Introduction to C++

Data Abstraction w/ Classes

Topic #3

Topic #3

- **Abstract Data Types**
  - Introduction to...Object Models
  - Introduction to...Data Abstraction
  - Using Data Abstraction in C++ ...an introduction to the class
- **Members of a Class**
  - The class interface, using the class, the class interface versus implementation
  - Classes versus Structures
  - Constructors, Destructors
  - Dynamic Memory and Linked Lists

Programming Paradigms

- The most important aspect of C++ is its ability to support many different programming paradigms
  - procedural abstraction
  - modular abstraction
  - data abstraction
  - object oriented programming (this is discussed later, in CS202)

Procedural Abstraction

- This is where you build a “fence” around program segments, preventing some parts of the program from “seeing” how tasks are being accomplished.
- Any use of globals causes <u>side effects</u> that may not be predictable, reducing the viability of procedural abstraction

Modular Abstraction

- With modular abstraction, we build a “screen” surrounding the internal structure of our program prohibiting programmers from accessing the data except through specified functions.
- Many times data structures (e.g., structures) common to a module are placed in a header files along with prototypes (allows external references)

Modular Abstraction

- The corresponding functions that manipulate the data are then placed in an implementation file.
- Modules (files) can be compiled separately, allowing users access only to the object (.o) files
- We progress one small step toward OOP by thinking about the actions that need to take place on data...

Modular Abstraction

- We implement modular abstraction by separating out various functions/structures/classes into multiple .cpp and .h files.
- .cpp files contain the implementation of our functions
- .h files contain the prototypes, class and structure definitions.

Modular Abstraction

- We then include the .h files in modules that need access to the prototypes, structures, or class declarations:
  - #include "myfile.h"
  - (Notice the double quotes!)
- We then compile programs (on UNIX) by:
  - g++ main.cpp myfile.cpp
  - (Notice no .h file is listed on the above line)

Data Abstraction

- Data Abstraction is one of the most powerful programming paradigms
- It allows us to create our own user defined data types (using the class construct) and
  - then define variables (i.e., objects) of those new data types.

Data Abstraction

- With data abstraction we think about <u>what</u> operations can be performed on a particular type of data and not <u>how</u> it does it
- Here we are one step closer to object oriented programming

Data Abstraction

- Data abstraction is used as a tool to increase the modularity of a program
- It is used to build walls between a program and its data structures
  - what is a data structure?
  - talk about some examples of data structures
- We use it to build new abstract data types

Data Abstraction

- An abstract data type (ADT) is a data type that we create
  – consists of data and operations that can be performed on that data
- Think about a `char` type
  – it consists of 1 byte of memory and operations such as assignment, input, output, arithmetic operations can be performed on the data

Data Abstraction

- An abstract data type is any type you want to add to the language over and above the fundamental types
- For example, you might want to add a new type called: list
  - which maintains a list of data
  - the data structure might be an array of structures
  - operations might be to add to, remove, display all, display some items in the list

Data Abstraction

- Once defined, we can create lists without worrying about how the data is stored
- We “hide” the data structure used for the data within the data type -- so it is transparent to the program using the data type
- We call the program using this new data type: the client program (or client)

Data Abstraction

- Once we have defined what data and operations make sense for a new data type, we can define them using the class construct in C++
- Once you have defined a class, you can create as many instances of that class as you want
- Each “instance” of the class is considered to be an “object” (variable)

Data Abstraction

- Think of a class as similar to a data type
  - and an object as a variable
- And, just as we can have zero or more variables of any data type...
  - we can have zero or more objects of a class!
- Then, we can perform operations on an object in the same way that we can access members of a struct...

What is a Class?

- Remember, we used a structure to group different types of data together under a common name
- With a class, we can go the next step an actually define a new data type
- In reality, structures and classes are 100% the same except for the default conditions
  - everything you can do with a class you can do with a structure!

What is a Class?

- First, let’s talk about some terminology
  - Think of a <u>class</u> as the same as a <u>data type</u>
  - Think of an <u>object</u> as the same as a <u>variable</u>
- An “object” is an <u>instance of a class</u>
  - Just like a “variable” is an instance of a specific data type
- We can zero or more variables (or objects) in our programs

When do we used Classes?

- I recommend using structures when you want to group different types of data together
  - and, to use a class when we are interested in building a new type of data into the language itself
  - to do this, I always recommend forming that data type such that it behaves in a consistently to how the fundamental data types work

But, What is a Data Type?

- We’ve been working with fundamental data types this term, such as ints, floats, chars...
- Whenever we define variables of these types,
  - memory is allocated to hold the data
  - a set of operations can now be performed on that data
  - different data types have different sets of operations that make sense (the mod operator doesn’t make sense for floats...)

Defining new Data Types...

- Therefore, when we define a new data type with the class construct
  - we need to specify how much memory should be set aside for each variable (or object) of this type
  - and, we need to specify which operations make sense for this type of data (and then implement them!!)
  - and, what <u>operators</u> makes sense (do be discussed with **operator overloading**)

Defining a Class...

- Once we have decided on how the new type of data should behave, we are ready to define a class:

```
class data_type_name {
  public:
        //operations go here
  private:
        //memory is reserved here
};
```

For Example, here is a Class Interface

```
class string {
  public:
          string();
          int copy(char []);
          int length();
          int display();
   private:
          char str[20];
          int len;
};
```

Then, the Class Implementation

```
string::string() {
   str[0]='\0';          len = 0;
}
int string::copy(char s []) [
  if (strlen(s) < 20)
        strcpy (str, s);
  else {
        for (int i = 0; i< 20; ++i)
             str[i] = s[i];
        str[20]='\0';
  len = strlen(str);   return len;     }
```

More of the Class Implementation

```
int string::length() {
   return len;
}
int string::display() {
   cout <<str;
   return len;
}
```

Defining Objects of this Class

- Notice how similar defining objects of class is to defining variables of any data type:

```
string  my_str;    vs.          int i;
```

- Defining an object causes the “constructor” to be invoked; a constructor is the same named function as the class (string) and is used to initialize the memory set aside for this object
- Think about how much memory is set aside?
- What initial values should it take on?

Using Objects of this Class

- Think about how you can use those objects

```
my_str.copy("hi!");
cout << my_str.length();
```

- We are limited to using only those operations that are defined within the public section of the class interface
- The only "built-in" operation that can be used with objects of a class is the assignment operation, which does a memberwise copy (as we learned with structures)

Using Objects of this Class

- Notice how similar the use of these operations is to the cin.get function.....

```
cin.get(ch);
```

- This should be a clue. cin therefore is an object of the istream class.
- The dot is the member access operator; it allows us to access a particular public member function defined within the istream class.
- The function get is therefore defined within the public section of the istream class

Limitations...

- But, there are limitations!
- If our goal is to really be able to use my string objects in a way consistent with the fundamental data types,
  - then I would expect to be able to read strings using the extraction operator
  - and to display strings by directly using the insertion operator
  - and to concatenate strings using +

Limitations...

- With the class as it is defined, none of these things can be done...
  - the only operations that can be performed are those specified within the public section of the class interface, and a memberwise copy with the assignment operator
  - No other operations are known
- Therefore, to be consistent, we must revise our class to use <u>operator overloading</u>

For Example, here is a Class Interface

```
class string {
  public:
        string();
        int length();
  friend ofstream & operator <<
        (ofstream &, const string &);
  friend ifstream & operator >>
                (ifstream &, string &);
   private:
        char str[20];
        int len;
};
```

List Example

- For a list of videos, we might start with a struct defining what a video is:

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

We will re-visit this example using dynamic memory once we understand the mechanics of classes

List Example

- For a list of videos data type:

```
class list {
   public:
      list();
      int add (const video &);
      int remove (char title[]);
      int display_all();
   private:
      video my_list[CONST_SIZE];  //for now...
      int num_of_videos;
};
```

List Example

- For a client to create a list object:

```
int main() {
  list home_videos;     //has an array of 100
  videos
  list kids_shows;      //another 100 videos
  here...

  •••

  video out_of_site;
  cin.get(out_of_site.title,100,'\n');
  cin.ignore(100,'\n');
  •••
  home_videos.add(out_of_site);      //use
  operation
```

Introduction to C++

Data Hiding

and

Member Functions

Data Abstraction in C++

- 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 <u>and</u> 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

```
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

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 <u>data hiding</u> and <u>encapsulation</u>
  - 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 <u>through</u> an object
    - object.display_all();
  - so the object is <u>implicitly</u> 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 Implementation list.c

```
#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"

Constructors

- Remember that when you define a local variable in C++, the memory is <u>not</u> 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!!
```

Dynamic Memory w/ Classes

- But, what if we didn't want to waste memory for the title (100 characters may be way too big (Big, with Tom Hanks)
- So, let's change our video structure to include a dynamically allocated array:

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

Dynamic Memory w/ Classes

- Let’s write a class that now allocates this list of videos dynamically, at run time
- This way, we can wait until we run our program to find out how much memory should be allocated for our video array

Dynamic Memory w/ Classes

- What changes in this case are the data members:

```
class list {
   public:
      list();
      int add (const video &);
      int remove (char title[]);
      int display_all();
   private:
      video *my_list;
      int video_list_size;
      int num_of_videos;
};
```

Replace the array with these

Default Constructor

- Now, let's think about the implementation.
- First, what should the constructor do?
  - initialize the data members

```
list::list() {
        my_list = NULL;
        video_list_size = 0;
        num_of_videos = 0;
}
```

Another Constructor

- Remember function overloading? We can have the same named function occur (in the same scope) if the argument lists are unique.
- So, we can have another constructor take in a value as an argument of the number of videos
  - and go ahead and allocate the memory, so that subsequent functions can use the array

2nd Constructor

```
list::list(int size) {
    my_list = new video [size];
    video_list_size = size;
    num_of_videos = 0;
}
```

Notice, unlike arrays of characters, we don't need to add one for the terminating nul!

Clients creating objects

- The client can cause this 2nd constructor to be invoked by defining objects with initial values

```
list fun_videos(20);  //size is 20
```

If a size isn't supplied, then no memory is allocated and nothing can be stored in the array....

Default Arguments

- To fix this problem, we can merge the two constructors and replace them with a single constructor:

```
list::list(int size=100) {
     my_list = new video [size];
     video_list_size = size;
     num_of_videos = 0;
}
```

(Remember, to change the prototype for the constructor in the class interface)

Destructor

- Then, we can deallocate the memory when the lifetime of a list object is over
- When is that?
- Luckily, when the client's object of the list class lifetime is over (at the end of the block in which it is defined) -- the **<u>destructor</u>** is implicitly invoked

Destructor

- So, all we have to do is write a destructor to deallocate our dynamic memory.

```
list::~list() {
     delete [] my_list;
     my_list = NULL;
     •••
}
(Notice the ~ in front of the function name)
(It can take NO arguments and has NO return type)
(This too must be in the class interface....)
```

Review of Classes

- What is the difference between a class and a struct
- What is a data member?
- Where should a data member be placed in a class? (what section)
- What is a member function?
- Where should member functions be placed, if clients should use them?

Review of Classes

- What is the difference between a member function and a regular-old C++ function?
- What is the purpose of the constructor?
- Why is it important to implement a constructor?
- What is the difference between a class and an object?

Review of Classes

- Show an example of how a client program defines an object of a list class
- How then would the client program call the constructor? (trick question!)
- How then would the client program call the display_all function?
- Why are parens needed?

Review of Classes

- Write a simple class interface (called `number`) that has the following members:
  - an integer private data member (containing a `value`)
  - a constructor
  - a `set` member function, that takes an integer as an argument and returns nothing
  - a `display` member function

Review of Classes

- Now, let's try our hand at implementing these functions
  - a constructor
  - a `set` member function, that takes an integer as an argument and returns nothing
  - a `display` member function

Review of Classes

- What happens if we forgot to put the keyword `public` in the previous class interface?
- Why is it necessary to place the class name, followed by the scope resolution operator (`::`) when we implement a member function outside of a class?