ITM 312, Fall 2013

Chapter 13 Lecture Notes

13.1. Procedural and Object-Oriented Programming

a. Procedural Programming
   i. Based on the processes needed to achieve a certain outcome
   ii. A procedure of actions in a program
   iii. Limitations:
      1. If the data structures change, many functions must also be changed
      2. Programs that are based on complex function hierarchies are:
         a. difficult to understand and maintain
         b. difficult to modify and extend
         c. easy to break

b. Object-Oriented Programming (OOP)
   i. Based on data and functions which operate on data
   ii. Objects are instances of an Abstract Data Type (ADT), which represents the data and its functions
   iii. Terminology:
      1. class: like a `struct` (allows bundling of related variables), but variables and functions in the class can have different properties than in a `struct`
      2. object: an instance of a class, in the same way that a variable can be an instance of a `struct`
      3. attributes: members of a class
      4. methods or behaviors: member functions of a class
      5. data hiding: restricting access to certain members of an object
      6. public interface: members of an object that are available outside of the object. This allows the object to provide access to some data and functions without sharing its internal details and design, and provides some protection from data corruption

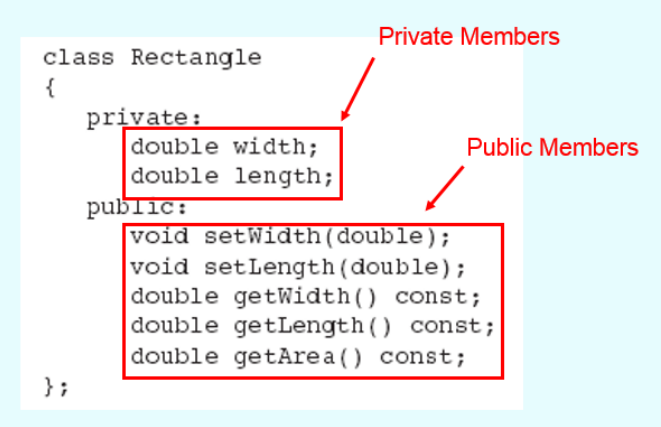
13.2. Introduction to Classes

a. A Class is like a blueprint and objects are like houses built from the blueprint; Objects are created from a class

b. Syntax:

```
class ClassName {
    declaration;
    declaration;
};
```

c. Example:



d. Access Specifiers - Used to control access to members of the class

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

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

iii. Can be listed in any order in a class

iv. Can appear multiple times in a class

v. If not specified, the default is `private`

e. `const` appearing after the parentheses in a member function declaration specifies that the function will not change any data in the calling object.

i. Example:

```
double getWidth() const;
double getLength() const;
double getArea() const;
```

f. Member Functions

i. When defining a member function:

1. Put prototype in class declaration
2. Define function using class name and scope resolution operator (`::`)

3. Example:

```
int Rectangle::setWidth(double w) {
    width = w;
}
```

ii. Mutator: a member function that stores a value in a private member variable, or changes its value in some way

iii. Accessor: function that retrieves a value from a private member variable. Accessors do not change an object's data, so they should be marked `const`.

13.3. Defining an Instance of a Class

a. An object is an instance of a class

b. Defined like structure variables:

```
Rectangle r;
```

c. Access members using the dot operator (`obj.member()`):

```
r.setWidth(5.2);
cout << r.getWidth();
```

d. Compiler error if attempt to access private member using dot operator

e. Avoiding Stale Data

i. Some data is the result of a calculation.

ii. In the `Rectangle` class the area of a rectangle is calculated: length * width

iii. If we were to use an `area` variable here in the `Rectangle` class, its value would be dependent on the length and the width.

iv. If we change `length` or `width` without updating `area`, then `area` would become *stale*.

v. To avoid stale data, it is best to calculate the value of that data within a member function rather than store it in a variable.

f. Pointers to Objects

i. Can define a pointer to an object:

```
Rectangle *rPtr;
```

ii. Can access public members via pointer:

```
rPtr = &otherRectangle;
rPtr->setLength(12.5);
cout << rPtr->getLenght() << endl;
```

iii. We can also use a pointer to dynamically allocate an object.

```
1  // Define a Rectangle pointer.
2  Rectangle *rectPtr;
3
4  // Dynamically allocate a Rectangle object.
5  rectPtr = new Rectangle;
6
7  // Store values in the object's width and length.
8  rectPtr->setWidth(10.0);
9  rectPtr->setLength(15.0);
10
11 // Delete the object from memory.
12 delete rectPtr;
13 rectPtr = 0;
```

13.4. Why Have Private Members?

a. Making data members `private` provides data protection
   i. If it was public, other classes could change the variable to an unknown and possibly erroneous value
b. Data can be accessed only through `public` functions
c. Public functions define the class's public interface

13.5. Separating Specification from Implementation

a. Place class declaration in a header file that serves as the class specification file. Name the file `ClassName.h`, for example, `Rectangle.h`
b. Place member function definitions in `ClassName.cpp`, for example, `Rectangle.cpp` – this file should `#include` the class specification file
c. Programs that use the class must `#include` the class specification file, and be compiled and linked with the member function definitions

13.6. Inline Member Functions

a. Member functions can be defined
   i. inline: in class declaration
   ii. or after the class declaration (e.g. in .cpp file)
b. Inline appropriate for short function bodies [where no data is modified]: `int getWidth() const { return width; }`
c. Tradeoffs:
   i. Regular functions – when called, compiler stores return address of call, allocates memory for local variables, etc.
   ii. Code for an inline function is copied into program in place of call – larger executable program, but no function call overhead, hence faster execution

13.7. Constructors

a. Member function that is automatically called when an object is created
b. Purpose is to construct an object
c. Constructor function name is class name
d. Has no return type, not even void
e. Default Constructors
    i. A default constructor is a constructor that takes no arguments.
        1. However, if all of a constructor's parameters have default arguments, then it is a default constructor. For example:
        `Rectangle(double = 0, double = 0);`
    ii. If you write a class with no constructor at all, C++ will write a default constructor for you, one that does nothing.
    iii. When all of a class's constructors require arguments, then the class has NO default constructor.
        1. When this is the case, you must pass the required arguments to the constructor when creating an object.
    iv. A simple instantiation of a class (with no arguments) calls the default constructor:
    `Rectangle r;`

13.8. Passing Arguments to Constructors

a. To create a constructor that takes arguments:
    i. indicate parameters in prototype:
    `Rectangle(double, double);`
    ii. Use parameters in the definition:
```
Rectangle::Rectangle(double w, double len) {
    width = w;
    length = len;
}
```
b. You can pass arguments to the constructor when you create an object:
`Rectangle r(10, 5);`

13.9. Destructors

a. Member function automatically called when an object is destroyed
b. Destructor name is `~classname`, *e.g.*, `~Rectangle`
c. Has no return type; takes no arguments
d. Only one destructor per class, *i.e.*, it cannot be overloaded

e. If constructor allocates dynamic memory, destructor should release it
    i. When an object is dynamically allocated with the new operator, its constructor executes:
    ```
    Rectangle *r = new Rectangle(10, 20);
    ```
    ii. When the object is destroyed, its destructor should execute:
    ```
    delete r;
    ```

13.10. Overloading Constructors

a. A class can have more than one constructor
b. Overloaded constructors in a class must have different parameter lists:
```
Rectangle();
Rectangle(double);
Rectangle(double, double);
```
c. Only one default constructor and one destructor
    i. Do not provide more than one default constructor for a class: one that takes no arguments and one that has default arguments for all parameters
    ```
    Square();
    Square(int = 0);  // will not compile
    ```
    ii. Since a destructor takes no arguments, there can only be one destructor for a class
d. Non-constructor member functions can also be overloaded:
```
void setCost(double);
void setCost(char *);
```
    i. Must have unique parameter lists as for constructors

13.11. Using Private Member Functions

a. A private member function can only be called by another member function
b. It is used for internal processing by the class, not for use outside of the class
c. See the `createDescription` function in `ContactInfo.h` (Version 2)

13.12. Arrays of Objects

a. Objects can be the elements of an array:
```
InventoryItem inventory[40];
```
b. Default constructor for object is used when array is defined
c. Must use initializer list to invoke constructor that takes arguments:
```
InventoryItem inventory[3] = { "Hammer", "Wrench", "Pliers" };
```

d. If the constructor requires more than one argument, the initializer must take the form of a function call:

```
InventoryItem inventory[3] = { InventoryItem("Hammer", 6.95, 12),
                               InventoryItem("Wrench", 8.75, 20),
                               InventoryItem("Pliers", 3.75, 10) };
```

e. It isn't necessary to call the same constructor for each object in an array:

```
InventoryItem inventory[3] = { "Hammer",
                               InventoryItem("Wrench", 8.75, 20),
                               "Pliers" };
```

   i. `inventory[0]` and `inventory[2]` are calling the constructor with one argument, while `inventory[1]` is using the constructor with 3 arguments

f. Objects in an array are referenced using subscripts

g. Member functions are referenced using dot notation:

```
inventory[2].setUnits(30);
cout << inventory[2].getUnits();
```