Programming techniques

Week 2

Topic 2: Data abstraction in C++

01/2014

What is in today?

- Terminology
- Data Hiding
- Class Constructors
- Defining and using functions in classes
- Where to place the class interface and implementation of the member functions

"`class`" Terminology

- Class
  - think data type
- Object
  - instance of a class, e.g., variable
- Members
  - like structures, the data and functions declared in a class
  - called "data members" and "member functions"

"`class`" Terminology

- A class could be a list, a string, a counter, a clock, a bank account, etc.
  - discuss a simple counter class on the board
- An object is as real as a variable, and gets allocated and deallocated just like variables
  - discuss the similarities of:

```
int i;                    list j;
```

"class" Terminology

- For the list of videos data type we used last time....

```
class list {        <--- the data type!!!
   public:
      list();       <--- the constructor
      int add (const video &);   <---- 3 member functions
      int remove (char title[]);
      int display_all();
   private:
      video my_list[CONST_SIZE];  <---- data members
       int num_of_videos;
};<--- notice like structures we need a semicolon
```

"`class`" Terminology

- If we examine the previous class,
  - notice that classes are really very similar to structures
  - a class is simply a generalized structure
  - in fact, even though we may not have used structures in this way...

    Structures and Classes are 100% identical except for their default conditions...
    - by default, all members in a structure are available for use by clients (e.g., main programs); they are public

"`class`" Terminology

- We have seen the use of structures in a more simple context,
  - as we examined with the **video** struct.
- It had three members (data members)
  - called title, category, and quantity.
- They are "public" by default,
  - so all functions that have objects of type video can directly access members by:

    video object;

    object.title object.category object.quantity

"`class`" Terminology

- This limited use of a structure was appropriate, because
  - it served the purpose of grouping different types of data together as a single unit
  - so, anytime we want to access a particular video -- we get all of the information pertaining to the video all at once
  - in fact, in your programming -- think about passing in structures instead of a million different arguments! **Think Grouping**

Structure Example

- Remember, anything you can do in a struct you can do in a class.
  - It is up to your personal style how many structures versus classes you use to solve a problem.
- Benefit: Using structures for simple “groupings” is compatible with C

```
struct video {
   char title[100];
   char category[5];
   int quantity;
};
```

"`class`" Terminology

- To accomplish data hiding and encapsulation
  - we usually turn towards classes
- What is data hiding?
  - It is the ability to protect data from unauthorized use
  - Notice, with the video structure, any code that has an object of the structure can access or modify the title or other members

Data Hiding

- With data hiding
  - accessing the data is restricted to authorized functions
  - “clients” (e.g., main program) can’t muck with the data directly
  - this is done by placing the **data members** in the private section
  - and, placing **member functions** to access & modify that data in the public section

Data Hiding

- So, the public section
  - includes the data and operations that are visible, accessible, and useable by all of the clients that have objects of this class
  - this means that the information in the public section is “transparent”; therefore, all of the data and operations are accessible outside the scope of this class
  - by default, nothing in a class is public!

Data Hiding

- The private section
  - includes the data and operations that are not visible to any other class or client
  - this means that the information in the private section is “opaque” and therefore is inaccessible outside the scope of this class
  - the client has no direct access to the data and must use the public member functions
  - this is where you should place all data to ensure the memory’s integrity

Data Hiding

- The good news is that
  - member functions defined in the public section can use, return, or modify the contents of any of the data members, directly
  - it is best to assume that member functions are the only way to work with private data
    - (there are “friends” but don’t use them this term)
  - Think of the member functions and private data as working together as a team

"class" Terminology

- Let's see how "display_all" can access the data members:

```
class list {
   public:                ←——  notice it is public
       int display_all() {
           for (int i=0; i<num_of_videos; ++i)
              cout <<my_list[i].title <<'\t'
                  <<my_list[i].category
                  <<'\t' <<my_list[i].quantity <<endl;
        }
    •••
  private:
      video my_list[CONST_SIZE];
      int num_of_videos;
};
```

Data Hiding

- Notice, that the display_all function can access the private my_list and num_of_videos members, directly
  - without an object in front of them!!!
  - this is because the client calls the display_all function through an object
    - object.display_all();
  - so the object is implicitly available once we enter “class scope”

Where to place….

- In reality, the previous example was misleading. We don’t place the implementation of functions with this this class interface
- Instead, we place them in the class implementation, and separate this into its own file

Class Interface (.h)

- Class Interface: list.h

```
class list {
   public:
      int display_all()
          ...
   private:
      video my_list[CONST_SIZE];
       int num_of_videos;
};
```

- list.h can contain:
  - prototype statements
  - structure declarations and definitions
  - class interfaces and class declarations
  - include other files

Class Implementation

- Class Implementationlist.cpp

```
#include "list.h"  <------ notice the double quotes
        int list::display_all() {
            for (int i=0; i<num_of_videos; ++i)
               cout <<my_list[i].title <<'\t'
                   <<my_list[i].category
                   <<'\t' <<my_list[i].quantity <<endl;
         }
```

  - Notice, the code is the same
  - But, the function is prefaced with the class name and the scope resolution operator!
  - This places the function in class scope even though it is implemented in another file
  - Including the list.h file is a "must"

Class Implementation

- Note:
  - the header file must be included in both the class implementation (list.cpp) and the client program (e.g., main.cpp)
- From now on, you will need to separate your code into these “modules”.....

Constructors

- Remember that when you define a local variable in C++, the memory is not automatically initialized for you
- This could be a problem with classes and objects
- If we define an object of our list class, we really need the “num_of_videos” data member to have the value *zero*
- *Uninitialized just wouldn’t work!*

Constructors

- Luckily, with a constructor we can write a function to initialize our data members
  - and have it implicitly be invoked whenever a client creates an object of the class
- The constructor is a strange function, as it has the same name as the class, and no return type (at all...not even void).

Constructor

- The list constructor was: (list.h)

```
class list {
  public:
      list();     <--- the constructor
      ...
};
```

- The implementation is: (list.cpp)

```
list::list(){
   num_of_videos = 0;
}
```

Constructor

- The constructor is implicitly invoked when an object of the class is formed:

```
int main() {

    list fun_videos;  implicitly calls the
                      constructor

  list all_videos[10];  implicitly calls the
                        constructor 10 times for
                        each of the 10 objects!!
```