OBJECT ORIENTED PROGRAMMING
OBJECT ORIENTED PROGRAMMING

• Design principles for organizing code into user-defined types

• Principles include:
  • Encapsulation
  • Inheritance
  • Polymorphism

USER DEFINED TYPES

- When solving problems often times it becomes easier for programmers to express algorithms in terms of real world objects or complex types
  - Ex. vectors, matrices, polygons, etc
  - Ex. Cars, planes, and trains
- Types define both the internal data and operations which can be done on the data
  - Ex. A vector might hold an x and a y data and define operations such vector addition and subtraction
- C++ allows for complex types in the form of classes and structs
CLASS EXAMPLE

• Can use struct or class to define a new type

```cpp
struct Point {
    float m_x, m_y;
    void Set(float x, float y) {
        m_x = x; m_y = y;
    }
};
```
```cpp
class Point {
    float m_x, m_y;
public:
    Point(float x, float y) :
        m_x(x), m_y(y) {} 
    void Set(float x, float y) {
        m_x = x; m_y = y;
    }
};
```
ENCAPSULATION

• A construct that facilitates the bundling of data with the methods (or other functions) operating on that data

• A mechanism for restricting access to some of the object's components

• Advise:
  • Helper functions and data should be “Encapsulated”, i.e., should be private or protected
  • Use Accessors and Modifiers to “Get” and “Set” the data
  • Bundle operations on an object inside the class, not as utility functions
CONSTRUCTORS AND DESTRUCTORS

• **Constructors** – “functions” which define how data is initialized
  • Same name as the class itself
  • Default constructor – takes no parameters, e.g., Point()
  • Copy constructor – takes an object of the class itself to initialize the data, e.g., Point(oldPoint)

• **Destructors** – define how data is deleted when an object goes out of scope
  • Same name as the class itself except with a ~, e.g., ~MyClass()
  • Usually not called in code you write. The compiler automatically calls it when an object goes out of scope.
PUBLIC, PROTECTED, AND PRIVATE SCOPES

• **Public**: functions and data which any part of a program may access
  • Ex. Data Accessors – GetData(), modifiers – SetData(newData), operations – ActionX()

• **Protected**: functions and data which only the class and any derived subclasses may access
  • Ex. Helper functions for operations. Most data in class hierarchies

• **Private**: functions and data which only the class itself may access
  • Ex. Helper functions and some data

• **By default**: classes have private scope while structs have public scope
QUICK NOTE ON “STATIC”

• Static data and functions are functions of a type which are not specific to an instance.

• Examples
  • Intrinsic type properties – value of PI, shared data of classes (container of all instances)
  • Functions on those properties
INHERITANCE

• Inheritance is a construct allowing reuse of code in existing objects and definition of subtypes
• Gives rise to class hierarchies
• Base class: class which is inherited from, e.g., Shape
• Derived class: class which does the inheriting, e.g., Circle
• By inheriting, derived classes have access to public and protected data and functions of the base class
ABSTRACT TYPES AND HIERARCHIES

• Often many types have things in common, e.g., all shapes can be drawn or all shapes have color

• However, you cannot concretely define a shape. In this case a shape is abstract

• We derive more concrete classes off of abstract ones, e.g., a circle is a concrete shape
INHERITANCE EXAMPLE

class Shape{
protected:
    Color m_c;
public:
    Shape(Color c) : m_c(c) {}
    void Draw(){
        //do nothing}
};

class Circle : public Shape {
    float m_radius;
public:
    Circle(Color c, float r) : Shape(c), m_radius(r) {}
    void Draw() {
        //draw circle of radius r
    }
};

class Square : public Shape {
    float m_side;
public:
    Square(Color c, float s) : Shape(c), m_side(s) {}
    void Draw() {
        //draw circle of radius r
    }
};

• Notice:
  • How to inherit a class
  • How to call the base class constructor
  • Data is inherited
POLYMORPHISM

• Defining and constructing data and algorithms in terms of abstract classes
• At runtime the specific type of the class is decided and its behavior is executed
• Abstract classes provide an interface to express algorithmic use, whereas derived classes provide the concrete implementation
• Abstract classes use “virtual” functions to ensure subtypes implement the correct interface
ABSTRACT CLASSES

- Abstract classes describe an interface
- “virtual” is a keyword used to allow subtypes to overload the function in the manner they see fit
- “pure virtual” functions MUST be overloaded in the base class

```cpp
class Shape {
  virtual void DrawNotPureVirtual() {
    // do nothing
  }
  virtual void DrawPureVirtual() = 0;
};

class Circle {
  // virtual void DrawNotPureVirtual()
  // does not need to be defined. By inheritance it will do nothing
  virtual void DrawPureVirtual() { // draw my circle
  }
};
```
POLYMORPHISM IN ALGORITHMS

• As stated before polymorphism allows algorithms to be expressed in terms of abstract classes

• Utilizes pointers (discussed in future review in more depth) and virtual lookup at runtime

• At runtime the concrete type’s implementation is used

```cpp
void DrawAllShapes(Shape** shapes, int numShapes){
    for(int i = 0; i<numShapes; i++){
        //virtual function call.
        //Lookup of concrete implementation occurs at runtime
        shapes[i]->Draw();
    }
}
```