UNIVERSITY of LOUISVILLE
dare to be great

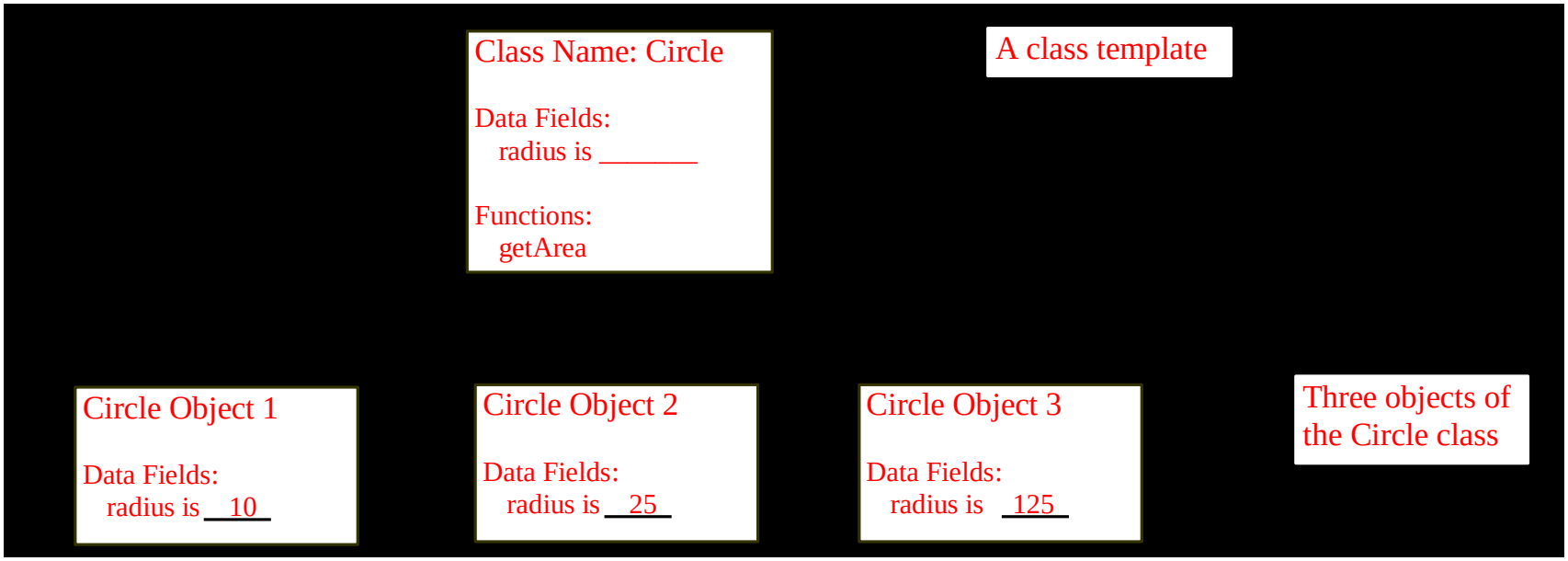
CPP Chapter 5:
Object Oriented Programming

CECS130
Introduction to Programming Languages
Dr. Roman V. Yampolskiy

OO Programming Concepts

- Object-oriented programming (OOP) involves programming using objects.
- An *object* represents an entity in the real world that can be distinctly identified.
- For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects.
- An object has a unique identity, state, and behaviors.
- The *state* of an object consists of a set of *data fields* (also known as *properties*) with their current values.
- The *behavior* of an object is defined by a set of functions.

Objects



An object has both a state and behavior. The state defines the object, and the behavior defines what the object does.

Classes

- Class: collection of a fixed number of components
- The components of a `class` are called members
- The general syntax for defining a `class`:

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

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`
  - 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)

- `class` is a reserved word
- Class defines a data type, no memory is allocated
- Don’t forget the semicolon after the closing brace of the `class`

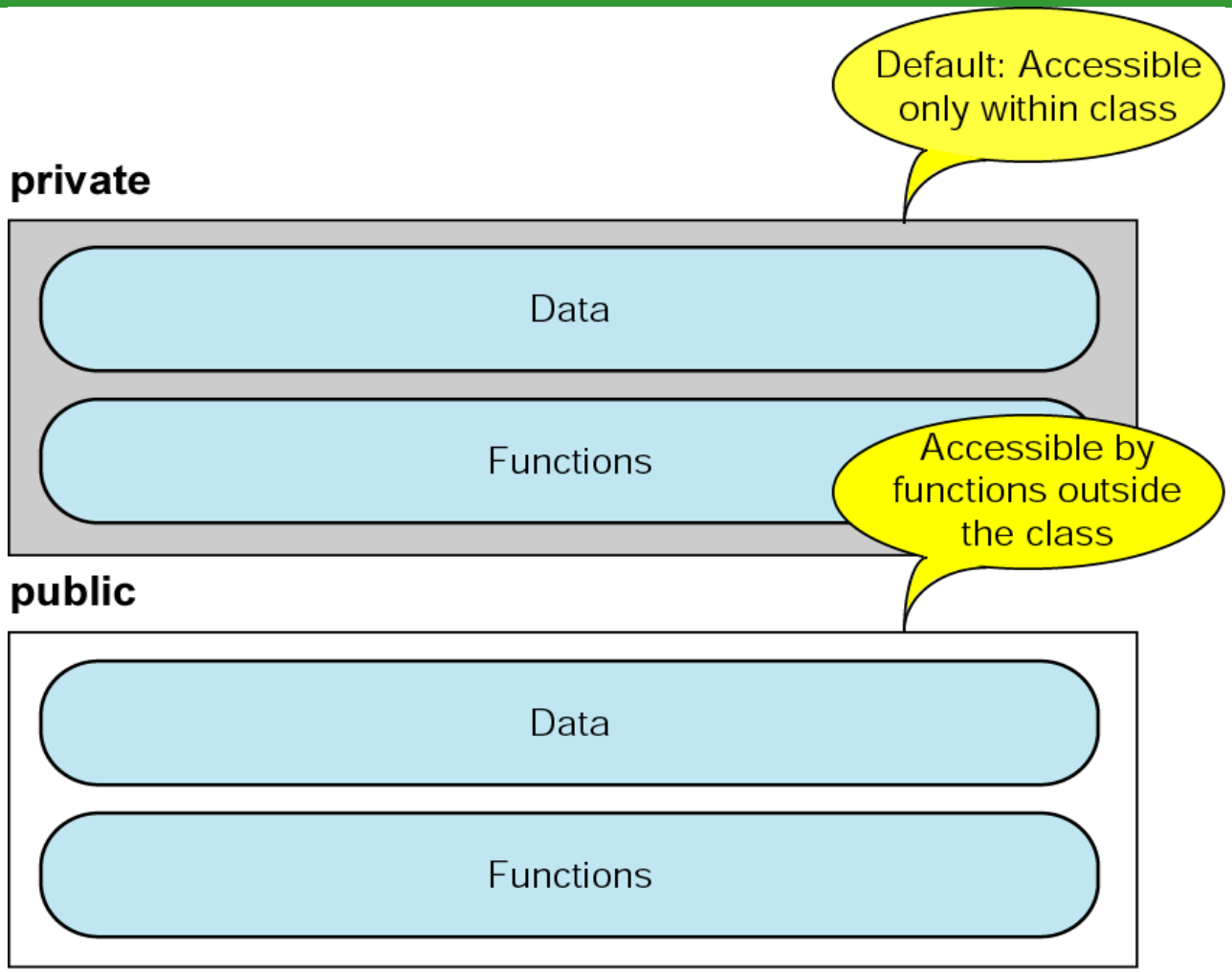
Classes (continued)

- Three categories of class members:
  - `private`
  - `public`
  - `protected`
- By default, all members of a class are `private`
- If a member of a class is `private`
  - It cannot be accessed outside the `class`

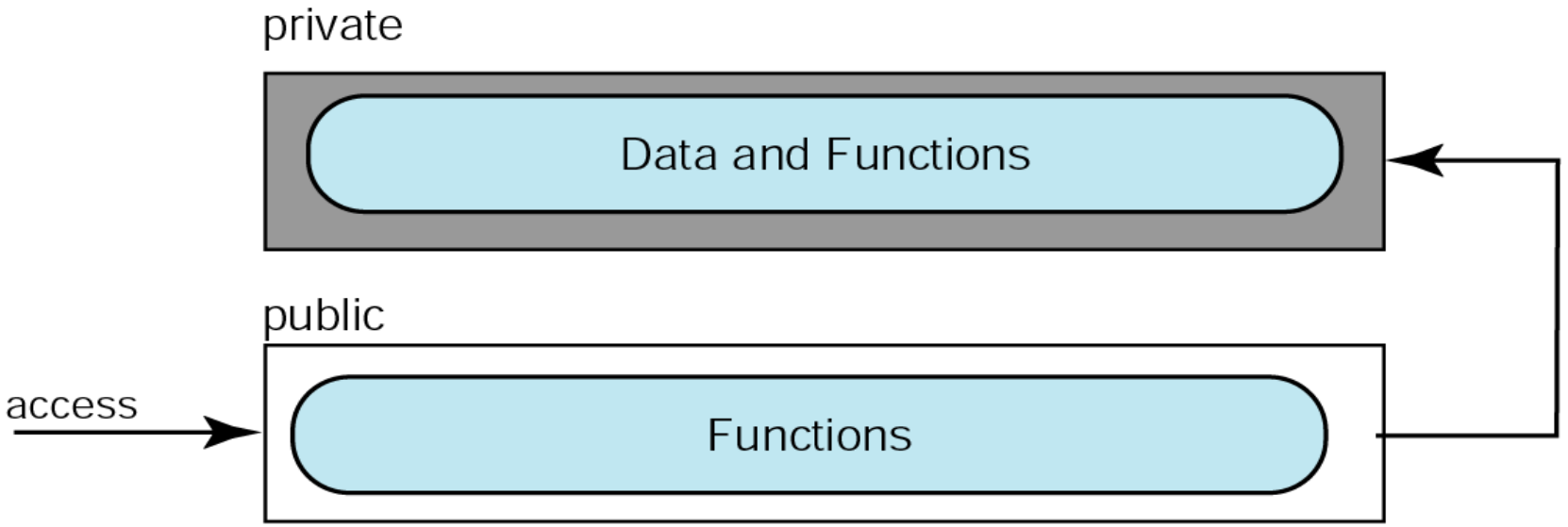
Classes (continued)

- A `public` member is accessible outside the class
- To make a member of a `class` `public`
  - Use the label `public` with a colon
- `private`, `protected`, and `public` are reserved words

Class access specifiers



Accessing data in a class



Class Example

```
class Fraction
{
 private:
   int numerator;

   int denominator;
public:
   void store (int numer,
               int denom);

   void print () const;
} ; // class Fraction
```

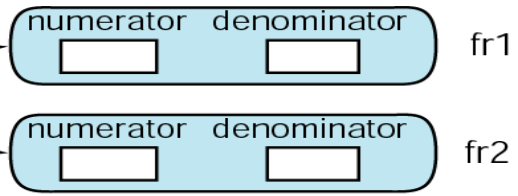
(a) Class Declaration

```
void Fraction :: store
     (int  numer,
      int  denom)
{
     numerator = numer;
```

```
void Fraction :: print() const
{
     cout << numerator
          << "/"
          << denominator;
     return;
} // Fraction print
```

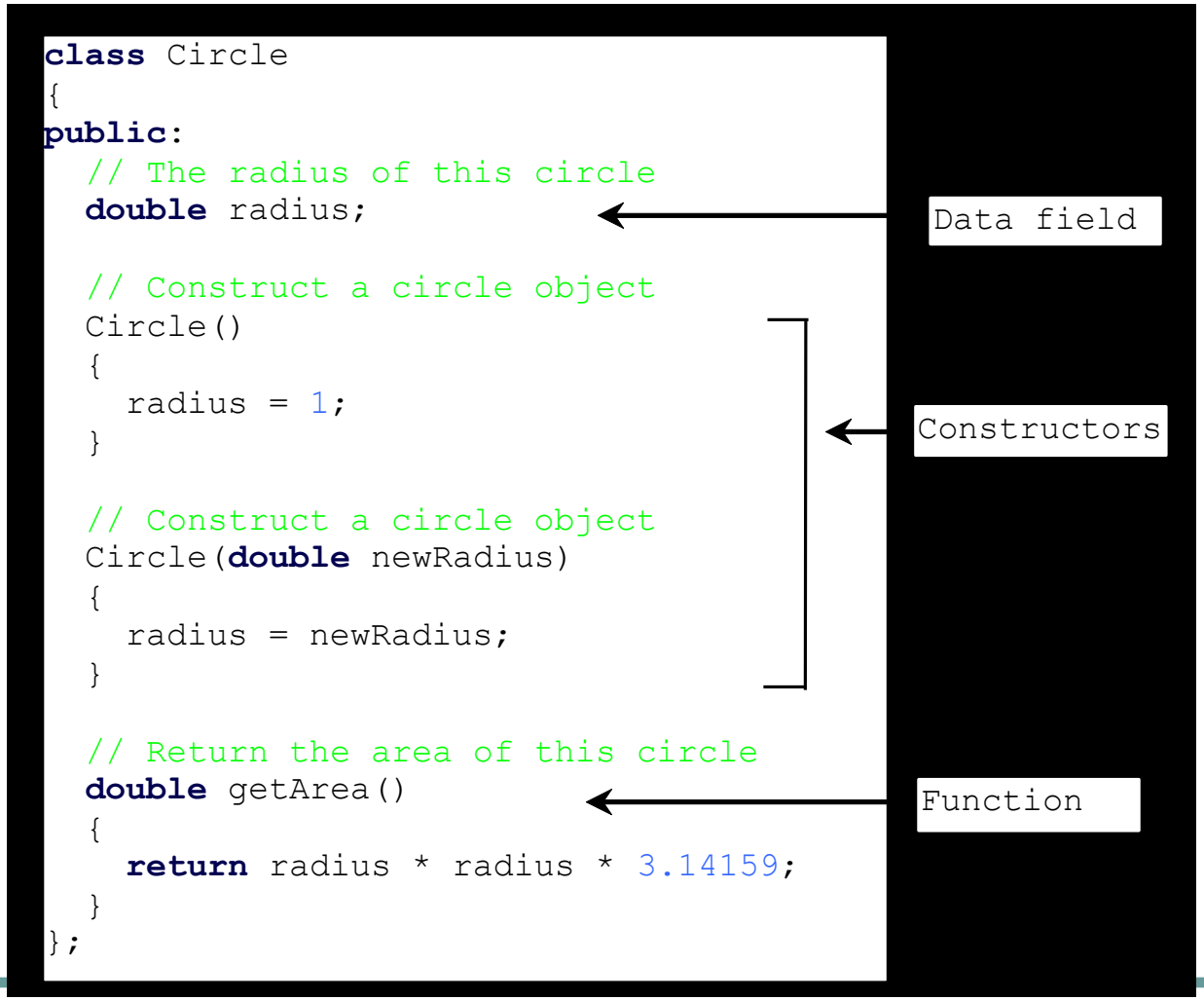
(b) Function Definitions

```
int main ()
{
    Fraction  fr1;
    Fraction  fr2;
    ...
} // main
```



(c) Class Instantiation

Class Example 2



Class Another Example

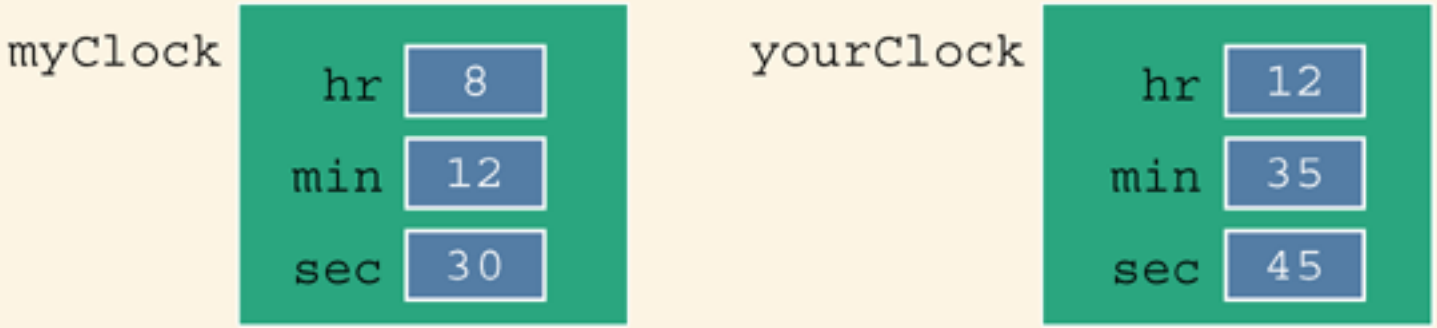
```
class clockType
{
public:
    void setTime(int, int, int);
    void getTime(int&, int&, int&) const;
    void printTime() const;
    void incrementSeconds();
    void incrementMinutes();
    void incrementHours();
    bool equalTime(const clockType&) const;

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

Variable (Object) Declaration

- Once a `class` is defined, you can declare variables of that type
- In C++ terminology, a `class` variable is called a `class` object or `class` instance
- The syntax for declaring a `class` object is the same as for declaring any other variable

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

Objects Visualized



Objects `myClock` and `yourClock`

Accessing Class Members

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

```
classObjectName.memberName
```

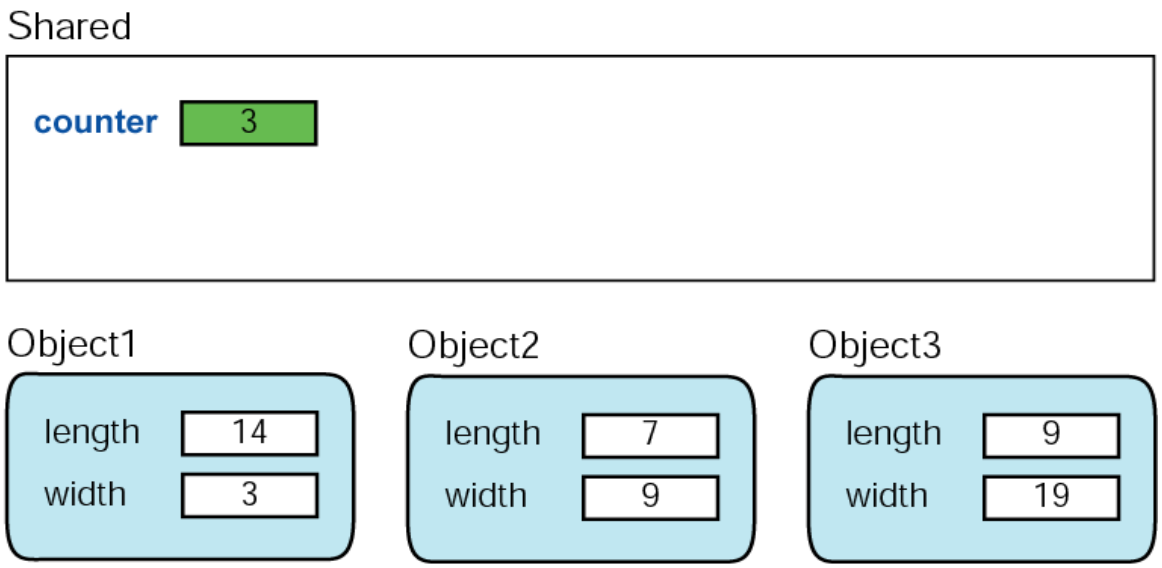
- The dot (`.`) is called the **member access operator**

Accessing Class Members (continued)

- The `class` members that a `class` object can access depend on where the object is declared.
  - If the object is declared in the definition of a member function of the `class`, then the object can access both the `public` and `private` members.
  - If the object is declared elsewhere (for example, in a user's program), then the object can access *only* the `public` members of the `class`.

Class Members Example

```
class Sample
{
      private:
      static  int counter;
              int length;
              int width;
      public:
         .
         .
         .
};    // Sample
         .
         .
         .
//    Function Definitions
      Sample object1;
      Sample object2;
      Sample object3;
```



Example: Working with Objects

```
clockType myClock;
clockType yourClock;

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))
.
.
.
```

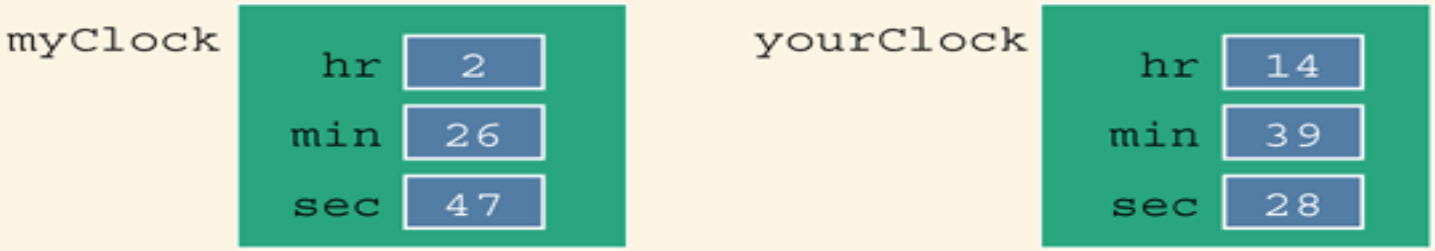
These statements are legal; that is, they are syntactically correct.

```
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
- The two built-in operations that are valid for `class` objects are member access (`.`) and assignment (`=`)

Assignment Operator and Classes



Objects `myClock` and `yourClock`

The statement:

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

copies the value of `yourClock` into `myClock`. That is,

- the value of `yourClock.hr` is copied into `myClock.hr`,
- the value of `yourClock.min` is copied into `myClock.min`, and
- the value of `yourClock.sec` is copied into `myClock.sec`.

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 & 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

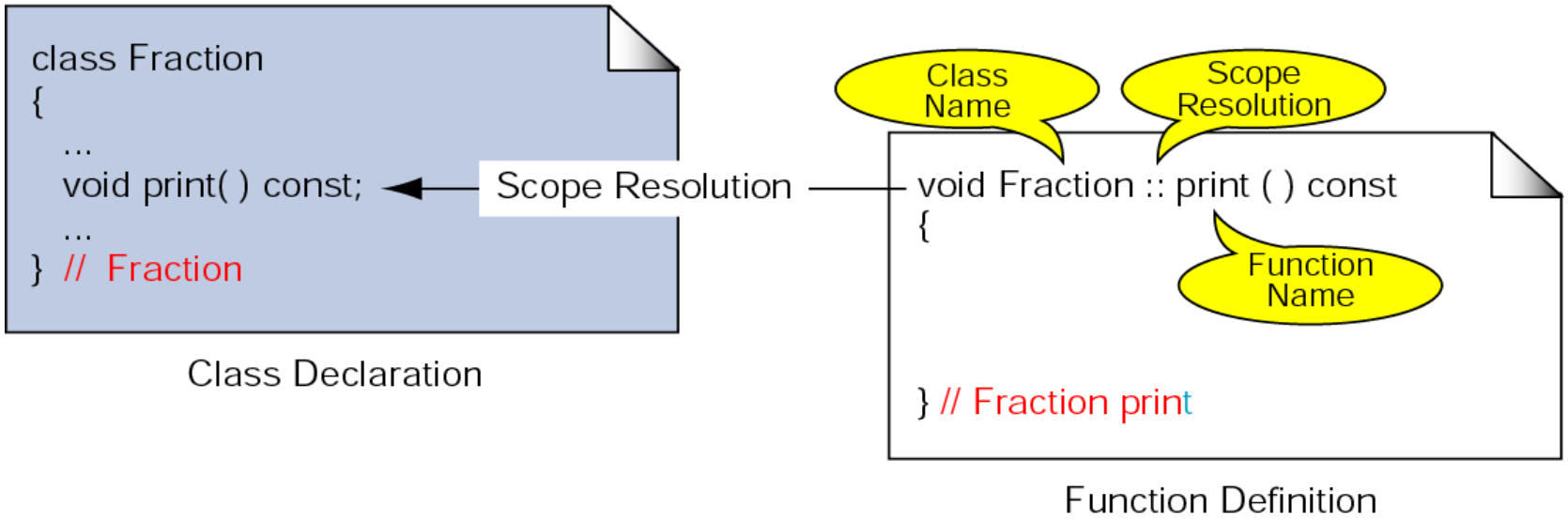
Reference Parameters & Variables

- Pass by reference is an efficient way to pass a variable as a parameter
- If a variable is passed by reference
  - Then the actual parameter changes when the formal parameter changes
- You can pass a variable by reference and still prevent the function from changing its value
  - Use the keyword `const` in the formal parameter declaration

Implementation of Member Functions

- The identifiers `setTime`, `printTime`, and so forth are local to the class; we cannot reference them (directly) outside the class.
- In order to reference these identifiers, we use the **scope resolution operator**, `::` (double colon).
- In the function definition's heading, the name of the function is the name of the class, followed by the scope resolution operator, followed by the function name.

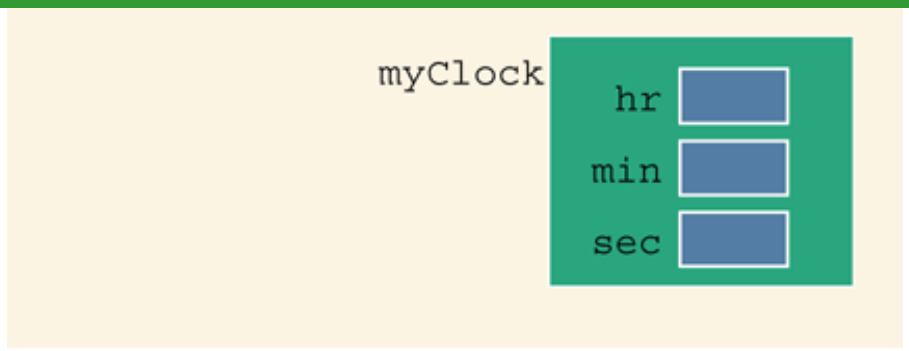
Scope resolution operator



Implementation of Member Functions

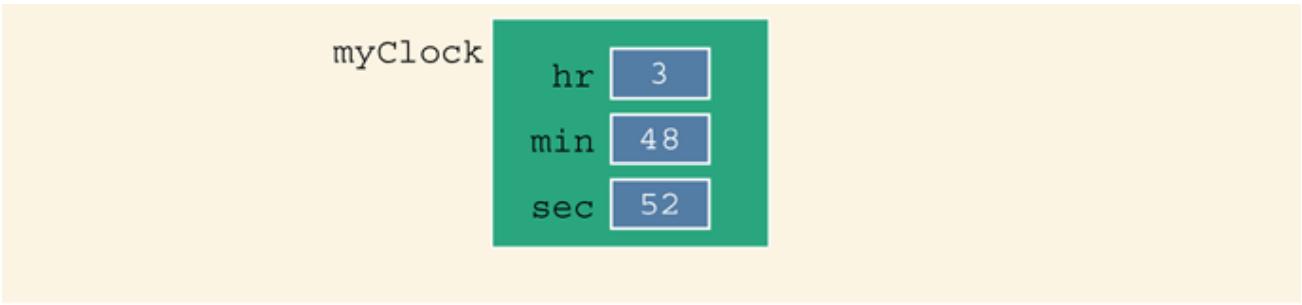
```
void clockType::setTime(int hours, int minutes, int seconds)
{
    if (0 <= hours && hours < 24)
        hr = hours;
    else
        hr = 0;

    if (0 <= minutes && minutes < 60)
        min = minutes;
    else
        min = 0;

    if (0 <= seconds && seconds < 60)
        sec = seconds;
    else
        sec = 0;
}
```

SetTime()



Object `myClock`

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



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

getTime() and printTime()

```
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;
}
```

incrementHours() and incrementMinutes()

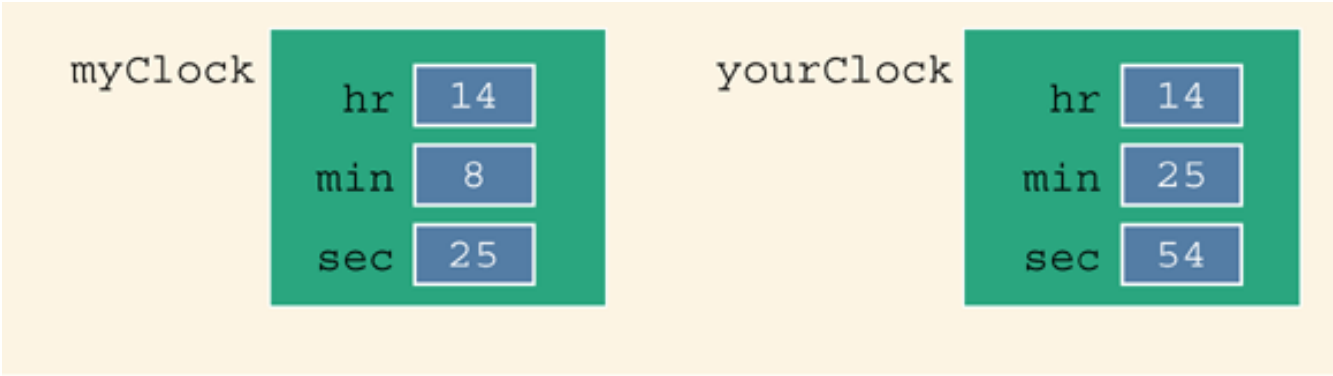
```
void clockType::incrementHours()
{
     hr++;
     if (hr > 23)
          hr = 0;
}

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

incrementSeconds()

```
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);
}
```

Suppose that `myClock` and `yourClock` are objects of type `clockType`, as declared previously. Further suppose that we have `myClock` and `yourClock` as shown in the next Figure.



Objects `myClock` and `yourClock`

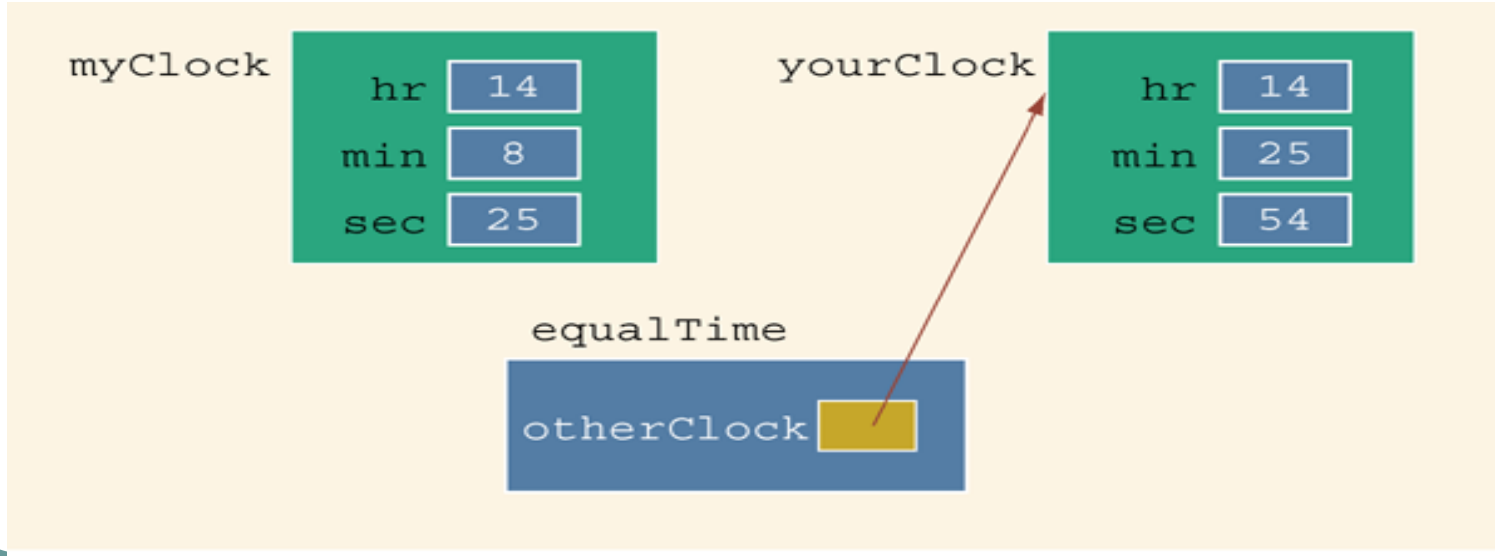
Consider the following statement:

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

In the expression:

`myClock.equalTime(yourClock)`

the object `myClock` accesses the member function `equalTime`. Because `otherClock` is a reference parameter, the address of the actual parameter `yourClock` is passed to the formal parameter `otherClock`.



Object `myClock` and parameter `otherClock`

Accessing Private Members

- Within the definition of this function, the object `otherClock` accesses the member variables `hr`, `min`, and `sec`.
- However, these member variables are `private`. So is there any violation? The answer is no.
- The function `equalTime` is a member of the `class` `clockType` and `hr`, `min`, and `sec` are the member variables.
- `otherClock` is an object of type `clockType`.
- Therefore, the object `otherClock` can access its `private` member variables within the definition of the function `equalTime`.

Instance Variables

- Once a `class` is properly defined and implemented, it can be used in a program.
- A program or software that uses and 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`.
- In object-oriented terminology, variables such as `hr`, `min`, and `sec` are called **instance variables** of the `class` because every object has its own instance of the data.

Accessor and Mutator Functions

- Accessor function: member function that only accesses (does not modify) the value(s) of the member variable(s)
- Mutator function: member function that modifies the value(s) of the member variable(s)
- Constant function:
  - Member function that cannot modify member variables
  - Include reserved word `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

Example: public before private

```
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;
};
```

Example: private before public

```
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;
};
```

Example: default - private

```
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
- Constructor without parameters is called the default constructor

Constructors (continued)

- The name of a constructor is the same as the name of the `class`.
- A constructor, even though it is a function, has no type. That is, 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.
- 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.

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;
};
```

Constructor

```
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;
}
```

Constructor (continued)

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

We can write the definition of the constructor with parameters by calling the function `setTime`, as follows:

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

Invoking a Constructor

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

```
className classObjectName;
```

The statement:

```
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`.

Invoking a Constructor with Parameters

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

where `argument1`, `argument2`, and so on, is either a variable or an expression.

Note the following:

- The number of arguments and their type should match the formal parameters (in the order given) of one of the constructors.
- If the type of the arguments does not match the formal parameters of any constructor (in the order given), C++ uses type conversion and looks for the best match.
- For example, an integer value might be converted to a floating-point value with a zero decimal part. Any ambiguity will result in a compile-time error.

Constructors and Default Parameters

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

If you replace the constructors of the `class clockType` with the constructor in Line 1 (the constructor with the default parameters), then 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
```

Arrays of Class Objects (Variables) and Constructors

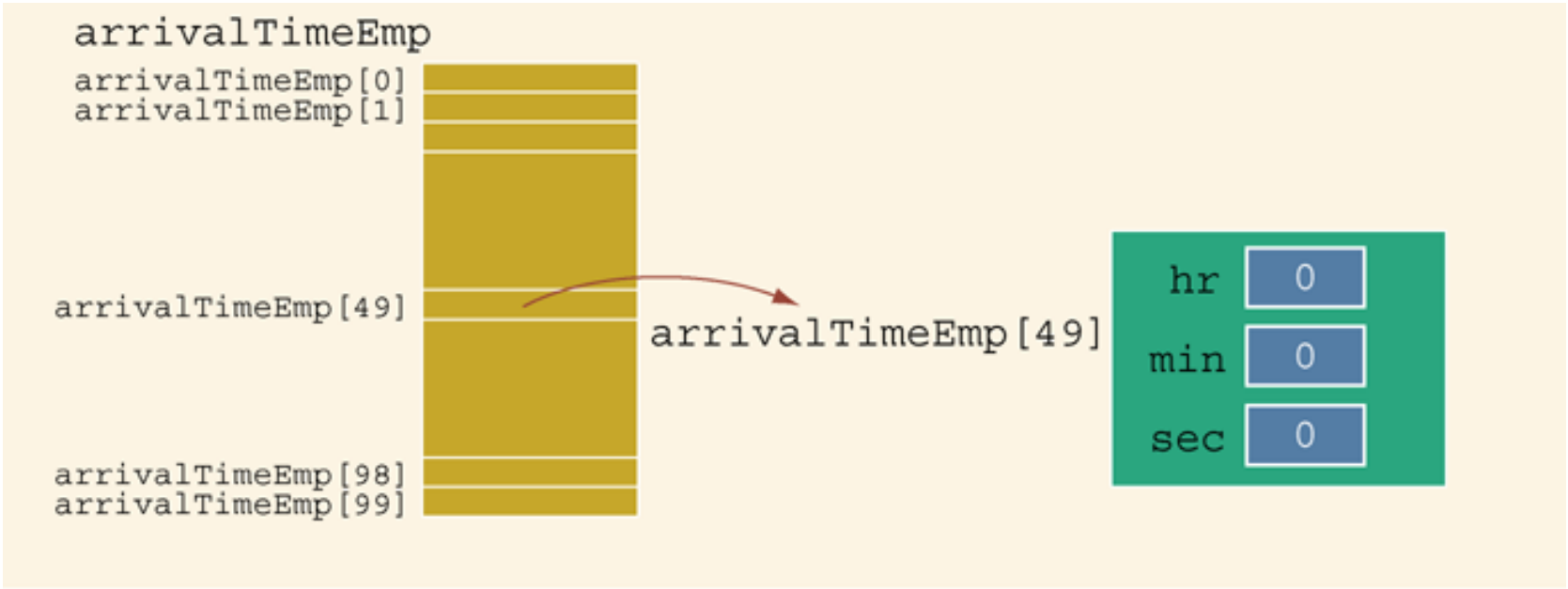
- If a class has constructors and you declare an array of that class's objects, the class should have the default constructor.
- The default constructor is typically used to initialize each (array) class object.

Consider the following statement:

```
clockType arrivalTimeEmp[100];                    //Line 1
```

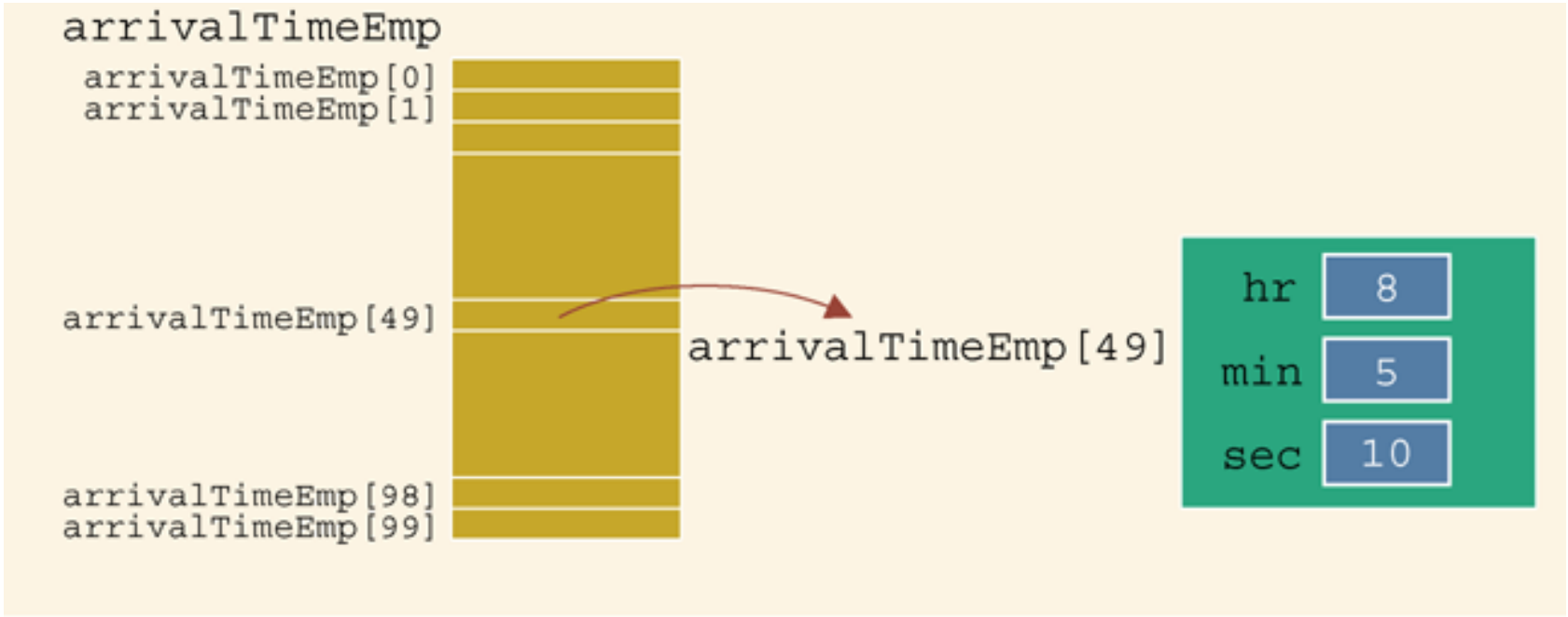
The statement in Line 1 creates the array of objects `arrivalTimeEmp[0]`, `arrivalTimeEmp[1]`, ..., `arrivalTimeEmp[99]`,

Arrays of Class Objects (Variables) and Constructors



Array arrivalTimeEmp

```
arrivalTimeEmp[49].setTime(8, 5, 10);          //Line 2
```



Array `arrivalTimeEmp` after setting the time of employee `49`

Classes and Constructors: A Precaution

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

Destructors

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

```
~clockType();
```

- A class can have only one destructor
  - It has no parameters
- The destructor is automatically executed when the class object goes out of scope

Data Abstract, Classes, and Abstract Data Types

- Abstraction
  - Separating design details from usage
  - Separating the logical properties from the implementation details
- Abstraction can also be applied to data

A struct Versus a Class

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

A struct Versus a Class (continued)

- 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`

Information Hiding

- Information hiding: hiding the details of the operations on the data
- Interface (header) file: contains the specification details
- Implementation file: contains the implementation details
- Include comments in the header file with the function prototypes that briefly describe the functions
  - Specify any preconditions and/or postconditions

Information Hiding (continued)

- **Precondition**: A statement specifying the condition(s) that must be true before the function is called
- **Postcondition**: A statement specifying what is true after the function call is completed

```
class clockType
{
public:
    void  setTime(int  hours, int  minutes, int seconds);
      //Function to set the time.
      //The time is set according to the parameters.
      //Postcondition: hr = hours; min = minutes;
      //                sec = seconds;
      //                The function checks whether the
      //                values of hours, minutes, and seconds
      //                are valid. If a value is invalid, the
      //                default value 0 is assigned.

    void getTime(int& hours, int& minutes, int& seconds) const;
      //Function to return the time.
      //Postcondition: hours = hr; minutes = min;
      //                seconds = sec;

    void printTime() const;
      //Function to print the time.
      //Postcondition: The time is printed in the form
      //                hh:mm:ss.
```

```
void incrementSeconds();
  //Function to increment the time by one second.
  //Postcondition: The time is incremented by one second.
  //                If the before-increment time is
  //                23:59:59, the time is reset to 00:00:00.

void incrementMinutes();
  //Function to increment the time by one minute.
  //Postcondition: The time is incremented by one minute.
  //                If the before-increment time is
  //                23:59:53, the time is reset to 00:00:53.

void incrementHours();
  //Function to increment the time by one hour.
  //Postcondition: The time is incremented by one hour.
  //                If the before-increment time is
  //                23:45:53, the time is reset to 00:45:53.

bool equalTime(const clockType& otherClock) const;
  //Function to compare the two times.
  //Postcondition: Returns true if this time is equal to
  //                otherClock; otherwise, returns false.
```

```
    clockType(int hours, int minutes, int seconds);
      //Constructor with parameters.
      //The time is set according to the parameters.
      //Postcondition: hr = hours; min = minutes;
      //                sec = seconds;
      //                The constructor checks whether the
      //                values of hours, minutes, and seconds
      //                are valid. If a value is invalid, the
      //                default value 0 is assigned.

    clockType();
      //Default constructor
      //The time is set to 00:00:00.
      //Postcondition: hr = 0; min = 0; sec = 0;

private:
    int hr;   //variable to store the hours
    int min; //variable to store the minutes
    int sec; //variable to store the seconds
};
```

Implementation File and User Program

```
//clockTypeImp.cpp, the implementation file

#include   <iostream>
#include   "clockType.h"

using namespace   std;
      .
      .
      .
//Place the definitions of the member functions of the class
//clockType here.
      .
      .

//The user program that uses the class clockType

#include   <iostream>
#include   "clockType.h"

using namespace   std;
      .
      .
      .
//Place the definitions of the function main and the other
//user-defined functions here
      .
      .
      .
```

Information Hiding (continued)

- Header file has an extension `.h`
- Implementation file has an extension `.cpp`
- Implementation file must include header file via include statement
- In an include statement
  - User-defined header files are enclosed in double quotes
  - System-provided header files are enclosed between angular brackets

The End!