CPS 533 Scientific Visualization

Wensheng Shen

Department of Computational Science
SUNY Brockport
Chapter 2: Object-Oriented Program in C++

- The goal of software design
- What is object-oriented programming?
- Why object-oriented programming?
- Object-oriented concepts
- Object-oriented modeling
- Object-oriented visualization
2.1 Introduction

- Today’s software systems try to solve complex, real-world problems. That involves many people, or even several companies.
- A large problem is then divided into lots of subproblems. Each group works on one of the subproblems. Each individual person works on just a small part of the project, focusing on very specified area.
- But, when all the codes written by individual programmer are putting together, they should work well.
- The newly-developed software should be compatible with the old ones.
- Visualization system is a good example of complex software than need to be designed with extensibility.
2.1 Goals of software design

- The quality of software design is difficult to measure.

- Some qualitative aspects of software design are: robust, understandable, extendible, modular, maintainable, and reusable.
Aspects of software design

- Robust: a software is consistent, and it can handle exceptional conditions successfully.
- Understandable: the software can be used by someone other than the original developer.
- Extendable: the software can be extended to accept new tasks, new functions, new data formats, and new algorithms.
- Modular: each component of the software should be independent, and should not have strong connection with other components.
- Maintainable: the repair of one problem will not introduce another problem to the software.
- Reusable: the implementation of new technology into the existing software should be a simple task.
2.2 What is object-oriented programming

- Object-oriented programming is to divide a complex task into small and simple pieces, each of them is an object.
Why object-oriented programming

- Easy to maintain
- Easy to design
- Easy to build an extremely large software system
The main program calls procedures, data are passed as parameters.
Each object is responsible to initialize and destroy itself independently, no need to call a creation and termination procedure. Data passing through messages.
Object-oriented programming

- What are the main differences between object-oriented programming and procedure programming?
- Procedural programming limits abstraction to data typing, while object-oriented programming creates abstractions for both data and operation.
- An operation to an object is done by the programming language’s dynamic-binding mechanism.
2.3 Object-oriented concepts

- Objects are abstractions that include properties and behaviors of the entities.
- Each object has an identity that distinguishes it from other objects.
- A pointer to allocated memory or a variable name in a system-managed symbol table are often used to distinguish objects in a system.
- An object-oriented system keeps the data and operations together in one programming construct --- object.
- Data and operations are an object’s properties.
The major difference between the object-oriented systems and the conventional procedural programming systems is that they treat data abstraction differently.

Conventional systems limit abstraction to data, but the object-oriented systems create abstractions for both the data and the operations that can be applied to the data.

An object-oriented system keeps the data and operations together in one programming construct of an object.

The data and operations together comprise an object’s properties.
When an operation is applied to an object, the programming language’s dynamic-binding mechanism executes the procedure of the object.

This is not the case in procedure-oriented systems. The programmer must supply logic to decide which procedure to call. Many case statements appear in systems that handle multiple types.

In conventional procedural programs, when new types are added to these systems, the code that dispatches operations based on data types must be extended to handle the new type.
A comparison between procedural and object-oriented programming

- Procedure oriented in C:
  - Primitive *aPrim;
  - drawPrimitive (aPrim)
  - Procedure drawPrimitive (aPrim)
    
    ```c
    if(aPrim->type == TRIANGLE) then
drawTriangle (aPrim)
    else if(aPrim->type == SQUARE)
then drawSquare (aPrim)
    else if (aPrim->type == CIRCLE)
then drawCircle(aPrim)
    else return
    }
    ```

- Object oriented in C++

  ```cpp
  aPrim->draw();
  ```
In the above code, if someone wants to add a new primitive, such as a quadratic, he has to search the existing system for all occurrences of the “if” statements, and add a test for the new quadratic type. This is a formidable task.

If a system is built by more than one programmer, it must be under a configuration management system, requiring a check-out, edit, and check-in cycle.

The object-oriented programmer has an easier task. He consults the design document that defines the object properties for a primitive, adds a draw operation to the quadratic object. The new primitive is available to the system without changing any existing code.
2.4 Object-oriented terminology

- **Object**: an object is an abstraction that models the state and behavior of entities in a system.

- The state of an object is its attributes, and the behavior of an object is defined by operations.

- Attributes have a name, a data type, and a data value. The data type of an attribute may be a primitive type in the programming language, such as a char or float in C++, or another object. Example: the `vtkTransform` object has an attribute of type `vtkMatrix4×4`, another object.

- Operations are functions or transformations of an object. Operations define the behavior of the object. The operations are implemented in procedures called methods.

- Attributes and operations of an object comprise its properties. A two-dimensional line graph could have attributes including an x and y axis, a legend, and a connected set of points. This graph has methods that draw the graph in a window, and the methods that let a user specify the axes, data to draw, and legend to use.
Mapping a real-world object into an object abstraction. The real-world objects are various types of trees. One of these objects (a pin oak tree) is mapped into the computer object, called PinOak.
class aTree
{
    private:
        int age;
        double height;
        double trunkDiameter;
        int habit;
    
    public:
        void setAge(int a){age = a;}
        void setHeight(int h){height = h;}
        void settrunkDiameter(double trunkDiameter){
            trunkDiameter = td;
        }
        void setHabit(int habit);
        void grow();
        void setSeason();
};

Main()
{
    aTree pinOak;
    pinOak.setAge(20);
    pinOak.setHeight(20.0);
    pinOak.grow();
    pinOak.setSeason();
    return (0);
}
Objects that share the same properties can be grouped using the process of classification. An object class specifies the properties that all objects in the class have.

The class only specifies the names of the properties, not their specific values. Different classes can have properties with names that exist in other classes. For example, many classes in the vtk visualization system have an attribute Position, but it has different effect, because they are different classes.

Attribute names are shared by all objects in a given class, but separate storage is allocated for each object’s attribute values.
The operation is called **polymorphic**, if an operation with the same name is applied to objects of different classes. For example, vtk has an operation named `Render()` that can be applied to many different objects.

- The implementation of an operation for a particular class is called a method. The print operation for a `vtkMatrix4x4` object is implemented in its print method. That is, there exists code that knows how to print objects of class `vtkMatrix4x4` but not objects of other classes.

- Objects know which method to use because they are kept within each object’s data structure. In most systems, the code for the methods is shared by all objects in the same class.

- Some program language, including C++, define a method by combining an operation name with its argument types. This process is called overloading an operation and is a powerful technique that permits the same name to be used for logically similar operations.
For example, the class below defines three methods for calculating the square of a number. Even though these methods have the same operation name, they are unique because C++ uses both the operation name and the operation argument types.

```cpp
Class math
{
    float square(float x);
    int square (int x);
    double square(double x);
}
```
To use a member of a class, we create an instance of the class. Instance creation establishes the identity of the instance including specifying its initial state. The instance’s class serves as a template for the instance during creation, defining the names of each of its attributes and operations.

Creation establishes the similarities and differences between this instance and other instances of the same class.

The similarities are the names and type of its attributes and the methods that implement its operations.

The differences are the specific values of the attributes.
Instance creation of VTK

- To create an object from the program stack:
  
  ```
  vtkActor aBall;
  ```

- To create an object from the program heap:
  
  ```
  vtkActor *aBall;
  aBall = new vtkActor;
  ```
Inheritance

- Most systems can be specified using a hierarchical classification system.
- Inheritance is a programming mechanism which makes it easy adding new classes that are different from existing classes only slightly.
- For example, to add one more apple to an existing apple orchard.
Class hierarchy

- Properties of a general classification are also properties of its subclassification.
- General class = superclass
  = base class
- Subclass = derived class
- Any instance variables and methods of a superclass are automatically inherited by its subclass.
The creation of inheritance

- Inheritance can be derived top-down by a process called specialization, or it can be created bottom-up by a process called generalization.

- Superclasses are also called abstract classes, and instance creation of an abstract class is generally prohibited.
Example of an abstract class

Assuming a data presentation application allows for many two-dimensional plotting, and the application support line charts and horizontal and vertical bar charts. The design process identifies properties common to all plots including title, axes, and legend. We then create an abstract class TwoDPlot to contain these common attributes. Common behavior can also be captured in TwoDPlot with its plot method:

```plaintext
Method Plot
{
  Draw the border
  scale the data
  draw the axes
  draw the data
  draw the title
  draw the legend
}
```
2.5 Object-oriented modeling and design

- Objects in computer animation are: actors, cameras, and lights.
- Objects in visualization are: models, contours, isosurfaces, streamlines, and cut planes.
Object modeling technique (OMT)

- The object model: the object model identifies each object in the system, its properties, and its relationships to other objects in the system.
- Dynamic model: the dynamic model describes the sequences of events and time dependencies of the system.
- The functional model: the functional model shows how data flows through the system, how processes and algorithms transform the data, as well as functional dependencies between processes.
- The object model is the dominating model.
Object model for locator devices

Locator
- Attribute: device
- Methods: open(), close()

Locator3D
- Attributes: position(x,y,z), orientation, front, up
- Method: locate()

locator2D
- Attribute: position(x,y)
- Method: locate()

- Articulated angles, linkages
- Flock
  - Position()
  - Orientation()
- Pixsys
  - Position()
  - Orientation()
- Logitek
  - Position()
  - Orientation()
- Immersion
  - Position()
  - Orientation()
- Phantom
  - Position()
  - Orientation()
- Touch screen
  - Position()
- Mouse
  - Position()
Data flow diagram

CT/MRI Scanner -> Slices -> Filter -> Slices -> Segment

Labelled slices -> Surface extraction

Surface extraction -> Triangles -> Render

Triangles -> Render

Render -> Pixels -> Image

Image -> Pixels

Write -> Triangle file

Triangle file -> Triangles

Read -> Triangles
2.6 C++ Review

- C++ is almost completely a superset of C, and it is better than C.

- It adds data abstraction and support for object oriented programming

- It does not force any particular programming style upon users (e.g. procedural vs OOP)

- It is constrained by the goal of supporting C-style programming (unlike languages with no legacy support such as Java).
The main function

- header is `int main ()` // NOT "void main"
- always have one in any C or C++ program
- is the marker for the start of the program execution
- empty () in header = no parameters
- braces around body {} (must balance)
- `return 0;` at the end of the function, matches the `int` in the header line
The heading of a function

```
int main()
```

type of returned value

name of function

says no parameters
Data

- has a name - either itself or an identifier
- has a type - integer, character, float, etc.
- has a space = an address = memory
- has a value - 7.2, 3, 'd', true
- can be stored as a variable (x) OR a named constant (PI) OR a literal constant (4)
Identifiers

- rules for making them
  - must start with a letter or underscore, and be followed by zero or more letters (A-Z, a-z), digits(0-9), or underscores

- case sensitivity
- used for names of variables, named constants, or functions
- all identifiers must be DECLARED before use
Identifiers

• VALID
  
age_of_dog         taxRateY2K
  PrintHeading      ageOfHorse

• NOT VALID (Why?)
  
age#              2000TaxRate    Age-Of-Cat
Data types

- the type determines which values can be used and the operations you can perform
- protect you from making logical mistakes
- int, float, double - numeric
  - exponential notation
  - overflow, underflow
- char - ONE character
- string - must include the string library
- bool - logical
Declarations

- **Variables**
  - Syntax to declare is “type identifier, identifier, ...
    ;”

- **Constants**
  - literals
  - named constants
    - Syntax to declare is "const type identifier = value;"
    - Why use them? documentation, easy to change, prevents typing errors
What Does a Variable Declaration Do?

A declaration tells the compiler to allocate enough memory to hold a value of this data type and to associate the identifier with this location.

```
int ageOfDog;
float taxRate;
char middleInitial;
```

4 bytes for taxRate

1 byte for middleInitial
A **string** is a sequence of characters enclosed in double quotes.

**Sample string values**

```
“Hello”  “Year 2000”  “1234”
```

The **empty string** (null string) contains no characters and is written as “”.
More About Type String

- **A string is not a built-in (standard) type**
  - It is provided in the C++ string library
  - Must include `<string>` - NOT `<string.h>`
    NOR `<cstring>`

- **String operations include**
  - Comparing 2 string values
  - Searching a string for a particular character
  - Joining one string to another - concatenation
What is a Named Constant?

A named constant is a location in memory that can be referred to by an identifier and in which a data value that cannot be changed is stored.

Valid constant declarations

```
const string STARS = "****";
const float NORMAL_TEMP = 98.6;
const char BLANK = ' ';
const int VOTING_AGE = 18;
const float MAX_HOURS = 40.0;
```
Comments

- used to explain code to author and other readers
- /* */
- //
- USE them!
- There is a bad example on class page: Other Useful Links
- Don't comment out code unless you mean to
General program format

- header comments at the top of the file
- one statement per line
- use spaces, vertical and horizontal, to make it easier to read
- `#include` statement
- `using namespace std;` statement
Block(Compound Statement)

- A block is a sequence of zero or more statements enclosed by a pair of curly braces `{ }`

**SYNTAX**

```
{  
    Statement (optional)  
    . . .  
}  
```

- Braces should line up and statements inside should be indented
- Blocks are used for the body of a function, as well as grouping statements together (like Alice uses boxes)
Variables and Arrays

- C++ has basic data types such as short, int, float, char, double etc.

- The statements

```c
int a;
float b[10];
double c[5][5];
```

define a single integer `a`, an one-dimensional array of floats `b`: `b[0] .. b[9]` and a two-dimensional array of doubles `c[0][0] .. c[4][4]` (All array indices start at 0.)

- Both `a`, `b` and `c` are implicitly allocated and will be deallocated when the function containing them exits. Their sizes are fixed at compile time.
Dynamic Memory Allocation

- Dynamic allocation allows the creation of arrays whose size is determined at runtime (e.g. loading an image whose size can vary).

- It is one of the key to writing memory efficient programs.

- It is, arguably, also the biggest source of problems and crashes in most C++ code. Most of the problems are caused by:

  1. Accessing/Deleting arrays/objects before they are allocated and initialized.
  2. Accessing/Deleting arrays/objects after they have been already deleted
  3. Neglecting to delete arrays/objects
Pointers

• Dynamic Memory Allocation reserves a portion of memory for a specific data structure (array or object).

• The allocation process returns the physical memory address of the data structure, and in turn the physical memory address is stored in a pointer variable.

• The type of the pointer variable indicates the kind of object that is being stored.

• The type \( T^* \) represents a pointer to memory holding objects of type \( T \) e.g. \( \text{float}^* \) represents a pointer a memory block holding an array of floats (arrays can have length 1 or greater!!!
Allocating/De-allocating Pointers

- Memory allocation is performed by the `new` command e.g.

```c
int* a = new int;
```

- `a` is NOT an integer. It is a pointer which contains a memory location. To access the value stored at the memory location pointed by `a` use the de-referencing operator `*` e.g.

```c
*a = 1
//This sets the value of the integer whose location is stored in a to 1
```

- When `a` is no longer needed the memory can be released using the `delete` command e.g.

```c
delete a;
```
Allocating/De-allocating Arrays of Pointers

• As before allocation is performed by the `new` command e.g.

```c++
int* a=new int[10]; // Allocate an array of 10 integers
```

• `a` is a pointer which contains the memory location of the first element of the array. To access the values stored at the memory locations use:

```c++
a[0] = 1 ; a[3]=2; etc.
```

• When `a` is no longer needed the memory can be released using the `delete []` command e.g.

```c++
delete [] a;
```

• Do not use the `delete` operator to delete arrays, it causes lots of problems use `delete []` !!!
Two-dimensional Arrays

- Two-dimensional arrays are created as an array of one-dimensional arrays: e.g. a 5x10 matrix could be stored as:

```c++
// allocate five rows of floats
float** matrix=new float*[5];
for (int i=0; i<=4; i++) {
    matrix[i]=new float[10];
}
```

- The array can be accessed in the usual way e.g. `matrix[0][2]=2.0`.
- Deallocation is tricky and messy:

```c++
for (int i=0; i<=4; i++) {
    delete [] matrix[i];
}
delete [] matrix;
```
• Often two-dimensional arrays are simulated using one dimensional arrays e.g.

```cpp
float* matrix=new float[5*10];
Matrix[10*row+column]=5;
```

• This is more cumbersome to use but avoids all issues with multi-dimensional pointers with respect to allocation/de-allocation.

• All VTK-arrays internally are stored as one-dimensional arrays.
Executable statements

- Output statements
- Input statements
- Assignment statements
- Return statement
Output statements

- `cout`
  - standard output stream
  - always goes to the screen / console
- insertion operator `<<`
  - can be chained in one statement
- outputting literals
- outputting variables and named constants
- escaped characters
- `endl`
Output Statements

SYNTAX

These examples yield the same output:

```
cout << "The answer is ";
cout << 3 * 4;
cout << "The answer is " << 3 * 4;
```
Input statements

- cin
  - standard input stream
  - always comes from the keyboard
- extraction operator
  - can chain them together in the same statement
Return statement

- always at the end of the sequence in the main function
- control turned back over to Operating System
- return value 0 for normal exit
Assignment statement

- syntax
- semantics
- operators
  - / and % - integer operations
  - precedence of operators
  - mixed types and their effect on results
  - type casts
  - concatenation of strings
- Math functions – pow, sqrt – cmath library
String Concatenation (+)

- **Concatenation** is a binary operation that uses the + operator.

- At least one of the operands must be a string variable or named string constant--the other operand can be a string literal or a char variable, literal, or constant.
Concatenation Example

```cpp
const string WHEN = "Tomorrow";
const char EXCLAMATION = '!'s;
string message1;
    string message2;

message1 = "Yesterday "s;
message2 = "and "s;
message1 = message1 + message2 +
    WHEN + EXCLAMATION;
```
Working with the IDE

- Build versus Rebuild versus Clean
- “build is up-to-date” warning - Watch out!!
- Errors and warnings
  - Warning level W2 is what will be used to grade your programs
Syntax Errors

- compiler may report lots of them
- fix the FIRST one then recompile
- try to understand what the error was
- error messages are not clear
  - keep a log
- do not ignore warning messages!
Logic (Semantic) Errors

- They are only found by testing
- Test plan
  - more than one case
  - each case tests different code
  - input for each case
  - expected output for the case worked out by hand
Runtime Errors

- something the program cannot control in its environment
  - example: division by zero
  - example: file not found
- usually crashes the program
- more advanced - read about 'exceptions'
Creating a Chessboard

**Problem** Your college is hosting a chess tournament, and the people running the tournament want to record the final positions of the pieces in each game on a sheet of paper with a chessboard preprinted on it. Your job is to write a program to preprint these pieces of paper. The chessboard is an eight-by-eight pattern of squares that alternate between black and white, with the upper left square being white. You need to print out squares of light characters (spaces) and dark characters (such as *) in this pattern to form the chessboard.
## Chessboard

### Constants

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACK</td>
<td>'********'</td>
<td>Characters forming one line of a black square</td>
</tr>
<tr>
<td>WHITE</td>
<td>' '</td>
<td>Characters forming one line of a white square</td>
</tr>
</tbody>
</table>

### Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>whiteRow</td>
<td>string</td>
<td>A row beginning with a white square</td>
</tr>
<tr>
<td>blackRow</td>
<td>string</td>
<td>A row beginning with a black square</td>
</tr>
</tbody>
</table>
Algorithm

Repeat four times
  Output five whiteRows
  Output five blackRows
C++ Program

//*********************************************
*****
// Chessboard program
// This program prints a chessboard pattern that is
// built up from basic strings of white and black
// characters.
//*********************************************
*****
#include <iostream>
#include <string>
using namespace std;
const string BLACK = "********"; // Define black square line
const string WHITE = "        "; // Define white square line
```cpp
int main()
{
    string whiteRow;   // White square beginning row
    string blackRow;   // Black square beginning row
    // Create a white-black row
    whiteRow = "WHITE + BLACK + WHITE + BLACK + WHITE + BLACK + WHITE + BLACK";
    // Create a black-white row
    blackRow = "BLACK + WHITE + BLACK + WHITE + BLACK + WHITE + BLACK + WHITE";
}
```
C++ Program

// Print five white-black rows
cout << whiteRow << endl;
cout << whiteRow << endl;
cout << whiteRow << endl;
cout << whiteRow << endl;
cout << whiteRow << endl;

// Print five black-white rows
cout << blackRow << endl;
cout << blackRow << endl;
cout << blackRow << endl;
cout << blackRow << endl;
cout << blackRow << endl;

// Print rest of the rows
...
return 0;
}