Classes and Data Abstraction

We have learned how to group of data items that are of different data type using a struct. C++ also provides another structured data type, called a classes extent structures in that they can group data as well as functions.

Definition: A class is a collection of a fixed number of components; the components of a class are called the members of a class.

The general syntax for defining a class is:

```
 class classIdentifier
{
  classMembersList
};
```

where classMembersList consists of a variable declaration and or functions. That is, a member of a class can be either a variable (to store data) or a function (to manipulate it).

•If a member of a class is a variable, you declare it just like any other variable. But, variables in classes cannot be initialized when you declare it.

- If a member of a class is a function, you typically use the function prototype to define that member.
- If a member of a class is a function, it can directly access any member of the class –data members and function members.

In C++, class is a reserved word, and it defines only a data type; no memory is allocated.

the ; after the right brace is part of the syntax.

The members of a class are classified into 3 categories private, public, and protected. These are reserved words and called member access specifiers. We will concentrate on the 2 first for now.

How to use them:

- By default, all members of a class are private. If a member is private, you cannot access it outside the class
- A public member is accessible outside the class. To make a member of a class public, you use the label public with a colon:

General Definition of a class

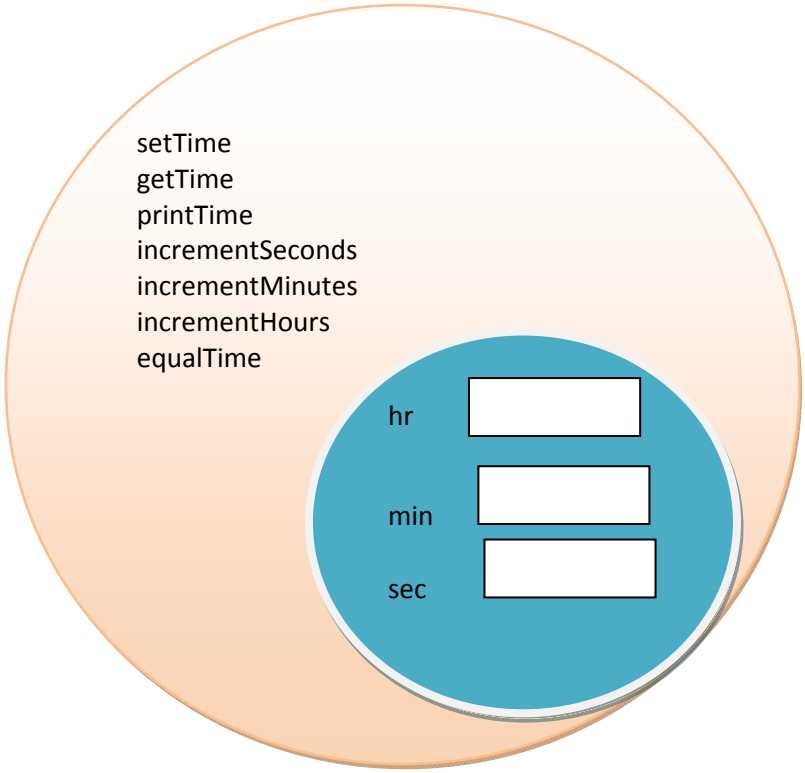
class class_name {

```
access_specifier_1:
  member1;
access_specifier_2:
  member2;
...
} object_names;
```

Suppose, we want to define a class to implement the time of day in a program, let's call this class ClockType. To represent time, we will use three int variables: one to represent the hours, one to represent the minutes and one to represent the seconds. We also want to perform the following operations on the time:

1. Set the time
2. Return the time
3. Print the time
4. Increment the time by one second
5. increment the time by one minute
6. increment the time by one hour.
7. Compare two times for equality

So, our class ClockType has 10 members: 3 data members and 7 function members



class ClockType and its members

Some members of the class ClockType will be private, others will be public.

Deciding which ones to make public and which ones to make private depends on the nature of the members.

General Access rule:

Any member that needs to be accessed outside the class is declared public, usually function members are declared public. For example, the user should be able to set the time, print the time, increment and compare times for equality.

Any member that should not be accessed directly by the user should be declared private. Usually data members are declared private. To control the direct manipulation of the data member's hr, min and sec, we will declare them private.

The following statements define the class ClockType

```
class ClockType
{
public:
  void setTime (int, int, int);
  void getTime(int&, int&, int&);
  void printTime() const;
  void incrementSeconds();
  void incrementMinutes();
  void incrementHours();
  bool  equalTime( const ClockType& otherClock) const;
private:
  int hr;
  int min;
  int sec;
};
```

This class has 3 data members and 7 function members. In the function equalTime, the parameter otherClock is a constant reference parameter.

The word const at the end of the functions specifies that the functions cannot modify the data members of a variable of the type ClockType.

Variable (Object) definition:

Once a class has been defined, you can declare variables of that type. In C++, a class variable is called a class object or a class instance or simply object.

The syntax for declaring an object is the same as for declaring a variable. The following statements declare two objects of the type ClockType.

```
ClockType  myClock;

ClockType yourClock;
```

Each object has 10 members: seven function members and three data members. Each object has a separate memory allocated for hr, min, sec

Accessing Class members

Once an object is declared, it can access the public members of a class. The general syntax to access the members of a class is:

```
classObjectName.memberName
```

The **.** (dot) operator, is an operator called the **member access operator.**

```
myClock.setTime(5, 2, 30);

myClock.printTime();

youClock.setTime(x, y, z);

if  (myClock.equalTime(yourClock))

{…..}
```

Illegal Operations:

An object can access only public members of the class. Members that have been declared private cannot be accessed directly.

Built-in Operations on Classes

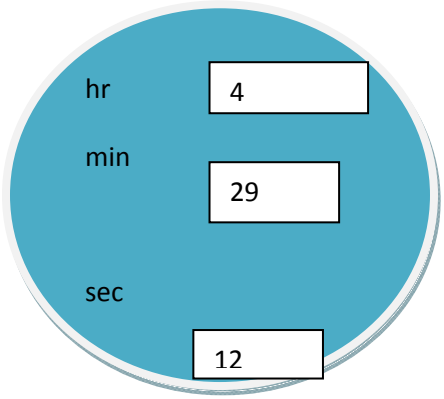
Assignment:

Most of the operations defined for variables **cannot** be used on objects. However, we can use the assignment operator to make the data members of one object equal the data members of another

Example:

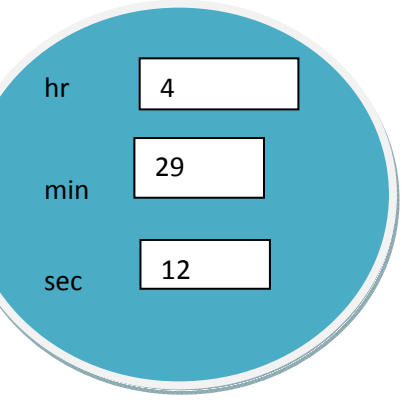
ClockType myClock, yourClock;

myClock.setTime(4, 29, 12);



myClock

yourClock=myClock;



yourClock

Class Scope:

A class object can be either automatic (created at declaration time, and destroyed when control exits) or static

A member of a class is local to the class. so it cannot be accessed outside of it.

Implementation of Member functions:

When we defined the class ClockType, we included only the function prototype for member functions. For these functions to work properly, we need to write the related code.

To write the definitions of the member functions of the class ClockType, we need to refer to the data members and

the function members. Because they are local to the class we cannot reference them directly outside of the class that is why we need to use the scope resolution operator::

For example the function setTime will be defined as:

```
void ClockType::setTime(int hours, int minutes, int seconds){
  if (0<= hours && hours < 24)
 hrs= hours;
 else hrs=0;
 if (0<= minutes && minutes < 60)
        min= minutes;
 else min=0;
 if (0<= seconds && seconds <60)
        sec=seconds;
 else sec =0; }
```

The function setTime checks for valid values of hours, minutes and seconds. If they are out of range, the member variable hr, min and sec are initialized to 0. The member function setTime is a void function.

Order of public and private Members of a class:

C++ has no fixed order in which you declare public and private members, you can declare them in any order. The only thing we need to remember is by default all members are private. We must use the access specifier public in order to make a member available for public access. If we start by listing the public class members, then we need to put the access specifier private to list the private members otherwise, they will be public as well.

Constructors:

Constructors are used to guarantee that the member variables of a class are initialized as they are declared. There are 2 types of constructors: with parameter and without parameters.

The constructor without parameter is called the default constructor

Constructors have the following properties:

- The name of the constructor is the same as the name of the class.
- A constructor, even though it is a function, has not type, that it is neither a value returning function nor a void function

- A class can have more than one constructor. However, all constructors of a class have the same name.
- If a class has more than one constructor, the constructors must have different formal parameter lists. That is either they have a different number of formal parameters or, if the number of parameters is the same, then the data type of at least one formal parameter should be different in each constructor.
- Constructors execute automatically when a class object enters its scope. Because they have no types, they cannot be called like other functions.
- Which constructor executes depends on the types of values passed to the class object when the class object is declared.

```
Class ClockType
{
public:
  void setTime (int, int, int);
  void getTime(int&, int&, int&);
  void printTime() const;
  void incrementSeconds();
```

```
  void incrementMinutes();
  void incrementHours();
  bool  equalTime( const ClockType&) const;
  ClockType(int, int, int);//constructor with parameters
  ClockType();  // Default constructor

private:
  int hr;
  int min;
  int sec;
};
```

The default constructor is usually designed to set the values of the data members to their default values

```
ClockType::ClockType(){
 hr=0;
min=0;
sec=0;
}
```

```
ClockType::ClockType(int hours, int minutes, int seconds)
  if (0<= hours && hours < 24)
        hr= hours;
  else hr=0;
  if (0<= minutes && minutes < 60)
        min= minutes;
  else min=0;
  if (0<= seconds && seconds <60)
        sec=seconds;
else sec =0;
 }
```

The constructor with parameters sets the member variables to whatever values are assigned to the formal parameters. Basically, it performs that same job as setTime. So we can redefine ClockType(int, int, int) as:

```
ClockType::ClockType(int hours, int minutes, int seconds)
{  setTime(hours, minutes, seconds);
}
```

Invoking a Constructor

When an object is declared, a constructor is automatically executed. Because a class might have more than one constructor, including the default constructor, we need to specify which one we want.

Invoking the default constructor:

Suppose that a class contains the default constructor. The way to invoke or call the default constructor is:

classNameclassObjectName;

For example:

ClockType yourClock;

declares yourClock to be an object of type ClockType. In this case, the default constructor executes and the member variables of yourClock are initialized to 0.

It is illegal to use the () after you declare the variable.

ClockType yourClock(); is illegal.

Invoking a constructor with Parameters:

Suppose a class contains constructors with parameters, the syntax to invoke a constructor with parameters is

className classObjectName(argument1, argument2,……);
where argument1, argument2 and so on are a value, a variable or an expression.

- The number of arguments and their type should match the formal parameters (in the order) of one of the constructors.
- If the data types of the provided values do not match the formal parameters, C++ will convert them.

ClockType myclock(5, 12, 40)

This statement declares an object myclock and initializes its data members to the values 5 for hrs, 12 for min and 40 for sec. This is done by providing 3 values of type integer, which match the type of the formal parameters.

Arrays of Class objects and Constructors:

To declare an array of objects we use the same procedure:

To initialize it, we need to have a default constructor which will initialize each array class object.

ClockType arrivalTime[100];

This statement creates the array of objects arrivalTime[0], arrivalTime[1],… arrivalTime[99] and initializes its data members to 0.

Destructors:

Like constructors, destructors are also functions. Furthermore, like constructors they do not have a return type. However a class can only have one destructor and the destructor has no parameters. The name of the destructor is the character (~) , followed by the name of the class. For example: the name of the destructor for the class ClockType is:

~ClockType();

Data Abstraction, Classes and Abstract Data Types:

Abstraction is the process of separating the design details of an object from its use. In other words, it focuses on what the object does and not how it does it.

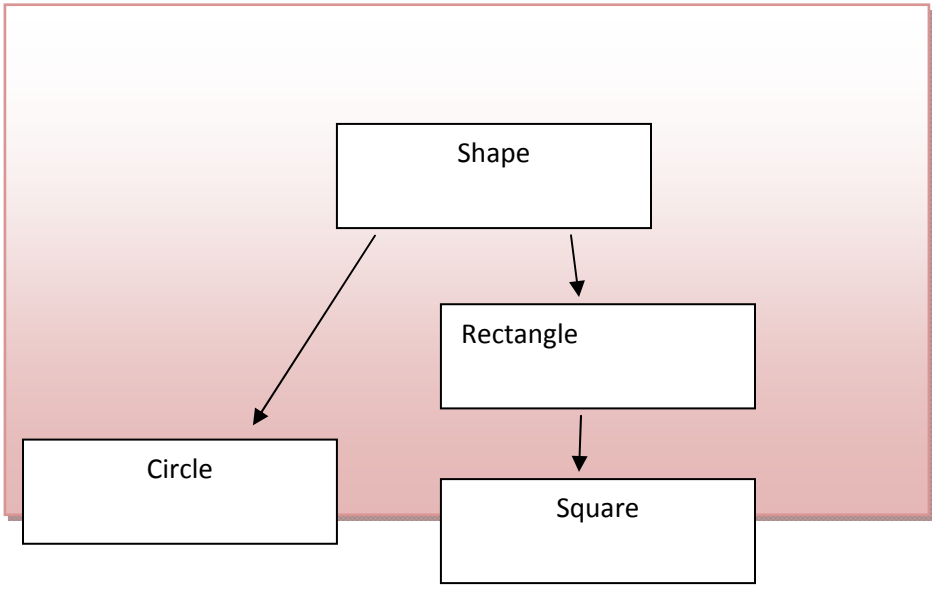
Abstract data type (ADT):

A data type that specifies the logical properties without the implementation details. The class definition and all the implementation details will be in a separate file called a header file with the name of the class.

The program with the main function –the driver program- will contain an include statement which adds the class definition to the program

Inheritance:

Inheritance let us create new classes from existing ones. The new class that we create from the existing ones are called the derived classes; the existing classes are called base classes. The derived classes inherit the properties of the base classes. So rather than create completely new classes from scratch, we can take advantage of inheritance and reduce software complexity.



In this diagram, Shape is the base class. The classes Circle and Rectangle are derived from shape, and the class Square is derived from Rectangle. Every Circle and every Rectangle is a Shape. Every Square is a Rectangle.

Inheritance and the "is a" relationship:

When an object is a specialized version of another one, there is an "is a" relationship between them.

- a poodle *is a* dog
- a car *is a* vehicle
- a tree *is a* plant….

When an "is a" relationship exists between classes, it means that the specialized class has all of the characteristics of the general class, plus additional characteristics that make it special.

In OOP the derived class is the specialized class and the base class is the more general or parent class.

The derived class inherits the member variables and member functions of the base class without any of them being rewritten. Furthermore, new members may be added

to the derived class to make it more specialized than the base class.

How to define a derived class from a base class:

class derivedClassName: memberAccessSpecifier baseClassName

{……};

Example: class Circle: public Shape

class Rectangle: public Shape

How would we declare the class Square?

The word *public* that precedes the base class name is the member access specifier. It affects how the members of the base class are inherited by the derived class.

where memberAccessSpecifier is public, private (or protected). When no memberAccessSpecifier is mentioned it is assumed to be a private inheritance.

class Circle: public Shape

{

…};

The word public in the heading is called the member access specifier. It specifies that all public members of the class Shape are inherited as public members by the class circle. In other words, public members of the class shape become public members of the class circle.

In this example, the class circle is derived from the class Shape, using public access specification. This means that the public members of the class Shape will become public members of the class Circle. The private members of the class Shape cannot be accessed directly by the code in the class Circle. Although the private members of the class Shape are inherited, it is as tough they are invisible to the code of the class Circle. They can only be accessed by the member functions of the Shape class.

class Circle: private Shape

{

…};

In this definition, the public members of Shape become private members of the class Circle.

If no member access specifier is mentioned, then the public members of a base class are inherited as private members.

class className: memberAccessSpecifier baseClassName

{……};

where memberAccessSpecifier is public, protected or private. When no memberAccessSpecifier is mentioned it is assumer to be a private inheritance.

Constructors in derived classes:

Constructors cannot be directly inherited from the base class to the derived class.

Inheritance cannot work in reverse. It is not possible for the base class to call a member function of the derived class.

Points to keep in mind:

1. The private members of the base class are private to the base class; hence the members of the derived class cannot directly access them.
2. The public members of a base class can be inherited either as public members or private members of the derived class.

3. The derived class can include additional data and/or function members.
4. The derived class can redefine the public member functions of the base class. That is, in the derived class we can have function members with the same name, number and type parameters as functions in the base class. However, this redefinition only applies to the objects of the derived access, nor to the objects of the base class.
5. All the data members of the base class are also data members of the derived class. Also, the member functions of the base class are also members of the derived class.

Protected Members and Class Access:

Protected members of a base class are like private members, but they may be accessed by derived classes. The base class access specification determines how private, public, and protected base class members are accessed when they are inherited by the derived class. For a member variable to be accessible to a derived class, it must be have "protected:' access specifier.

```
class Rectangle {
 protected:
```

```
  double length;
   double width;

public:
//Constructor:
Rectangle (double a, double b) {
        length=a;
        width=b; }
……

};
```

Avoiding multiple inclusion Header Files:

Usually the specification file of a derived class the(.h) file begins with an #include directive to insert the specification file(.h) of the base class.

Now, there may be cases in which we need both the base class and the derived classes, in which case a program using both will have include statements such as:

#include "BaseClass.h"

#include "DerivedClass.h"

When the preprocessor encounters these 2 directives, it will insert the file BaseClass.h, then DerivedClass.h, then

BaseClass.h a second time, since DerivedClass.h also includes BaseClass.h. The result is a compile-time error because the file BaseClass.h is defined twice.

The solution to this problem is to write BaseClass.h this way:

```
#ifndef BASECLASS_H

#define BASECLASS_H

 class BaseClass

{

……

};

#endif
```

The lines beginning with # are directives to the preprocessor. BASECLASS_H is a preprocessor identifier. In effect these directives say:

if the preprocessor identifier BASECLASS_H is not already define then

1. define BASECLASS_H as an identifier known to the processor
   and

2. let the declaration of BaseClass class be compiled.
   if a subsequent #include "BaseClass.h" is encountered, the test #ifndef BASECLASS_H will fail. The BaseClass class declaration will not pass through to the compiler again.

Class Composition:

Two classes typically are:

1) Independent of one another, 2) they are related by inheritance, or 3) they are related by composition.

Composition or containment is the relationship in which the internal data of one class A includes an object of another class B. Stated differently, an B object is contained within an A object.

In C++, there is no need for any language notation for composition. We simply declare an object of one class to be one of the data members of another class.

Example:

The Land Sales:

We are developing a program to represent Land lots. A lot of land usually has 3 types of information: The Id of the lot, a string, the dimensions of the lot- assumed to

be rectangle, and a price per square foot. We may need to add more, but we will start with these for now. Constructing this object LandLot should be easy because we already have a class rectangle.

```
#include “Rectangle.h”
#include <string>

class Land
{
        private :
       string Id;
       Rectangle  Lot;
       double price;

       public:
       Land(); // Default constructor
        Land( string ID, int length, int width, double price)
        //parameterized constructor
        LotPrice();
        PrintLot()
}
```

Instance and Static Members:

Each class object is an instance of a class. It has its own copy of the class member variables. Each object is separate, distinct and independent of another object of the same class.

Static Members:

It is possible to create a member variable or a member function that does not belong to any instance of a class. Such members are known as static member variables and static member functions.

When a value is stored in a static member variable, it is not stored in any instance of the class, but with the class itself. Likewise, a static member function can only operate on static member variables and not on instance variables.

Static Member Variables:

When a member variable is declared with the keyword static, there will be only one copy of the member variable in memory, regardless of the number of instances that might exist. A single copy of the class'static member variable is shared by all instances of the class.