Records (`structs`) (1 of 3)

- **`struct`**: a collection of a fixed number of components in which the components are accessed by name
  - The components may be of different types and are called the members of the **`struct`**
- Syntax

```
struct structName
{
    dataType1 identifier1;
    dataType2 identifier2;
       .
       .
       .
    dataTypen identifiern;
};
```

Records (structs) (2 of 3)

- A **struct** is a definition, not a declaration
  - Must declare a variable of that type to use it

```
struct houseType
{
    string style;
    int numOfBedrooms;
    int numOfBathrooms;
    int numOfCarsGarage;
    int yearBuilt;
    int finishedSquareFootage;
    double price;
    double tax;
};

//variable declaration
houseType newHouse;
```

CENGAGE Learning

Records (`structs`) (3 of 3)

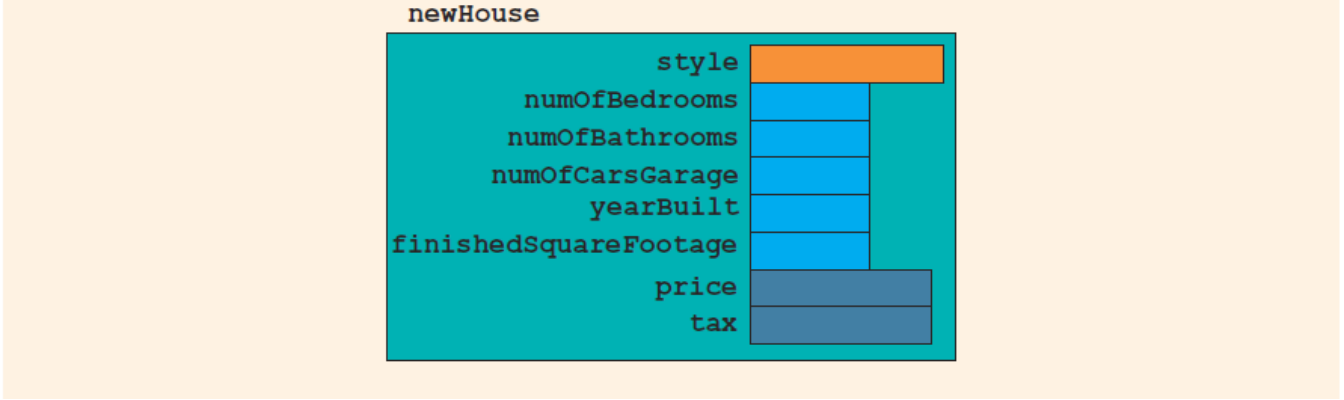

FIGURE 9-1 `struct newHouse`

Accessing `struct` Members (1 of 2)

- Syntax to access a **`struct`** member:

```
structVariableName.memberName
```

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

Accessing `struct` Members (2 of 2)

- To initialize the members of **`newStudent`**:

```
newStudent.GPA = 0.0;
newStudent.firstName = "John";
newStudent.lastName = "Brown";
```

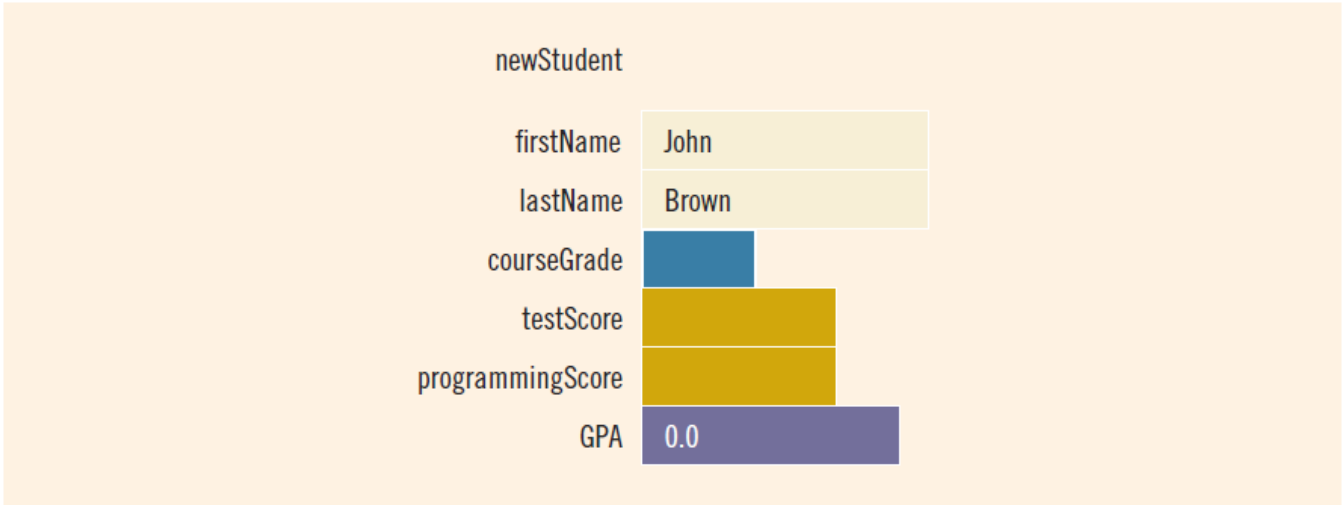

FIGURE 9-2 `struct` newStudent

Assignment (2 of 2)

- The assignment statement:

```
student = newStudent;
```

- is equivalent to the following statements:

```
student.firstName = newStudent.firstName;
student.lastName = newStudent.lastName;
student.courseGrade = newStudent.courseGrade;
student.testScore = newStudent.testScore;
student.programmingScore = newStudent.programmingScore;
student.GPA = newStudent.GPA;
```

CENGAGE Learning

Comparison (Relational Operators)

- Compare **`struct`** variables member-wise
  - No aggregate relational operations are allowed
- To compare the values of **`student`** and **`newStudent`**:

```
if (student.firstName == newStudent.firstName &&
    student.lastName == newStudent.lastName)
    .
    .
    .
```

Input/Output

- No aggregate input/output operations are allowed on a **`struct`** variable
- Data in a **`struct`** variable must be read or written one member at a time
- The following code would output **`newStudent`** contents:

```
cout << newStudent.firstName << " " << newStudent.lastName
     << " " << newStudent.courseGrade
     << " " << newStudent.testScore
     << " " << newStudent.programmingScore
     << " " << newStudent.GPA << endl;
```

CENGAGE Learning

struct Variables and Functions

- A **struct** variable can be passed as a parameter by value or by reference
- A function can return a value of type **struct**
- The following function displays the contents a **struct** variable of type **studentType**:

```
void printStudent(studentType student)
{
      cout << student.firstName << " " << student.lastName
           << " " << student.courseGrade
           << " " << student.testScore
           << " " << student.programmingScore
           << " " << student.GPA << endl;
}
```

Classes (1 of 4)

- Object-oriented design (OOD): a problem solving methodology
- Object: combines data and the operations on that data in a single unit
- Class: a collection of a fixed number of components
- Member: a component of a class

Classes (2 of 4)

- The general syntax for defining a **`class`**:

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

- A 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
  - You cannot initialize a variable when you declare it

CENGAGE Learning

Classes (3 of 4)

- If a member of a **`class`** is a function
  - A function prototype declares that member
  - Function members can (directly) access any member of the **`class`**
- A class definition defines only a data type
  - No memory is allocated
  - Remember the semicolon (`;`) after the closing brace

Classes (4 of 4)

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

CENGAGE Learning

Variable (Object) Declaration

- Once defined, you can declare variables of that `class` type
  - `clockType myClock;`
  - `clockType yourClock;`
- A `class` variable is called a class object or class instance

FIGURE 10-2 Objects `myClock` and `yourClock`

Accessing Class Members

- Once an object is declared, it can access the members of the class
- The general syntax for an object to access a member of a class:

```
classObjectName.memberName
```

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

CENGAGE Learning

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
  - Relational operators cannot be used to compare two class objects for equality
- Built-in operations that are valid for class objects:
  - Member access (.)
  - Assignment (=)

Assignment Operator and Classes

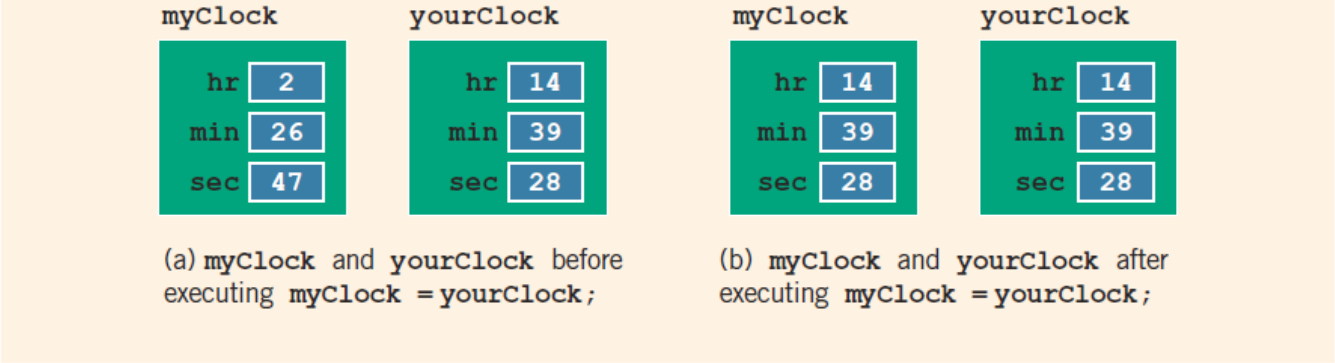

FIGURE 10-3 `myClock` and `yourClock` before and after executing the statement `myClock = yourClock;`

Class Scope (1 of 2)

- A **class** object can be automatic or static
  - Automatic: created when the declaration is reached and destroyed when the surrounding block is exited
  - Static: created when the declaration is reached and destroyed when the program terminates
- A member of a `class` has the same scope as a member of a `struct`

Class Scope (2 of 2)

- 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:
  - Class 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) (1 of 2)

- 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) (2 of 2)

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

Implementation of Member Functions (1 of 4)

- Must write the code for functions defined as function prototypes
- Prototypes are left in the class to keep the class smaller and to hide the implementation
- To access identifiers local to the class, use the scope resolution operator, (: :)

Implementation of Member Functions (2 of 4)

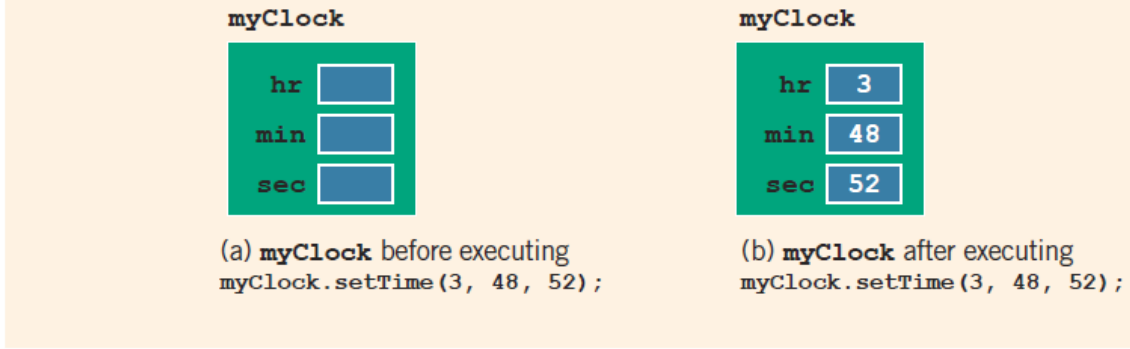

FIGURE 10-4 `myClock` before and after executing the statement `myClock.setTime(3, 48, 52);`

Implementation of Member Functions (3 of 4)

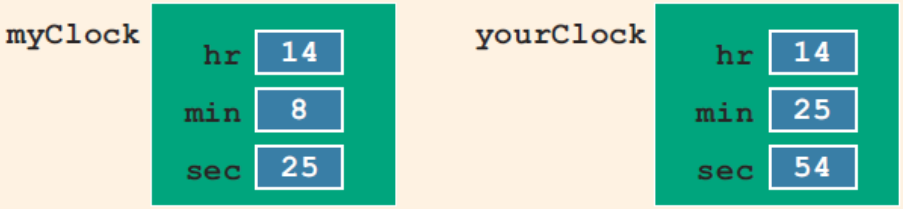

FIGURE 10-5 Objects **myClock** and **yourClock**

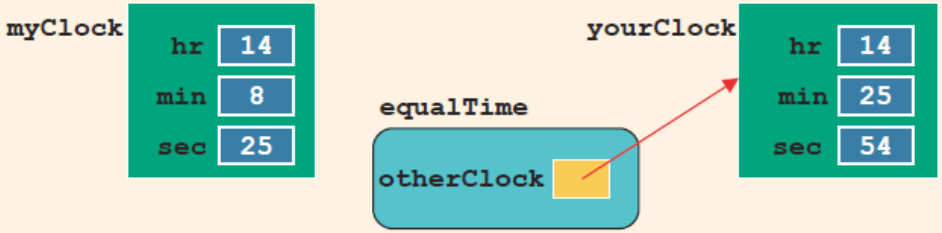

FIGURE 10-6 Object **myClock** and parameter **otherClock**

Implementation of Member Functions (4 of 4)

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

Accessor and Mutator Functions

- Accessor function: member function that only accesses the value(s) of member variable(s)
- Mutator function: member function that modifies the value(s) of member variable(s)
- Constant member function
  - Member function that cannot modify member variables of that class
  - Member function heading with `const` at the end

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

- C++ has no fixed order in which to 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

Constructors (1 of 2)

- Use constructors to guarantee that member variables of a class are initialized
- Two types of constructors
  - With parameters
  - Without parameters (default constructor)
- Other properties of constructors
  - Name of a constructor is the same as the name of the class
  - A constructor has no type

Constructors (2 of 2)

- 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

Invoking a Constructor

- A constructor is automatically executed when a class variable is declared
- Because a class may have more than one constructor, you can invoke a specific constructor

Invoking the Default Constructor

- Syntax to invoke the default constructor is:

```
className classObjectName;
```

- The statement:

  `clockType yourClock;`

  declares **`yourClock`** to be an object of type **`clockType`** and the default constructor executes

Invoking a Constructor with Parameters

- The syntax to invoke a constructor with a parameter is:

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

- Number and type of arguments 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 causes a compile-time error

Constructors and Default Parameters

- A constructor can have default parameters
  - Rules for declaring formal parameters are the same as for declaring default formal parameters in a function
  - Actual parameters are passed according to the same rules for functions
- A default constructor is a constructor with no parameters or with all default parameters

Destructors

- Destructors are functions without any type
- A class can have only one destructor
  - The destructor has no parameters
- The name of a destructor is the tilde character (~) followed by the class name
  - Example: `~clockType();`
- The destructor automatically executes 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 also applicable to data
- Abstract data type (ADT): a data type that separates the logical properties from the implementation details
- Three things associated with an ADT
  - Type name: the name of the ADT
  - Domain: the set of values belonging to the ADT
  - Set of operations on the data

CENGAGE Learning

Arrays versus `structs`

TABLE 9-1 Arrays vs. `structs`

Data Type	Array	`struct`
Arithmetic	No	No
Assignment	No	Yes
Input/output	No (except strings)	No
Comparison	No	No
Parameter passing	By reference only	By value or by reference
Function returning a value	No	Yes

A `struct` versus a `class` (1 of 2)

- 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`**
- **`class`**es and **`struct`**s have the same capabilities

A `struct` versus a `class` (2 of 2)

- 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`**