CSC2212

C++ Programming

Class and Object

Ch 7: Gaddis, Starting Out with C++ Early Objects 8th ed

Lab Six

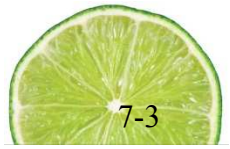
Abstract Data Types

- Programmer-created data types that specify
  - legal values that can be stored
  - operations that can be done on the values

- 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 operations on it are carried out)

Abstraction in Software Development

- Abstraction allows a programmer to design a solution to a problem and to use data items without concern for how the data items are implemented
- This has already been encountered in the book:
  - To use the **`pow`** function, you need to know what inputs it expects and what kind of results it produces
  - You do not need to know how it works

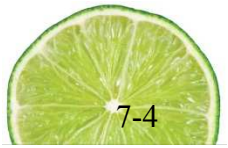


Abstraction and Data Types

- **Abstraction:** a definition that captures general characteristics without details

  Example: *An abstract triangle is a 3-sided polygon. A specific triangle may be scalene, isosceles, or equilateral*

- **Data Type:** defines the kind of values that can be stored and the operations that can be performed on it



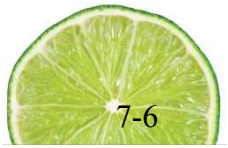
Object-Oriented Programming

- **Procedural Programming** uses variables to store data, and 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 (cont.)

- **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 Example

Square

```
Member variables (attributes)
        int side;
Member functions
    void setSide(int s)
    {  side = s;    }

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

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 data type used to define objects

- It is a pattern for creating objects

  Example:
  ```
  string fName, lName;
  ```
  creates two objects of the **`string`** class

Introduction to Classes

- Class declaration format:

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

Notice the required ;

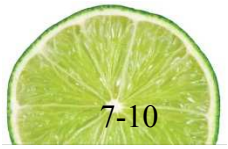
Access Specifiers

- 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

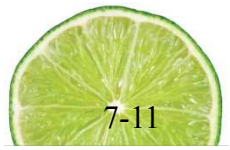
  or

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



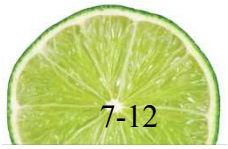
Class Example


```
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`**



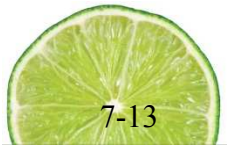
Creating and Using Objects

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

```
Square sq1, sq2;
```

- It can access members using dot operator

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

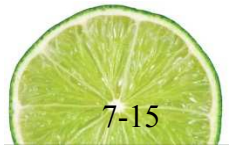
Defining Member Functions

- Member functions are part of a class declaration
- Entire function definition can be place inside class declaration
- Or using 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; }
};
```

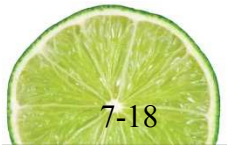
Defining Member Functions After the Class Declaration

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

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


Constructors

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



Constructor (examples)

Inline:

```
class Square{
        . . .

     public:

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

Declaration outside the class:

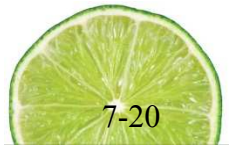
```
Square(int);//prototype
            //inside class

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


Overloading Constructors

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

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



The Default Constructor

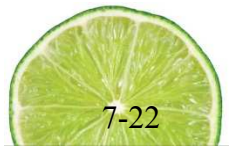
- Constructors can have any number of parameters, including none
- A **default constructor** is one that takes no arguments either due to
  - No parameters or
  - All parameters have default values
- If a class has any programmer-defined constructors, it must have a programmer-defined default constructor

Default Constructor Example

```
class Square
{
  private:
    int side;

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

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



Another Default Constructor (example 2)

```
class Square
{
  private:
    int side;

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

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

Has parameter but it has a default value

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

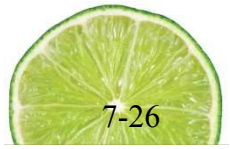
- Is a public member function automatically called when an object is destroyed
- The destructor name is ~*className*, *e.g.*,

  `~Square`
- It has no return type
- It takes no arguments
- Only 1 destructor is allowed per class

  (*i.e.*, it cannot be overloaded)

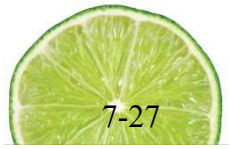
Private Member Functions

- 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



NOTE

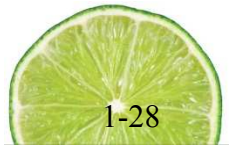
Separating class declaration, member function definitions, and the program that uses the class into separate files is considered good design



EXERCISE

Write a simple class to represent a circle, with three functions: one to set radius, another one to compute area, another one for computing circumference.

Modify it with adding default constructor that can set the radius and also destructor.



Inside vs. Outside the Class

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

Structures

- Structure: Programmer-defined data type 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;
};
```

structure name

structure members

Notice the required ;

`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

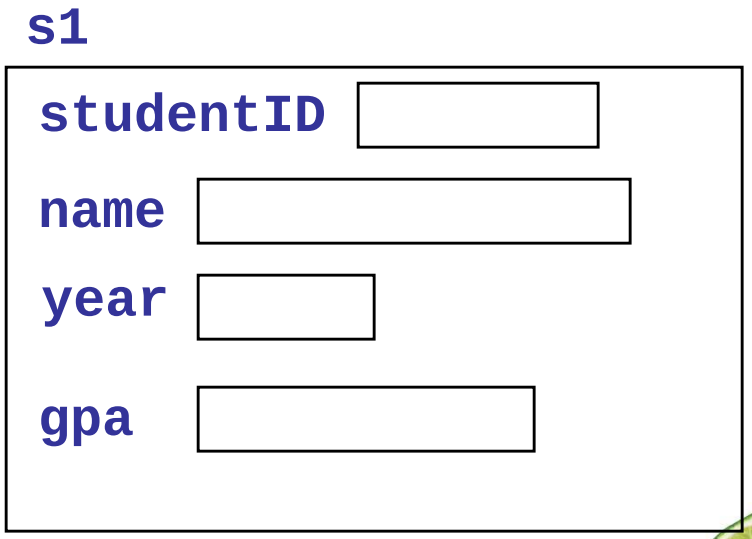
```
string name,
       address;
```

- Fields in a structure are all public by default

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
```

NOTE

It is not allowed to 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

- 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

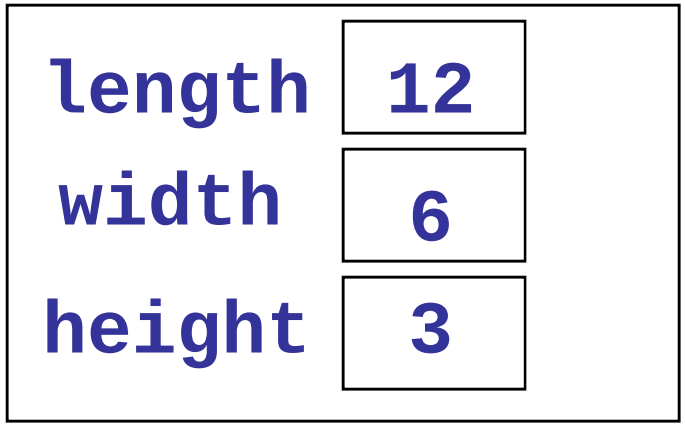
Initialization List Example

Structure Declaration

```
struct Dimensions
{ int length,
      width,
      height;
};
```

Structure Variable

box




```
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;
       }
};
```

EXERCISE

*W*rite a simple program that use structure to store school information.

*M*odify it with adding constructor that can initialise all its variable.