CSCI 240 Lecture Notes - Part 10

Object-Oriented Programming

Object-Oriented Programming (**OOP**) is the next step in modern programming practice after functional programming (60's) and structured programming (70's).

OOP is complex. The first of three main principles of OOP is that of the class, which is the basis of OOP.

A **class** is a kind of a package that consists of functions and the data they work on. However, in OOP, functions are called **methods**, and the individual variables defined in a class are called **data members**.

An **object** is an instance of a class that contains data and methods.

So, a class is a template for the creation of multiple instances of that class, each of which is called an object.

The generic format for a class:

```
class class_name
{
method_prototype_1;
method_prototype_2;
...

data_type data_member_1;
data_type data_member_2;
...
};
```

A class for a bank account might resemble:

```
class Acct
{
//The class variables

string acctNum;
string acctName;
char acctType;
double acctBal;


//The class methods

Acct( string, string, char, double );      //the "constructor"

void printAcct();       //display the account information
};
```

The constructor is a special method that is used to supply initial values for the data members.

Before the class can be used, code must be written for the "constructor" and the printAcct() method:

```
Acct::Acct( string newAcctNum, string newAcctName, char newAcctType, double newAcctBal )
{
acctNum = newAcctNum;
acctName = newAcctName;
acctType = newAcctType;
acctBal = newAcctBal;
}


void Acct::printAcct()
{
cout << "Account Number: " << acctNum << "       Account Name: " << acctName << endl;

if( acctType == 'C' )
  {
  cout << "Account Type:   Checking";
  }
else
  {
  cout << "Account Type:   Savings";
  }

cout << "     Account Balance: " << acctBal << endl;
}
```

Now before we get to a program that uses these, a few notes:

The "constructor" is the method called when you create an instance of a class. Here, its arguments are stored into the data members of the object. So it is a kind of special initialization method.

The name of the constructor is always the same as the name of the class.

The notation of **Acct::** in front of the constructor and the method are used to specify what class the methods belong to, since the code for the methods are not defined inside the class definition. It would be possible to have some other class with a printAcct() method, for example. It would be defined as OtherClass::printAcct(). This way the compiler can tell which class a given method definition belongs to.

Now that the Acct class is defined, we can write a program that uses it. Just type in the class definition before int main() (the actual methods can go before or after main).

```
int main()
{
Acct account = Acct( "123456", "Mouse, Mighty", 'C', 1000000.00 );

account.printAcct();

return 0;
}
```

That's it. Clearly, there's some work up front to design and define the Acct class, but once that's done, the program becomes simpler.

Note that to add another Acct, we need 1 more variable, 1 initialization statement, and 1 (simple) method call to print it.

Notice the way the method is called - it's just like the structure dot notation - in this case, the **objectname.methodname(args if any).**

Also, notice that the call to printAcct() does not need any arguments! The Acct account "knows" its account number, account name, etc... (they are data members of account). So since the code in printAcct() (which is inside account) can "see" the data members (which are also inside account), no arguments are needed.

It's like main is telling the Acct account to print itself. In fact, in OOP, calling a method of an object is called "**sending a message to the object**". In this case, it's the "print yourself" message.

Another thing: each object of the Acct class has its own set of variables (data members). So calling the printAcct() method for different Acct objects will cause different information to be printed:

```
int main()
{
Acct account1 = Acct( "987654", "Amonte, Tony", 'S', 5000000.00);

Acct account2 = Acct( "321098", "Belfour, Ed", 'C', 6000000.25);

account1.printAcct();    //prints the account information for Amonte, Tony

account2.printAcct();    //prints the account information for Belfour, Ed

return 0;
}
```

Additional Methods for class Acct

Now that we have the Acct class, we can add functionality to it by adding some useful methods.

All that needs to be done is to add a prototype to the class definition and then code the new method.

Guarding the Data

One important advantage of OOP is that since the data for an object is kept inside that object, we can make sure that the data is always valid and consistent by making checks in the object's code.

For example, suppose we wanted to be sure that all of our Accts don't have negative balances. We could write the constructor so that we would initialize the salary values no matter what the passed-in values were:

```
Acct::Acct( string newAcctNum, string newAcctName, char newAcctType, double newAcctBal )
{
acctNum = newAcctNum;
acctName = newAcctName;
acctType = newAcctType;

if( newAcctBal < 0 )
  {
  acctBal = 0;
  }
else
  {
  acctBal = newAcctBal;
  }
}
```

Now it's impossible for an Acct to have a negative balance.

Access Methods and Private/Public

There is also another action that can be taken to guard the data. Typically, we do not want outside methods or functions to be able to access the individual data members directly. To guard against the access, we will make some class items **public** and others **private**.

Public data and methods are "visible" and "accessible" to code outside the object - that is, to code that is not in the object's methods.

Private data and methods are "invisible" and "inaccessible" to code outside the object. **This is the default access.**

The new definition for the class might resemble:

```
class Acct
{
private:
  string acctNum;
  string acctName;
  char acctType;
  double acctBal;

public:
  Acct( string, string, char, double );      //the "constructor"

  void printAcct();       //display the account information
};
```

However, it is possible that an outside method or function will need access to the private data members or methods. To grant the access...

We will write some public methods to allow other code access to the account number, account name, etc..., but we write the code to guard against invalid values.

A new version, with a couple of new public access methods follows:

```
class Acct
{
private:
  string acctNum;
  string acctName;
  char acctType;
  double acctBal;

public:
  Acct( string, string, char, double );      //the "constructor"

  void printAcct();       //display the account information

  void set_acctNum( string );
  void set_acctName( string );
  void set_acctType( char );
  void set_acctBal( double );
};


void Acct::set_acctNum( string newAcctNum )
{
acctNum = newAcctNum;
}


void Acct::set_acctName( string newAcctName )
{
acctName = newAcctName;
}


void Acct::set_acctType( char newAcctType )
{
if( newAcctType == 'C' || newAcctType == 'S' )
  {
  acctType = newAcctType;
  }
else
  {
  acctType = 'C';
  }
}


void Acct::set_acctBal( double newAcctBal )
{
if( newAcctBal < 0 )
  {
  acctBal = 0;
  }
else
  {
  acctBal = newAcctBal;
  }
}
```

Now main() (or any other function with access to an Acct object) can reset one or more of the data members - but never to an invalid value:

```
account1.set_acctType( 'C' );

account2.set_acctBal( 100.01 );
```

It's also possible that an outside method or function just wants to retrieve the value in one of the data members. This can be easily achieved by writing a set of *getXXXX* access methods that simply return the value in the specific data member:

```
double Acct::get_acctBal()
{
return acctBal;
}


char Acct::get_acctType()
{
return acctType;
}
```

You get the idea (hopefully). Now, to use them:

```
cout << account1.get_acctBal();


char type;

type = account1.get_acctType();
```

Most classes will have a set of getXXX and setXXX access methods.

The getXXX methods usually have no arguments and simply return the value of a particular data member.

The setXXX methods usually take one or more arguments and do some kind of validity or consistency checking of their argument(s) before assigning the argument's value to a data member. They may or may not print an error message, return a success/error code, or "throw an exception" (another error handling system in C++).

Method Overloading

Sometimes it's convenient to have different versions of a method that do the same thing. That is, several methods with the same name, but which take different arguments, and perform the same task.

For example, maybe the balance for the account object is stored as a string in a character array rather than in a float/double variable. As the class is currently written, the string cannot be passed to the set_acctBal method because the data types don't match.

However, if the set_acctBal method is overloaded, the string can be used.

```
void Acct::set_acctBal( const char newbalance[] )
{
double balance;

balance = atof( newbalance );  //convert the string to a double value

if( balance < 0 )
  {
  acctBal = 0;
  }
else
  {
  acctBal = balance;
  }
}
```

One of the most common types of overloading is to overload the constructor.

```
Acct::Acct()
{
acctNum = "";
acctName = "";

acctType = ' ';

acctBal = 0.00;
}
```

So in a program that uses Accts, you are now free to create Accts using either of the constructors:

```
Acct account5 = Acct( "147852", "Olesyk, Eddie", 'C', 65412.90 );

Acct account6 = Acct();
```

Note 1: keep in mind that account6 doesn't have meaningful data at this point.

Note 2: anytime you write any new method, you must write the prototype of the method in the class definition.

The rule for creating different methods with the same name is that the ***number***, ***order***, or ***data type*** of the arguments must be different in each case (so that the compiler can tell which one to call). The return value has nothing to do with it. So you could have the following:

```
void dog(int, int)
int dog(int, double);
void dog(double, int);
double dog(double);
etc.
```

Here's another example. Suppose you have a private int data member in an object called num, and you want to write a setNum() method. That's easy:

```
void ClassType::setNum( int i )
{
num = i;   // you might have some "guarding" code also
}
```

Now you also realize that sometimes the calling program will have the numeric value they want to use stored in a double. So, for the convenience of the calling program, you can write:

```
void ClassType::setNum( double i )
{
num = (int) i;
}
```

The informal rule is that functions with the same name should do the same thing. You can sometimes see the following:

```
int length();
void length(int);
```

The first is used to return (i.e. get) the length of something:

```
int len = obj.length();
```

The second is used to set the length of that thing.

```
obj.length( 10 );
```

Although they both have something to do with length, they do very different things, so this is not a good use of this feature.