**Class:**

**What?**

Class is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

**When and why?**

When we want many properties and data to be stored in a variable we can use class and the variable is the object in the class.

**How?**

Syntax:

**class** class\_Name {

//access specifier

// Data\_Member

// Member\_Function

};

class\_Name ObjectName;

Accessing data members and member function:

Data member and member function of a class can be accessed using (.)dot operator. Ex, lets say obj is the object name and trying to access the function printData() in the class.

obj.printData();

There are three access specifier,

* public
* protected
* private

**Program**

**#01\_Pgm**

#include<iostream>

#include<string>

class Data

{

public:

std::string name;

void printName(){

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

}

};

int main()

{

Data obj;

obj.name = "Kumar";

obj.printName();

return 0;

}

Output:

Name is Kumar

**Constructor:**

Its a special member function that is called by the compiler when ever a object for a class is instantiated.

Types of constructor:

1. Default constructor

2. Parameterized constructor

3. Copy constructor

**why constructor?**

It can be used to initialize the data members of a class.

**Ex:**

class Point

{

public:

Point(){

x = 0.0;

y = 1.1;

}

double x, y;

};

**Default constructor:**

It is the constructor which doesn’t take any argument. It has no parameters.

**#02\_Pgm**

#include <iostream>

using namespace std;

class construct {

public:

int a, b;

// Default Constructor

construct()

{

a = 10;

b = 20;

}

};

int main()

{

// Default constructor called automatically

// when the object is created

construct c;

cout << "a: " << c.a << endl

<< "b: " << c.b << endl;

return 1;

}

Output:

a: 10

b: 20

**Parameterized Constructor:**

It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor’s body, use the parameters to initialize the object.

Uses:

1. It is used to initialize the various data elements of different objects with different values when they are created.   
2. It is used to overload constructors.

**#03\_Pgm**

#include <iostream>

using namespace std;

class Point {

private:

int x, y;

public:

// Parameterized Constructor

Point(int x1, int y1)

{

x = x1;

y = y1;

}

int getX()

{

return x;

}

int getY()

{

return y;

}

};

int main()

{

// Constructor called

Point p1(10, 15);

// Access values assigned by constructor

cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

return 0;

}

Output:

p1.x = 10, p1.y = 15

**Copy Constructor:**

Its a member function which initialize an object using another object of the same class.

Syntax: ClassName (const ClassName &old\_obj){ //define }

#### When is copy constructor called:

1. When object of class is returned by value.

2. When an object of the class is passed (to a function) by value as an argument.

3. When an object is constructed based on another object of the same class.

4. When the compiler generates a temporary object.

#### #04\_Pgm

#include<iostream>

using namespace std;

class Point

{

private:

int x, y;

public:

Point(int x1, int y1) { x = x1; y = y1; }

// Copy constructor

Point(const Point &p2) {x = p2.x; y = p2.y; }

int getX() { return x; }

int getY() { return y; }

};

int main()

{

Point p1(10, 15); // Normal constructor is called here

Point p2 = p1; // Copy constructor is called here

// Let us access values assigned by constructors

cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY() << endl;

return 0;

}

Output:

p1.x = 10, p1.y = 15

p2.x = 10, p2.y = 15

**#05\_Pgm**

#include<iostream>

#include<cstring>

using namespace std;

class String

{

private:

char \*s;

int size;

public:

String(const char \*str = NULL); // constructor

~String() { delete [] s; }// destructor

String(const String&); // copy constructor

void print() { cout << s << endl; } // Function to print string

void change(const char \*); // Function to change

};

String::String(const char \*str)

{

size = strlen(str);

s = new char[size+1];

strcpy(s, str);

}

void String::change(const char \*str)

{

delete [] s;

size = strlen(str);

s = new char[size+1];

strcpy(s, str);

}

String::String(const String& old\_str)

{

size = old\_str.size;

s = new char[size+1];

strcpy(s, old\_str.s);

}

int main()

{

String str1("Hello World!");

String str2 = str1;

str1.print(); // what is printed ?

str2.print();

str2.change("Hello Universe!");

str1.print(); // what is printed now ?

str2.print();

return 0;

}

Output:

Hello World!

Hello World!

Hello World!

Hello Universe!

Lets try the same program without the copy constructor in it,

**#06\_Pgm**

#include<iostream>

#include<cstring>

using namespace std;

class String

{

private:

char \*s;

int size;

public:

String(const char \*str = NULL); // constructor

~String() { delete [] s; }// destructor

void print() { cout << s << endl; } // Function to print string

void change(const char \*); // Function to change

};

String::String(const char \*str)

{

size = strlen(str);

s = new char[size+1];

strcpy(s, str);

}

void String::change(const char \*str)

{

delete [] s;

size = strlen(str);

s = new char[size+1];

strcpy(s, str);

}

int main()

{

String str1("Hello World!");

String str2 = str1;

str1.print(); // what is printed ?

str2.print();

str2.change("Hello Universe!");

str1.print(); // what is printed now ?

str2.print();

return 0;

}

Output:

Hello World!

Hello World!

Hello Universe!

Hello Universe!

Aborted (core dump)

Reason:

----------

Shallow copy is happening here hence a change in the str2 is reflected in str1, in the #05\_Pgm we create a new memory and copied the data (deep copy) hence the data was not modified.

Can we make copy constructor private?

Yes, a copy constructor can be made private. When we make a copy constructor private in a class, objects of that class become non-copyable.

Why argument to copy constructor must be passed as a reference?

Without reference we cannot have a copy constructor as pass by value would lead to recursive call of the copy constructor infinitely, Hence we should go with the reference.

Why do we need const in the copy constructor?

In order to prevent accidental modification of the data that has to be copied to constructor.

Note:

#include <iostream>

using namespace std;

class Point

{

int x, y;

public:

Point(const Point &p) { x = p.x; y = p.y; }

int getX() { return x; }

int getY() { return y; }

};

int main()

{

Point p1;

Point p2 = p1;

cout << "x = " << p2.getX() << " y = " << p2.getY();

return 0;

}

Output:

There is compiler error in line "Point p1;". The class Point doesn't have a constructor without any parameter. If we write any constructor, then compiler doesn't create the default constructor. It is not true other way, i.e., if we write a default or parameterized constructor, then compiler creates a copy constructor. See the next question.

i.e, when default constructor will not be created automatically if copy constructor is defined in the class.

**#07\_Pgm**

#include <iostream>

using namespace std;

class Point

{

int x, y;

public:

Point(int i = 0, int j = 0) { x = i; y = j; }

int getX() { return x; }

int getY() { return y; }

};

int main()

{

Point p1;

Point p2 = p1;

cout << "x = " << p2.getX() << " y = " << p2.getY();

return 0;

}

Output:

x= 0 y = 0

Eventhough parameter are not provided the compiler uses the default arguments and creates a default copy constructor.

**#08\_Pgm**

#include<iostream>

#include<stdlib.h>

using namespace std;

class Test

{

public:

Test()

{ cout << "Constructor called"; }

};

int main()

{

Test \*t = (Test \*) malloc(sizeof(Test));

return 0;

}

Output:

Empty

Note: malloc does not call constructor.

**#09\_Pgm**

#include <iostream>

using namespace std;

class Test

{

public:

Test() { cout << "Hello from Test() "; }

} a;

int main()

{

cout << "Main Started ";

return 0;

}

Output:

Hello from Test() Main Started

Note: The object calls the constructor first even before the constructor is been called

**Destructor:**

What is destructor:

Its a member function which destructs or delets an object.

When it is called:

Its called when object goes out of scope,

1. Function ends.

2. Program ends.

3. Block containing local variable ends.

4. Delete operator is called.

How:

Destructor have same name as class name and preceded by (~) tilde operator. It dosent take any argument and dosen’t return anything.

Can there be more than one destructor:

No there can be only one destructor.

When user defined destructor is needed:

Compiler automatically writes default destructor, But when we have dynamically allocated memory or pointer in the class. During such scenario we need to release memory before instance is being destroyed.

**Friend class and function:**

What is friend function:

It has permission to access the private and protected members. A friend function can be a method of another class or global function.

**#10\_Pgm**

#include <iostream>

using namespace std;

class Rectangle {

int width, height;

public:

Rectangle(int w = 1, int h = 1):width(w),height(h){}

friend void display(Rectangle &);

};

void display(Rectangle &r) {

cout << "The friend function accessing provate data:\n"<< r.width \* r.height << endl;

}

int main () {

Rectangle rect(5,10);

display(rect);

return 0;

}

Output:

The friend function accessing provate data:

50

Note:

Here we use the display friend function to access the private data of the class Rectangle and friend function are not members of the class and cannot be accessed using dot(.) operator.

Prototype:

friend return\_type function\_Name(parameter);

why do we need a friend function:

It can be friendly to 2 or more classes. The friend function does not belong to any class and used to access the protected and private data of two or more classes. It deviates from the necessity of oops methodology, so needs to used when no other ways are available.

**#11\_Pgm**

#include <iostream>

using namespace std;

class Square; // forward declaration

class Rectangle {

int width, height;

public:

Rectangle(int w = 1, int h = 1):width(w),height(h){

cout << "The width " << w << " height " << h << endl;

}

friend void display(Rectangle &, Square &);// friend function prototype

};

class Square {

int side;

public:

Square(int s = 1):side(s){

cout << "The side of square " << s << endl;

}

friend void display(Rectangle &, Square &);// friend function prototype

};

void display(Rectangle &r, Square &s) { // friend function definition

cout << "Rectangle: widht \* height = " << r.width \* r.height << endl;

cout << "Square: side \* side = " << s.side \* s.side << endl;

cout << "Adding the the rect width and square side: " << r.width + s.side;

cout << "\nMult the rect height and sq side: " << r.height \* s.side << endl;

}

int main () {

Rectangle rec(5,10);

Square sq(5);

display(rec,sq);// Calling friend function

return 0;

}

Output:

The width 5 height 10

The side of square 5

Rectangle: widht \* height = 50

Square: side \* side = 25

Adding the the rect width and square side: 10

Mult the rect height and sq side: 50

What is friend class:

Friend class is used to access the private and protected members of other class in which its

declared as friend.

**#12\_Pgm**

#include <iostream>

using namespace std;

class Square;

class Rectangle {

int width, height;

public:

Rectangle(int w = 1, int h = 1):width(w),height(h){}

void display() {

cout << "Rectangle: " << width \* height << endl;

cout << "width : " << width << "\nHeight " << height << endl;

};

void morph(Square &);

};

class Square {

int side;

public:

Square(int s = 1):side(s){}

void display() {

cout << "Square: " << side \* side << endl;

};

friend class Rectangle;

};

void Rectangle::morph(Square &s) {

width = s.side;

height = s.side;

}

int main () {

Rectangle rec(5,10);

Square sq(5);

cout << "Before:" << endl;

rec.display();

sq.display();

rec.morph(sq);

cout << "\nAfter:" << endl;

rec.display();

sq.display();

return 0;

}

Output:

Before:

Rectangle: 50

width : 5

Height 10

Square: 25

After:

Rectangle: 25

width : 5

Height 5

Square: 25

Note:

Square considers rectangle to be friend, But rectangle does not consider square as its friend. So rectangle can access square private data but square cannot access data of rectangle.