ENGINEERING PROGRAMMING I

Chapter 07: Class and OOP

2017

Topics

7.1 Abstract Data Types
7.2 Object-Oriented Programming
7.3 Introduction to Classes
7.4 Introduction to Objects
7.5 Defining Member Functions
7.6 Constructors
7.7 Destructors
7.8 Private Member Functions
7.9 Passing Objects to Functions
7.10 Object Composition
7.11 Separating Class Specification, Implementation, and Client Code
7.12 Input Validation Objects
7.13 Structures
7.15 Introduction to Object-Oriented Analysis and Design

Abstract Data Types

- ADT = Programmer-created data types
  - ADT specifies values and operations
    - Values: member variables
    - Operations: member functions
  - ADTs in C++: struct and class

- The user of an abstract data type (ADT) does not need to know any implementation details
  - *e.g.*, how the data is stored or how the functions are carried out
  - E.g.) string type in C++
    - You don't need to know how **`substr()`**, **`compare()`** are implemented.

Object-Oriented Programming

- Procedural programming uses variables to store data, focuses on the processes/ functions that occur in a program. Data and functions are separate and distinct.
- Object-oriented programming is based on objects that encapsulate the data and the functions that operate on it.

Object-Oriented Programming Terminology

- **object**: software entity that combines data and functions that act on the data in a single unit

- **attributes**: the data items of an object, stored in **member variables**

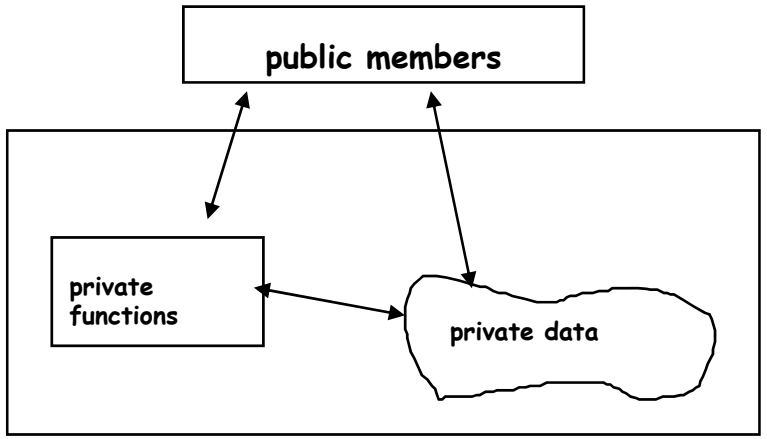
- **member functions (methods)**: procedures/ functions that act on the attributes of the class

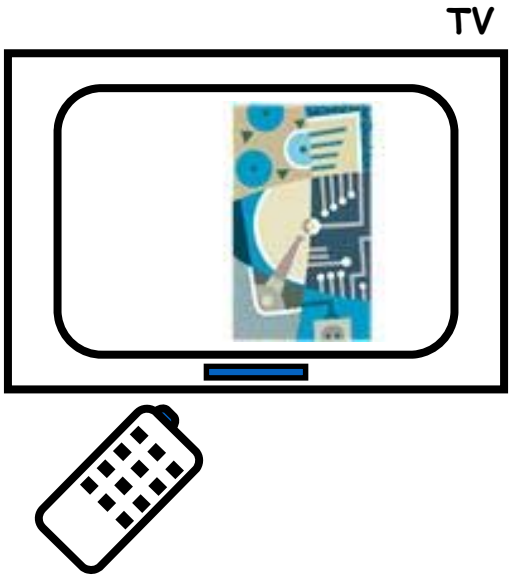
Object-Oriented Terminology

- A programming language is called OOP if it supports
  - **Encapsulation**
  - **Inheritance**
    - Ability to create a new class data type from an existing class.
    - I.e., we can crate a class "Dog" from an existing class "Animal"
    - Purpose of inheritance is reusability.
  - **Polymorphism**
    - Ability to take on many forms.
    - Harry is a dog, and it is also an animal.
  - EP1 will not cover inheritance and polymorphism.
    - If interested, please take Object Oriented Programming class (CSE241) in Computer Science major.

Encapsulation

- Encapsulation requires functions, modules and classes
  - Hide implementation details
  - Have clearly defined external interfaces

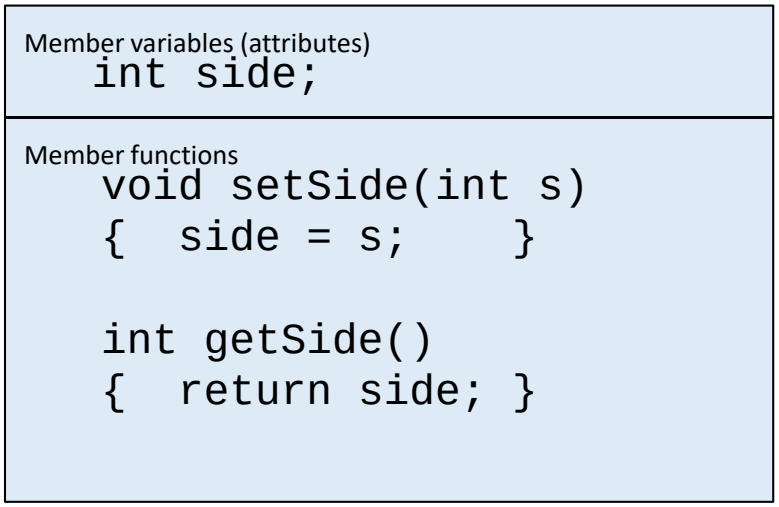




- Also referred to as "data hiding".
  - "data hiding" prevents inadvertent data modification

Object Example

Square



Square object's data item: **`side`**

Square object's functions: **`setSide`** - set the size of the side of the square,

`getSide` - return the size of the side of the square

Introduction to Classes

- **Class**: a programmer-defined datatype used to define objects
- It is a pattern for creating objects
- Class declaration format:

```
class className
{
   declaration;
   declaration;
};
```

Notice the required ;

Access Specifiers

- Used to control access to members of the class.
- Each member is declared to be either

  `public`: can be accessed by functions outside of the class

  or

  `private`: can only be called by or accessed by functions that are members of the class

Class Example


```
class Square
{
  private:
    int side;
  public:
    void setSide(int s)
    { side = s; }
    int getSide()
    { return side; }
};
```

More on Access Specifiers

- Can be listed in any order in a class
- Can appear multiple times in a class
- If not specified, the default is **`private`**

Introduction to Objects

- An object is an instance of a class
- Defined just like other variables

```
Square sq1, sq2;
```

- Can access members using dot operator

```
sq1.setSide(5);
cout << sq1.getSide();
```

Types of Member Functions

- **Acessor, get, getter function**: uses but does not modify a member variable

  ex: **`getSide`**

- **Mutator, set, setter function**: modifies a member variable

  ex: **`setSide`**

Defining Member Functions

- Member functions are part of a class declaration
- Can place entire function definition inside the class declaration

  or
- Can place just the prototype inside the class declaration and write the function definition after the class

Defining Member Functions Inside the Class Declaration

- Member functions defined inside the class declaration are called inline functions
- Only very short functions, like the one below, should be inline functions

```
int getSide()
{ return side; }
```

Inline Member Function Example

```
class Square
{
  private:
    int side;
  public:
    void setSide(int s)
    { side = s; }
    int getSide()
    { return side; }
};
```

Defining Member Functions After the Class Declaration

- Put a function prototype in the class declaration
- In the function definition, precede function name with class name and scope resolution operator (::)

```
int Square::getSide()
{
  return side;
}
```

Conventions and a Suggestion

- Conventions:
- Member variables are usually private
- Accessor and mutator functions are usually public
- Use 'get' in the name of accessor functions, 'set' in the name of mutator functions

- Suggestion: calculate values to be returned in accessor functions when possible, to minimize the potential for stale data

Tradeoffs of Inline vs. Regular Member Functions

- When a regular function is called, control passes to the called function
  - the compiler stores return address of call, allocates memory for local variables, etc.
- Code for an inline function is copied into the program in place of the call when the program is compiled
  - larger executable program, but
  - less function call overhead, possibly faster execution

Constructors

- A **constructor** is a member function that is used to initialize data members of a class
- Is called automatically when an object of the class is created
- Must be a **`public`** member function
- Must be named the same as the class
- Must have no return type

Constructor – 2 Examples

Inline:

```
class Square
{
   . . .
   public:
     Square(int s)
     { side = s; }
   . . .
};
```

Declaration outside the class:

```
Clas Square
{
  Square(int); //prototype
               //in class
}

Square::Square(int s)
{
   side = s;
}
```

Overloading Constructors

- A class can have more than 1 constructor
- Overloaded constructors in a class must have different parameter lists

```
Square();
Square(int);
```

The Default Constructor

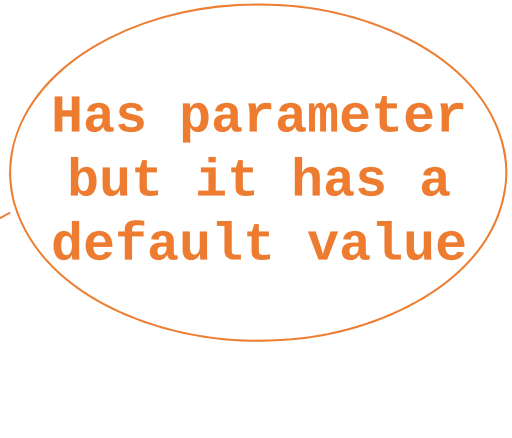
- Constructors can have any number of parameters, including none
- A **default constructor** is one that takes no arguments either due to
  - No parameters or
  - All parameters have default values
- If a class has any programmer-defined constructors, it must have a programmer- defined default constructor

Default Constructor Example

```
class Square
{
  private:
    int side;

  public:
    Square()          // default
    { side = 1; }     // constructor

    // Other member
    // functions go here
};
```

Another Default Constructor Example

```
class Square
{
  private:
    int side;

  public:
    Square(int s = 1) // default
    { side = s; }     // constructor

    // Other member
    // functions go here
};
```



Invoking a Constructor

- To create an object using the default constructor, use no argument list and no **()**

```
Square square1;
```

- To create an object using a constructor that has parameters, include an argument list

```
Square square1(8);
```

Destructors

- Public member function automatically called when an object is destroyed
- Destructor name is ~*className*, *e.g.*, **`~Square`**
- Has no return type
- Takes no arguments
- Only 1 destructor is allowed per class
  (*i.e.*, it cannot be overloaded)

Private Member Functions

- A **`private`** member function can only be called by another member function of the same class
- It is used for internal processing by the class, not for use outside of the class

Passing Objects to Functions

- A class object can be passed as an argument to a function
- When passed by value, function makes a local copy of object. Original object in calling environment is unaffected by actions in function
- When passed by reference, function can use 'set' functions to modify the object.

Notes on Passing Objects

- Using a value parameter for an object can slow down a program and waste space
- Using a reference parameter speeds up program, but allows the function to modify data in the structure
- To save space and time, while protecting structure data that should not be changed, use a **const** reference parameter

```
void showData(const Square &s)
                                  // header
```

Returning an Object from a Function

- A function can return an object

```
Square initSquare();   // prototype
s1 = initSquare();     // call
```

- Function must define a object
  - for internal use
  - to use with **return** statement

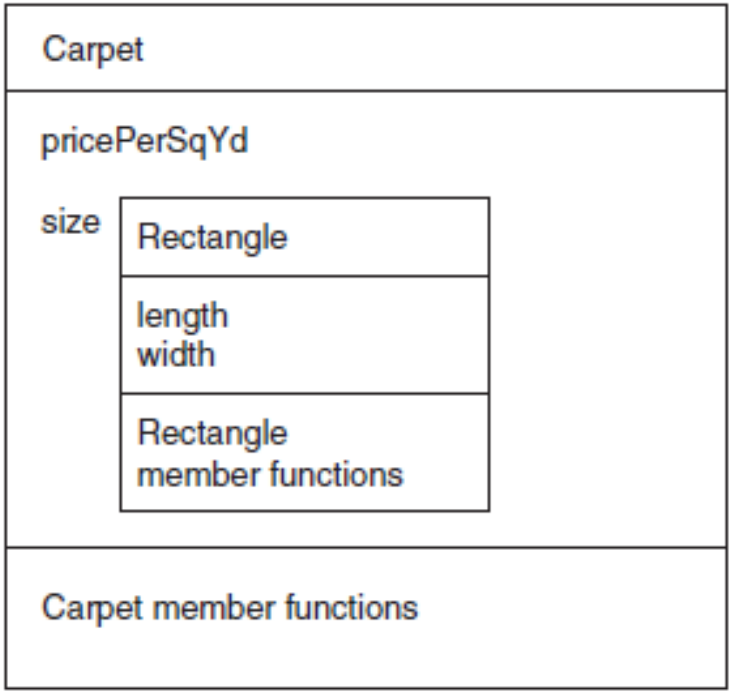
Returning an Object Example

```
Square initSquare()
{
  Square s;    // local variable
  int inputSize;
  cout << "Enter the length of side: ";
  cin >> inputSize;
  s.setSide(inputSize);
  return s;
}
```

Object Composition

- Occurs when an object is a member variable of another object.
- Often used to design complex objects whose members are simpler objects
- ex. (from book): Define a rectangle class. Then, define a carpet class and use a rectangle object as a member of a carpet object.

Object Composition, cont.



Separating Class Specification, Implementation, and Client Code

- Separating class declaration, member function definitions, and the program that uses the class into separate files is considered good design

Using Separate Files

- Place class declaration in a header file that serves as the class specification file. Name the file ***`classname.h`*** (for example, **`Square.h`**)

- Place member function definitions in a class implementation file. Name the file ***`classname.cpp`*** (for example, **`Square.cpp`**) This file should **`#include`** the class specification file.

- A client program (client code) that uses the class must **`#include`** the class specification file and be compiled and linked with the class implementation file.

Include Guards

- Used to prevent a header file from being included twice
- Format:

```
#ifndef symbol_name
#define symbol_name
. . .   (normal contents of header file)
#endif
```

- *symbol_name* is usually the name of the header file, in all capital letters:

```
#ifndef SQUARE_H
#define SQUARE_H
. . .
#endif
```

What Should Be Done Inside vs. Outside the Class

- Class should be designed to provide functions to store and retrieve data
- In general, I/O should be done by functions that use class objects, rather than by class member functions

Input Validation Objects

Classes can be designed to validate user input

- to ensure acceptable menu choice
- to ensure a value is in range of valid values
- etc.

Structures

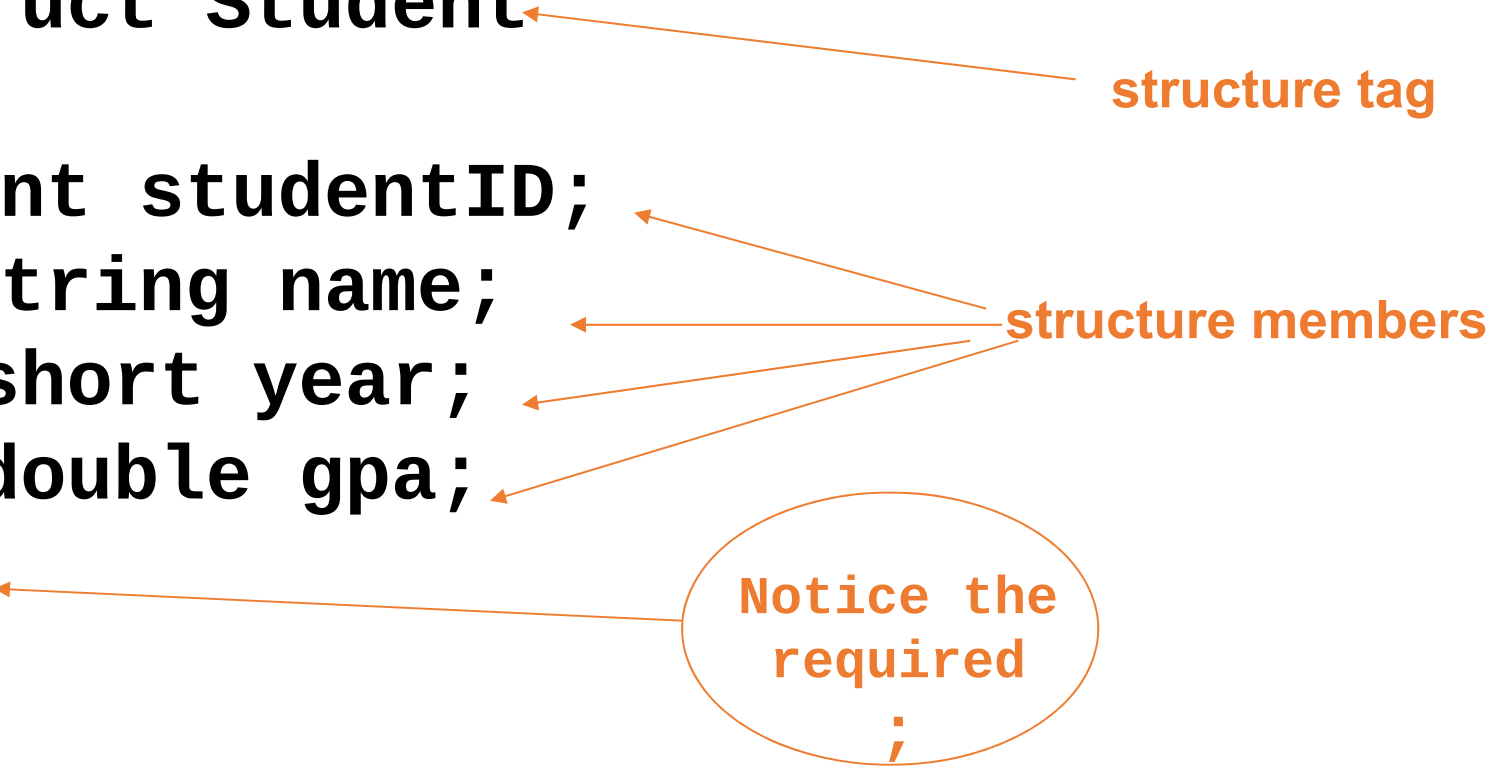
- Structure: C++ construct that allows multiple variables to be grouped together

- Structure Declaration Format:

```
struct structure name
{
 type1 field1;
 type2 field2;
   …
 typen fieldn;
};
```

Example `struct` Declaration

```
struct Student
{
  int studentID;
  string name;
   short year;
   double gpa;
};
```



`struct` Declaration Notes

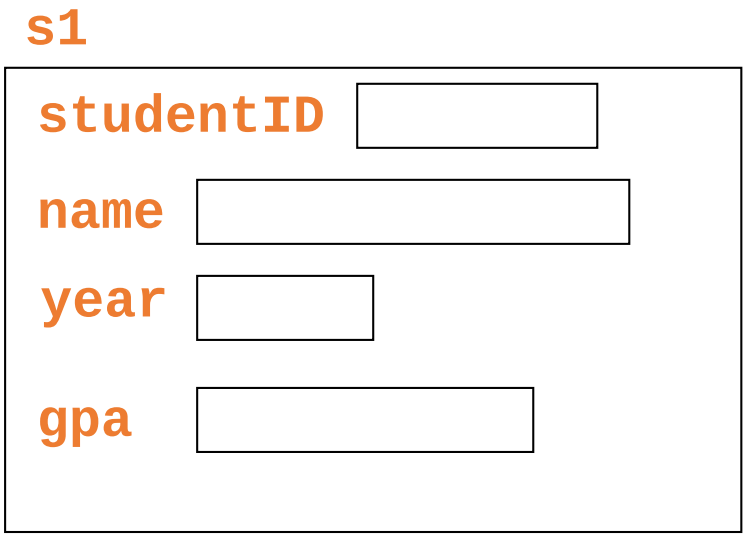
- `struct` names commonly begin with an uppercase letter
- The structure name is also called the tag
- Multiple fields of same type can be in a comma-separated list

```
string name,
       address;
```

Defining Structure Variables

- **`struct`** declaration does not allocate memory or create variables
- To define variables, use structure tag as type name

```
Student s1;
```



Accessing Structure Members

- Use the dot **(.)** operator to refer to members of **struct** variables

```
getline(cin, s1.name);
cin >> s1.studentID;
s1.gpa = 3.75;
```

- Member variables can be used in any manner appropriate for their data type

Displaying `struct` Members

To display the contents of a **`struct`** variable, you must display each field separately, using the dot operator

Wrong:

```
cout << s1; // won't work!
```

Correct:

```
cout << s1.studentID << endl;
cout << s1.name << endl;
cout << s1.year << endl;
cout << s1.gpa;
```

Comparing `struct` Members

- Similar to displaying a **`struct`**, you cannot compare two **`struct`** variables directly:

```
if (s1 >= s2) // won't work!
```

- Instead, compare member variables:

```
if (s1.gpa >= s2.gpa) // better
```

Initializing a Structure

Cannot initialize members in the structure declaration, because no memory has been allocated yet

```
struct Student      // Illegal
{                   // initialization
  int studentID = 1145;
  string name = "Alex";
  short year = 1;
  float gpa = 2.95;
};
```

Initializing a Structure (continued)

- Structure members are initialized at the time a structure variable is created
- Can initialize a structure variable's members with either
  - an initialization list
  - a constructor

Using an Initialization List

An initialization list is an ordered set of values, separated by commas and contained in { }, that provides initial values for a set of data members

```
{12, 6, 3}  // initialization list
            // with 3 values
```

More on Initialization Lists

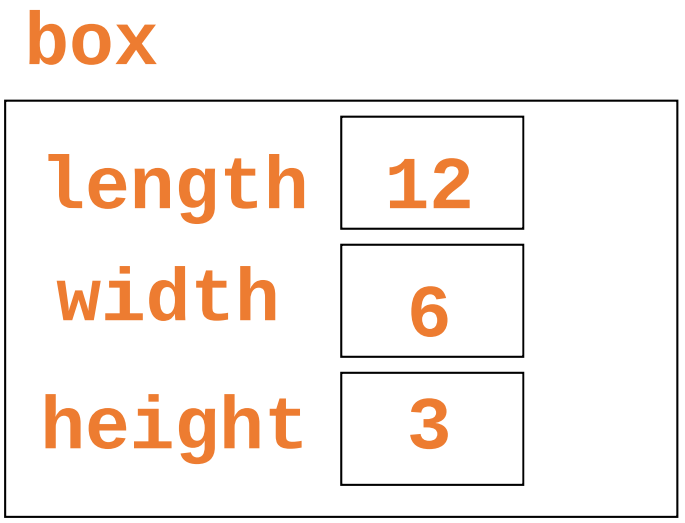
- Order of list elements matters: First value initializes first data member, second value initializes second data member, etc.
- Elements of an initialization list can be constants, variables, or expressions

```
{12, W, L/W + 1} // initialization list
                 // with 3 items
```

Initialization List Example

Structure Declaration

```
struct Dimensions
{
   int length,
       width,
       height;
};
Dimensions box = {12,6,3};
```



Partial Initialization

Can initialize just some members, but cannot skip over members

```
Dimensions box1 = {12,6};  //OK
Dimensions box2 = {12,,3}; //illegal
```

Problems with Initialization List

- Can't omit a value for a member without omitting values for all following members
- Does not work on most modern compilers if the structure contains any string objects
  - Will, however, work with C-string members

Using a Constructor to Initialize Structure Members

- Similar to a constructor for a class:
  - name is the same as the name of the struct
  - no return type
  - used to initialize data members
- It is normally written inside the **`struct`** declaration

A Structure with a Constructor

```
struct Dimensions
{
  int length,
      width,
      height;
  // Constructor
  Dimensions(int L, int W, int H)
  {length = L; width = W; height = H;}
};
```

Passing Arguments to a Constructor

- Create a structure variable and follow its name with an argument list
- Example:

```
Dimensions box3(12, 6, 3);
```

Nested Structures

A structure can have another structure as a member.

```
struct PersonInfo
{  string name,
           address,
           city;
};
struct Student
{  int          studentID;
   PersonInfo   pData;
   short        year;
   double       gpa;
};
```

Members of Nested Structures

Use the dot operator multiple times to access fields of nested structures

```
Student s5;
s5.pData.name = "Joanne";
s5.pData.city = "Tulsa";
```

Structures as Function Arguments

- May pass members of **`struct`** variables to functions

```
computeGPA(s1.gpa);
```

- May pass entire **`struct`** variables to functions

```
showData(s5);
```

- Can use reference parameter if function needs to modify contents of structure variable

Notes on Passing Structures

- Using a value parameter for structure can slow down a program and waste space
- Using a reference parameter speeds up program, but allows the function to modify data in the structure
- To save space and time, while protecting structure data that should not be changed, use a **const** reference parameter

```
void showData(const Student &s)
                                  // header
```

Returning a Structure from a Function

- Function can return a **`struct`**

```
Student getStuData();  // prototype
s1 = getStuData();     // call
```

- Function must define a local structure variable
  - for internal use
  - to use with **`return`** statement

Returning a Structure Example

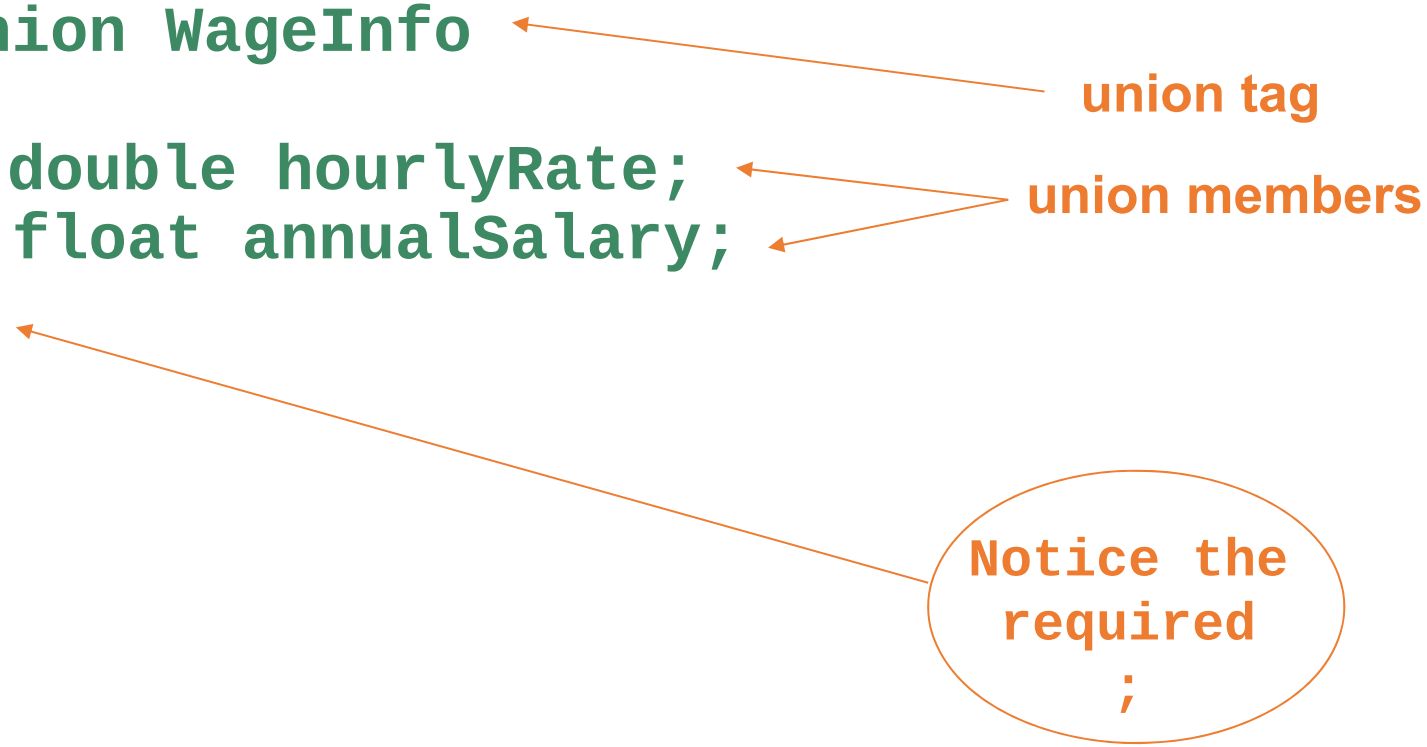
```
Student getStuData()
{ Student s;    // local variable
  cin >> s.studentID;
  cin.ignore();
  getline(cin, s.pData.name);
  getline(cin, s.pData.address);
  getline(cin, s.pData.city);
  cin >> s.year;
  cin >> s.gpa;
  return s;
}
```

Unions

- Similar to a **`struct`**, but
  - all members share a single memory location, which saves space
  - only 1 member of the union can be used at a time
- Declared using key word **`union`**
- Otherwise the same as **`struct`**
- Variables defined and accessed like **`struct`** variables

Example **union** Declaration

```
union WageInfo
{
  double hourlyRate;
  float annualSalary;
};
```



Introduction to Object-Oriented Analysis and Design

- Object-Oriented Analysis: that phase of program development when the program functionality is determined from the requirements
- It includes
  - identification of objects and classes
  - definition of each class's attributes
  - identification of each class's behaviors
  - definition of the relationship between classes

Identify Objects and Classes

- Consider the major data elements and the operations on these elements
- Candidates include
  - user-interface components (menus, text boxes, *etc.*)
  - I/O devices
  - physical objects
  - historical data (employee records, transaction logs, *etc.*)
  - the roles of human participants

Finding the Classes

Technique:

- Write a description of the problem domain (objects, events, etc. related to the problem)
- List the nouns, noun phrases, and pronouns. These are all candidate objects
- Refine the list to include only those objects that are relevant to the problem

Define Class Attributes and Behaviors

- For each class,
  - determine the data elements needed by an object of that class
  - determine the behaviors or activities that the class must perform

Determine Class Responsibilities

Class responsibilities:

- What is the class responsible to know?
- What is the class responsible to do?

Use these to define the member functions

Relationships Between Classes

Possible relationships

- Access ("uses-a")
- Ownership/Composition ("has-a")
- Inheritance ("is-a")

Object Reuse

- A well-defined class can be used to create objects in multiple programs
- By re-using an object definition, program development time is shortened
- One goal of object-oriented programming is to support object reuse