# Computer Programing [Lab]

## Lab # 05-A: Classes & Objects

### Agenda

- Classes and Objects
- Accessing Class Members
- Public and Private Members
- Messages and C++ Physical Objects
- Accessor & Mutator Functions
- Constructor Implicit & Explicit
- Initializer List
- Object as Function Arguments
- Default Copy Constructor
- Returning Objects from Functions

## Classes & Objects

A class is a collection of a fixed number of components. The components of a class are called the members of the class. An object has the same relationship to a class that a variable has to a data type. Class variable is also called an object or an instance of a class, in the same way car is an instance of a vehicle. The general syntax for defining a class is:

```
class className
{
	private:
	// private members if any

	public:
	// public members if any
}
```

## Accessing Class Members:

In C++, the dot (.) is an operator called the **member access operator**.

To access any member of a class, we use the **member access operator (.)**.

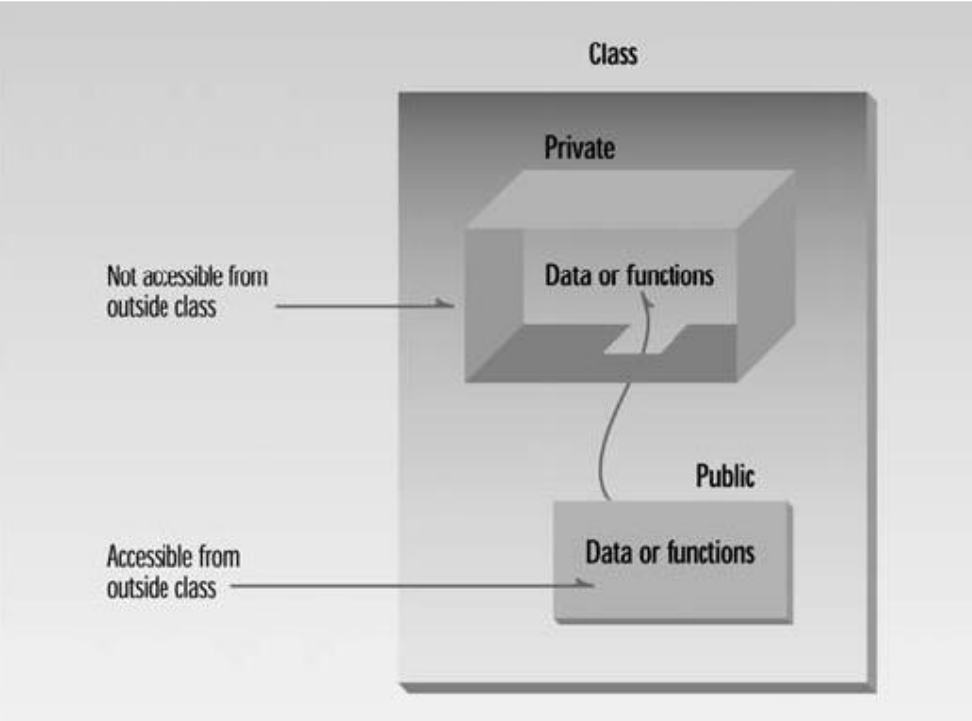
## Sample Program

```
//	Sample program for understanding classes and objects

#include <iostream>
using namespace std;
class smallobj //defines a class
{
	private:
	int somedata; //class data
	public:
	void setdata(int d) //member function to set data
	{ somedata = d; }
	void showdata() //member function to display data
	{
		cout << "Data is “ << somedata" << endl; }
	};
////////////////////////////////////////////////////////////
int main()
{
	smallobj s1, s2; //define two objects of class smallobj
	s1.setdata(1066); //call member function to set data
	s2.setdata(1776);
	s1.showdata(); //call member function to display data
	s2.showdata();
	return 0;
}
```

## Public and Private Members of a Class

The members of a class are classified into three categories: private, public, and protected. In C++, private, protected, and public are reserved words and are called member access specifiers.




## Using the Class

Now that the class is defined, let's see how main () makes use of it. We'll see how objects are defined, and, once defined, how their member functions are accessed

## Defining Objects

The first statement in main()

smallobj s1, s2;

## Calling Member Functions

The next two statements in main() call the member function setdata():

s1.setdata(1066);

s2.setdata(1776);

Similarly, the following two calls to the showdata() function will cause the two objects to display their values:

s1.showdata();

s2.showdata();

## Messages

Some object-oriented languages refer to calls to member functions as messages. Thus the call

s1.showdata();

can be thought of as sending a message to s1 telling it to show its data. The term message is not a formal term in C++, but it is a useful idea to keep in mind as we discuss member functions.

## C++ Objects as Physical Objects

In many programming situations, objects in programs represent physical objects: things that can be felt or seen. These situations provide vivid examples of the correspondence between the program and the real world.

### Example # 01

```
// This program demonstrates the use of classes and objects
#include <iostream>
using namespace std;
class part //define class
{
	private:
	int modelnumber; //ID number of widget
	int partnumber; //ID number of widget part
	float cost; //cost of part
	public:
	void setpart(int mn, int pn, float c) //set data
	{
	modelnumber = mn;
	partnumber = pn;
	cost = c;
	}
	void showpart() //display data
	{
	cout << "Model " << modelnumber;
	cout << ", part " << partnumber;
	cout << ", costs $" << cost << endl;
	}
};
int main()
{
	part part1; //define object
	// of class part
	part1.setpart(6244, 373, 217.55F); //call member function
	part1.showpart(); //call member function
	return 0;
}
```

## Accessor & Mutator Functions

Every class has member functions that only access and do not modify the member variables, called accessor functions, and member functions that modify the member variables, called mutator functions.

**Accessor function:** A member function of a class that only accesses (that is, does not modify) the value(s) of the member variable(s) i.e. all functions for getting/printing values of member variables.

**Mutator function:** A member function of a class that modifies the value(s) of the member variable(s) i.e. all functions for setting values to member variables.

## Constructor Implicit & Explicit

```
// This program demonstrates the use of constructors
#include <iostream>
using namespace std;
class Counter
{
	private:
	unsigned int count; //count
	public:
	Counter() : count(0) //constructor
	{ /*empty body*/ }
	void inc_count() //increment count
	{ count++; }
	int get_count() //return count
	{ return count; }
};
int main()
{
	Counter c1, c2; //define and initialize
	cout << "\nc1=" << c1.get_count(); //display
	cout << "\nc2=" << c2.get_count();
	c1.inc_count(); //increment c1
	c2.inc_count(); //increment c2
	c2.inc_count(); //increment c2
	cout << "\nc1=" << c1.get_count(); //display again
	cout << "\nc2=" << c2.get_count();
	cout << endl;
	return 0;
}
```

## Initializer List

One of the most common tasks a constructor carries out is initializing data members. In the Counter class the constructor must initialize the count member to 0. You might think that this would be done in the constructor's function body, like this:

count() { count = 0; }

However, this is not the preferred approach (although it does work). Here's how you should initialize a data member:

count() : count(0)

{ }

The initialization takes place following the member function declarator but before the function body. It's preceded by a colon. The value is placed in parentheses following the member data. If multiple members must be initialized, they're separated by commas. The result is the initializer list (sometimes called by other names, such as the member-initialization list).

someClass() : m1(7), m2(33), m2(4) ←□□□ initializer list
{ }

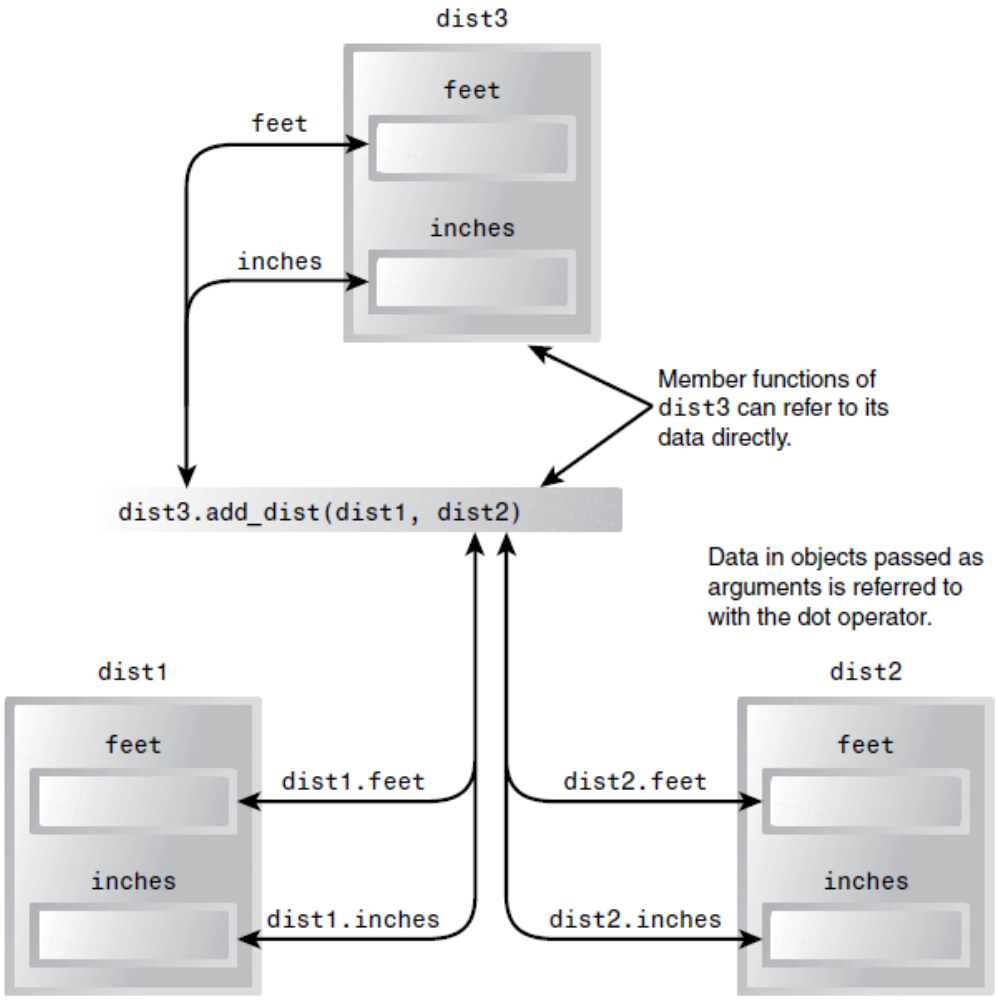
## Destructors

We've seen that a special member function—the constructor—is called automatically when an object is first created. You might guess that another function is called automatically when an object is destroyed. This is indeed the case. Such a function is called a destructor. A destructor has the same name as the constructor (which is the same as the class name) but is preceded by a tilde:

```
class Foo
{
	private:
	int data;
	public:
	Foo() : data(0) //constructor (same name as class)
	{ }
	~Foo() //destructor (same name with tilde)
	{ }
};
```

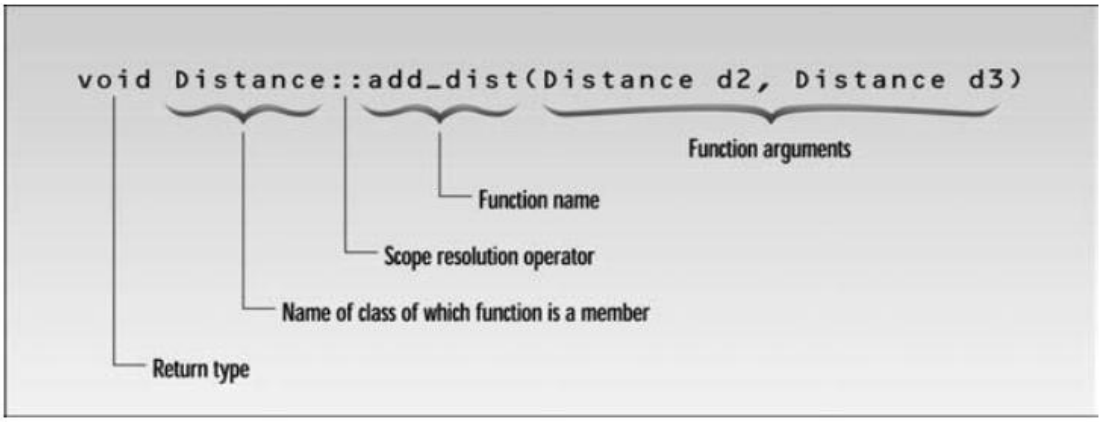
## Object as Function Arguments

```
// this program demonstrates the object as function arguments
#include <iostream>
using namespace std;
class Distance //English Distance class
{
	private:
		int feet;
	float inches;
	public:
	Distance() : feet(0), inches(0.0)
	{ }
	//constructor (two args)
	Distance(int ft, float in) : feet(ft), inches(in)
	{ }
	void getdist() //get length from user
	{
	cout << "\nEnter feet: "; cin >> feet;
	cout << "Enter inches: "; cin >> inches;
	}
	void showdist() //display distance
	{ cout << feet << "'-" << inches << "'"; }
	void add_dist( Distance, Distance ); //declaration
	};
	//-----------------------------------------------------------
	//add lengths d2 and d3
	void Distance::add_dist(Distance d2, Distance d3)
	{
	inches = d2.inches + d3.inches; //add the inches
	feet = 0; //(for possible carry)
	if(inches >= 12.0) //if total exceeds 12.0,
	{ //then decrease inches
	inches -= 12.0; //by 12.0 and
	feet++; //increase feet
	} //by 1
	feet += d2.feet + d3.feet; //add the feet
	}
////////////////////////////////////////////////////////////////
int main()
{
	Distance dist1, dist3; //define two lengths
	Distance dist2(11, 6.25); //define and initialize dist2
	dist1.getdist(); //get dist1 from user
	dist3.add_dist(dist1, dist2); //dist3 = dist1 + dist2
	//display all lengths
	cout << "\ndist1 = "; dist1.showdist();
	cout << "\ndist2 = "; dist2.showdist();
	cout << "\ndist3 = "; dist3.showdist();
	cout << endl;
	return 0;
}
```




## Member Functions Defined Outside the Class

```
void Distance::add_dist(Distance d2, Distance d3)
{
       inches = d2.inches + d3.inches; //add the inches
       feet = 0; //(for possible carry)
       if(inches >= 12.0) //if total exceeds 12.0,
       { //then decrease inches
       inches - = 12.0; //by 12.0 and
       feet++; //increase feet
       } //by 1
       feet + = d2.feet + d3.feet; //add the feet
}
```



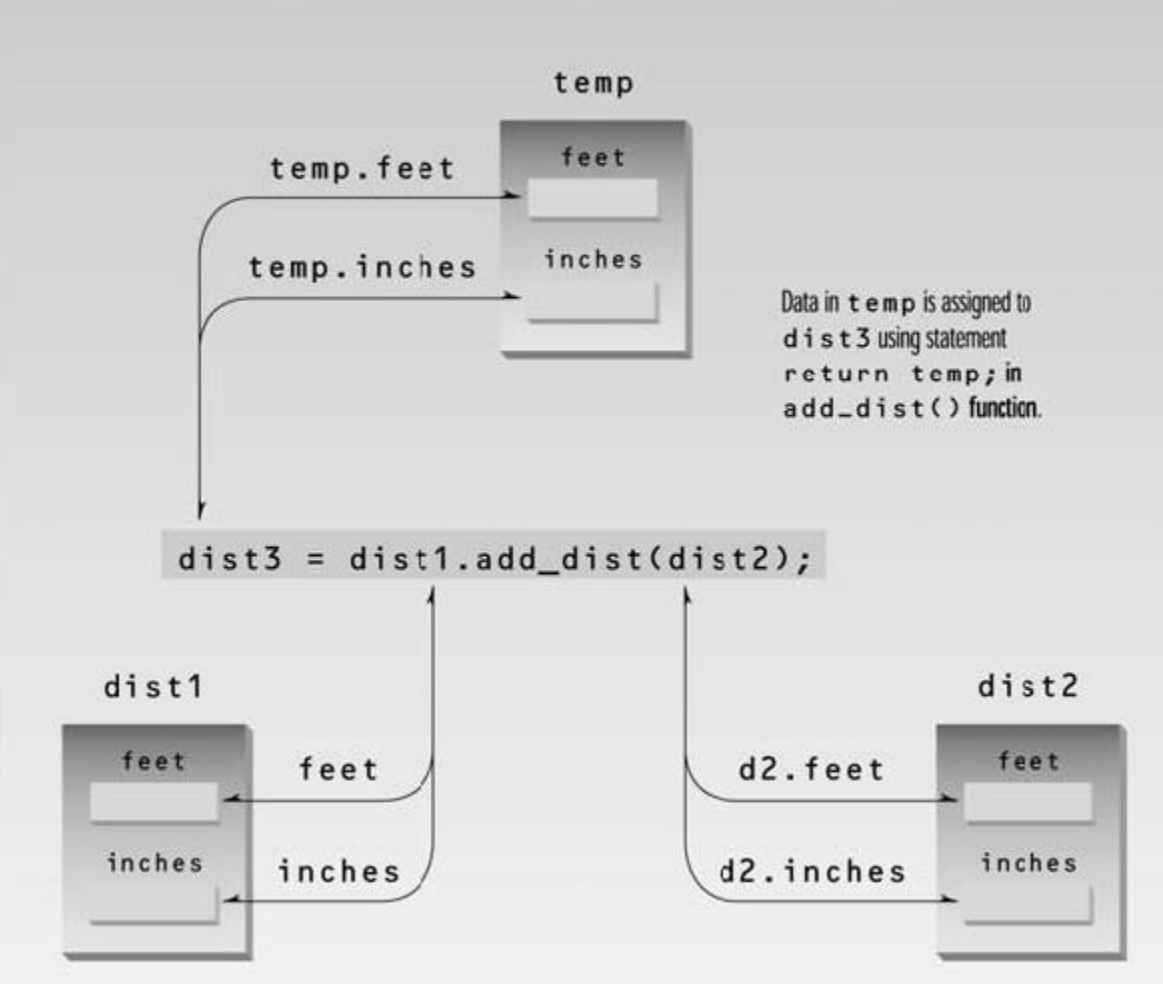

## Default Copy Constructor

We've seen two ways to initialize objects. A no-argument constructor can initialize data members to constant values, and a multi-argument constructor can initialize data members to values passed as arguments. Let's mention another way to initialize an object: you can initialize it with another object of the same type. Surprisingly, you don't need to create a special constructor for this; one is already built into all classes. It's called the default copy constructor. It's a one argument constructor whose argument is an object of the same class as the constructor.

```
#include <iostream>
using namespace std;
class Distance //English Distance class
{
        private:
        int feet;
        float inches;
        public:
        //constructor (no args)
        Distance() : feet(0), inches(0.0)
        { }
        //Note: no one-arg constructor
        //constructor (two args)
        Distance(int ft, float in) : feet(ft), inches(in)
        { }
        void getdist() //get length from user
        {
        cout << "\nEnter feet: "; cin >> feet;
        cout << "Enter inches: "; cin >> inches;
        }
        void showdist() //display distance
        { cout << feet << "\'-" << inches << "\""; }
};
int main()
{
        Distance dist1 (11, 6.25); //two-arg constructor
        Distance dist2 (dist1); //one-arg constructor
        Distance dist3 = dist1; //also one-arg constructor
        //display all lengths
        cout << "\ndist1 = "; dist1.showdist();
        cout << "\ndist2 = "; dist2.showdist();
        cout << "\ndist3 = "; dist3.showdist();     cout << endl; return 0;
}
```

## Returning Objects from Functions

```
#include <iostream>
using namespace std;
class Distance //English Distance class
{
	private:
	int feet;
	float inches;
	public: //constructor (no args)
	Distance() : feet(0), inches(0.0)
	{ } //constructor (two args)
	Distance(int ft, float in) : feet(ft), inches(in)
	{ }
	void getdist() //get length from user
	{
	cout << "\nEnter feet: "; cin >> feet;
	cout << "Enter inches: "; cin >> inches;
	}
	void showdist() //display distance
	{ cout << feet << "\'-" << inches << "\""; }
	Distance add_dist(Distance); //add
};
//-----------------------------------------------------------
//add this distance to d2, return the sum
Distance Distance::add_dist(Distance d2)
{
	Distance temp; //temporary variable
	temp.inches = inches + d2.inches; //add the inches
	if(temp.inches >= 12.0) //if total exceeds 12.0,
	{	//then decrease inches
		temp.inches -= 12.0; //by 12.0 and
		temp.feet = 1; //increase feet by 1
	}
	temp.feet += feet + d2.feet; //add the feet
	return temp;
}
int main()
{
	Distance dist1, dist3; //define two lengths
	Distance dist2(11, 6.25); //define, initialize dist2
	dist1.getdist(); //get dist1 from user
	dist3 = dist1.add_dist(dist2); //dist3 = dist1 + dist2
	//display all lengths
	cout << "\ndist1 = "; dist1.showdist();
	cout << "\ndist2 = "; dist2.showdist();
	cout << "\ndist3 = "; dist3.showdist();
	cout << endl;
	return 0;
}
```


temp
feet
inches
temp.feet
temp.inches
Data in temp is assigned to dist3 using statement return temp; in add_dist() function.
dist3 = dist1.add_dist(dist2);
dist1
feet
inches
feet
inches
dist2
feet
inches
d2.feet
d2.inches