Introduction to OOP

Instructor: Dr. Danish Shehzad

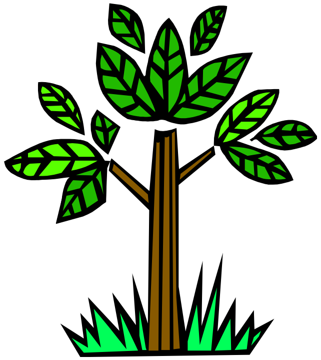
What is Object-Orientation?

- A technique for system modeling
- OO model consists of several interacting objects

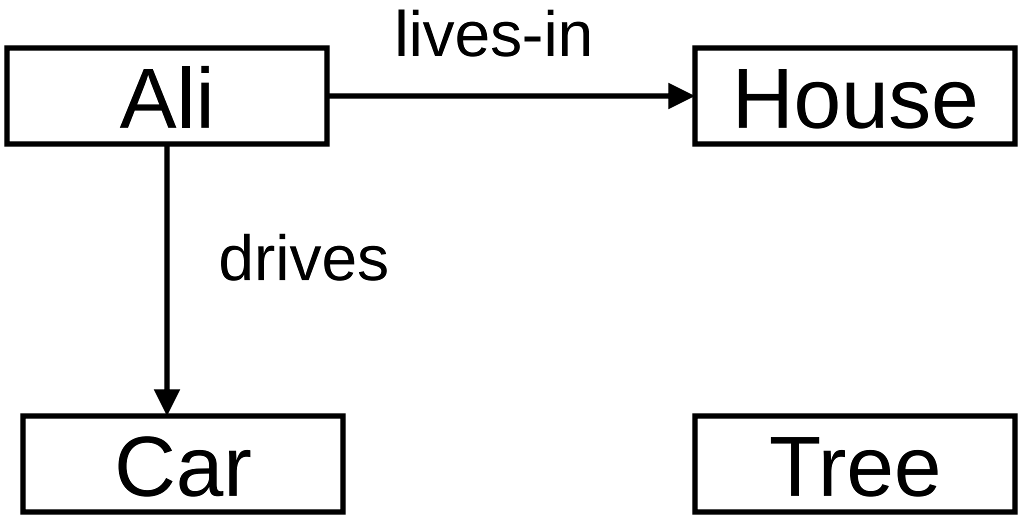
What is a Model?

- A model is an abstraction of something
- Purpose is to understand the product before developing it

Example – OO Model



- Objects
  - Ali
  - House
  - Car
  - Tree
- Interactions
  - Ali lives in the house
  - Ali drives the car



Object-Orientation - Advantages

- People think in terms of objects

- OO models map to reality

- Therefore, OO models are
  - easy to develop
  - easy to understand

OO- Languages

- Everything is an object.
- A program is a bunch of objects telling each other what to do, by sending messages.
- Each object has its own memory, and is made up of other objects.
- Every object has a type (class).
- All objects of the same type can receive the same messages.

What is an Object?

An object is

- Something tangible (Ali, Car)
- Something that can be apprehended intellectually (Time, Date)

… What is an Object?

An object has

- State (attributes)
- Well-defined behaviour (operations)
- Unique identity

Example – Ali is a Tangible Object

- State (attributes)
  - Name
  - Age
- behaviour (operations)
  - Walks
  - Eats
- Identity
  - His name

Example – Car is a Tangible Object

- State (attributes)
  - - Color
  - - Model
- behaviour (operations)
  - - Accelerate
  - - Start Car
  - - Change Gear
- Identity
  - - Its registration number

Why is OOP Popular?

- Quickly and easily leads to increase productivity and improve reliability (solve the software crisis)
- Hope for easy transition from existing languages
- Mimic problem solving in real worlds

INTRODUCTION

- Object-oriented programming (OOP)
  - Encapsulates **data (attributes)** and **functions (behavior)** into packages called classes

- Classes
  - Classes are the standard unit of programming
  - A class is like a blueprint – reusable
  - Objects are instantiated (created) from the class
  - Consists of data members and member functions

Basic Syntax

In C++:

- *class class_name{ Member_list };//Class*

  Member_List includes MemberVariables and MemberFunctions

- *class_name identifier;//Object*

Structure

A structure has

1. data members only

Class

A class has

1. data
2. functions

Class

- A Class is a user defined data type.
- Template for creating objects.

OBJECT

The instances of the class are called Objects.

Structure of a class

```
class name_of_class
{

  // definition of a class

};
```

Example 1

```
struct Date
{
   int day ;
   int month ;
   int year ;
} ;

Date mydate ;
mydate.month = 1 ;
mydate.day = 21 ;
mydate.year = 1979 ;
```

Example 2

```
class Date
{
  int day ;
  int month ;
  int year ;
} ;
```

```
main ( )
{
        Date mydate;
        mydate.month = 10 ; // Error
}
```

Private

Default visibility of all data and function inside a class is private

Member Access Specifiers

- **`public`**
  - Presents clients with a view of the services the class provides (interface)
  - Data and member functions are accessible

- **`private`**
  - Default access mode
  - Data only accessible to member functions and **`friend`**s
  - **`private`** members only accessible through the **`public`** class interface using **`public`** member functions

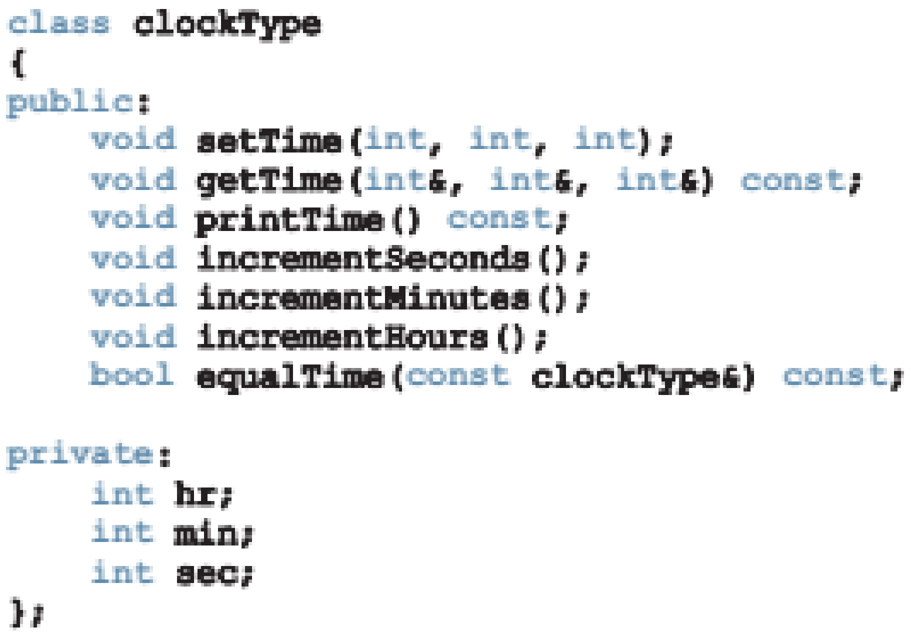
Public

```
class Date
{
   private :

          // private data and functions

  public :

          // public data and functions
};
```

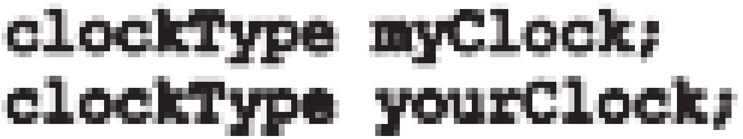
Date Class

```
class Date
{
  private :
            int day , month , year ;
  public :
            setMonth ( ) ;
            print ( ) ;
};
```

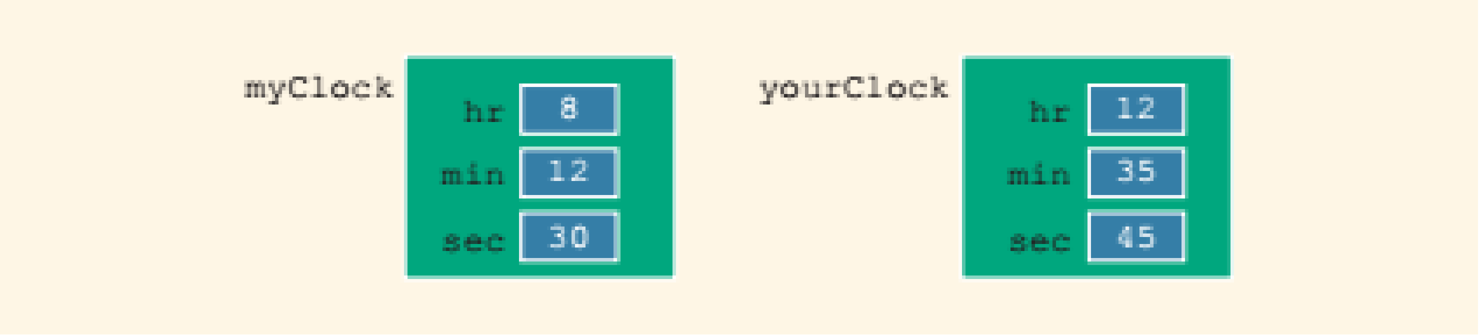
```
main ( )
{
  Date mydate ;
  mydate.setMonth ( 10 ) ;
  mydate.print ( ) ;
}
```



```
class clockType
{
public:
    void setTime(int, int, int);
    void getTime(int&, int&, int&) const;
    void printTime() const;
    void incrementSeconds();
    void incrementMinutes();
    void incrementHours();
    bool equalTime(const clockType&) const;

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

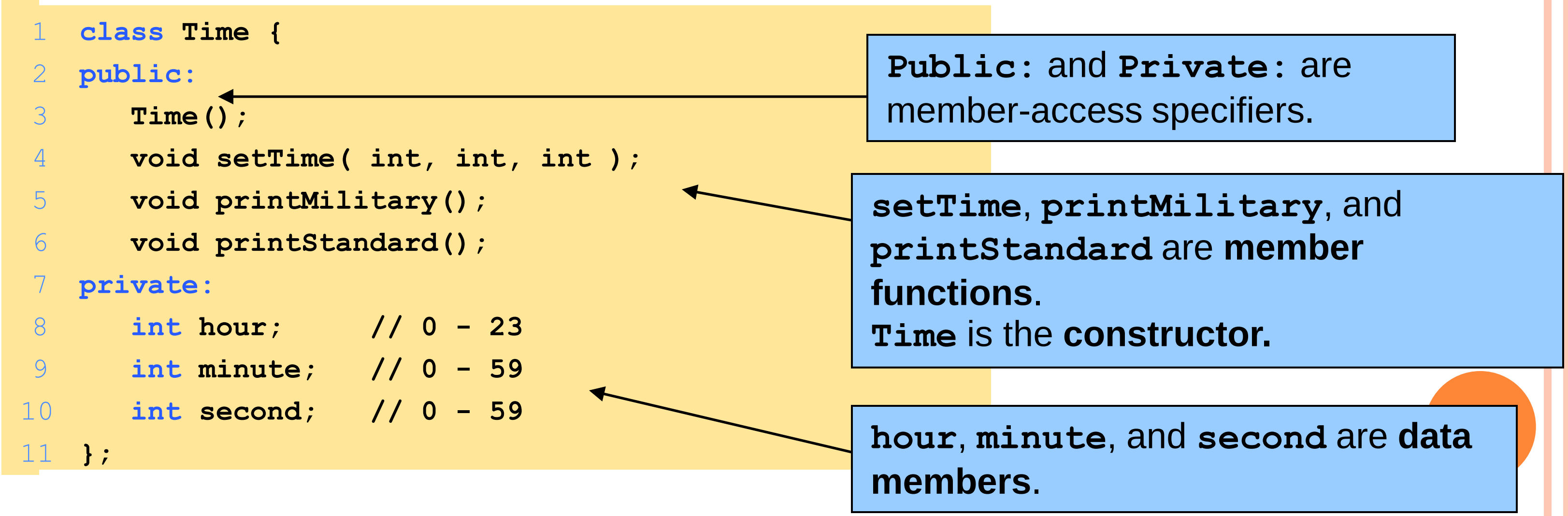


```
clockType myClock;
clockType yourClock;
```



TIME CLASS

- Classes
  - Model objects that have attributes (data members) and behaviors (member functions)
  - Defined using keyword **`class`**
  - Have a body delineated with braces (**`{`** and **`}`**)
  - Class definitions terminate with a semicolon



```
class Time {
public:
   Time();
   void setTime( int, int, int );
   void printMilitary();
   void printStandard();
private:
   int hour;     // 0 - 23
   int minute;   // 0 - 59
   int second;   // 0 - 59
};
```

`Public:` and **`Private:`** are member-access specifiers.

`setTime`, **`printMilitary`**, and **`printStandard`** are **member functions.**
`Time` is the **constructor.**

`hour`, **`minute`**, and **`second`** are **data members.**

Time Class

```cpp
class Time {
public:
   Time();
   void setTime( int, int, int );
   void printMilitary();
   void printStandard();
private:
   int hour;     // 0 - 23
   int minute;   // 0 - 59
   int second;   // 0 - 59
};
```

`setTime`, `printMilitary`, and `printStandard` are **member functions.**
`Time` is the **constructor.**

Member Functions

- Inline
  - The functions are defined within the body of the class definition.
- Out-of-line:
  - The functions are declared within the body of the class definition and defined outside.

Member Functions

- If a member function is defined outside the class
  - Scope resolution operator (::) and class name are needed
  - Defining a function outside a class does not change it being **`public`** or **`private`**

- Binary scope resolution operator (: :)
  - Combines the class name with the member function name
  - Different classes can have member functions with the same name
- Format for defining member functions

```
        ReturnType ClassName::MemberFunctionName( ){
            …
          }
```

```cpp
// Fig. 6.3: fig06_03.cpp
// Time class.
#include <iostream>

using std::cout;
using std::endl;

// Time abstract data type (ADT) definition
class Time {
public:
   Time();                           // constructor
   void setTime( int, int, int ); // set hour, minute, second
   void printMilitary();          // print military time format
   void printStandard();          // print standard time format
private:
   int hour;     // 0 - 23
   int minute;   // 0 - 59
   int second;   // 0 - 59
};

// Time constructor initializes each data member to zero.
// Ensures all Time objects start in a consistent state.
Time::Time() { hour = minute = second = 0; }

// Set a new Time value using military time. Perform validity
// checks on the data values. Set invalid values to zero.
void Time::setTime( int h, int m, int s )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0;
   minute = ( m >= 0 && m < 60 ) ? m : 0;
   second = ( s >= 0 && s < 60 ) ? s : 0;
}
```

Note the `::` preceding the function names.

```
// Print Time in military format
void Time::printMilitary()
{
   cout << ( hour < 10 ? "0" : "" ) << hour << ":"
        << ( minute < 10 ? "0" : "" ) << minute;
}

// Print Time in standard format
void Time::printStandard()
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
        << ":" << ( minute < 10 ? "0" : "" ) << minute
        << ":" << ( second < 10 ? "0" : "" ) << second
        << ( hour < 12 ? " AM" : " PM" );
}

```

CREATING OBJECTS OF THE *TIME* CLASS

- Class definition and declaration
  - Once a class has been defined, it can be used as a type in object, array and pointer declarations
  - Example:

```
Time sunset,                    // object of type Time
     arrayOfTimes[ 5 ],         // array of Time objects
     *pointerToTime,            // pointer to a Time object
```

Note: The class name becomes the new type specifier.

CONSTRUCTOR

A constructor is a special member function of a class. The name of a constructor is same as the class name, and it doesn’t have a return value.

We can use constructors to initialize an object. A constructor is called automatically for each class object when the object is created.

Initializing Class Objects: Constructors

- Constructors
  - Initialize class members
  - Same name as the class
  - No return type
  - Member variables can be initialized by the constructor or set afterwards
- Passing arguments to a constructor
  - When an object of a class is declared, initializers can be provided
  - Format of declaration with initializers:

    Class-type ObjectName(value1,value2,…);
  - Default arguments may also be specified in the constructor prototype

```cpp
// Time abstract data type definition
class Time {
public:
   Time( int = 0, int = 0, int = 0 );  // default constructor
   void setTime( int, int, int ); // set hour, minute, second
   void printMilitary();          // print military time format
   void printStandard();          // print standard time format
private:
   int hour;     // 0 - 23
   int minute;   // 0 - 59
   int second;   // 0 - 59
};

```

Notice that default settings for the three member variables are set in constructor prototype. No names are needed; the defaults are applied in the order the member variables are declared.

DEFAULT CONSTRUCTOR AND OVERLOADED CONSTRUCTOR

A default constructor is one that can be called without supplying any arguments.

Constructor with a parameter list is called parameterized constructor.

Constructors can be overloaded.

```
class BankAccount{
public:
BankAccount(); // default constructor
BankAccount(double);
BankAccount(string, string, double);
// ...
private:
string name, account_number;
double account_balance;
};
// the user-defined default constructor
// BankAccount() is invoked.
BankAccount a1;
// constructor BankAccount(string, string, double)
// is invoked.
BankAccount a2("Jerry Wang", "123888", 2000);
```

Using Destructors

- Destructors
  - Are member function of class
  - Perform termination housekeeping before the system reclaims the object's memory
  - Complement of the constructor
  - Name is tilde (**~**) followed by the class name (i.e., **`~Time`**)
    - Recall that the constructor's name is the class name
  - Receives no parameters, returns no value
  - One destructor per class
    - No overloading allowed

```cpp
class CreateAndDestroy {
public:
   CreateAndDestroy( int );  // constructor
   ~CreateAndDestroy();      // destructor
private:
   int data;
};


```

```
CreateAndDestroy::CreateAndDestroy( int value )
{
   data = value;
   cout << "Object " << data << "   constructor";
}

CreateAndDestroy::~CreateAndDestroy()
   { cout << "Object " << data << "   destructor " << endl; }
```

Constructor and Destructor changed to print when they are called.

Consider the following class definition:

```
class inventory
{
public:
    inventory();                                //Line 1
    inventory(string);                          //Line 2
    inventory(string, int, double);             //Line 3
    inventory(string, int, double, int);        //Line 4

    //Add additional functions

private:
    string name;
    int itemNum;
    double price;
    int unitsInStock;
};
```

This class has four constructors and four member variables. Suppose that the definitions of the constructors are as follows:

```
inventory::inventory() //default constructor
{
    name = "";
    itemNum = -1;
    price = 0.0;
    unitsInStock = 0;
}

inventory::inventory(string n)
{
    name = n;
    itemNum = -1;
    price = 0.0;
    unitsInStock = 0;
}

inventory::inventory(string n, int iNum, double cost)
{
    name = n;
    itemNum = iNum;
    price = cost;
    unitsInStock = 0;
}
```

```
inventory::inventory(string n, int iNum, double cost, int inStock)
{
    name = n;
    itemNum = iNum;
    price = cost;
    unitsInStock = inStock;
}
```

Consider the following declarations:

```
inventory item1;
inventory item2("Dryer");
inventory item3("Washer", 2345, 278.95);
inventory item4("Toaster", 8231, 34.49, 200);
```

For **item1**, the default constructor in Line 1 executes because no value is passed to this variable. For **item2**, the constructor in Line 2 executes because only one parameter, which is of type **string**, is passed, and it matches with the constructor in Line 2. For **item3**, the constructor in Line 3 executes because three parameters are passed to **item3**, and they match with the constructor in Line 3. Similarly, for **item4**, the constructor in Line 4 executes (see Figure 10-7).