Brief Notes on Object-Oriented Software Design and Programming with C++

OpenSees Developers Workshop

Pacific Earthquake Engineering Research Center

August 21-23, 2001

Gregory L. Fenves

1. Introduction

OpenSees is a software framework for creating models and analysis methods to simulate structural and geotechnical systems. The framework has been designed and implemented using object-oriented techniques because, when properly used, this process produces software that is more modular, flexible, and extensible than software designed using other methodologies. To implement software designed using object-oriented techniques, it is desirable (although not essential) to use a language that supports object-oriented concepts. The most widely used object-oriented programming languages are C++ and Java; OpenSees is implemented in C++.

For OpenSees developers interested in material and element models, it is important to understand the overall design of OpenSees, but many of the details are not critical to using the framework (this is one of the advantage of object-oriented design). In terms of programming, developers will need to learn the rudiments of C++, but at the material and element level, the organization and calculations are similar to Fortran programs, although the syntax of statements and operators is different. For developers interested in tackling more challenging problems, more familiarity with the OpenSees architecture and object-oriented techniques is necessary. The objective of the Developers Workshop is to provide a self-contained description of object-oriented software design, the OpenSees architecture, and implementing materials and elements using C++. This should meet the needs of the first group of developers and provide an introduction for the second group.

These notes constitute the first step in the learning process. They do not discuss OpenSees directly; rather they provide an overview of software design using object-oriented techniques. Through two examples, object designs are expressed as C++ classes. Implementation of the classes is not directly covered in the lecture, but is provided in appendices for further study (especially after learning about the C++ language). In a short lecture it is not possible, and it would be a conceit, to claim that this is a comprehensive document on software design and programming. The scope is more modest in providing an introduction to modern concepts in software engineering for those interested in using and contributing to OpenSees. A bibliography for further study is in the last section.

2. The Problem of Software Design

The key concepts for designing and implementing software are decomposition, abstraction, and modularity. Before examining the object-oriented approaches based on these principles, it is worth reviewing the problems that software engineering addresses.

Software design must address the complexity of systems. As outlined by Booch (1994), software is complex because:

- <u>The problems are complex</u>. In our engineering domain, the design and analysis of structural and geotechnical systems requires understanding how the physical materials and components behave and how they interact together. There are numerous assumptions about physical behavior, requirements for engineering design, and uncertainty about loading. Robust and flexible software must deal with algorithmic issues, modeling accuracy, solution procedures, numerical stability, computational performance, databases for performance evaluation, and interfaces with scientific visualization tools and design software.

- Requirements change. During the software design process the requirements will likely change as the application is developed in more depth. This may be "feature creep" or more fundamentally an improved understanding of the needs as engineers use the software. After a software system is developed, it must grow and change as new uses and extensions are needed. The operating systems, user-interface systems, and networks change over the life of the application. The software application must be designed to be insulated from current technology, yet take advantage of increased capabilities of new information technology.

- A program may be unconstrained. A programmer starting with a blank screen can do anything. At first the unconstrained nature of a program appears to be an advantage, but for large systems the flexibility can be detrimental because software becomes handcoded without generalization considered. Software developed in an unconstrained manner is generally the antithesis of reusable software components that can form the building blocks of a large software system.

- The development process introduces complexity. Teams develop large software systems. In actual applications it is not possible for one person to understand all aspects of the software (at least at one time). Managing the complexity, and remaining cognizant of the end-users needs, is a paramount concern of software engineering.

Software design is usually a top-down process. *Decomposition* of a problem into smaller problems is the key to controlling the complexity of software. Our experience with numerical computation has emphasized procedural decomposition by breaking an algorithm into simpler steps. Whereas this type of decomposition is appropriate for pure algorithmic processes, as software addresses non-algorithmic problems (such as structural design) or relationships between algorithms, algorithmic abstraction is limited or even inapplicable. A more general view emphasizes decomposition by data abstractions. The purpose of data abstractions is to represent the essential (software) behavior of the data. Good abstractions lead to modular programs, in which software components are as independent as possible. Modularity is the best way to develop reliable software that is easier to change. A *framework* is a collection of modules (representing important data abstractions) that can be used to develop specific applications in a domain.

3. Data Abstraction

A fundamental aspect of software design and programming is data abstraction. Data abstraction involves determining the properties of the data and the operations on the data necessary to represent a problem. A data abstraction is formalized as an abstract data type (ADT). Object-oriented programming provides language support for implementing abstract data types.

Data abstraction can be compared and contrasted with procedural abstraction of a problem. Procedural abstraction emphasizes the process or algorithm with less emphasis on the data. Procedural programming languages have control structures (such as iteration and selection) but generally weak support for data. In the teaching of computer programming in engineering disciplines, procedural abstraction is emphasized and data abstraction is often neglected. Data abstraction is essential for developing modern engineering software. What often seems to be procedural abstraction in a program are really operations on data. Careful design of a data abstraction often results in programs that are easier to write, understand, and modify.

3.1 Abstract Data Type

Abstract data types are used to define data abstractions. An abstract data type describes the *behavior* of objects of that type. The description, or *specification*, of the behavior is separated from the *implementation* of the behavior. This separation allows the software designer to concentrate on the important question of what the data represents from how the data are implemented in a program.

An abstract data type consists of:

- A set of objects
- Operations that can be applied to objects

For example, integers are an abstract data type used in mathematics. The integer type is the set of all integers. Mathematical operations such as addition and subtraction are defined, and all operations are closed. The type `int` in C/C++ is an implementation of the abstract data type integers. The implementation does not represent all integers, only the ones within in the word length and bit representation for `int`, which is system dependent. We use variables of `int` in a program with full confidence that they will behave like abstract, mathematical integers.

The goal of data abstraction is to select data types appropriate for a particular problem. The behavior of the abstract data type is carefully specified, i.e. what does the set include and what are valid operations. The implementation then involves producing the specified behavior with what we will see is the concept of class and object in C++.

3.2 Example of Specification of an Abstract Data Type

A vector is a mathematical quantity representing a magnitude and direction in N-dimensional space of real numbers (for this example). We would like to use vector variables in our program that have the correct mathematical behavior. A specification and use of an abstract data type for vectors follows.

The abstract data type vector is a set of all vectors in N–dimensional space. The specification includes the following operations on members of the set:

- define a vector
- magnitude of a vector, returning a scalar
- addition of two vectors, returning a vector
- multiplication by a scalar, returning a vector
- dot product of two vectors, returning a scalar

A formal specification would precisely indicate what is the meaning of these operations. For vectors the mathematical operations are straightforward. In a practical data type many more operators would be defined.

Let's look a t C++ program that uses a class called Vector that has the operations listed above. For the time being, we will not be too concerned with the C++ syntax, but concentrate on the important point that objects of the class are created (constructed) and operated upon according the specification for the class. The definition of the vector class in these notes is similar but not identical to the class of the same name in OpenSees.

```
#include"vector.h"

void main ( void )
{
    Vector v1(4,1.0), v2(4,2.0);      // v1 initialed to 1; v2 to 2
    Vector s1, s2, v3;                // vectors of undefined size.
    float d;

    s1 = v1.vAdd(v2);                 // addition with vAdd operator
    s2 = v1 + v2;                     // addition with overloaded + operator

    Vector v4(4,10);                  // create vector, initial to 10.
    d = v4*v1;                        // inner product with overloaded *

    v2[0]=v2[0]+v2[1];                // example of index operation

    v2+=v1;                           // compound assignment, v2=v2+v1
}
```

C++ is a strongly type language. All data and objects must be declared to have a type. Built-in types include `int`, `float`, and `double`. In this example an new type is included, `Vector`, so it must be `defined` and the general practice is to define the type in a header (`*.h`) file, which is included using the `#include` precompiler directive.

The header file for `Vector` contains the specification for the data type. For this example it is:

```
// ADT for Vector

class Vector {

     public:
          Vector ( int sz=3, float val=0.0); // default to 3D vector
          Vector ( const Vector& );          // copy constructor
          ~Vector ( void );                  // destructor

          Vector& operator= ( const Vector& w );// assignment

          Vector& operator= ( float s ); // assign vector constant

          float vMag   ( void )    const;
          Vector vAdd  ( const Vector& w ) const;
          Vector vMult ( float s         ) const;
          float vDot   ( const Vector &w ) const;

          int vGetSize ( void  ) const;


          // Overloaded operators
          Vector operator+ ( const Vector& w) const; // add
          Vector operator- ( const Vector& w) const; // subtract
          Vector operator* ( float s ) const; // mult. by scalar
          Vector operator/ ( float s ) const; // div. by scalar
          float operator*  ( const Vector& w ) const; // dot product

          // Subscript operators
          float& operator[] ( int i );             // LH side
          const float& operator[] (int i ) const;  // RHS

          // Compound assignment operators
          Vector& operator+= ( const Vector& w ); // add to object
          Vector& operator-= ( const Vector& w ); // sub. from object
          Vector& operator*= ( const float s  ); // multiply by scalar
          Vector& operator/= ( const float s  ); // divide by scalar

          // Equality operations
          int operator== ( const Vector& w ) const;
          int operator!= ( const Vector& w ) const;

     private:
          float *vec;
          int size;
};
```

There are two constructors available for public use. The first declares the size, with a default of 3, and initial value, with a default of zero. The second constructor is needed to define copying of a vector object. It takes an argument that is a reference to a vector. The destructor for the class is `~Vector`. The next operator is an assignment operator defining what is meant to assign one vector to another.

This class uses dynamic memory allocation to request memory for a vector. *Classes that have dynamic memory allocation should have a destructor, a copy constructor, and an assignment operator defined*. This is important to handle the dynamically allocated memory properly and avoid dangling pointers and memory leaks.

The next group of member functions is the standard mathematical operators, overloaded to define vector operations. These are followed by subscript operators for subscripting access to vector elements. Finally, the last group of functions includes the overloaded compound assignment operators.

The overloaded mathematical operators return vectors to preserve the notion that the result of an operation is a valid type to which another operator can be applied. For example, the definition allows the proper evaluation of an expression like:

```
v2 = (v2 + v1 – v3) * v4;
```

The overloaded operators return actual vectors, which involves copying (a significant overhead that is best avoided for critical operations). The compound assignment operators are more efficient because the vector (and memory) for the result of the operation already exists.

The last section of the class definition gives the instance variables for the class. Each object of this class has two instance variables. The variable `size` is the number of elements in the vector and `vec` is a pointer to dynamically allocated memory of `size` floats. The last section is private, meaning that only objects created from vector can access the data. The privacy allows the implementation of the how vectors are represented and how the operations are performed to be isolated (hidden) from how the vectors are used in an application.

Since the purpose of these notes is to present the design concepts, implementation of the Vector class will not be covered. However, Appendix A of these notes does provide the implementation.

4. Object-oriented Software Design

Object-oriented software design views a system as a collection of objects that coordinate and communicate with each other to provide the desired functionality. There are various ways to identify the objects, which is the craft of software engineering. Gamma et al. (1995) provides one of the best approaches for object-oriented software design. The key issues are:

- Abstraction. Abstraction represents the essential behavior of a software component independent of how the behavior is implemented or computed. The abstraction is the essential properties that distinguish one component from another in the system.
- Hierarchy. Hierarchy is a ranking of abstractions from high-level to low-level, where more specific functionality is provided moving from high to low levels.
- Encapsulation. Encapsulation is a general term for hiding information within a component. It the mechanism for isolating implementation decisions from specification.
- Concurrency. Concurrency deals with multiple processes (or threads) interacting with each other. Interoperability of software components is becoming an important characteristic of networked systems.
- Persistence. Information associated with an application must live beyond the activation of a a process.

Prior to object-oriented programming in the mid-1980's, much work had been done on modules and the use of modules to support abstraction. The so-called Parnas (1972) module was one that required each module to have a specific interface and functionality, below which some information was hidden or private. An important benefit of modules is the opportunity for *software reuse*. Software reuse addresses the problem of a completely unconstrained design space for software. It provides a foundation of components that are general enough to use in more than one application.

The idea of an object was anextension of a module. Objects have the properties of modules but in addition they support hierarchies and encapsulation. A software object represents a part of the real-world addressed by the application. The correspondence between the real-world and the software is the key to defining objects and an advantage of the object-oriented approach. In addition, objects may represent convenient behavior that abstracts important operations (such as the vectors) or processes (such as a time integration algorithm).

Objects are specific instances of a class. Objects are instantiated (or constructed) from a class. Objects of a class are independent and separate, but they all have the behavior defined by the class. Objects package data and operations that together provide the desired software behavior. The external behavior may be implemented in a variety of ways, but the representation and calculations are hidden (or encapsulated) inside the object. The user of an object is not concerned with how the behavior is realized.

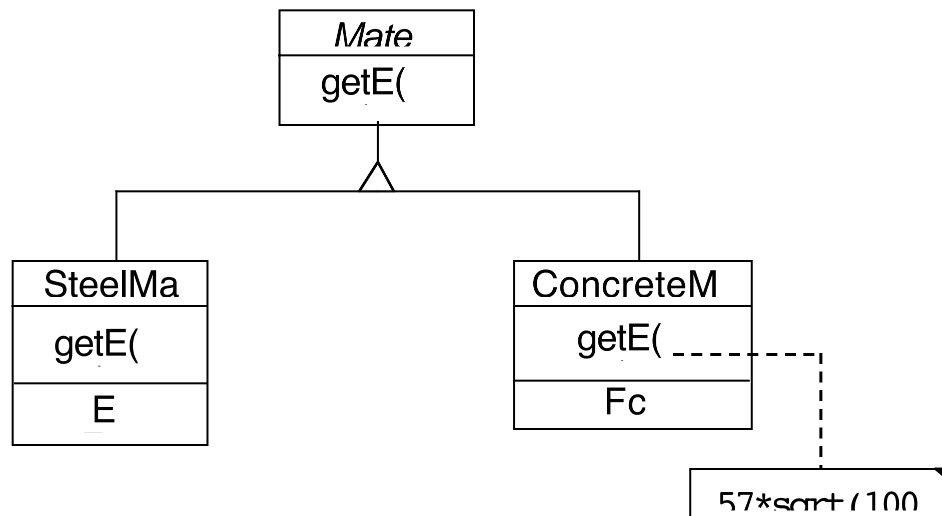
Operations on objects in a class are variously referred to a method or member function. Invoking an object may be called sending a message to the object or selecting a function. Methods (or member functions) operate on the object for which they are invoked. As we will see later, there are different ways functions are bound to an object based on the class of the object.

5. Example of Object-oriented Software Design

We now consider software design using the object-modeling technique. The methodology has been referred to as the object modeling technique and, more recently, the unified modeling language (UML) as described in the books by Rumbaugh and Booch. After the object-oriented design is completed we will design an interface for C++ classes and use then in a program. The implementation of the classes is presented in Appendix B.

5.1 Material Class

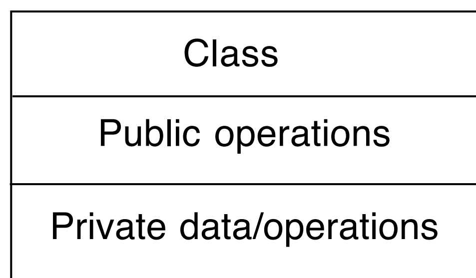
First we develop a material class with specialization to steel and concrete. At the top level is a material that provides functionality to obtain the elastic modulus. It should be noted that the functionality of the material class in this example is to represent basic properties. In contrast the Material class in OpenSees is used for state determination of material models.



Material is a class that has one operation on objects of that class, `getE()`, to determine the modulus of elasticity of the object. Because the class is very general, details must be provided for specific materials such as steel and concrete. The Material is an *abstract* class because it does not provide specific information about the representation of all materials, it only has the specification of behavior common to all materials. To provide specific information about actual materials, we use the concept of specializing material classes using class inheritance.

Inheritance means that a derived class, such as ConcreteMaterial, inherits the public interface of the base class (Material) while providing specific implementations appropriate for the concrete. In traditional object-oriented programming terminology, Material is termed a superclass of ConcreteMaterial and SteelMaterial, and ConcreteMaterial and SteelMaterial are subclasses of Material. The terminology in C++ is that Material is a based class for ConcreteMaterial and SteelMaterial. The latter two are derived classes.

For designing classes, we adopt a simplified form of the unified modeling language and its precursor the object modeling language. More examples will be presented.



With the classes defined, let's examine the class declaration to show the correspondence between the design and the specification of the class. The declaration is given below and we will then go through the various aspects of it.

```
class Material
{
    public:
        virtual double getE ( void ) const = 0;
        virtual ~Material (void);
};

class SteelMaterial : public Material
{
    public:
        SteelMaterial ( double f, double e=29000 );
        virtual double getE  ( void ) const;
        virtual double getFy ( void ) const;

    private:
        double E;     // Modulus of elasticity
        double Fy;    // Nominal yield stress
};

class ConcreteMaterial : public Material
{
    public:
        ConcreteMaterial ( double f );
        virtual double getE  ( void ) const;
        virtual double getFc ( void ) const;
```

```
        private:
            double Fc;    // Compressive strength
    };
```

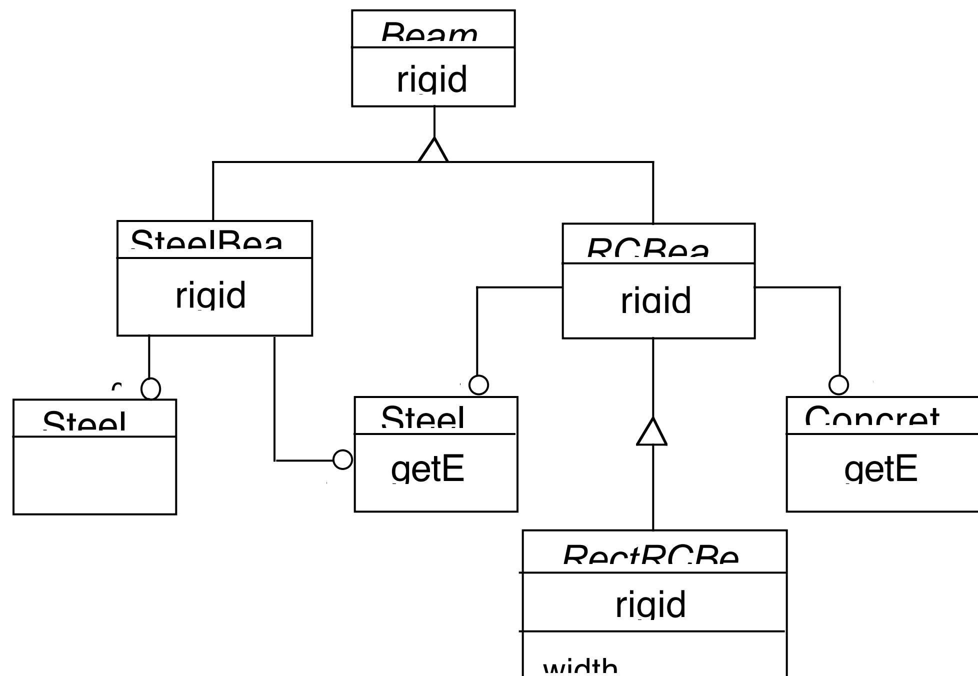
The material class is as an abstract class. It does not have a constructor because no instances will be created of this class. It must have a destructor, which destroys objects when they are no longer needed. An abstract class must have a destructor because of a subtle point having to do with deleting memory when subclasses are deleted. The member functions for the Material class have the keyword virtual. This means that subclasses may or may not be required to implement the function. The C++ compiler creates the underlying code needed to support dynamic binding of virtual functions an object depending on the class of the object. In the Material class, the function `getE()` is declared with the "=0" convention. This means that `getE()` is a *pure virtual function*. Subclasses *must* implement the `getE()` function or it is a compile error. Since Material is an abstract class it has no private variables because it does not provide a representation of actual materials.

SteelMaterial is a derived class (subclass) of Material and it is declared in the class specification using *public inheritance*. The SteelMaterial class has a constructor and access functions, and it has private variables for this simple representation of steel. A similar process is done for the derived class ConcreteMaterial. Notice that each derived class declares an implementation of the pure virtual function `getE()` and adds additional member functions appropriate to the representation of the specific material.

The functions for the derived classes are also declared to be virtual (but not pure virtual) since at a later time there may be even more specific derived classes. However, if the software designer is sure this would not happen then the virtual is not needed and there is a very small gain in runtime efficiency for the dynamic binding.

5.2 Classes for Beam Section

The design for representing beam sections is a follows:



To represent different types of steel sections, a simple hierarchy is used:

```
class BeamSection
{
    public:
        virtual double rigidity ( void ) const = 0;
        virtual double flexCap  ( void ) const = 0;
        virtual ~BeamSection ( void );
};

class SteelBeamSection : public BeamSection
{
    public:
        SteelBeamSection ( void );
        SteelBeamSection ( const SteelMaterial& b );
        SteelBeamSection ( const SteelMaterial& b, const SteelSection& a );

        virtual double rigidity ( void ) const;
        virtual double flexCap  ( void ) const;
        virtual void setSteelSection ( const SteelSection& a );

    private:
        SteelSection*  aSteelSec;
        const SteelMaterial* aSteelMat;
};

class RConcreteBeamSection : public BeamSection
{
    public:
        virtual double rigidity ( void ) const = 0;
        virtual double flexCap  ( void ) const = 0;

    protected:
        RConcreteBeamSection ( const SteelMaterial& a,
            const ConcreteMaterial& b );
        virtual double getIcr ( void ) const = 0;

        const SteelMaterial*    aSteelMat;
        const ConcreteMaterial* aConcreteMat;
};

class RectRConcreteBeamSection : public RConcreteBeamSection
{
    public:
        virtual double rigidity ( void ) const = 0;
        virtual double flexCap  ( void ) const = 0;
        virtual void setWidth ( double w );
        virtual void setDepth ( double h );
        virtual void setEffectiveDepth ( double d );
        virtual double getWidth ( void ) const;
        virtual double getDepth ( void ) const;
        virtual double getEffectiveDepth ( void ) const;

    protected:
        RectRConcreteBeamSection ( const SteelMaterial& a,
            const ConcreteMaterial& b,
            double w = 0, double h = 0, double d = 0 );
        virtual double getIcr ( void ) const;
```

```
    private:
        double width;
        double depth_h;
        double depth_d;
};

class RectSingleRConcreteBeamSection : public RectRConcreteBeamSection
{
    public:
        RectSingleRConcreteBeamSection ( const SteelMaterial& a,
            const ConcreteMaterial& b,
            double w = 0, double h = 0, double d = 0, double A = 0 );
        virtual double rigidity ( void ) const;
        virtual double flexCap  ( void ) const;
        virtual void setAs ( double A );
        virtual double getAs ( void ) const;

    private:
        double As;
};
```

The SteelBeamSection has two constructors depending on whether the SteelSection is defined at the time the beam object is defined; otherwise the public interface includes a setSection operation.

RconcreteBeamSection and RectRconcreteBeam section providing increasingly detailed representation of RC beams, but hey are still abstract classes. Notice that these two classes have an additional type of functions and data termed *protected*. Protected means that subclasses have access to the functions and data. The constructors are provided because there is a specific representation that is useful for the derived classes (subclasses).

Private data are strictly private so even subclasses do not have access. This is sometimes a difficult design decision. Moving functionality up the hierarchy increases code re-use. However, it makes the derived classes more dependent on decisions with the base class.

Finally for this example problem, the class SteelSection provides an abstraction of different types of structural steel cross sections. The class declaration is:

```
class SteelSection
{
    public:
        virtual double getZ ( void ) const;
        virtual double getI ( void ) const;
        virtual ~SteelSection ( void );

    protected:
        SteelSection ( double zxx, double ixx );

    private:
        double Z;
        double I;
};

class WFSteelSection : public SteelSection
{
    public:
        WFSteelSection ( double zxx, double ixx, double depth, double
width );
```

```
            double getDepth ( void ) const;
            double getWidth ( void ) const;

        private:
            double h;
            double b;
    };
```

For this example, the subclasses for specific shapes only store additional data. Further design would provide additional functionality that depends on the shape of the section, and additional subclasses could be defined.

5.3 Sample Application

With the interfaces defined, the classes can be used. Here is an example that creates two beam sections, one steel and one reinforced concrete, and then computes the flexural rigidity and strength. Notice that the programming is self-explanatory (with the aid of the interface) and the same calculations are done for the different types of beams.

```
int main ( void )
{
    // Create material objects
    SteelMaterial      a36 = SteelMaterial(36);
    SteelMaterial      a60 = SteelMaterial(60);
    ConcreteMaterial   f4  = ConcreteMaterial(4);

    // Create a steel WF section
    SteelSection sec1 = WFSteelSection(400,300,12,8);

    // Create beam sections with material only
    SteelBeamSection beam1 = SteelBeamSection (a36);
    RectSingleRConcreteBeamSection beam2 =
                      RectSingleRConcreteBeamSection(a60,f4);

    // Set section for steel beam
    beam1.setSteelSection(sec1);

    // Define a singly reinforced concrete beam
    double h = 24;
    beam2.setWidth(h/2);
    beam2.setDepth(h);
    beam2.setEffectiveDepth(h-3);
    beam2.setAs(6);

    // Cast upward to test dynamic binding of member functions
    BeamSection *beam3 = dynamic_cast<BeamSection*>(&beam2);

    // Get flexural properties of two beams
    double EI = beam1.rigidity();
    double Mp = beam1.flexCap();

    double EI2=beam3->rigidity();
    double Mn =beam3->flexCap();

    cout << "Steel Beam:    EI=" << EI  << "  Mp=" << Mp << endl;
    cout << "Concrete Beam: EI=" << EI2 << "  Mn=" << Mn << endl;
}
```

6. Bibliography

C/C++ and Object-Oriented Programming

Deitel, H., and Deitel, P., *C++ How to Program*, 2nd Edition, Prentice-Hall, 1997.
Perry, J., Levin, H., Gamma, E., *An Introduction to Object-Oriented Design in C++*, Addison-Wesley, 1999.
Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, 1997.

Object-Oriented Software Design

Gamma, E., Elm, R., Johnson, R., Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
Rumbaugh, J., Jacobson, I., Booch, G., The Unified Modeling Language Reference Manual, Addison-Wesley, 1998.
Booch, G., Jacobson, I., ., Rumbaugh, J., *The Unified Modeling Language User Guide*, Addison-Wesley, 1998.
Booch, G, *Object Oriented Design with Applications*, 2nd Edition, Benjamin/Cummings, 1994.
Rumbaugh, J., Blaha, M, Premerlani, W., Eddy, F., and Lorenson, W., *Object–Oriented Modeling and Design*, Prentice–Hall, 1991.
Goldberg, A., Robson, D., *Smalltalk–80, The Language and its Implementation*, Addison–Wesley, 1983.

Structured Programming, Modularity and Abstraction

Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R., *Structured Programming*, Academic Press, 1972.
Dijkstra, E.W., *A Discipline of Programming,* Prentice–Hall, 1976.
Knuth, D.E., "Structured Programming with Go To Statements," *Computing Surveys*, ACM, Vol. 6, No. 4, pp. 261-301.
Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, Vol. 15, Dec., 1972, pp. 330-336.
Shaw, M., "Abstraction Techniques in Modern Programming Languages,", *IEEE Software*, Vol. 1, No. 4, Oct., 1984, p. 10.

Computing Algorithms and Data Structures

Horowitz, E., Sahni, S., Rajasekaran, S., *Computer Algorithms*, Computer Science Press, 1997.
Knuth, D., *Fundamental Algorithms*, The Art of Computer Programming, Vol.1, 2nd Ed., Addison–Wesley,1973.
Wirth, N., *Algorithms and Data Structures*, Prentice–Hall, 1986.

Appendix A - Implementation of the Vector Class

The class is defined in the file `vector.C`. The following description presents sections of code from that file. The beginning has the include files and a declaration of a private module function for error handling:

```
#include<math.h>
#include<stdlib.h>
#include<iostream.h>
#include"vector.h"

// Private functions

static void vectorError ( const char *);
```

The constructors include a standard one and, very importantly, a copy constructor. A destructor must be defined to release dynamically allocated memory.

```
// Constructor and intialize
Vector::Vector ( int sz, float val )
{
     if (sz < 1 )
          vectorError("Invalid size");

     vec = new float[size=sz];
     if (!vec)
          vectorError("Insufficient memory.");

     for ( int i = 0; i < size; i++)
          vec[i]=val;
}

// Copy constructor
Vector::Vector( const Vector& w ) : size( w.size )
{
     vec= new float[size];
     if (!vec)
          vectorError("Insufficient memory.");

     // copy w into new object
     for (int i = 0; i < size; i++ )
          vec[i] = w.vec[i];
}


// Destructor
Vector::~Vector ( void )
{
     delete [] vec;
}
```

The constructor checks that the size is valid, uses new to allocate memory, checks that the returned pointer is valid, and initializes the vector. The copy constructor is very similar. Given the reference to a vector, the size is initialized (notice, using the initializer), memory allocated, checked, and initialized. The destructor returns the dynamically allocated memory with the `delete` operator.

To allow assignments of vectors, such as `v2=v1`, the assignment operator must be overloaded to say how copies are made. The assignment operator, like the copy constructor, takes the reference to a vector (on the right hand side of the assignment). First it checks that the operation is not assigning a vector to itself, in which case nothing needs to be done. If the assignment is valid and the lefthand side is a different size from the right, it deletes the current definition of the lefthand side vector, including releasing memory (avoiding a memory leak), and allocates new memory. Finally, it copies the values from the right to the left. The return is a reference to the left side vector with the `this` pointer.

```
// Overloaded assignment operator
Vector& Vector::operator=( const Vector& w )
{
     if ( &w != this )      // Don't assign an array to itself
     {
          if ( size != w.size )
          {
               delete [] vec;                // delete current storage
```

```
                                         // and allocate new storage
        vec = new float[size=w.size]; // of correct size
        if (!vec)
            vectorError("Insufficient memory.");
    }

    for ( int i = 0; i < size; i++ )
        vec[i] = w.vec[i];            // copy w to self
}

return *this; // return reference to self
}
```

Any class that has dynamically allocated memory should include a destructor, copy constructor, and assignment to handle the memory properly. If functions are not provided, C++ will use default operations, which will fail in many cases, or at least cause memory leaks and dangling pointers.

For convenience, assignment is overloaded to assign an array to a constant value, such as the expression: `v6=8`. The declaration Vector `v7(100)=3` is equivalent to `Vector v7(100,3)`.

```
Vector& Vector::operator=( float s )
{
    for ( int i = 0; i < size; i++ )
        vec[i] = s;

    return *this;
}
```

The access functions are straightforward.

```
// Access functions
int   Vector::vGetSize ( void ) const { return size; }
```

The standard vector operations are defined with overloaded operators. The functions take the right hand side as a constant reference to a vector and return a vector. It is important to note that the functions construct a new vector for the result of the operation and then returns the new vector by value (by making a copying). When needed, vectors are checked for size compatibility.

```
Vector Vector::vAdd ( const Vector& w ) const
{
    Vector sum(size);      // create vector for sum

    if ( size == w.size )
    {
        for ( int i = 0; i < size; i++ )
            sum.vec[i] = vec[i] + w.vec[i];
    }
    else
        vectorError("Invalid addition.");

    return sum;       // return new vector for sum
}

// Multiply by scalar
Vector Vector::vMult ( float s ) const
{
    Vector pr(size);       // create vector for product
```

```
    for ( int i = 0; i < size; i++ )
            pr.vec[i] = vec[i]*s;

    return pr;              // return new vector
}
```

An automatic variable of type Vector is created, with the constructor, of the size of the vector receiving the operation (the left operand). Then the operation is performed defining the new vector. The new vector is returned by value, which involves a copy operation. After the return the `sum` and `pr` are destroyed and the destructor releases the dynamically allocated memory.

The dot product related operations do not require creating vectors for the result:

```
// Dot product
float Vector::vDot ( const Vector& w) const
{
    float sum = 0.0;

    if ( size == w.size )
    {
        for ( int i = 0; i < size; i++ )
            sum += vec[i] * w.vec[i];
    }
    else
        vectorError("Invalid dot product.");

    return sum;
}

// Magnitude
float Vector::vMag ( void ) const
{
    return sqrt ( this->vDot( *this ) );
}
```

Next, we will examine the overloaded compound assignment operators. Since the memory is allocated for the left operand, no additional allocation is needed. A reference to the left hand side is returned. Addition and subtraction need to check for compatibility.

```
// Compound assignment operators
Vector& Vector::operator+= ( const Vector& w)
{
    if ( size == w.size )
    {
        for ( int i = 0; i < size; i++ )
            vec[i] += w.vec[i];
    }
    else
        vectorError("Invalid addition.");

    return *this;  // return vector with sum
}

Vector& Vector::operator-= ( const Vector& w)
{
    if ( size == w.size )
```

```
    {
        for ( int i = 0; i < size; i++ )
            vec[i] -= w.vec[i];
    }
    else
        vectorError("Invalid subtraction.");

    return *this;      // return vector with difference
}

// Scale vector
Vector& Vector::operator*= ( const float s)
{
    for ( int i = 0; i < size; i++ )
        vec[i] *= s;

    return *this;      // return scaled vector
}

Vector& Vector::operator/= ( const float s)
{
    for ( int i = 0; i < size; i++ )
        vec[i] /= s;

    return *this;      // return scaled vector
}
```

The overloaded algebraic operations can now be defined. The overloaded binary operators can be implemented in an efficient way by utilizing the corresponding compound assignment operators (+=, -=, etc.). As shown below a variable of type Vector is created using the copy constructor to duplicate the left operand. This creates the memory of the correct size and the right operand can be applied using the corresponding overloaded compound assignment.

```
Vector Vector::operator+ ( const Vector& w ) const
{
    Vector sum(*this);

    sum += w;
    return sum;
}

Vector Vector::operator- ( const Vector& w ) const
{
    Vector sum(*this);

    sum -= w;
    return sum;
}


Vector Vector::operator* ( float s ) const
{
    Vector sum(*this);

    sum *= s;
    return sum;
}
```

```
Vector Vector::operator/ ( float s ) const
{
    Vector sum(*this);

    sum /= s;
    return sum;
}
```

The asterisk is overloaded for a dot product, using the previously defined function:

```
// Dot product
float Vector::operator* ( const Vector& w ) const
{
    return this->vDot(w);
}
```

The subscript operators allow vector indexing for expressions on the left and right of an assignment. The first operation will modify the i'th element of the vector when it appears to the left of an assignment operator. The second operation is constant, and it is used to access a value on the right size of an assignment.

```
// Subscript operators
float& Vector::operator[] ( int i )
{
    if ( i<0 || i>=size)
        vectorError("Invalid index.");
    return vec[i];    // Modify left hand side of assignment
}

const float& Vector::operator[] (int i ) const
{
    if ( i<0 || i>=size)
        vectorError("Invalid index.");
    return vec[i];        // return value
}
```

Note, we could have defined ()'s to be the syntax for indexing with operator(). It is also possible to change the range of index starting at 1.

The equality operators check to see if two vectors are the same. Notice that `operator==` violates the one-entry, one-return rule, but for simple functions the logic is clear. The `operator!=` is defined in terms of `operator==`.

```
// Equality operators
bool Vector::operator== ( const Vector& w) const
{
    if ( size == w.size )
    {
        for ( int i = 0; i < size ; i++ )
            if ( vec[i] != w.vec[i] ) return false;

        return true;
    }

    return false;
}
```

```
bool Vector::operator!= ( const Vector& w) const
{
    return ! ( *this == w );
}
```

The private function is not defined as a member function for the class:

```
// Define private functions
static void vectorError ( const char *mess)
{
    cerr << "Program error in Vector class: " << mess << endl;
    exit(-1);
}
```

To summarize the discussion about return type for operations that produce a new intermediate vector, it is best to give up the clarity of binary mathematical operations and accumulate the results of operations with the compound operators. For example:

```
Vector v1, v2, v3;

// define vectors with same order

v1 = v1 * 10 + v2 * (v2*v3);
```

is valid, but results in copying the return value from three operations. The programmer can avoid the overhead of copying intermediate values of the expression by:

```
d = v2*v3; // avoid side effects
v2 *= d;
v1 *= 10;
v1 += v2;
```

This requires no additional memory allocation, but it obviously overwrites `v2`.

Appendix B – Implementation of Classes for Structural Beam Example

Material Classes

The implementation of the Material classes is shown below. There are no methods for class Material because it is abstract (except for the destructor). The constructors for the derived classes check for valid ranges of the parameters to preserve invariants of the classes.

```
// Default destructor
Material::~Material ( void )
{ }

// SteelMaterial methods
SteelMaterial::SteelMaterial ( double f, double e)
{
    if ( e > 0 )
        E = e;
    else
        errorExit("SteelMaterial", "Invalid modulus of elasticity.");

    if ( f > 0 )
        Fy = f;
```

```
    else
        errorExit("SteelMaterial", "Invalid yield strength.");
}

double SteelMaterial::getE  ( void ) const { return E;  }
double SteelMaterial::getFy ( void ) const { return Fy; }


// ConcreteMaterial methods
ConcreteMaterial::ConcreteMaterial ( double f )
{
    if ( f > 0 )
        Fc = f;
    else
        errorExit("ConcreteMaterial","Invalid compressive strength");
}

double ConcreteMaterial::getE ( void ) const
{
    return 57.0*sqrt(Fc*1000); // per ACI,  normal weight concrete
}

double ConcreteMaterial::getFc ( void ) const { return Fc; }
```

Beam Section Classes

The implementation of the classes that represent beam sections is:

```
// Default destructor
BeamSection::~BeamSection ( void )
{ }

// SteelBeamSection methods
SteelBeamSection::SteelBeamSection ( void)
    : aSteelSec(0), aSteelMat(0)
{ }

SteelBeamSection::SteelBeamSection ( const SteelMaterial& b )
{
    aSteelSec = 0;

    if (&b)
        aSteelMat = &b;
    else
        errorExit("SteelBeamSection","Invalid steel material.");
}

SteelBeamSection::SteelBeamSection ( const SteelMaterial& b,
        const SteelSection& a )
{
    if (&a)
        setSteelSection(a);

    if (&b)
        aSteelMat = &b;
    else
        errorExit("SteelBeamSection","Invalid steel material.");
}
```

```
void SteelBeamSection::setSteelSection ( const SteelSection& a )
{
     if (&a)
     {
          if (!aSteelSec)
               aSteelSec = const_cast<SteelSection*>(&a);
          else
               errorExit("SteelBeamSection",
                    "Steel section previously defined.");
     }
     else
          errorExit("SteelBeamSection","Invalid steel section.");
}

double SteelBeamSection::rigidity ( void ) const
{
     double I = 0;

     if (aSteelSec)
          I = aSteelSec->getI();

     return aSteelMat->getE() * I;
}


double SteelBeamSection::flexCap ( void ) const
{
     double Z = 0;

     if (aSteelSec)
          Z = aSteelSec->getZ();

     return aSteelMat->getFy() * Z;
}


// ConcreteBeamSection methods

RConcreteBeamSection::RConcreteBeamSection ( const SteelMaterial& a,
                           const ConcreteMaterial& b)
{
     if (&a)
          aSteelMat = &a;
     else
          errorExit("RConcreteBeamSection","Invalid steel material.");

     if (&b)
          aConcreteMat = &b;
     else
          errorExit("RConcreteBeamSection","Invalid concrete material.");
}


RectRConcreteBeamSection::RectRConcreteBeamSection
          ( const SteelMaterial& a, const ConcreteMaterial& b,
            double w, double h, double d )
     : RConcreteBeamSection(a,b)
```

```
{
     setWidth(w);
     setDepth(h);
     setEffectiveDepth(d);
}


void RectRConcreteBeamSection::setWidth ( double w )
{
     if ( w >= 0 )
          width = w;
     else
          errorExit("RectRConcreteBeamSection","Invalid width.");
}

void RectRConcreteBeamSection::setDepth ( double h )
{
     if ( h >= 0 )
          depth_h = h;
     else
          errorExit("RectRConcreteBeamSection","Invalid depth.");
}

void RectRConcreteBeamSection::setEffectiveDepth ( double d )
{
     if ( d >= 0 )
          depth_d = d;
     else
          errorExit("RectRConcreteBeamSection","Invalid effective
depth.");
}

double RectRConcreteBeamSection::getWidth ( void ) const
     { return width; }

double RectRConcreteBeamSection::getDepth ( void ) const
     { return depth_h; }

double RectRConcreteBeamSection::getEffectiveDepth ( void ) const
     { return depth_d; }

double RectRConcreteBeamSection::getIcr ( void ) const
{
     // cracked modulus factor
     const static double crackFactor = 0.50;

     return crackFactor*width*pow(depth_h,3)/12;
}

// Single reinforced rectangular concrete beam methods

RectSingleRConcreteBeamSection::RectSingleRConcreteBeamSection
          ( const SteelMaterial& a, const ConcreteMaterial& b,
            double w, double h, double d, double A )
     : RectRConcreteBeamSection(a,b,w,h,d)
{
     setAs(A);
}
```

```
void RectSingleRConcreteBeamSection::setAs ( double A )
{
     if ( A >= 0 )
          As = A;
     else
          errorExit("RectSingleRConcreteBeamSection","Invalid steel
area.");
}

double RectSingleRConcreteBeamSection::getAs ( void ) const { return As;
}

double RectSingleRConcreteBeamSection::rigidity ( void ) const
{
     return aConcreteMat->getE() * getIcr();
}

double RectSingleRConcreteBeamSection::flexCap ( void ) const
{
     double b  = getWidth();
     double d  = getEffectiveDepth();
     double Ft = aSteelMat->getFy() * As;
     double Mn = 0;

     if ( b > 0 && d > 0 )
     {
          double ablock = Ft/(0.85 * aConcreteMat->getFc() * b );
          Mn = Ft * ( d - 0.50*ablock);
     }

     return Mn;
}
```

In the last function, the properties of the section and materials are obtained by sending messages. Notice that no assumptions are made about how the rectangular section is represented. The access functions are used to obtain the data since it is private to the base class RectRConcreteBeamSection.

Steel Section Classes

```
// default destructor
SteelSection::~SteelSection  ( void )
{ }

// construct steel section
SteelSection::SteelSection ( double zxx, double ixx )
{
     if ( zxx > 0 )
          Z = zxx;
     else
          errorExit("SteelSection","Invalid Z.");

     if ( ixx > 0 )
          I = ixx;
     else
          errorExit("SteelSection","Invalid I.");
}
```

```
double SteelSection::getZ ( void ) const { return Z; }
double SteelSection::getI ( void ) const { return I; }


WFSteelSection::WFSteelSection ( double zxx, double ixx, double depth,
                      double width )
     : SteelSection(zxx,ixx)
{
     if ( depth > 0 )
          this->h = width;
     else
          errorExit("WFSteelSection","Invalid depth.");

     if ( width > 0 )
          b = width;
     else
          errorExit("WFSteelSection","Invalid width.");
}

double WFSteelSection::getDepth ( void ) const { return h; }
double WFSteelSection::getWidth ( void ) const { return b; }
```