Object-Oriented Programming

- *Object-oriented programming* (or *OOP*) attempts to allow the programmer to use data objects in ways that represent how they are viewed in the real world with less attention to the implementation method.
- An *object* is characterized by its name, its properties (values that it contains) and its methods (procedures and operations performed on it).
Principles of OOP

There are four main principles of OOP:

- **Data Abstraction** - our main concern is what data represents and not how it is implemented.
- **Encapsulation** - Private information about an object ought not be available the outside world and used only in prespecified ways.
- **Polymorphism** - There may be more than version of a given function, where the different functions share a name but have different parameter lists.
- **Inheritance** - New classes of objects can be built from other classes of objects.

Defining A Class of Objects

- Defining a class of objects is similar to defining a type of structure, except that we can add functions as well as variables.
- By default, any item included is **private** (available only to functions within the class of objects) unless we include the keyword **public**.
Syntax For a Class Definition

- The syntax is:
  ```
  class classname {
  public:
      declarations for public functions and variables
  private:
      declarations for private functions and variables
  }
  ```

Example of a Class definition

```
class point {
public:
    void read(void);
    void write(void);
    float distance(void);
    float distance(point p);
private:
    float x, y;
};
```
Writing a Class Function

```cpp
float point::distance(void)
{
    float a;
    a = sqrt(x * x + y * y);
    return(a);
}
```

Example: Rewriting the Original Age Program

- Let’s rewrite the program that asked for name and age and then printed these items.
- There are two data items, both of which should be private: name and age.
- There are two procedures, both of which should be public: read and write.
#include <iostream>
using namespace std;

const int namelen = 12;

class oldguy {
    char name[namelen];
    int age;
    public:
        void read(void)
        {
            cout << "What's your name?";  
            cin >> name;
            cout << "How old are you?";  
            cin >> age;
        }

        void write(void)
        {
            cout << name << " is " << age  
                 << " years old." << endl;
        }
};

int main(void)
{
    oldguy me;
    me.read();
    me.write();
    return (0);
}
Putting **public** Before **private**

- It is better to place the public members of the class before the private member because these are the ones that we want programmers to think in term of.
- To ensure this, we place the keyword public at the top followed by the public members.
- Below the public members, we place the word private followed by the private members.
- We can write the body of the member function inside the declaration (if it’s short) or outside (where we must write `ClassName::` to indicate membership).

```cpp
#include <iostream>
using namespace std;

const int namelen = 12;

class oldguy {
public:
    void read(void);
    void write(void);

private:
    char name[namelen];
    int age;
};
```
void oldguy::read(void)
{
    cout << "What\'s your name\t?";
    cin >> name;
    cout << "How old are you\t?";
    cin >> age;
}

void oldguy::write(void)
{
    cout << name << " is " << age << " years old." << endl;
}

int main(void)
{
    oldguy me;
    me.read();
    me.write();
    return(0);
}
Member Functions and Parameters

- Functions belonging to a class can have parameters, including other objects of the same class or different class.
- If you pass as a parameter an object of the same class, you must use the name of the object when specifying its members.
- E.g., `y` is an item in this object, `q.y` is an item in object `q`.

```cpp
#include <iostream>
using namespace std;
const int namelen = 12;

// A class of object which contains name and age
class oldguy {
public:
    void read(void);
    void write(void);
    bool older(oldguy him);
    bool younger(oldguy him);
private:
    char name[namelen];
    int age;
};
```
// read() - Reads in name and age
void oldguy::read(void)
{
    cout << "What\'s your name\t?";
    cin >> name;
    cout << "How old are you\t?";
    cin >> age;
}
// write() - Writes name and age
void oldguy::write(void)
{
    cout << name << " is " << age
    << " years old." << endl;
}

// older() - Returns true if this guy is older
//           Returns false if this guy is younger
//           or the same age
bool oldguy::older(oldguy him)
{
    return(age > him.age);
}

// younger() - Returns true if this guy is
//              younger
//              Returns false if this guy is
//              older or the same age
bool oldguy::younger(oldguy him)
{
    return(age < him.age);
}
int main(void)
{
    oldguy me, him;

    me.read();
    him.read();

    if (me.older(him))
        cout << "I'm older." << endl;
    else if (me.younger(him))
        cout << "I'm younger." << endl;
    return(0);
}

Example: Complex Numbers

- Complex numbers are number of the type
  \[ w = x + iy \]
  where \( x \) and \( y \) are real and \( i \) is the square root of -1.

- We can define the operations addition, subtraction and multiplication.
Complex Number Operations

• If our two complex numbers are u and v:
  - If \( w = u + v \)
    • \( \text{Re } w = \text{Re } u + \text{Re } v \)
    • \( \text{Im } W = \text{Im } u + \text{Im } v \)
  - If \( w = u - v \)
    • \( \text{Re } w = \text{Re } u - \text{Re } v \)
    • \( \text{Im } W = \text{Im } u - \text{Im } v \)
  - If \( w = u \cdot v \)
    • \( \text{Re } w = \text{Re } u \cdot \text{Re } v - \text{Im } u \cdot \text{Im } v \)
    • \( \text{Im } w = \text{Re } u \cdot \text{Im } v - \text{Im } u \cdot \text{Re } v \)

complx1.cpp

#include <iostream>
using namespace std;

class complex {
  public:
    void read(void);
    void write(void);
    complex add(complex v);
    complex sub(complex v);
    complex mult(complex v);
  private:
    float real;
    float imag;
};
// read() - Read in a Complex value
void complex::read(void)
{
    cout << "Real\t?";
    cin >> real;
    cout << "Imaginary\t?";
    cin >> imag;
}

// write() - Write a complex value
void complex::write(void)
{
    cout << '(' << real << ', ' << imag << ')';
}

// add() - Returns the sum of this value + v
complex complex::add(complex v)
{
    complex w;
    w.real = real + v.real;
    w.imag = imag + v.imag;
    return(w);
}

// sub() - Returns the difference of
// this value - v
complex complex::sub(complex v)
{
    complex w;
    w.real = real - v.real;
    w.imag = imag - v.imag;
    return(w);
}
// Mult() - Returns the product of
//          this value times v
complex complex::mult(complex v)
{
    complex w;
    w.real = real * v.real - imag * v.imag;
    w.imag = real * v.imag + imag * v.real;
    return(w);
}

// ComplexDemo() - Demonstrate the complex class
int main(void)
{
    complex u, v, w;
    u.read();
    v.read();
    w = u.add(v);
    w.write();
    w = u.sub(v);
    w.write();
    w = u.mult(v);
    w.write();
    return(0);
}
Constructors

- Sometimes we need an object to have some initial values set when we define it. This can be done implicitly by writing a constructor.
- Constructors are called automatically when the program enters the function where the object is declared.
- Constructors share a name with the class and have no result type, not even void.

Default Constructors

- If an object is declared without any parameters, the default constructor is called.
- A default constructor has no parameters.
Conversion Constructors

• Thus, a conversion constructor initialize some or all of the values within the object.
• To use a conversion constructor, an object must be declared including (in parentheses) the initial values:

```
MyClass MyObject(2, "name");
```

```
#include <iostream>
using namespace std;
class complex {
public:
    complex(void);
    complex(float a, float b);
    complex(int a, int b);
    void read(void);
    void write(void);
    complex add(complex v);
    complex sub(complex v);
    complex mult(complex v);
};
```
private:
    float real;
    float imag;
};

// complex() - A Default Constructor
complex::complex(void)
{
    real = imag = 0.0;
}

// complex() - A Conversion Constructor
complex::complex(float a, float b)
{
    real = a;
    imag = b;
}

// complex() - A Conversion Constructor
complex::complex(int a, int b)
{
    real = (float) a;
    imag = (float) b;
}

// read() - Read in a Complex value
void complex::read(void)
{
    cout << "Real\t?";
    cin >> real;
    cout << "Imaginary\t?";
    cin >> imag;
}
// write() - Write a complex value
void complex::write(void)
{
    cout << '(' << real << ', ' << imag << ')';
}

// add() - Returns the sum of this value + v
complex complex::add(complex v)
{
    complex w;
    w.real = real + v.real;
    w.imag = imag + v.imag;
    return(w);
}

// sub() - Returns the difference of
// this value - v
complex complex::sub(complex v)
{
    complex w;
    w.real = real - v.real;
    w.imag = imag - v.imag;
    return(w);
}
// mult() - Returns the product of this value times v
complex complex::mult(complex v)
{
    complex w;
    w.real = real * v.real - imag * v.imag;
    w.imag = real * v.imag - imag * v.real;
    return(w);
}

// ComplexDemo() - Demonstrate the complex class
int main(void)
{
    complex u(1, 1), v, w;
    v.read();
    w = u.add(v);
    w.write();
    w = u.sub(v);
    w.write();
    w = u.mult(v);
    w.write();
    return(0);
}
Rewriting `average.cpp`

- Let’s rewrite `average` so that it is an object with the private data items (first and last name and four exam grades in an array) and three functions (`read`, `write` and `findaverage`).
- By writing two versions of `write` (one with the average, one without), we overload the function. It will choose the one matching the parameter list.

```
#include <iostream>

using namespace std;

const int namelen = 15, numexams = 4;

class student {
    public:
        void read(void);
        void write(void);
        void write(float average);
        float findaverage();
    private:
        char firstname[namelen],
        lastname[namelen];
        int exam[numexams];
};
```
// read() - Read the input about the student
void student::read(void)
{
    int i;
    cout << "First name\t?";
    cin >> firstname;
    cout << "Last name\t?";
    cin >> lastname;
    for (i = 0; i < numexams; i++) {
        cout << "Enter grade for exam #" << i+1 << "\t?";
        cin >> exam[i];
    }
}

// FindAverage() - Returns the average of n exam scores
float student::findaverage(void)
{
    int i, sum = 0;
    for (i = 0; i < numexams; i++)
        sum += exam[i];
    return((float) sum/numexams);
}
// WriteStudent() - Print the data about the
// student without the
// average
void student::write(void)
{
    int i;
    cout << firstname << ' ' << lastname
        << " scored : " << endl;
    for (i = 0; i < numexams; i++)
        cout << exam[i] << endl;
}

// WriteStudent() - Print the data about the
// student including the
// average
void student::write(float average)
{
    int i;
    cout << firstname << ' ' << lastname
        << " scored : " << endl;
    for (i = 0; i < numexams; i++)
        cout << exam[i] << 't';
    cout << "\n\twhich resulted in an average of "
        << average << endl;
}
// AvgGrade() - Averages the grades on n exams
int main(void)
{
    student s;
    float average;
    // Read the student's name and test scores
    s.read();
    // Find the average
    average = s.findaverage();
    // Print the results
    s.write(average);
    return(0);
}