

// Constructor definition

Box(double l = 2.0, double b = 2.0, double h = 2.0) {

cout <<"Constructor called."<< endl;

length = l;

breadth = b;

height = h;

}

double Volume() {

return length \* breadth \* height;

}

int compare(Box box) {

returnthis->Volume() > box.Volume();

}

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

int main(void) {

Box Box1(3.3, 1.2, 1.5); // Declare box1

Box Box2(8.5, 6.0, 2.0); // Declare box2

if(Box1.compare(Box2)) {

cout <<"Box2 is smaller than Box1"<<endl;

} else {

cout <<"Box2 is equal to or larger than Box1"<<endl;

}

return 0;

}

## Virtual Destructor

### Definition

* Deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behavior.
* To correct this situation, the base class should be defined with a virtual destructor. For example, following program results in undefined behavior.

## Virtual Function

### Definition

A virtual function a member function which is declared within base class and is re-defined (Overridden) by derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class’s version of the function.

* Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
* They are mainly used to achieve[Runtime polymorphism](https://www.geeksforgeeks.org/polymorphism-in-c/)
* Functions are declared with a virtual keyword in base class.
* The resolving of function call is done at Run-time.

**Rules for Virtual Functions**

* They must be declared in public section of class.
* Virtual functions cannot be static and also cannot be a friend function of another class.
* Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
* The prototype of virtual functions should be same in base as well as derived class.
* They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
* A class may have [virtual destructor](https://www.geeksforgeeks.org/virtual-destructor/) but it cannot have a virtual constructor.

### Syntax

virtual returntype function\_name()

{

....

}

**Compile-time(early binding) VS run-time(late binding) behavior of Virtual Functions**

Consider the following simple program showing run-time behavior of virtual functions.

### Example

// CPP program to illustrate

// concept of Virtual Functions

#include<iostream>

using namespace std;

class base

{

public:

virtual void print()

{

cout << "print base class" << endl;

}

void show()

{

cout << "show base class" << endl;

}

};

class derived :public base

{

public:

void print()

{

cout << "print derived class" << endl;

}

void show()

{

cout << "show derived class" << endl;

}

};

int main()

{

base \*bptr;

derived d;

bptr = &d;

//virtual function, binded at runtime

bptr->print();

// Non-virtual function, binded at compile time

bptr->show();

}

**Output:**

print derived class

show base class

## Virtual Table(vtable) and \_vptr

### Definition

**Virtual Table (vtable)**

* Virtual Table is a lookup table of function pointers used to dynamically bind the virtual function to objects at runtime.
* Every class that uses virtual functions (or is derived from a class that uses virtual functions) is given its own virtual table as a secret data member.
* This table is set up by the compiler at compile time.
* A virtual table contains one entry as a function pointer for each virtual function that can be called by objects of the class.
* Virtual table stores NULL pointer to pure virtual functions.
* Virtual Table is created even for classes that have virtual base classes. In this case, the vtable has pointer to the shared instance of the base class along with the pointers to the classes’ virtual functions if any.

\_**vptr**

* This vtable pointer or \_vptr, is a hidden pointer added by the Compiler to the base class. And this pointer is pointing to the virtual table of that particular class.
* This \_vptr is inherited to all the derived classes.
* Each object of a class with virtual functions transparently stores this \_vptr.
* Call to a virtual function by an object is resolved by following this hidden \_vptr.
* Here is a simple example with a vtable representation.
* Here we have 3 classes Base, D1 and D2. Where D1 and D2 are derived from class Base.

### Syntax / Example

#include<iostream>

using namespace std;

class Base

{

public:

virtual void function1() { cout << "Base :: function1()\n"; };

virtual void function2() { cout << "Base :: function2()\n"; };

virtual ~Base(){};

};

class D1 : public Base

{

public:

~D1(){};

virtual void function1() { cout << "D1 :: function1()\n"; };

};

class D2 : public Base

{

public:

~D2(){};

virtual void function2() { cout << "D2 :: function2\n"; };

};

int main()

{

D1 \*d = new D1;

Base \*b = d;

b->function1();

b->function2();

delete (b);

return (0);

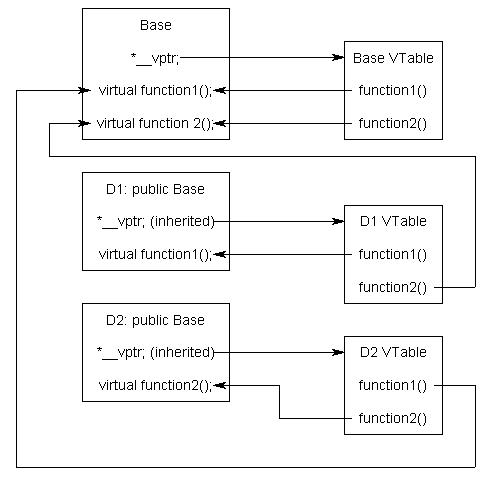
}

**output:**

D1::function1()

Base :: function2()

Here is a pictorial representation of Virtual Table and \_vptr for the above code:



Explanation :

Here in function main **b** pointer gets assigned to D1's \_vptr and now starts pointing to D1's vtable. Then calling to a **function1()**, makes it's \_vptr startightway calls D1's vtable **function1()** and so in turn calls D1's method i.e. **function1()** as D1 has it's own **function1()** defined it's class.   
   
Where as pointer **b** calling to a**function2()**, makes it's \_vptr points to D1's vatble which in-turn pointing to **Base**class's vtable **function2 ()** as shown in the diagram (as D1 class does not have it's own definition or **function2()**).   
   
So, now calling delete on pointer **b** follows the \_vptr - which is pointing to D1's vtable calls it's own class's destructor i.e. D1 class's destructor and then calls the destrcutor of Base class - this as part of when dervied object gets deleted it turn deletes it's emebeded base object. **Thats why we must always make Base class's destrcutor as virtual if it has any virtual functions in it.**   
   
Bootom Line:   
Thats how the Run-time or Dynamic binding happens on calling virtual functions of different derived objects.

## Volatile

### Definition

A volatile keyword is a qualifier which prevents the objects, from the compiler optimization and tells the compiler that the value of the object can be change at any time without any action being taken by the code. It prevents from the cache a variable into a register and ensures that on every access variable is fetched from the memory.

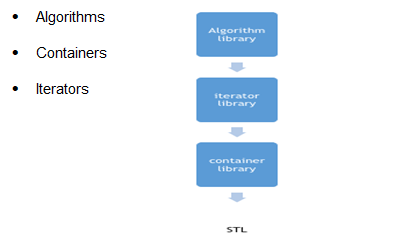
### Syntax / Example

int volatile iData;

volatile int iData;

# STL(Standard Template Library)

STL is an acronym for standard template library. It is a set of C++ template classes that provide generic classes and function that can be used to implement data structures and algorithms .STL is mainly composed of:



STL is a generic library , i.e. a same container or algorithm can be operated on any data types , you don’t have to define the same algorithm for different type of elements.

For example, sort algorithm will sort the elements in the given range irrespective of their data type , we don’t have to implement different sort algorithm for different data types.

**Containers in STL**

Container library in STL provide containers that are used to create data structures like arrays, linked list, trees etc.

These container are generic, they can hold elements of any data types, for example: vector can be used for creating dynamic arrays of char, integer, float and other types.

**Algorithms in STL**

STL provide number of algorithms that can be used of any container, irrespective of their type. Algorithms library contains built in functions that performs complex algorithms on the data structures.

For example: one can reverse a range with reverse() function, sort a range with sort() function, search in a range with binary\_search() and so on.

Algorithm library provides abstraction, i.e. you don't necessarily need to know how the algorithm works.

**Iterators in STL**

Iterators in STL are used to point to the containers. Iterators actually acts as a bridge between containers and algorithms.

For example: sort() algorithm have two parameters, starting iterator and ending iterator, now sort() compare the elements pointed by each of these iterators and arrange them in sorted order, thus it does not matter what is the type of the container and same sort() can be used on different types of containers.

## STL Container

### Array

Arrays, as we all know, are collection of homogenous objects. array container in STL provides us the implementation of static array, though it is rarely used in competitive programming as its static in nature but we'll still discuss array container cause it provides some member functions and non-member functions which gives it an edge over the array defined classically like, int array\_name[array\_size].

SYNTAX of array container:

array <object\_type, array\_size> array\_name;

The above code creates an empty array of object\_type with maximum size of array\_size. However, if you want to create an array with elements in it, you can do so by simply using the = operator.

Example of array container:

#include <vector>

int main()

{

array<int, 4> odd\_numbers = { 2, 4, 6, 8 };

}

**Member Functions of array template**

### at ( )

This method returns value in the array at the given range. If the given range is greater than the array size, out\_of\_range exception is thrown. Here is a code snippet explaining the use of this operator :

Example of at ( )

#include <iostream>

#include <array>

using namespace std;

int main()

{

array<int, 10> array1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

cout << array1.at(2); // prints 3

cout << array1.at(4); // prints 5

return 0;

}

### [ ] Operator

The use of operator [ ] is same as it was for normal arrays. It returns the value at the given position in the array. Example : In the above code, statement cout << array1[5]; would print 6 on console as 6 has index 5 in array1.

### front()

This method returns the first element in the array.

### back()

This method returns the last element in the array. The point to note here is that if the array is not completely filled, back() will return the rightmost element in the array.

#include <iostream>

#include <array>

using namespace std;

int main()

{

array<int, 10> array1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

cout << array1.front(); // prints 1

cout << array1.back(); // prints 9

return 0;

}

### fill()

This method assigns the given value to every element of the array, example :

#include <array>

int main(){

array<int, 8> myarray;

myarray.fill(1);

cout << "myarray is : ";

for (int i = 0; i < 8; i++) {

cout << myarray[i] << " ";

}

return 0;

}

/\* ouput will be

myarray is : 1 1 1 1 1 1 1 1\*/

### swap()

This method swaps the content of two arrays of same type and same size. It swaps index wise, thus element of index i of first array will be swapped with the element of index i of the second array, and if swapping any of the two elements throws an exception, swap() throws exception. Below is an example to demonstrate its usage :

#include<iostream>

#include<array>

using namespace std;

int main(){

array <int, 8> a = { 1, 2, 3, 4, 5, 6, 7, 8 };

array <int, 8> b = { 8, 7, 6, 5, 4, 3, 2, 1 };

a.swap(b); // swaps array a and b

cout << "a is : ";

for (int i = 0; i < 8; i++) {

cout << a[i] << " ";

}

cout << endl;

cout << "b is : ";

for (int i = 0; i < 8; i++) {

cout << a[i] << " ";

}

return 0;

/\* ouput will be

a is : 8 7 6 5 4 3 2 1

b is : 1 2 3 4 5 6 7 8 \*/

}

### operators ( == , != , > , < , >= , <= )

All these operators can be used to lexicographically compare values of two arrays.

### Empty()

This method can be used to check whether the array is empty or not.

Syntax : array\_name.empty(), returns true if array is empty else return false.

### Size()

This method returns the number of element present in the array.

### max\_size()

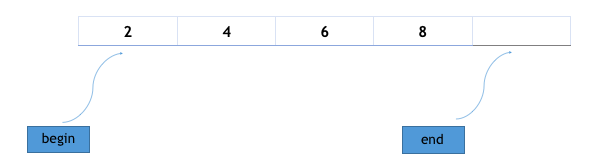
This method returns the maximum size of the array.

### begin()

This method returns the iterator pointing to the first element of the array. Iterators are just like pointers and we’ll discuss them later in the lessons, for now you can just think of an iterator like a pointer to the array.

### End

This method returns an iterator pointing to an element next to the last element in the array, for example the above array has 4 elements and the end() call will return the iterator pointing to the 4th index of the array.



# Vs (Differences)

## Inline Function Vs Macros

|  |  |  |
| --- | --- | --- |
| **Basis for Comparison** | **Inline Function** | **Macro** |
| Basic | Inline functions are parsed by the compiler. | Macros are expanded by the preprocessor. |
| Syntax | inline return\_typefunct\_name( parameters ){ . . . } | #define macro\_namechar\_sequence |
| Keywords Used | inline | #define |
| Defined | It can be defined inside or outside the class. | It is always defined at the start of the program. |
| Evaluation | It evaluates the argument only once. | It evaluates the argument each time it is used in the code. |
| Expansion | The compiler may not inline and expand all the functions. | Macros are always expanded. |
| Automation | The short functions, defined inside the class are automatically made onto inline functions. | Macros should be defined specifically. |
| Accessing | An inline member function can access the data members of the class. | Macros can never be the members of the class and can not access the data members of the class. |
| Termination | Definition of inline function terminates with the curly brackets at the end of the inline function. | Definition of macro terminates with the new line. |
| Debugging | Debugging is easy for an inline function as error checking is done during compilation. | Debugging becomes difficult for macros as error checking does not occur during compilation. |
| Binding | An inline function binds all the statements in the body of the function very well as the body of the function start and ends with the curly brackets. | A macro faces the binding problem if it has more than one statement, as it has no termination symbol. |

## Pointer Vs Reference

|  |  |  |
| --- | --- | --- |
| **Basis for Comparison** | **Pointer** | **Reference** |
| Basic | The pointer is the memory address of a variable. | The reference is an alias for a variable. |
| Returns | The pointer variable returns the value located at the address stored in pointer variable which is preceded by the pointer sign '\*'. | The reference variable returns the address of the variable preceded by the reference sign '&'. |
| Operators | \*, -> | & |
| Null Reference | The pointer variable can refer to NULL. | The reference variable can never refer to NULL. |
| Initialization | An uninitialized pointer can be created. | An uninitialized reference can never be created. |
| Time of Initialization | The pointer variable can be initialized at any point of time in the program. | The reference variable can only be initialized at the time of its creation. |
| Reinitialization | The pointer variable can be reinitialized as many times as required. | The reference variable can never be reinitialized again in the program. |

## new Vs malloc()

|  |  |  |  |
| --- | --- | --- | --- |
| **Basis for Comparison** | **new** | | **malloc()** |
| Language | | The operator new is a specific feature of C++, Java, and C#. | The function malloc( ) is a feature of C. |
| Nature | | "new" is an operator. | malloc( ) is a function. |
| sizeof( ) | | new doesn't need the sizeof operator as it allot enough memory for specific type | malloc requires the sizeof operator to know what memory size it has to allot. |
| Constructor | | Operator new can call the constructor of an object. | malloc( )can not at all make a call to a constructor. |
| Initialization | | The operator new could initialize an object while allocating memory to it. | Memory initialization could not be done in malloc. |
| Overloading | | Operator new can be overloaded. | The malloc( ) can never be overloaded. |
| Failure | | On failure, operator new throws an exception. | On failure, malloc( ) returns a NULL. |
| Deallocation | | The memory allocation by new, deallocated using "delete". | The memory allocation by malloc( ) is deallocated using a free( ) function. |
| Reallocation | | The new operator does not reallocate memory. | Memory allocated by malloc() can be reallocated using realloc(). |
| Execution | | The operator new cuts the execution time. | The malloc( ) requires more time for execution. |

## Copy Constructor Vs Assignment Operator

|  |  |  |
| --- | --- | --- |
| **Basis for Comparison** | **Copy Constructor** | **Assignment Operator** |
| Basic | The copy constructor is an overloaded constructor. | The assignment operator is a bitwise operator. |
| Meaning | The copy constructor initializes the new object with an already existing object. | The assignment operator assigns the value of one object to another object both of which are already in existence. |
| Syntax | class\_name(const class\_name& object\_name) { //body of the constructor } | class\_name Ob1, Ob2; Ob2=Ob1; |
| Invokes | (1)Copy constructor invokes when a new object is initialized with existing one. (2)The object passed to a function as a non-reference parameter. (3)The object is returned from the function.  (4)When C++ Compilecreats a temporary object. | The assignment operator is invoked only when assigning the existing object a new object. |
|  |  |  |
| Memory Allocation | Both the target object and the initializing object shares the different memory locations. | Both the target object and the initializing object shares same allocated memory. |
| Default | If you do not define any copy constructor in the program, C++ compiler implicitly provides one. | If you do not overload the "=" operator, then a bitwise copy will be made. |

## Semaphore Vs Mutex

|  |  |  |
| --- | --- | --- |
| **Basis for Comparison** | **Semaphore** | **Mutex** |
| Basic | Semaphore is a signalling mechanism. | Mutex is a locking mechanism. |
| Existence | Semaphore is an integer variable. | Mutex is an object. |
| Function | Semaphore allow multiple program threads to access a finite instance of resources. | Mutex allow multiple program thread to access a single resource but not simultaneously. |
| Ownership | Semaphore value can be changed by any process acquiring or releasing the resource. | Mutex object lock is released only by the process that has acquired the lock on it. |
| Categorize | Semaphore can be categorized into counting semaphore and binary semaphore. | Mutex is not categorized further. |
| Operation | Semaphore value is modified using wait() and signal() operation. | Mutex object is locked or unlocked by the process requesting or releasing the resource. |
| Resources Occupied | If all resources are being used, the process requesting for resource performs wait() operation and block itself till semaphore count become greater than one. | If a mutex object is already locked, the process requesting for resources waits and queued by the system till lock is released. |

# Summary in Tables

## Storage Classes

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Automatic** | **Local Static** | **Global** | **Global Static** |
| **Visibility/Scope** | Function | Function | Program (used by other file using extern) | File |
| **Lifetime** | Function | Program | Program | Program |
| **Initialization Value** | Not Initialized | 0 | 0 | 0 |
| **Storage** | Stack | Static Memory | Static Memory | Static Memory |
| **Purpose** | Variables used by single function | Retains value when the function terminates | Variable used by several functions | Variable used by several functions on same file. |
| **Linkage** | No Linkage | No Linkage | External | Internal |

## Visibility After Inheritance

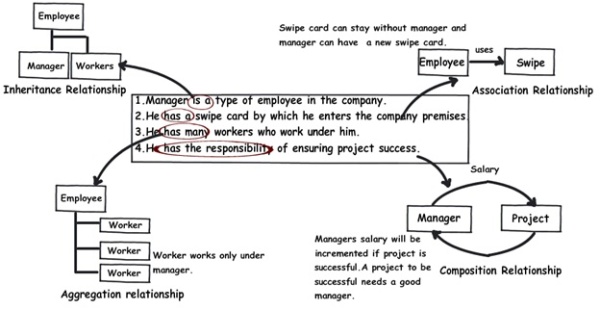
|  |  |  |  |
| --- | --- | --- | --- |
|  | **Derived Class** | **Derived Class** | **Derived Class** |
| **Base class** | **Public Mode** | **Private Mode** | **Protected Mode** |
| **Private** | Not Inherited | Not Inherited | Not Inherited |
| **Protected** | Protected | Private | Protected |
| **Public** | Public | Private | Protected |

## Summary of Function Behavior

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Function Type** | **Is Function Inherited from Base Class?** | **Can Function Be Virtual?** | **Can Function Return a Value?** | **Is Function a Member or Friend?** | **Will Compiler Generate Function if User Does Not?** |
| **Constructor** | No | No | No | Member | Yes |
| **Copy Constructor** | No | No | No | Member | Yes |
| **Destructor** | No | Yes | No | Member | Yes |
| **Conversion** | Yes | Yes | No | Member | No |
| **Assignment (operator=)** | No | Yes | Yes | Member | Yes |
| **new** | Yes | No | void\* | Static member | No |
| **delete** | Yes | No | void | Static member | No |
| **Other member functions** | Yes | Yes | Yes | Member | No |
| **Friend functions** | No | No | Yes | Friend | No |

## Association, Aggregation, and Composition

Below is a visual representation of how the relationships have emerged from the requirements.



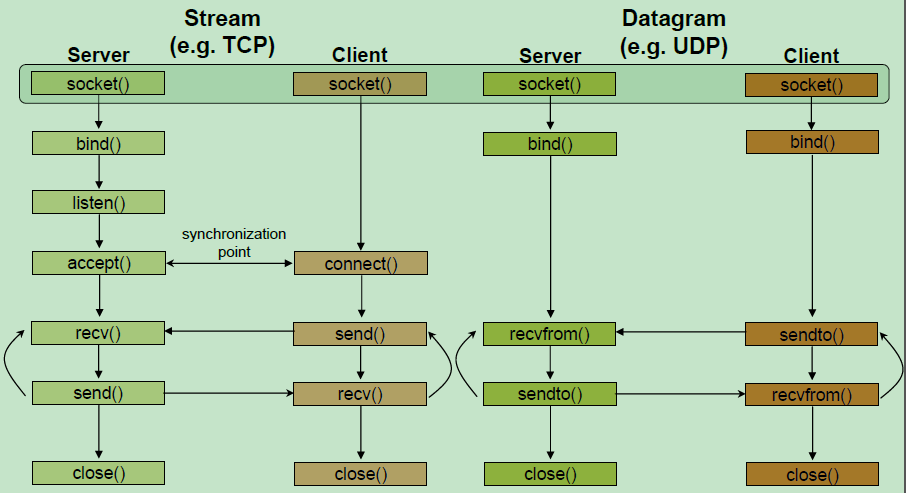
|  |  |  |  |
| --- | --- | --- | --- |
|  | **Association** | **Aggregation** | **Composition** |
| **Owner** | No owner | Single owner | Single owner |
| **Life time** | Have their own lifetime | Have their own lifetime | Owner's life time |
| **Child object** | Child objects all are independent | Child objects belong to a single parent | Child objects belong to a single parent |

## Sequence Containers (Vector, List, Deque)

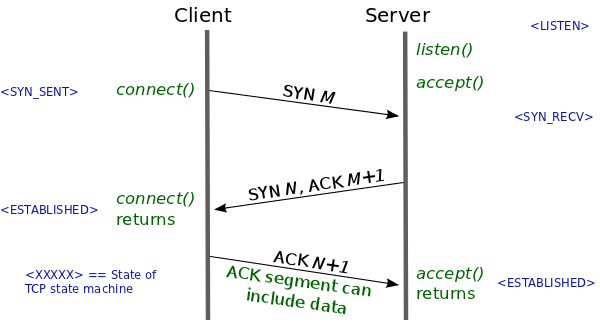
|  |  |  |  |
| --- | --- | --- | --- |
|  | **Chareteristics** | **Features** | **Iterators** |
| **Vector** | Relocating, Expandable Array | * Quick random access by index number. * Slow insert/delete in the middle. * Constant time insert/delete at the end. | Random Access |
| **List** | Doubly Linked List | * Slow random access. * Constant Time insert/delete at any location. | Bidirectional |
| **Deque** | Like Vector but can be accessed at either end. | * Quick random access by index number. * Slow insert/delete in the middle. * Constant time insert/delete (Push/Pop) at either beginning or at the end. | Random Access |

# Networking

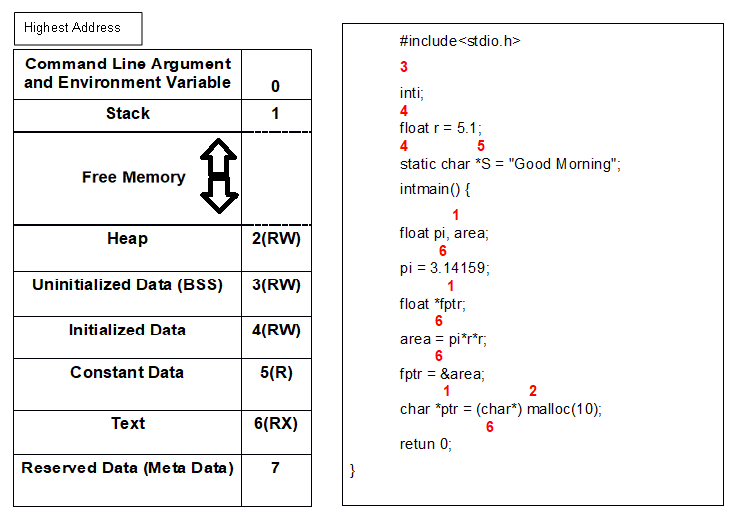
## Client-Server Communication



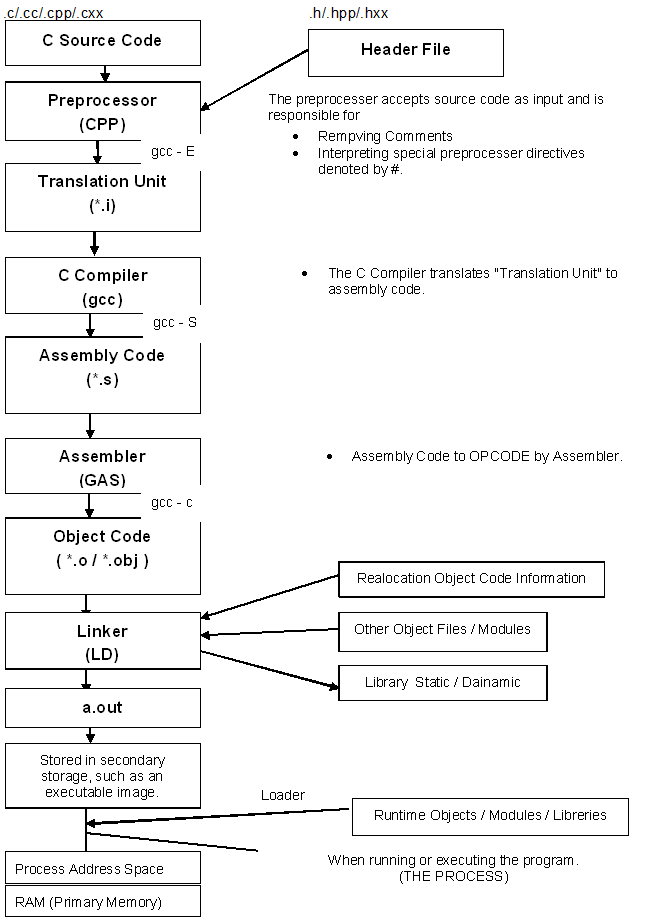
## TCP 3-Way HandShake



# Memory Layout of C Programs



# The Build Process - C/C++



# Important C Programs

## String Functions

### strlen()

#include<stdio.h>

#include<string.h>

#include "cassert"

int my\_strlen(constchar\*);

int main(){

int n=0;

constchar \*str="Jyoti";

n=my\_strlen(str);

printf("The length of the string \"%s\" is %d\n",str,n);

return 0;

}

int my\_strlen(constchar \*lptr){

assert (lptr);

int length=0;

while(\*lptr++){

length++;

}

return length;

}

### strcpy()

#include<stdio.h>

#include<stdlib.h>

#include "cassert"

char\* my\_strcpy(char\*,constchar\*);

int main(){

constchar source[] = "Jyoti";

char destination[40]="Ranjan";

my\_strcpy(destination,source);

printf("The Destination is \"%s\" \n",destination);

return 0;

}

char\* my\_strcpy(char \*dest,constchar \*src){

assert (dest && src);

char\* temp=dest;

while(\*dest++ = \*src++);

return temp;

}

### strcat ()

#include<stdio.h>

#include<stdlib.h>

#include "cassert"

char\* my\_strcat(char\*,constchar\*);

int main(){

constchar source[] = "ranjan";

char destination[40]="Jyoti";

my\_strcat(destination,source);

printf("The Destination is \"%s\" \n",destination);

return 0;

}

char\* my\_strcat(char \*dest,constchar \*src){

assert (dest && src);

char\* temp=dest;

if(\*dest!=0){

while(\*++dest);

}

while(\*dest++ = \*src++);

return temp;

}

### strcmp ()

#include<stdio.h>

#include<stdlib.h>

#include "cassert"

int my\_strcmp(constchar\*,constchar\*);

int main(){

int n=0;

constchar source[] = "J";

constchar destination[]="j";

n=my\_strcmp(destination,source);

printf("N is \"%d\" \n",n);

return 0;

}

int my\_strcmp(constchar \*s1, constchar \*s2) {

assert (s1 && s2);

while (\*s1 && \*s2) {

if (\*s1 == \*s2) {

++s1;

++s2;

}

else {

break;

}

}

return \*s1 - \*s2;

}

### atoi ()

#include<stdio.h>

#include<stdlib.h>

int my\_atoi(constchar\*);

int main(){

const char \*str = "1123.56";

int n = my\_atoi(str);

//int n = atoi(str);

printf("The string %s as an integer is = %d\n",str,n);

return 0;

}

int my\_atoi(const char \*lptr){

int result=0;

while(1){

if ((\*lptr >= '0') && (\*lptr <= '9')){

result \*= 10;

result += \*lptr - '0';

lptr++;

}

else{

break;

}

}

return result;

}

## count how many tokens are there in given string

#include <stdio.h>

#include <string.h>

char str[100], sub[100];

int count = 0, count1 = 0;

void main(){

int i, j, len\_str, len\_sub;

printf("\nEnter a string : ");

scanf("%[^\n]s", str);

len\_str = strlen(str);

printf("\nEnter a substring : ");

scanf(" %[^\n]s", sub);

len\_sub = strlen(sub);

for (i = 0; i < len\_str;){

j = 0;

count = 0;

while ((str[i] == sub[j]) && (str[i])){

count++;

i++;

j++;

}

if (count == len\_sub){

count1++;

count = 0;

}

else

i++;

}

printf("%s occurs %d times in %s", sub, count1, str);

}

## Swap two variables using macro

#include<stdio.h>

#include<conio.h>

#define SWAPE(x,y) int t;t=x;x=y;y=t;

int main()

{

int a,b;

printf("\n Enter two numbers");

scanf("%d%d",&a,&b);

printf("\n Before swaping the Value of a=%d and b=%d",a,b);

SWAPE(a,b);

printf("\n After swap value of a=%d and b=%d",a,b);

return 0;

}

## Multiply by 5 with given number using bitwise operator

#include<stdio.h>

#include<conio.h>

int main(){

int a;

printf("\n Enter One number");

scanf("%d",&a);

printf("\n After multiply by 5, the number is %d",(a<<2)+a);

return 0;

}

## Toggle the k- th bit of a given number using bitwise operator

// CPP program to toggle k-th bit of n

#include<iostream>

using namespace std;

int toggleKthBit(int n, int k)

{

return (n ^ (1 << (k - 1)));

}

// Driver code

int main()

{

int n = 5, k = 1;

cout << toggleKthBit(n, k);

return 0;

**}**

## REVERSE A NUMBER USING ARITHMATIC OPERATOR

#include<stdio.h>

int main() {

int num,rev=0;

printf("Enter a number:");

scanf("%d", &num);

while(num>0){

rev=(rev\*10)+ (num%10);

num=num/10;

}

printf("Reverse of the given number: %d", rev);

return 0;

}

## PRINT THE GIVEN COMMAND LINE ARGUMENTS

#include<stdio.h>

int main(int argc,char \*\*argv){

int i;

printf("argc=%d\n",argc);

for(i=0;i<argc;++i)

printf("argv[%d] = %s\n",i,argv[i]);

return 0;

}

## Write a line backward

#include<stdio.h>

void wrt\_it();

int main(){

printf("Type a line ...");

wrt\_it();

return 0;

}

void wrt\_it(){

int c;

if((c=getchar())!='\n')

wrt\_it();

putchar(c);

}

## Print 12345678910987654321

#include<stdio.h>

int main(){

int i=1,j=1;

while(i<20){

if(i<=10){

printf("%d",i);

j=i-1;

i++;

}else{

printf("%d",j);

j--;

i++;

}

}

return 0;

}

## Variable Length Arguments

#include<stdio.h>

#include<stdarg.h>

void sum(char \*, int, ...);

int main(void){

sum("The sum of 10+15+13 is %d.\n",3,10,15,13);

return 0;

}

void sum(char \*string,int num\_args,...){

int sum=0;

va\_list ap;

int loop;

va\_start(ap,num\_args);

for(loop=0;loop<num\_args;loop++)

sum+=va\_arg(ap,int);

printf(string,sum);

va\_end(ap);

}

## Variable Length Arguments

#include<stdio.h>

#include<stdarg.h>

void printargs(int arg1, ...) /\*print all int type args, finishing with -1 \*/

{

va\_list ap;

int i;

va\_start(ap, arg1);

for (i = arg1; i != -1; i = va\_arg(ap, int))

printf("%d ", i);

va\_end(ap);

putchar('\n');

}

int main(void)

{

printargs(5, 2, 14, 84, 97, 15, 24, 48, -1);

printargs(84, 51, -1);

printargs(-1);

printargs(1, -1);

return 0;

}

## swap two numbers without using temp variable

#include<stdio.h>

int main(){

int a=5;

int b=10;

a=a^b;

b=a^b;

a=a^b;

printf("a=%d",a);

printf("b=%d",b);

return 0;

}

## function to dynamically allocate 2-dimensional array using malloc.

void allocate2D(int\*\* arr, int nrows, int ncols) {

arr = (int\*\*)malloc(nrows\*sizeof(int\*));

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

arr[i]=(int\*)malloc(ncols\*sizeof(int));

}

}

## Palindrome Check

#include<stdio.h>

#include<string.h>

int check\_Palindrome(char\*);

int main(){

char name[1024];

printf("Enter a string for Palindrome Check \n");

scanf("%s",name);

if(check\_Palindrome(name))

printf("This is a Palindrome\n");

else

printf("This is not a Palindrome\n");

return 0;

}

int check\_Palindrome(char\* str){

int num=(int)strlen(str);

for(int i=0;i<num/2;i++){

if(str[i]!=str[num-(i+1)]){

return 0;

}

}

return 1;

}

## Count no of ‘1’ in a given integer

#include<stdio.h>

int main (void){

int var = 7;

int i, count = 0;

while(var > 0)

{

count += var & 1;

var >>= 1;

}

printf("%d\n", count);

return 0;

}

## Without Using “sizeof” operator print size of given structure

#include<stdio.h>

struct MyStruct

{

int i;

int j;

char a;

};

int main()

{

struct MyStruct \*p=0;

int size = ((char\*)(p+1))-((char\*)p);

printf("\nSIZE : %d\n", size);

return 0;

}

## Arrenge 0’s and 1’s in given array.

#include<stdio.h>

int main(){

int i,a[10]={0};

int \*r,\*w;

a[2]=a[4]=a[6]=a[8]=1;

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

printf("%d",a[i]);

r=w=a;

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

if(\*r==1){

\*w=1;

\*r=0;

++w;

}

++r;

}

printf("\n\nafter\n");

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

printf("%d",a[i]);

return 0;

}

## Factorila Of a Number Using Recursion

#include<stdio.h>

int my\_factorial(int);

int main(){

printf("%d",my\_factorial(10));

return 0;

}

int my\_factorial(int a){

if(a <=1){

return 1;

}else{

return (a\*my\_factorial(a-1));

}

}

## fibonacci Series

#include <stdio.h>

/\*

0 1 2 3 4 ...

0 1 1 2 3 ...

\*/

// Recursive Methhod

static int fibonacci\_r(int n)

{

if (n <= 1) {

return n;

}

return fibonacci\_r(n - 1) + fibonacci\_r(n - 2);

}

// Iteration Method

static int fibonacci(unsigned int n)

{

if (n <= 1) {

return n;

}

int n1 = 1, n2 = 1;

for (int ii = 2; ii < n; ++ii){

int n3 = n1 + n2;

n1 = n2;

n2 = n3;

}

return n2;

}

int main(int argc, char\* argv[])

{

int number = atoi(argv[1]);

printf("fibonacci of %d = %d\n", fibonacci(number));

retrun 0;

}

## Arrays: Left Rotation

#include<iostream>

int main(){

int arr[] = { 1, 2, 3, 4, 5 };

int n = 5; //Number of elements in array

int k = 7; //Number of Left\_Rotation

k = k%n;

for (int i = k; i < n; i++)

std::cout << arr[i];

for (int j = 0; j < k;j++)

std::cout << arr[j];

return 0;

}

## string concatenation (strcat) in c using macro

#include <stdio.h>

#define conc(str1,str2) #str1 ## #str2

int main(){

printf("The Destination is \"%s\" \n", conc(idan, oop));

return 0;

}

## What is the sizeof() empty main() or any empty function() ?

#include <stdio.h>

int fun(){

return 0;

}

int main(void)

{

printf("%u\n", sizeof (&main));

printf("%u\n", sizeof (main()));

printf("%u\n", sizeof (fun()));

return 0;

}

**OUTPUT:**

4

4

4

## Implement Your Own sizeof()

#include <stdio.h>

#define my\_sizeof(type) (char \*)(&type+1)-(char\*)(&type)

int main()

{

int x;

printf("%d", my\_sizeof(x));

return 0;

}

**OUTPUT:**

4

## Implement Your Own toupper() and tolower()

#include <stdio.h>

int my\_toupper(int ch){

if (ch >= 'a' && ch <= 'z')

return('A' + ch - 'a');

else

return(ch);

}

int my\_tolower(int ch){

if (ch >= 'A' && ch <= 'Z')

return('a' + ch - 'A');

else

return(ch);

}

int main(){

char \*ch = "Jyotiranjan Mohanty";

while (\*ch != '\0') {

// converting char to upper case

if (\*ch >= 'A' && \*ch <= 'Z'){

char ch1 = my\_tolower(\*ch);

printf("%c", ch1);

//printf("%c is in Upper case", ch);

}

else if (\*ch >= 'a' && \*ch <= 'z'){

char ch2 = my\_toupper(\*ch);

printf("%c", ch2);

//printf("%c is in Lower case", ch);

}

else

printf(" ");

ch++;

}

return 0;

}

## How to find length of a string without string.h and loop in C?

#include <stdio.h>

int main(){

char str[100];

printf("Enter a string: ");

printf("\rLength is: %d",

printf("Entered string is: %s\n", gets(str)) - 20

);

//"Entered string is: \n" -> Length is 20

return 0;

}

**OUTPUT:**

Enter a string: hello

Entered string is: hello

Length is: 5

## Write a C program to find the GCD of two numbers

#include < stdio.h >

int gcd(int a, int b);

int gcd\_recurse(int a, int b);

int main(){

printf("\nGCD(%2d,%2d) = [%d]", 6, 4, gcd(6, 4));

printf("\nGCD(%2d,%2d) = [%d]", 4, 6, gcd(4, 6));

printf("\nGCD(%2d,%2d) = [%d]", 3, 17, gcd(3, 17));

printf("\nGCD(%2d,%2d) = [%d]", 17, 3, gcd(17, 3));

printf("\nGCD(%2d,%2d) = [%d]", 1, 6, gcd(1, 6));

printf("\nGCD(%2d,%2d) = [%d]", 10, 1, gcd(10, 1));

printf("\nGCD(%2d,%2d) = [%d]", 10, 6, gcd(10, 6));

printf("\n----------------");

printf("\nGCD(%2d,%2d) = [%d]", 6, 4, gcd\_recurse(6, 4));

printf("\nGCD(%2d,%2d) = [%d]", 4, 6, gcd\_recurse(4, 6));

printf("\nGCD(%2d,%2d) = [%d]", 3, 17, gcd\_recurse(3, 17));

printf("\nGCD(%2d,%2d) = [%d]", 17, 3, gcd\_recurse(17, 3));

printf("\nGCD(%2d,%2d) = [%d]", 1, 6, gcd\_recurse(1, 6));

printf("\nGCD(%2d,%2d) = [%d]", 10, 1, gcd\_recurse(10, 1));

printf("\nGCD(%2d,%2d) = [%d]", 10, 6, gcd\_recurse(10, 6));

return(0);

}

**// Iterative algorithm**

int gcd(int a, int b){

int temp;

while (b)

{

temp = a % b;

a = b;

b = temp;

}

return(a);

}

**// Recursive algorithm**

int gcd\_recurse(int a, int b){

int temp;

temp = a % b;

if (temp == 0)

{

return(b);

}

else

{

return(gcd\_recurse(b, temp));

}

}

# Important DataStructure Programs

## To get the middle of the linked list

/\* Link list node \*/

struct Node

{

int data;

struct Node\* next;

};

/\* Function to get the middle of the linked list\*/

void printMiddle(struct Node \*head)

{

struct Node \*slow\_ptr = head;

struct Node \*fast\_ptr = head;

if (head!=NULL)

{

while (fast\_ptr != NULL && fast\_ptr->next != NULL)

{

fast\_ptr = fast\_ptr->next->next;

slow\_ptr = slow\_ptr->next;

}

printf("The middle element is [%d]\n\n", slow\_ptr->data);

}

}

## Program to check loop in single linked list

void findloop(struct node \*head) {

struct node \*slow, \*fast;

slow = fast = head;

while(slow && fast && fast->next) {

/\* Slow pointer will move one node per iteration whereas

fast node will move two nodes per iteration \*/

slow = slow->next;

fast = fast->next->next;

if (slow == fast) {

printf("Linked List contains a loop\n");

return;

}

}

printf("No Loop in Linked List\n");

}

## Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Example:

First Pass:

( 5 1 4 2 8 ) –> ( 1 5 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.

( 1 5 4 2 8 ) –> ( 1 4 5 2 8 ), Swap since 5 > 4

( 1 4 5 2 8 ) –> ( 1 4 2 5 8 ), Swap since 5 > 2

( 1 4 2 5 8 ) –> ( 1 4 2 5 8 ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

( 1 4 2 5 8 ) –> ( 1 4 2 5 8 )

( 1 4 2 5 8 ) –> ( 1 2 4 5 8 ), Swap since 4 > 2

( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )

( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Third Pass:

( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )

( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )

( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )

( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )

**// C program for implementation of Bubble sort**

#include <stdio.h>

void swap(int \*xp, int \*yp){

int temp = \*xp;

\*xp = \*yp;

\*yp = temp;

}

// A function to implement bubble sort

**void bubbleSort(int arr[], int n){**

**int i, j;**

**for (i = 0; i < n - 1; i++)**

**// Last i elements are already in place**

**for (j = 0; j < n - i - 1; j++)**

**if (arr[j] > arr[j + 1])**

**swap(&arr[j], &arr[j + 1]);**

**}**

/\* Function to print an array \*/

void printArray(int arr[], int size){

int i;

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

printf("%d ", arr[i]);

}

// Driver program to test above functions

int main(){

int arr[] = { 64, 34, 25, 12, 22, 11, 90 };

int n = sizeof(arr) / sizeof(arr[0]);

bubbleSort(arr, n);

printf("Sorted array: \n");

printArray(arr, n);

return 0;

}

OUTPUT::

Sorted array:

11 12 22 25 34 64 90

# Multi-Threading

## What Is a Thread?

* A thread is a semi-process, that has its own stack, and executes a given piece of code.
* Unlike a real process, the thread normally shares its memory with other threads (where as for processes we usually have a different memory area for each one of them).
* A Thread Group is a set of threads all executing inside the same process. They all share the same memory, and thus can access the same global variables, same heap memory, same set of file descriptors, etc.
* All these threads execute in parallel (i.e. using time slices, or if the system has several processors, then really in parallel).
* Threads in the same process share:
  + Process instructions
  + Most data
  + open files (descriptors)
  + signals and signal handlers
  + current working directory
  + User and group id
* Each thread has a unique:
  + Thread ID
  + set of registers, stack pointer
  + stack for local variables, return addresses
  + signal mask
  + priority
  + Return value: errno
* pthread functions return "0" if OK.
  + Because threads within the same process share resources:
  + Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
  + Two pointers having the same value point to the same data.
  + Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.

## pthread\_create()

### SYNOPSIS

#include <pthread.h>

int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start)(void \*), void \*arg);

### PARAMETERS

* thread

Is the location where the ID of the newly created thread should be stored, or NULL if the thread ID is not required.

* attr

Is the thread attribute object specifying the attributes for the thread that is being created. If attr is NULL, the thread is created with default attributes.

* start

Is the main function for the thread; the thread begins executing user code at this address.

* arg

Is the argument passed to start.

### Program

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* FILE: Pthread\_Creation.c

\* DESCRIPTION:

\* A "hello world" Pthreads program. Demonstrates thread creation and termination.

\* AUTHOR: Jyotiranjan

\* LAST REVISED: 14/Dec/2017

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#define NUM\_THREADS 5

void \*PrintHello(void \*threadid)

{

long tid;

tid = (long)threadid;

printf("Hello World! It's me, thread #%ld!\n", tid);

pthread\_exit(NULL);

}

int main(int argc, char \*argv[])

{

pthread\_t threads[NUM\_THREADS];

int rc;

long t;

for(t=0;t<NUM\_THREADS;t++){

printf("In main: creating thread %ld\n", t);

rc = pthread\_create(&threads[t], NULL, PrintHello, (void \*)t);

printf(" thread\_ID %ld\n", threads[t]);

if (rc){

printf("ERROR; return code from pthread\_create() is %d\n", rc);

exit(-1);

}

}

/\* Last thing that main() should do \*/

pthread\_exit(NULL);

}

### Run The Program

[root@localhost threads]# gcc Pthread\_Creation.c -o Pthread\_Creation -lpthread

[root@localhost threads]# ./Pthread\_Creation

### Output

In main: creating thread 0

thread\_ID 139851122005760

In main: creating thread 1

thread\_ID 139851113613056

In main: creating thread 2

thread\_ID 139851105220352

In main: creating thread 3

thread\_ID 139851096827648

In main: creating thread 4

thread\_ID 139851088434944

Hello World! It's me, thread #4!

Hello World! It's me, thread #3!

Hello World! It's me, thread #2!

Hello World! It's me, thread #1!

Hello World! It's me, thread #0!

# SHELL Scripting

## Environment Variable

|  |  |
| --- | --- |
| **Variable name** | **Purpose** |
| $0 | the name of the shell program |
| $1 thru $9 | the first thru to ninth parameters |
| $# | the number of parameters |
| $\* | all the parameters passed represented as a single word with individual parameters separated |
| $@ | all the parameters passed with each parameter as a separate word |
| $? | hold the exit status of the previous command |
| $$ | the process id of the current process |

### Example

shift.sh

#!/bin/sh

echo $@

echo $# $\*

shift

echo $# $\*

shift

echo $# $\*

shift

echo Script Name is $0

echo $?

echo $$

**Output**

./shift.sh hello jyoti ranjan

hello jyoti ranjan

3 hello jyoti ranjan

2 jyoti ranjan

1 ranjan

Script Name is ./shift.sh

0

52248

## " if " conditional commands

In the following

command represents a single command,

command-list represents one or more commands,

The numbers 1 and 2 are used to indicate different commands.

### Syntax

if command1

then

command-list

fi

------------------------------------------------------

if command1

then

command-list1

elif command1

command-list2

fi

-------------------------------------------------------

if command1

then

command-list1

else

command-list2

fi

### Example

#!/bin/sh

a=10

b=20

if [ $a == $b ]

then

echo "a is equal to b"

elif [ $a -gt $b ]

then

echo "a is greater than b"

elif [ $a -lt $b ]

then

echo "a is less than b"

else

echo "None of the condition met"

fi

**Output**

a is less than b

# Design Patterns

## Singleton Design Pattern

### Definition

* Only one instance for a singleton class.
* The class has to provide global point of access to the unique instance.

# Algorithms

## Time Complexities of all Sorting Algorithms

Following is a quick revision sheet that you may refer at last minute

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | |
|  | **Best** | **Average** | **Worst** |
| [Selection Sort](http://geeksquiz.com/selection-sort/) | Ω(n^2) | θ(n^2) | O(n^2) |
| [Bubble Sort](http://geeksquiz.com/bubble-sort/) | Ω(n) | θ(n^2) | O(n^2) |
| [Insertion Sort](http://geeksquiz.com/insertion-sort/) | Ω(n) | θ(n^2) | O(n^2) |
| [Heap Sort](http://geeksquiz.com/heap-sort/) | Ω(n log(n)) | θ(n log(n)) | O(n log(n)) |
| [Quick Sort](http://geeksquiz.com/quick-sort/) | Ω(n log(n)) | θ(n log(n)) | O(n^2) |
| [Merge Sort](http://geeksquiz.com/merge-sort/) | Ω(n log(n)) | θ(n log(n)) | O(n log(n)) |
| [Bucket Sort](https://www.geeksforgeeks.org/bucket-sort-2/) | Ω(n+k) | θ(n+k) | O(n^2) |
| [Radix Sort](https://www.geeksforgeeks.org/radix-sort/) | Ω(nk) | θ(nk) | O(nk) |

# git

## Commands

### git init

Initialize a local Git repository. .git folder will be created in your local drive.

### git remote add origin

* Example

git remote add origin "https://github.com/MohantyOne/mohantys\_git.git"

* Description

Add a remote repository

### git pull <branch\_name> <remote\_URL/remote\_name>

* Example

git pull origin master

* Description

To get the latest version of a repository run git pull. This pulls the changes from the remote repository to the local computer.

### git status

* Example

git status

* Description

git status will return the current working branch. If a file is in the staging area, but not committed, it shows with git status. Or, if there are no changes it’ll return nothing to commit, working directory clean.

### git add

* Example

# To add all files not staged:

$ git add **.**

**OR**

$ git add **-**A

# To stage a specific file:

$ git add index.html

# To stage an entire directory:

$ git add css

* Description

Adds files in to the staging area for Git. Before a file is available to commit to a repository, the file needs to be added to the Git index (staging area). There are a few different ways to use git add, by adding entire directories, specific files, or all unstaged files.

### git commit

* Example

# Adding a commit with message

$ git commit -m "Commit message in quotes"

# Adding all commit with message

$ git commit **.** -m "Commit message in quotes"

OR

$ git commit -a -m "Commit message in quotes"

* Description

Record the changes made to the files to a local repository. For easy reference, each commit has a unique ID.

Its best practice to include a message with each commit explaining the changes made in a commit. Adding a commit message helps to find a particular change or understanding the changes.