WEEK 7 - LECTURE NOTES

Chapter 1(16-17pp): Object Orientated Programming

Chapter 10: Defining Classes

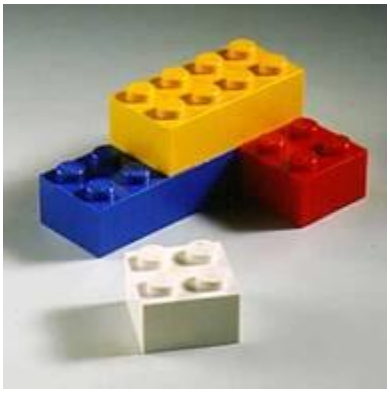
DATA MODELING – PROCEDURAL DESIGN

- Oftentimes when designing programs, we tend to think first about the tasks (or procedures) necessary to solve a problem and second on the data (variables) necessary to store information.
- In other words, we design our programs by thinking about procedures first, data second. This can be termed a procedural method of program design. Consider, for example, designing a bowling simulator program.
    - A first inclination might be to think about the functions necessary to implement the scoring system rather than on say the balls, pins, and the other "data" necessary for the program.
    - Or consider writing a program to simulate a clock. A procedural method of design would first focus on the functions (and how they work together) to keep the time rather than on the data to store the time. In procedural design, the focus is primarily on the end application and the functions necessary to solve the particular problem at hand.

OBJECT-ORIENTED DESIGN

- An alternative method of program design is to think about the data first, procedures second. Rather than first asking *how*, we ask *what*.
    - What are we working with in the problem description?
    - What is the data involved in the problem description?
    - In other words, we concern ourselves first with the data (or objects) and second with the procedures. In this way, we orient our design toward the data (objects) in the problem. In this way, our design becomes data-centered or object-oriented.
    - Although the design order of procedures and data might seem like a minor point, it can make a major impact in the overall usefulness of program code.
    - Most importantly, it can affect how reusable code can be for other applications. In this course, one of our goals is effective object-oriented program design. Among other advantages, object-oriented design methods can result in code that can be more effectively reused in other applications – an important factor in efficient software development.

- By properly designing objects, we develop code modules than can not only be used to solve the problem at hand, but also mixed and matched with others to form new applications. Similar to Lego™ pieces (see figure below), object-oriented design creates individual, reusable code modules.



- Designed properly, these code modules can be mixed with other code modules to solve not only the problem at hand, but other applications. In this way, the focus becomes more on developing effective pieces rather than on the end application. With enough proper pieces, we can build almost any kind of application (see figure below).

- Object-oriented program design begins with an analysis of the data models in a problem description.
- Data models represent the objects, data, or nouns in a problem description. For each main object identified in a problem description, we ask two basic questions:
    - "what does the object know?"
    - "what does the object do?"
- Consider the problem description of writing a clock simulation program. The primary object or noun in such a description is the clock. We can use a two-column table below to develop the data model for a clock. Under each column we list information to answer our data model questions.

CLOCK

What does it know?	What does it do?
Hours	Moves hands
Minutes	Sets time
Seconds	Keeps time
Analog or Digital	Displays time
Military time	Sets alarm
Alarm	Sounds alarm
Color	...
Size	
...	

- Our next step is to realize this data model by translating this information into actual programming code. By carefully structuring our code, we can implement our data model as a reusable module that can be used to solve not only our original problem but any problem requiring a clock. Using this object-oriented design method, we first develop data models then build applications using these data models. In this way, we can start to build up a collection of reusable objects that can be used in a variety of applications.

- To implement data models we could use any programming language. Design methods are independent of programming languages. Although some languages (Java, C++) are more conducive to object-oriented design, we can structure code in any language to reflect this design methodology.

CLASSES AND OBJECTS

- After identifying the data in a problem description, we saw how to define a data model using both classes and structures in C++. Defining a class in C++ allows us to create a new data type that encapsulates what the data model knows (the variables stored in the class) and what the object does (the member functions stored in the class). Below is an example of a C++ class definition that defines a new data type named `Solution`. This data type can be used to describe a solution that (as we saw earlier) can be used to solve problems associated with mixing different types of fluids.

```
class Solution
{
private:
  float vol;
  float per;

public:
  Solution()
  {
    vol = 0.0;
    per = 0.0;
  }
  void Init(float v, float p);
  void Print();
  void Mix(Solution *, Solution *);
  void setVol( float v ) { vol = v; }
  void setPer( float p ) { per = p; }
  float getVol() { return vol; }
  float getPer() { return per; }
};
```

Constructor

We can declare variables of this new data type in a program as in the example below.

```
int main()
```

```
{
  Solution x;
  ...
}
```

- When we declare variables of class definitions, we refer to them as objects. Note the difference between a class and an object. A class is the definition of an object, not the actual object itself. An object is a variable that has memory allocated to store the information defined in the class definition. We refer to this as the instantiation of the class. An object is an instantiation of a specific class data type. Think of the analogy of building a house. The blueprint defines what is needed to build the house and how the house is to be built. Nothing is actually built yet, rather just the definition of the house has been specified. The blueprint is equivalent to a class definition. No memory is allocated with a class definition (nothing is "built" yet). To build the house, construction workers look to the blueprint and use the specified material to build the house (lumber, nails, concrete). We could say that the actual house is the instantiation of the blueprint. Similarly, when a variable (or object) is declared, memory is allocated according to the class definition and ready for use within a program. The object is instantiated from the class definition. In our example above, we say that the object `x` is an instantiation of the `Solution` class.

- As another example of a class, consider writing code to manage a window on a screen typical of any computer application. We identify a window as a primary data model. Our design begins by asking "what does a window know?" and "what does a window do?" Our initial design specification might look like the following. Such a table is a good method to use when designing classes.

WINDOW

What does it know?	What does it do?
Position Color Border type ...	Moves horizontally Moves vertically ...

A class definition is used to implement such a design by encapsulating what the window knows (under the keyword `private`) and what a window does (under the `public` keyword).

```
class window
 {
 private:
     int xpos, ypos;       // position on screen
     int window_type;      // window design
     int border_type;      // border type
     int window_color;     // window color

 public:
     // Move horizontally
     move_hz( int amt );

     // Move vertically
     move_vt( int amt );
 };
```

INTERNAL DATA

- Objects store, in memory, the actual values for each of the variables in a class definition. These values define the state of an object at any point in time during the execution of a program. We also refer to this information as internal data since the values of the variables are internal to an individual object. Consider the analogy of a person. At any given time, we all have information that is internal to us (blood pressure, weight, heart beat). At different points in time, these values may change as they define our state. Likewise, declared objects may take on different values defining their state. As such, we refer to these as state variables. In our example above, the state variables for a window object would include the position, window type, border type, and color.

INFORMATION HIDING

- Depending on the class, state variables might contain some very important information (imagine a class storing information describing bank accounts!). Because of this importance, we must consider how to protect and hide information inside of an object. In object-oriented programming, it helps to consider two types of programmers, class developers and application developers.

CLASS DEVELOPER VS. APPLICATION DEVELOPER

- A class developer is responsible for the design, creation, and protection of a class. This is the software developer that writes the actual class definition and member functions. After the class is written, it might be used within a program or application by either the same individual or by other programmers. For our discussion, let's call these programmers application developers. These programmers use the class to declare objects to solve some problem in an application. They are only using the class, they are not writing the class. This is often the case in organizations with software development teams. One group of developers might be responsible for designing and writing reusable classes while other programmers use the classes to write applications. This is also the case with application programming interfaces (API) or class libraries. Oftentimes, some specific type of functionality is provided as a software package that can be used to create a variety of applications. A class developer might write, for example, several C++ classes to create and run a 3D game engine. These classes can be organized and stored in a library or software package that can be used by application developers to write specific types of 3D games. Finally, sometimes the class developer and the application developer might be the same person (as in the case for student programming assignments).

- The distinction between class and application developer comes into play when considering the security of objects. Who should have access to the values in the internal data (or state variables) of objects? When designing classes, it is good practice to restrict access to an object's state variables only to the member functions of the class itself. In other words, the member functions, designed and written by the class developer, should only be able to directly modify state variables. All other access to the values of state variables in applications is accomplished by calling member functions. In this way, information is said to be hidden (or protected) from the object users. C++ provides information hiding through the use of keywords `private, public, protected`. Consider the example below defining a class to describe a bank account. An initial design written by a class developer might look like the following:

```
class Account
{
public:
      int balance;            // stores the amount of money

      // Constructor
      Account( int initial ) { balance = initial; }

      // Withdraw money
      int withdraw( int amount )
      {
```

```
    if( balance >= amount )
      balance -= amount;
    else
      cerr << "Insufficient funds." << endl;
    return balance;
  }

  // Deposit money
  int deposit( int amount )
  {
    balance += amount;
    return balance;
  }
};
```

- An application developer using this class might write a program that looks like the following. Without protecting the value of the state variable, `balance`, any application developer could embezzle money using this unprotected class!

```
int main()
{
  Account Sue( 30 );
  Sue.balance += 10000; // embezzle some money
}
```

- To protect and hide the information from a user, a class developer might re-design the definition to look like the following. Note the addition of the `private` keyword to protect access (by users) to the `balance` state variable.

```
class Account
{
private:
      int balance;
public:
      Account(int initial) { balance = initial; }
      int withdraw(int amount) ...
      int deposit(int amount) ...
};
```

- With this protection, the program above will not even compile and report the following error. In this way, the class developer has hidden information about the balance to users of the class.

```
error: main() cannot access Account::balance: private member
```

- Access to the balance is now only provided through the `withdraw` and `deposit` member functions. If this is the case, would the following application be any different from the previous program?

```
int main()
{
      Account Sue( 30 );
      Sue.deposit( 10000); // embezzle some money
}
```

- The answer would be no! What might we add to our `deposit` function, then, to restrict such an illegal deposit to an account? We could add some type of security check within the function to make sure the person depositing is the owner of an account. We might add something like the following as the class developer. In this way not only can we create objects that are secure but we can hide information to users of the class.

```
public:

      // Deposit money
      int deposit( int amount )
```

```
    {
      // Check personal id number
      if( id number matches )
      {
          balance += amount;
          return balance;
      }
      else
      {
          // Call the police
      }
    }
```

- Finally, a good design practice is to allow users of objects (application developers) access to state variables through what is referred to as access functions. These functions allow users to set (assign) and get (retrieve) values from state variables. The `get` and `set` functions in the `Solution` class above (repeated below) above are examples of such access member functions.

```
class Solution
{
private:
  float vol;
  float per;

public:
  Solution()
  {
    vol = 0.0;
    per = 0.0;
  }
  void Init(float v, float p);
```

```
  void Print();
  void Mix(Solution *, Solution *);
  void setVol( float v ) { vol = v; }
  void setPer( float p ) { per = p; }
  float getVol() { return vol; }
  float getPer() { return per; }
};
```

INHERITANCE

After further analysis of our window design above, we might notice a few additional characteristics. For example, we might notice that a window can be a variety of different types. For example a window could be one of the following:

- pop-up windows
- menu windows
- help message windows
- dialog box windows

In addition, we might also notice that many of these different types of windows have many characteristics in common. For example, each would include

- position
- color
- border
- ability to move horizontally
- ability to move vertically

C++ provides a feature that allows similar classes to be grouped and organized efficiently. Organized hierarchically, classes can inherit properties (variables and functions) from other classes. Inheritance allows classes to be built from other classes. In this way, classes can be built upon what is already in place (another advantage of re-usability). As we'll see in depth later, inheritance provides the following properties:

- define a basic class with common traits, then create slight derivations
- each derivation created will automatically receive attributes of its parent
- this basic class is termed a *base class*
- the slight derivations from the base class are termed *derived classes*
- derived classes are *inherited* from base classes
- inheritance allows creation of a hierarchy of classes

- each subclass inherits or shares properties with parent
- derived classes can...
    - share data and code from base class
    - add its own special code and data to it
    - change those items that need to be different

MESSAGES, METHODS, AND BEHAVIORS

When member functions are called by objects in an application, we refer to this as message passing. Think of it as the method by which an application developer communicates with objects. A message, or command, sent to an object, telling the object to do something. For example, the program below is sending a message to a window object to "move itself horizontally."

```
int main()
{
	window w;
	w.move_hz( 10 );
}
```

The manner in which an object responds to a message is termed its behavior. Behaviors are implemented in member functions, or methods. Different objects can have different behaviors responding to the same message.

IMPLEMENTATION AND INTERFACE

Another feature of object-oriented programming is the ability to separate the implementation of an object from the interface of an object. Consider again the example below.

```
int main()
{
	window w;
	w.move_hz( 10 );
}
```

Note the user of the object (application developer) does not tell the window object how to move. The object knows how to do that. The object is simply passed a message. We can say that a user communicates with an object through an interface of a set of provided messages. We are given, by the class developer, a variety of member functions that define how it will communicate. These member functions (methods) provide the interface to

anyone using an object instantiated from the class. The user of an object does not need to know (or care) about how the message will be performed, only that it will be performed. How an object performs a task (the instructions in member functions) is referred to as the implementation. In this way, we say that the interface of an object is separated from the implementation of an object. The internal details of the implementation can and should be hidden from users of an object.

Why is separating the interface from the implementation so important? Consider the example above where the message, “move horizontally,” is repeated thousands of time in various files that form some large application using the window class. Now suppose the class developer has developed a new, more efficient way to move a window horizontally. The class developer can simply re-write the implementation (the member function `move_hz`) without ever requiring the application developers to modify the large program using this message. The message (or interface) never changes, only the implementation. The implementation is hidden and separated from the interface. Consider also proprietary classes that are written and sold as part of a software product. Users writing programs using such classes would only be given access to the list of messages provided by the class and NOT the actual implementation (code) of each of the messages. In this way, the separation allows the class to be used without giving access to the inner-workings of the code.