Misr International University (MIU)

Faculty of Computer Science

Computer Science Program

CSC 105: Introduction to Programming & Problem Solving

Dr. Ayman Ezzat and Dr. Ashraf AbdelRaouf

Lecture 7

Classes and Data Abstraction

Objectives

In this Lecture, you will:

- Learn about classes
- Learn about `private`, `protected`, and `public` members of a class
- Explore how classes are implemented
- Examine constructors and destructors
- Learn about the abstract data type (ADT)
- Explore how classes are used to implement ADTs
- Learn about information hiding
- Explore how information hiding is implemented in C++
- Learn about the `static` members of a class

Classes

- Class: collection of a fixed number of components (members)
- Definition syntax:

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

  - Defines a data type; no memory is allocated
  - Don't forget the semicolon after closing brace

Classes (continued)

- Class member can be a variable or a function
- If a member of a `class` is a variable
  - It is declared like any other variable
- In the definition of the `class`
  - You cannot initialize a variable when you declare it
- If a member of a class is a function
  - Function prototype is listed
  - Function members can (directly) access any member of the `class`

Classes (continued)

- Three categories of class members
  - `private` (default)
    - Member cannot be accessed outside the `class`
  - `Public`
    - Member is accessible outside the class
  - `protected`

Classes (continued)

Suppose that we want to define a class to implement the time of day in a program. Because a clock gives the time of day, let us call this `class clockType`. Furthermore, to represent time in computer memory, we use three `int` variables: one to represent the hours, one to represent the minutes, and one to represent the seconds.

Suppose these three variables are:

```
int hr;
int min;
int sec;
```

We also want to perform the following operations on the time:

1. Set the time.
2. Retrieve 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 the two times for equality.

Classes (continued)

```
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;
};
```

These functions cannot modify the member variables of a variable of type `clockType`

`const`: formal parameter can't modify the value of the actual parameter

`private` members, can't be accessed from outside the class

Unified Modeling Language Class Diagrams

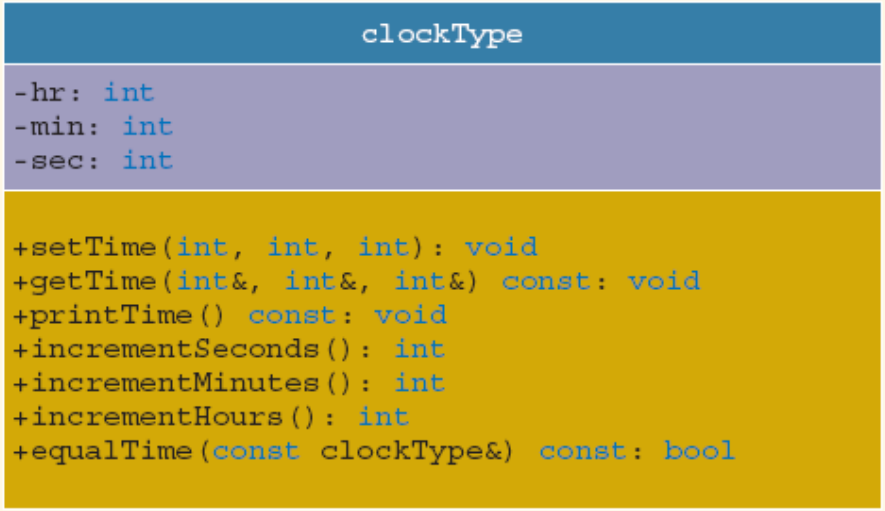

FIGURE 12-1 UML class diagram of the `class clockType`

+: member is `public`

-: member is `private`

#: member is `protected`

Variable (Object) Declaration

- Once a `class` is defined, you can declare variables of that type

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

- A `class` variable is called a `class` object or `class` instance

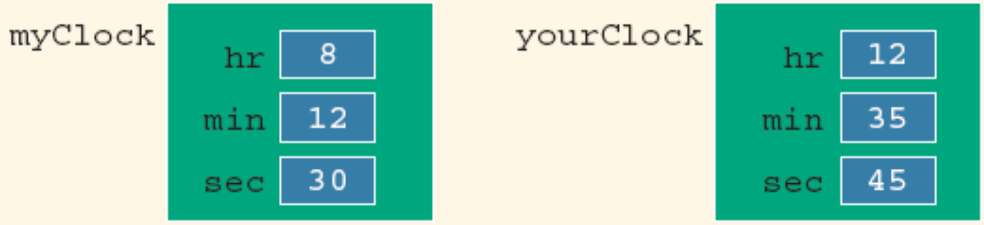

FIGURE 12-2 Objects `myClock` and `yourClock`

Accessing Class Members

- Once an object is declared, it can access the `public` members of the class
- Syntax:

```
classObjectName.memberName
```

  - The dot (`.`) is the **member access operator**
- If object is declared in the definition of a member function of the `class`, it can access the `public` and `private` members

EXAMPLE 12-1

Suppose we have the following declaration (say, in a user's program):

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

Consider the following statements:

```
myClock.setTime(5, 2, 30);
myClock.printTime();
yourClock.setTime(x, y, z);    //assume x, y, and z are
                               //variables of type int
```

```
if (myClock.equalTime(yourClock))
.
.
.
```

These statements are legal; that is, they are syntactically correct.

The objects `myClock` and `yourClock` can access only **`public`** members of the class. Thus, the following statements are illegal because `hr` and `min` are declared as **`private`** members of the **`class`** `clockType` and, therefore, cannot be accessed by the objects `myClock` and `yourClock`:

```
myClock.hr = 10;                //illegal
myClock.min = yourClock.min;    //illegal
```

Built-in Operations on Classes

- Most of C++'s built-in operations do not apply to classes
  - Arithmetic operators cannot be used on `class` objects unless the operators are overloaded
  - You cannot use relational operators to compare two `class` objects for equality
- Built-in operations valid for `class` objects:
  - Member access (`.`)
  - Assignment (=)

Assignment Operator and Classes

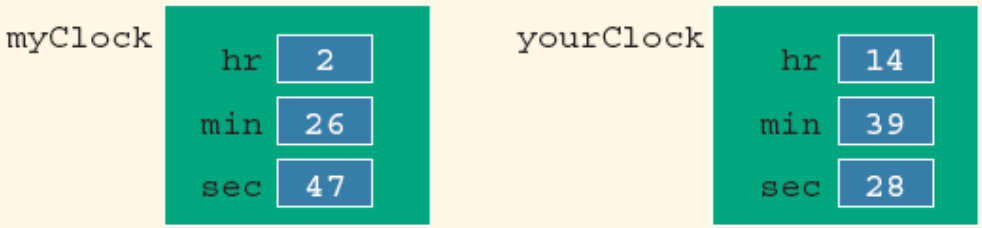

FIGURE 12-3 Objects `myClock` and `yourClock`

```
myClock = yourClock;          //Line 1
```

copies the value of `yourClock` into `myClock`.

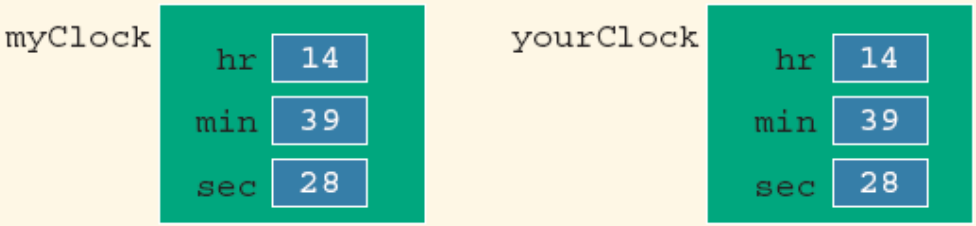

FIGURE 12-4 Objects `myClock` and `yourClock` after the assignment statement `myClock = yourClock;` executes

Class Scope

- An object can be automatic or static
- A member of the `class` is local to the `class`
- You access a `class` member outside the `class` by using the `class` object name and the member access operator (`.`)

Functions and Classes

- Objects can be passed as parameters to functions and returned as function values
- As parameters to functions
  - Objects can be passed by value or by reference
- If an object is passed by value
  - Contents of data members of the actual parameter are copied into the corresponding data members of the formal parameter

Reference Parameters and Class Objects (Variables)

- Passing by value might require a large amount of storage space and a considerable amount of computer time to copy the value of the actual parameter into the formal parameter
- If a variable is passed by reference
  - The formal parameter receives only the address of the actual parameter

Reference Parameters and Class Objects (Variables) (continued)

- Pass by reference is an efficient way to pass a variable as a parameter
  - Problem: when passing by reference, the actual parameter changes when formal parameter changes
  - Solution: use `const` in the formal parameter declaration

Implementation of Member Functions

Scope resolution operator

```
void clockType::setTime(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;
}
```

Implementation of Member Functions (continued)

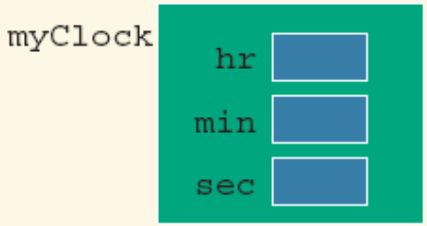

FIGURE 12-5 Object `myClock`

```
myClock.setTime(3, 48, 52);
```

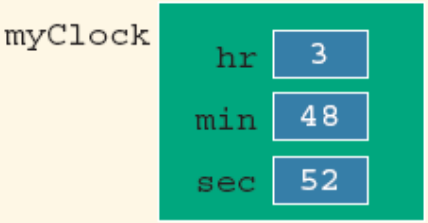

FIGURE 12-6 Object `myClock` after the statement `myClock.setTime(3, 48, 52);` executes

```
void clockType::getTime(int& hours, int& minutes,
                        int& seconds) const
{
    hours = hr;
    minutes = min;
    seconds = sec;
}

void clockType::printTime() const
{
    if (hr < 10)
        cout << "0";
    cout << hr << ":";

    if (min < 10)
        cout << "0";
    cout << min << ":";

    if (sec < 10)
        cout << "0";
    cout << sec;
}
```

```
void clockType::incrementHours()
{
    hr++;
    if (hr > 23)
        hr = 0;
}

void clockType::incrementMinutes()
{
    min++;
    if (min > 59)
    {
        min = 0;
        incrementHours(); //increment hours
    }
}

void clockType::incrementSeconds()
{
    sec++;

    if (sec > 59)
    {
        sec = 0;
        incrementMinutes(); //increment minutes
    }
}
```

```
bool clockType::equalTime(const clockType& otherClock) const
{
    return (hr == otherClock.hr
            && min == otherClock.min
            && sec == otherClock.sec);
}
```

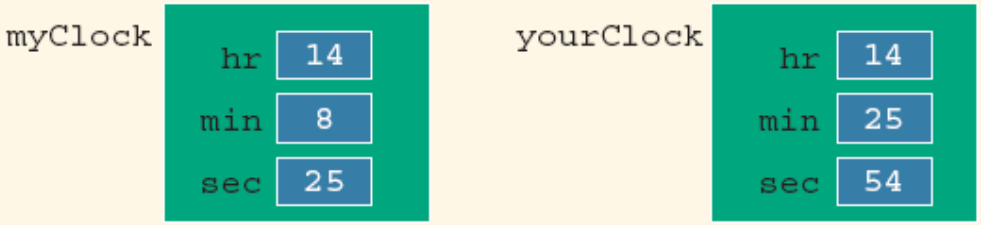

FIGURE 12-7 Objects `myClock` and `yourClock`

```
if (myClock.equalTime(yourClock))
```

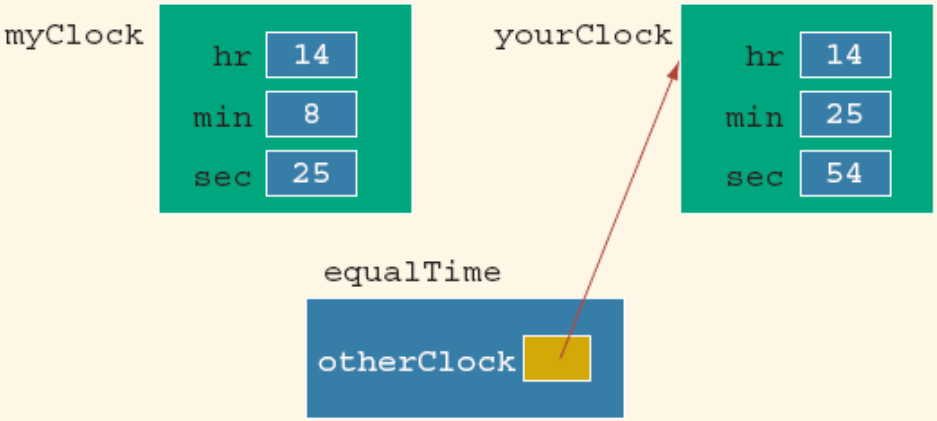

FIGURE 12-8 Object `myClock` and parameter `otherClock`

Implementation of Member Functions (continued)

- Once a `class` is properly defined and implemented, it can be used in a program
  - A program that uses/manipulates the objects of a class is called a **client** of that class
- When you declare objects of the `class` *`clockType`*, every object has its own copy of the member variables (`hr`, `min`, and `sec`)
- Variables such as `hr`, `min`, and `sec` are called **instance variables** of the class
  - Every object has its own instance of the data

Accessor and Mutator Functions

- Accessor function (Getters): member function that only accesses the value(s) of member variable(s)
- Mutator function (Setters): member function that modifies the value(s) of member variable(s)
- Constant function:
  - Member function that cannot modify member variables
  - Use `const` in function heading

Order of `public` and `private` Members of a Class

- C++ has no fixed order in which you declare `public` and `private` members
- By default, all members of a `class` are `private`
- Use the member access specifier `public` to make a member available for `public` access

Order of public and private Members of a Class (continued)

EXAMPLE 12-3

This declaration is the same as before. For the sake of completeness, we include the class definition:

```
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;
};
```

Order of public and private Members of a Class (continued)

EXAMPLE 12-4

```
class clockType
{
private:
    int hr;
    int min;
    int sec;
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;
};
```

Order of public and private Members of a Class (continued)

EXAMPLE 12-5

```
class clockType
{
    int hr;
    int min;
    int sec;

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;
};
```

Constructors

- Use constructors to guarantee that data members of a class are initialized
- Two types of constructors:
  - With parameters
  - Without parameters (default constructor)
- The name of a constructor is the same as the name of the class
- A constructor has no type

Constructors (continued)

- A class can have more than one constructor
  - Each must have a different formal parameter list
- Constructors execute automatically when a class object enters its scope
  - 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

Constructors (continued)

```
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;
    clockType(int, int, int);  //constructor with parameters
    clockType();  //default constructor

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

```
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;
}

clockType::clockType()  //default constructor
{
   hr = 0;
   min = 0;
   sec = 0;
}
```

Can be replaced with:

```
setTime(hours, minutes, seconds);
```

Invoking a Constructor

- A constructor is automatically executed when a class variable is declared
- To invoke the default constructor:

- Example:

```
clockType yourClock;
```

Invoking a Constructor with Parameters

- Syntax:

```
className classObjectName(argument1, argument2, ...);
```

- The number of arguments and their type should match the formal parameters (in the order given) of one of the constructors
  - Otherwise, C++ uses type conversion and looks for the best match
  - Any ambiguity leads to a compile-time error

Constructors and Default Parameters

```
clockType clockType(int = 0, int = 0, int = 0);    //Line 1
```

- If you replace the constructors of `clockType` with the constructor in Line 1, you can declare `clockType` objects with zero, one, two, or three arguments as follows:

```
clockType clock1;                    //Line 2
clockType clock2(5);                 //Line 3
clockType clock3(12, 30);            //Line 4
clockType clock4(7, 34, 18);         //Line 5
```

Classes and Constructors: A Precaution

- If a class has no constructor(s), C++ provides the default constructor
  - However, object declared is still uninitialized
- If a class includes constructor(s) with parameter(s), but not the default constructor
  - C++ does not provide the default constructor

Arrays of Class Objects (Variables) and Constructors

- If a class has constructors and you declare an array of that class's objects, the class should have the default constructor

```
clockType arrivalTimeEmp[100];
```

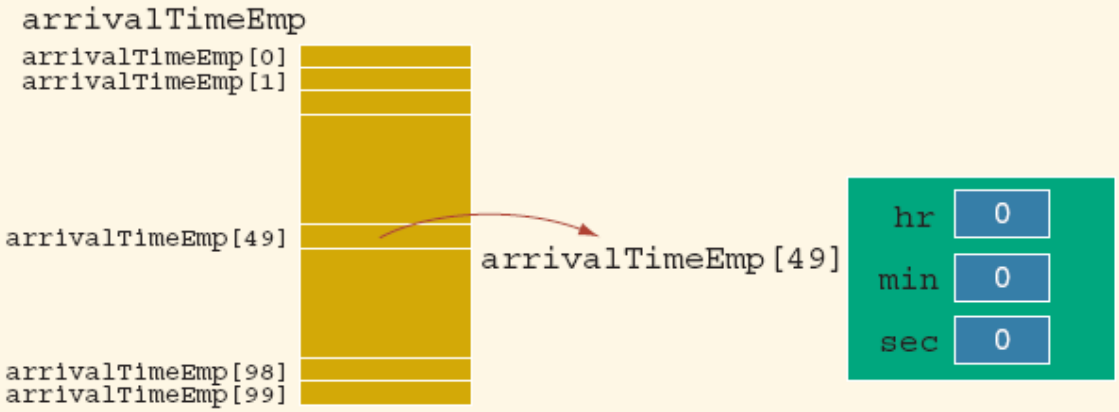

FIGURE 12-9 Array arrivalTimeEmp

Arrays of Class Objects (Variables) and Constructors (continued)

```
arrivalTimeEmp[49].setTime(8, 5, 10);
```

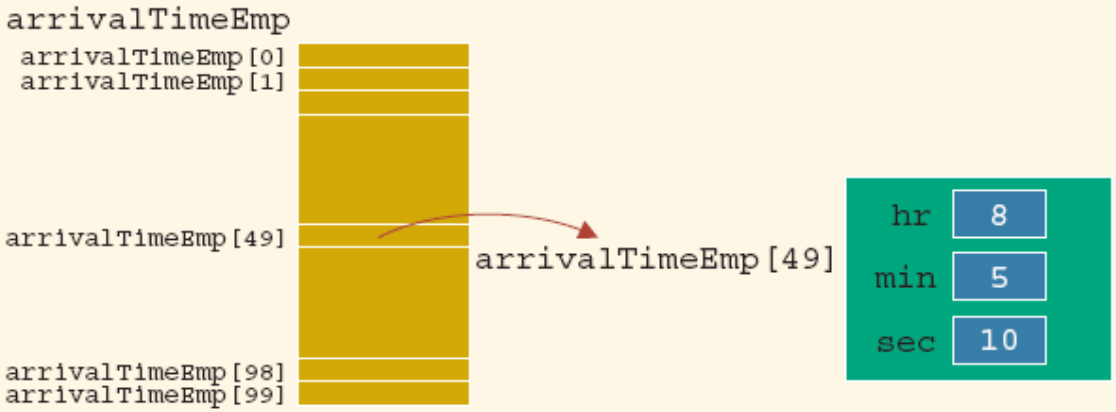

FIGURE 12-10 Array `arrivalTimeEmp` after setting the time of employee 49

```
for (int j = 0; j < 100; j++)                    //Line 3
{
    cout << "Employee " << (j + 1)
         << " arrival time: ";
    arrivalTimeEmp[j].printTime();               //Line 4
    cout << endl;
}
```

Destructors

- Destructors are functions without any type
- The name of a destructor is the character '~' followed by class name
  - For example:

    ```
    ~clockType();
    ```
- A class can have only one destructor
  - The destructor has no parameters
- The destructor is automatically executed when the class object goes out of scope

A struct versus a class

- By default, members of a `struct` are `public`
  - `private` specifier can be used in a `struct` to make a member private
- By default, the members of a `class` are `private`
- `classes` and `structs` have the same capabilities

A struct versus a class (continued)

- In C++, the definition of a `struct` was expanded to include member functions, constructors, and destructors
- If all member variables of a `class` are `public` and there are no member functions
  - Use a `struct`

Static Members of a Class

- Use the keyword `static` to declare a function or variable of a class as `static`
- A `public static` function or member of a class can be accessed using the class name and the scope resolution operator
- `static` member variables of a class exist even if no object of that `class` type exists