

Encapsulation.

One of the most important concepts in OO programming is called encapsulation. I intend to demonstrate encapsulation by example rather than by giving an abstract definition. However I will tell you something about encapsulation before I begin.

When C++ was born in 1984 (credit for this goes to Bjarne Stroustrup) he first called the language "C with classes". A class is the most important difference between C and C++. With classes you can extend the built-in data types that we have discussed with your own. You can define your own functions and operators on those classes.

Did I ever mention how a floating point number was stored on a computer? No. Why not? It wasn't important. Encapsulation roughly means data hiding: clients of a new data type should know its protocol, but not its insides.

A good analogy is an integrated circuit. Users of an integrated circuit need to read the TTL data book (for example) and to understand the pinout diagram. They do not need to have see detailed circuit diagrams of the inside of the integrated circuits.

Think of classes, at first, as data types similar to int or double. A class like "vector" is a good example to start with, because it has well defined properties that are relatively easy to capture in a class.

Later, you'll realize that all of the work in a C++ program can be accomplished through classes, -not just low-level work. But, just, like the the vector example that we'll discuss now, your classes should have a simple well-define interface and represent a single abstraction.

This part of the seminar will be presented as a series of programs for calculating the sum of two vectors, with each example moving one step closer to Object-oriented programming.

A FORTRAN or C programmer's first instinct would be this:

```
// A program to take the sum of two vectors
#include <iostream.h>

// Here is the prototype for the vector addition function
//
void vectorSum(double a0, double a1, double a2,
               double b0, double b1, double b2,
               double & c0, double & c1, double & c2);

// main routine just tests the vector addition function
main() {
    double x0 = 3.0 , y0=2.4, z0=-3.5;
    double x1 = -2.1, y1=-5.9, z1 = 3.4;
    double x2, y2, z2;

    vectorSum(x0,y0,z0,x1,y1,z1,x2,y2,z2);

    cout << x0 << '\t' << y0 << '\t' << z0 << '+' << endl
         << x1 << '\t' << y1 << '\t' << z1 << '=' << endl
         << x2 << '\t' << y2 << '\t' << z2 << endl;

}

// here is the implementation of the vector addition function
void vectorSum(double a0, double a1, double a2,
               double b0, double b1, double b2,
               double & c0, double & c1, double & c2){
    c0 = a0 + b0;
    c1 = a1 + b1;
    c2 = a2 + b2;
}
```

```

// A program to take the sum of two vectors
#include <iostream.h>

// We introduce a structure to agglomerate the
// x, y, and z variables of a vector:

struct vect3D {

    double _x;
    double _y;
    double _z;
};

// Vector sum is declared in term of the struct:
void vectorSum(vect3D a, vect3D b, vect3D & c);

main() {
    vect3D x0, x1, x2;

    x0._x = 3.0 , x0._y=2.4, x0._z=-3.5;
    x1._x = -2.1, x1._y=-5.9, x1._z = 3.4;

    // Vector sum now is simpler to invoke:
    vectorSum(x0,x1,x2);

    cout
        << x0._x << '\t' << x0._y << '\t' << x0._z << '+' << endl
        << x1._x << '\t' << x1._y << '\t' << x1._z << '=' << endl
        << x2._x << '\t' << x2._y << '\t' << x2._z << endl;

}

// Here is the implementation of vector sum:
void vectorSum(vect3D a, vect3D b, vect3D & c) {
    c._x = a._x + b._x;
    c._y = a._y + b._y;
    c._z = a._z + b._z;
}

```

A slight rearrangement (return the new data type) makes the function look more natural:

```
// A program to take the sum of two vectors
#include <iostream.h>

// We introduce a structure to agglomerate the
// x, y, and z variables of a vector:

struct vect3D {

    double _x;
    double _y;
    double _z;
};

// Vector sum is declared in term of the struct:
vect3D vectorSum(vect3D a, vect3D b);

main() {
    vect3D x0, x1, x2;

    x0._x = 3.0 , x0._y=2.4, x0._z=-3.5;
    x1._x = -2.1, x1._y=-5.9, x1._z = 3.4;

    // Vector sum now is simpler to invoke:
    x2=vectorSum(x0,x1);

    cout
        << x0._x << '\t' << x0._y << '\t' << x0._z << '+' << endl
        << x1._x << '\t' << x1._y << '\t' << x1._z << '=' << endl
        << x2._x << '\t' << x2._y << '\t' << x2._z << endl;

}

// Here is the implementation of vector sum:
vect3D vectorSum(vect3D a, vect3D b) {
    vect3D c;
    c._x = a._x + b._x;
    c._y = a._y + b._y;
    c._z = a._z + b._z;
    return c;
}
```

Next, add initialization and vector addition to the definition of a structure:

```
struct vect3D {
    // data members:
    double _x;
    double _y;
    double _z;
    // now let the structure handle addition:
    vect3D operator+ (const vect3D & right) {
        vect3D c;
        c._x = _x + right._x;
        c._y = _y + right._y;
        c._z = _z + right._z;
        return c;
    }
    // give the structure a means of initializing itself
    // (a constructor)
    vect3D (double x=0, double y=0, double z=0) {
        _x=x;
        _y=y;
        _z=z;
    }
};

// give the structure a print method:
inline ostream & operator<< ( ostream & o, const vect3D & v){
    o << v._x << '\t' << v._y << '\t' << v._z ;
    return o;
}

main() {
    // Declare & initialize three vectors.
    vect3D x0(3.0,2.4,-3.5),
           x1(-2.1,-5.9,3.4),
           x2;

    // Vector sum now is simpler to invoke:
    x2=x0 + x1;

    cout << x0 << "+" << endl
         << x1 << "=" << endl
         << x2 << endl;
}
```

This is OK, but unfortunately main gets to read (and write!) the guts of this structure. If we change the data type to a class, we can control this. We'll separate the class into "guts" and "interface". The "client" only gets to see the interface.

The main program doesn't change. The definition of Vect3D does.

```
class vect3D {  
  
    // data members:  
private:  
  
    double _x;  
    double _y;  
    double _z;  
  
public:  
    // now let the structure handle addition:  
    vect3D operator+ (const vect3D & right) {  
        vect3D c;  
        c._x = _x + right._x;  
        c._y = _y + right._y;  
        c._z = _z + right._z;  
        return c;  
    }  
  
    // give the structure a means of initializing itself  
    // (a constructor)  
    vect3D (double x=0, double y=0, double z=0) {  
        _x=x;  
        _y=y;  
        _z=z;  
    }  
  
    friend inline ostream & operator << ( ostream & o,  
                                           const vect3D & v);  
  
};  
  
// give the structure a print method:  
inline ostream & operator <<( ostream & o, const vect3D & v){  
    o << v._x << '\t' << v._y << '\t' << v._z ;  
    return o;  
}
```

The class can only be addressed through it's interface. The next steps would be:

- * Make a "header file" for the vector class.
- * Separate the implementation of the vector classes member functions and make a library.

Then, the header file would contain only this:

```
#include <iostream.h>
class vect3D {

    // data members, invisible

private:

    double _x;
    double _y;
    double _z;

public:

    // addition:
    vect3D operator + (const vect3D & right);

    // (a constructor)
    vect3D (double x=0, double y=0, double z=0);

    // (a print method)
    friend inline ostream & operator << ( ostream & o,
                                           const vect3D & v);

};

inline ostream & operator<< ( ostream & o, const vect3D & v);
```

This header file is the class definition. The "public" part is called the **public interface** of the class. Clients can only go through the public interface to the class! The public interface defines the data type.

The implementation can change. For example, instead of x, y, z we could use r, theta, and phi to describe a vector. This would not affect any clients of the class.

Client code now looks like this:

```
#include <vect3D.h>
#include <iostream.h>

main() {
    // Declare & initialize three vectors.
    vect3D x0(3.0,2.4,-3.5),
           x1(-2.1,-5.9,3.4),
           x2;

    // Vector sum now is simpler to invoke:
    x2=x0 + x1;

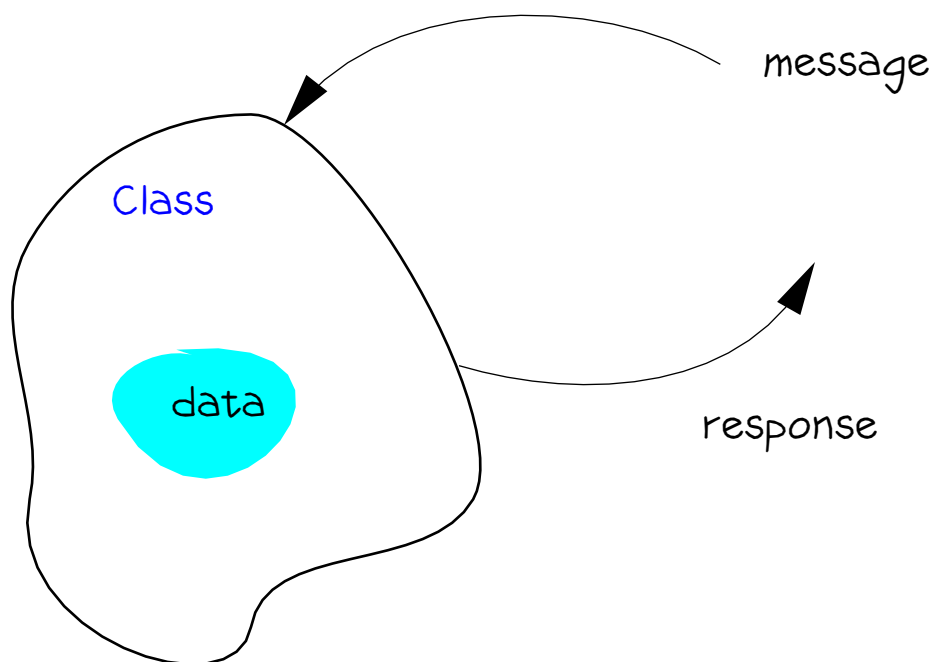
    cout << x0 << "+" << endl
         << x1 << "=" << endl
         << x2 << endl;
}
```

What could be simpler?

In designing classes, one should think first of the abstract interface. Then, he or she should choose a representation that supports the interface. Finally, he or she should implement the entire interface. In the case of vectors, we still lack a cross product, a dot product, subtraction, multiplication by a scalar, and many more.

The act of designing the class is often more difficult than implementing it.

One more look at encapsulation:



The data within a class is encapsulated so that the clients see only the class's public interface. Arbitrary data types can be made to behave in arbitrary ways. The details of how to achieve the desired behaviour do not "spill out" of the class and into client code.

We will continue this discussion by learning how to build classes.