The University Of Michigan

EECS402 Lecture 06

Andrew M. Morgan

Savitch Ch. 6
Intro To OOP
Classes
Objects
ADTs

A Little History

Complexity Increases

- Early computing
  - Cheap Programmers, Expensive Computers
  - Binary instructions (fast, complex) written by programmers
- Later, as complexity increases
  - Programmers get more expensive
  - Assembly language - computer translated symbols (cost effective)
- Programs continue to get more complex, computers cheaper
  - Introduction of high level languages (Fortran)
  - Computers do more work, as they get cheaper
- Need for clearer, easy-to-understand programs
  - Introduction of structured programming languages (C)

Pattern Continues

EECS 402

- Program complexity continues to increase
- C can no longer handle complexity satisfactorily
- Programs of 50 KLOC are considered too complex to grasp as a totality
- Programmers need a new paradigm to handle new complexity

Object Oriented Paradigm

EECS 402

- New paradigm: Object Oriented Programming
- C++: Developed by Bjarne Stroustrup, in 1980
- C++ was developed as an extension of C
  - Superset of C to provide object oriented capabilities
- C++ aims to enable larger, more complex programs to be:
  - Better organized
  - Easier to comprehend
  - Easier and better managed
- Stroustrup says C++ allows "programs to be structured for clarity, extensibility, and ease of maintenance, without loss of efficiency."

Why OOP Helps Complexity

- The world consists of objects and actions
- Programmers are object-oriented beings
  - Programmers want to "program like we think"
- Programs become a collection of objects and how they act
  - No longer just a "set of instructions"
  - Programs are easier to think about as "chunks"
- Can program to an interface, even if implementation is incomplete
  - Different developers can develop functionality associated with the objects that will be used in the program

OOP Properties

- There are properties that a language must provide to be considered an object-oriented language
- These are properties that are not implicitly provided by languages such as C, Pascal, etc.
- Languages that are OO languages: C++, Java, etc.
- Three OOP properties are:
  - Encapsulation
  - Inheritance
  - Polymorphism

Encapsulation

- Definition: Group data and functionality together
- The C language used structs to group data together - why not group functionality along with it?
- Allows a programmer to explicitly provide the interface to an object
- Allows hiding of implementation details
- Allows programmer to think in an OO way
  - The world consists of objects that do things
  - Programs become a collection of objects and how they act, instead of a set of instructions

Inheritance

- Another "real world" property
- Definitions: Allows one data type (class) to acquire properties of other data types (classes)
- Allows a hierarchical structure of data types
- Is an apple edible?
  - An apple is fruit
  - Fruit is food
  - Food is edible
  - Therefore, an apple is edible

Polymorphism

- Another "real world" property
- Definition: Allows one common interface for many implementations
- Allows objects to act different under different circumstances
- Example:
  - Steering wheel - learn how to use one, know how to use them all
  - Steering mechanism (power steering, manual steering, some new form of steering mechanism) does not matter when using the steering wheel.

Grouping Data Together - struct

- In C/C++, different pieces of data can be grouped together
- A “structure” is such a grouping
  - Data which is different, but related in that each attribute describes one item, is often put into a structure
  - Data need not be of the same data type

```
struct circle
{
  int xLoc;
  int yLoc;
  int zLoc;
  double radius;
};
```

This creates a new data type

The data type is called “circle” and groups together different data, all of which are attributes that describe a circle

ANY circle has its own center (x,y,z) and a radius

*Note the semi-colon after the closing brace. It is required syntax.

Accessing Data In structs

EECS 402

- New variables can be declared of struct types
- The dot operator (.) is used to access individual elements

```
int main()
{
  circle circ1;
  circle circ2;

  circ1.xLoc = 5;
  circ1.yLoc = 0;
  circ1.zLoc = 15;
  circ1.radius = 5.5;
  circ2.xLoc = 6;
  circ2.yLoc = -5;
  circ2.zLoc = 10;
  circ2.radius = 3.2;

  return 0;
}
```

Memory associated with circ1
Memory associated with circ2

1000	xLoc	1012	radius	1024	zLoc
1001	5	1013	5.5	1025	10
1002		1014		1026	
1003		1015		1027	
1004	yLoc	1016	xLoc	1028	Radius
1005	0	1017	6	1029	3.2
1006		1018		1030	
1007		1019		1031	
1008	zLoc	1020	yLoc	1032	
1009	15	1021	-5	1033	
1010		1022		1034	
1011		1023		1035	

OOP Basic Building Blocks

EECS 402

- The C++ Class
  - Similar to a struct
  - Defines what objects of a class are (what attributes describe them)
  - Usually contains functionality in addition to attributes
- The Object
  - An *instance* of a class
  - A variable declared to be of a "class type"
- Class Vs Object
  - Classes do not have memory associated with them
    - A class is only a definition – i.e. a data type
  - Objects do have memory associated with them
    - Like structure variables
    - Each object has its own set of attributes in memory

The First C++ Class, p.1

```
class IntClass
{
  public:
    int val;  //Member variable/attribute

    //Member function/method.  Is able to
    //access attributes of the class.
    void add(int addVal)
    {
      val = val + addVal;
    }
};
```

In the main function:

```
  IntClass intObject1;
  IntClass intObject2;
  int      intVar;          <-----
  intObject1.val = 6;
  intObject2.val = 14;
  intVar = 4;
  intObject1.add(intVar);
  intObject2.add(intVar);
  cout << "1. Value is: " << intObject1.val << endl;
  cout << "2. Value is: " << intObject2.val << endl;
```

Memory assoc. with intObject1
Memory assoc. with intObejct2
Memory assoc. with other vars

1000	val	1012	
1001		1013	
1002		1014	
1003		1015	
1004	val	1016	
1005		1017	
1006		1018	
1007		1019	
1008	intVar	1020	
1009		1021	
1010		1022	
1011		1023	

* Memory that is accessible from current statement

The First C++ Class, p.2

```
class IntClass
{
  public:
    int val;  //Member variable/attribute

    //Member function/method.  Is able to
    //access attributes of the class.
    void add(int addVal)
    {
      val = val + addVal;
    }
};
```

In the main function:

```
  IntClass intObject1;
  IntClass intObject2;
  int      intVar;
  intObject1.val = 6;       <-----
  intObject2.val = 14;
  intVar = 4;
  intObject1.add(intVar);
  intObject2.add(intVar);
  cout << "1. Value is: " << intObject1.val << endl;
  cout << "2. Value is: " << intObject2.val << endl;
```

Memory assoc. with intObject1
Memory assoc. with intObejct2
Memory assoc. with other vars

1000	val	1012	
1001	6	1013	
1002		1014	
1003		1015	
1004	val	1016	
1005		1017	
1006		1018	
1007		1019	
1008	intVar	1020	
1009		1021	
1010		1022	
1011		1023	

* Memory that is accessible from current statement

The First C++ Class, p.3


```
class IntClass
{
  public:
    int val;  //Member variable/attribute

    //Member function/method.  Is able to
    //access attributes of the class.
    void add(int addVal)
    {
      val = val + addVal;
    }
};
```

In the main function:

```
  IntClass intObject1;
  IntClass intObject2;
  int      intVar;
  intObject1.val = 6;
  intObject2.val = 14;
  intVar = 4;
  intObject1.add(intVar);
  intObject2.add(intVar);
  cout << "1. Value is: " << intObject1.val << endl;
  cout << "2. Value is: " << intObject2.val << endl;
```

Memory assoc. with intObject1
Memory assoc. with intObejct2
Memory assoc. with other vars

1000	val	1012	
1001	6	1013	
1002		1014	
1003		1015	
1004	val	1016	
1005	14	1017	
1006		1018	
1007		1019	
1008	intVar	1020	
1009		1021	
1010		1022	
1011		1023	

* Memory that is accessible from current statement

The First C++ Class, p.4


```
class IntClass
{
  public:
    int val;  //Member variable/attribute

    //Member function/method.  Is able to
    //access attributes of the class.
    void add(int addVal)
    {
      val = val + addVal;
    }
};
```

In the main function:

```
  IntClass intObject1;
  IntClass intObject2;
  int      intVar;
  intObject1.val = 6;
  intObject2.val = 14;
  intVar = 4;
  intObject1.add(intVar);
  intObject2.add(intVar);
  cout << "1. Value is: " << intObject1.val << endl;
  cout << "2. Value is: " << intObject2.val << endl;
```

Memory assoc. with intObject1
Memory assoc. with intObejct2
Memory assoc. with other vars

1000	val	1012	
1001	6	1013	
1002		1014	
1003		1015	
1004	val	1016	
1005	14	1017	
1006		1018	
1007		1019	
1008	intVar	1020	
1009	4	1021	
1010		1022	
1011		1023	

* Memory that is accessible from current statement

The First C++ Class, p.5


```
class IntClass
{
  public:
    int val;  //Member variable/attribute

    //Member function/method.  Is able to
    //access attributes of the class.
    void add(int addVal)
    {
      val = val + addVal;
    }
};
```

In the main function:

```
  IntClass intObject1;
  IntClass intObject2;
  int      intVar;
  intObject1.val = 6;
  intObject2.val = 14;
  intVar = 4;
  intObject1.add(intVar);
  intObject2.add(intVar);
  cout << "1. Value is: " << intObject1.val << endl;
  cout << "2. Value is: " << intObject2.val << endl;
```

Memory assoc. with intObject1
Memory assoc. with intObejct2
Memory assoc. with other vars

1000	val	1012	addVal
1001	6	1013	4
1002		1014	
1003		1015	
1004	val	1016	
1005	14	1017	
1006		1018	
1007		1019	
1008	intVar	1020	
1009	4	1021	
1010		1022	
1011		1023	

* Memory that is accessible from current statement

The First C++ Class, p.6


```
class IntClass
{
  public:
    int val;  //Member variable/attribute

    //Member function/method.  Is able to
    //access attributes of the class.
    void add(int addVal)
    {
      val = val + addVal;
    }
};
```

In the main function:

```
  IntClass intObject1;
  IntClass intObject2;
  int      intVar;
  intObject1.val = 6;
  intObject2.val = 14;
  intVar = 4;
  intObject1.add(intVar);
  intObject2.add(intVar);
  cout << "1. Value is: " << intObject1.val << endl;
  cout << "2. Value is: " << intObject2.val << endl;
```

Memory assoc. with intObject1
Memory assoc. with intObejct2
Memory assoc. with other vars

1000	val	1012	addVal
1001	10	1013	4
1002		1014	
1003		1015	
1004	val	1016	
1005	14	1017	
1006		1018	
1007		1019	
1008	intVar	1020	
1009	4	1021	
1010		1022	
1011		1023	

* Memory that is accessible from current statement

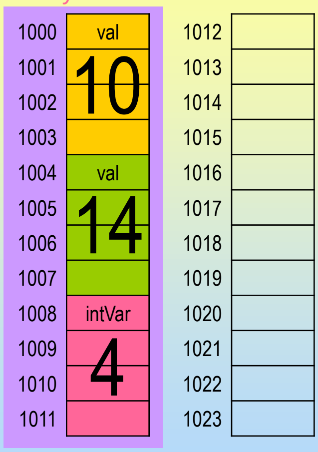
The First C++ Class, p.7

```
class IntClass
{
  public:
    int val;  //Member variable/attribute

    //Member function/method.  Is able to
    //access attributes of the class.
    void add(int addVal)
    {
      val = val + addVal;
    }
};
```

In the main function:

```
  IntClass intObject1;
  IntClass intObject2;
  int      intVar;
  intObject1.val = 6;
  intObject2.val = 14;
  intVar = 4;
  intObject1.add(intVar);
  intObject2.add(intVar);
  cout << "1. Value is: " << intObject1.val << endl;
  cout << "2. Value is: " << intObject2.val << endl;
```

Memory assoc. with intObject1
Memory assoc. with intObejct2
Memory assoc. with other vars



* Memory that is accessible from current statement

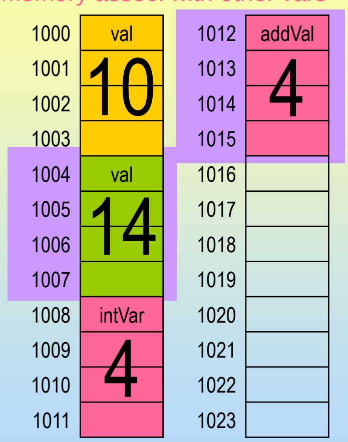
The First C++ Class, p.8

```
class IntClass
{
  public:
    int val;  //Member variable/attribute

    //Member function/method.  Is able to
    //access attributes of the class.
    void add(int addVal)
    {
      val = val + addVal;
    }
};
```

In the main function:

```
  IntClass intObject1;
  IntClass intObject2;
  int      intVar;
  intObject1.val = 6;
  intObject2.val = 14;
  intVar = 4;
  intObject1.add(intVar);
  intObject2.add(intVar);
  cout << "1. Value is: " << intObject1.val << endl;
  cout << "2. Value is: " << intObject2.val << endl;
```

Memory assoc. with intObject1
Memory assoc. with intObejct2
Memory assoc. with other vars



* Memory that is accessible from current statement

The First C++ Class, p.9

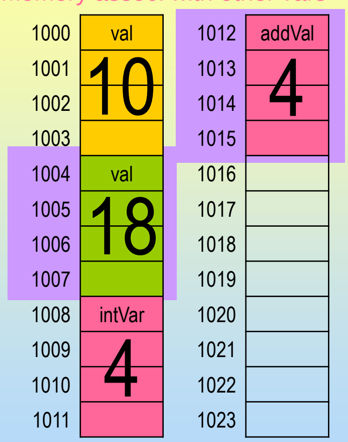
EECS 402

```
class IntClass
{
  public:
    int val;  //Member variable/attribute

    //Member function/method.  Is able to
    //access attributes of the class.
    void add(int addVal)
    {
      val = val + addVal;
    }
};
```

In the main function:

```
  IntClass intObject1;
  IntClass intObject2;
  int      intVar;
  intObject1.val = 6;
  intObject2.val = 14;
  intVar = 4;
  intObject1.add(intVar);
  intObject2.add(intVar);
  cout << "1. Value is: " << intObject1.val << endl;
  cout << "2. Value is: " << intObject2.val << endl;
```

Memory assoc. with intObject1
Memory assoc. with intObejct2
Memory assoc. with other vars



* Memory that is accessible from current statement

The First C++ Class, p.10

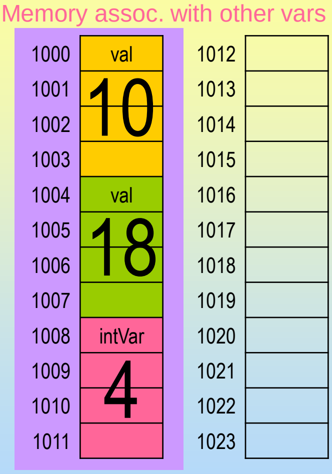
EECS 402

```
class IntClass
{
  public:
    int val;  //Member variable/attribute

    //Member function/method.  Is able to
    //access attributes of the class.
    void add(int addVal)
    {
      val = val + addVal;
    }
};
```

In the main function:

```
  IntClass intObject1;
  IntClass intObject2;
  int      intVar;
  intObject1.val = 6;
  intObject2.val = 14;
  intVar = 4;
  intObject1.add(intVar);
  intObject2.add(intVar);
  cout << "1. Value is: " << intObject1.val << endl;
  cout << "2. Value is: " << intObject2.val << endl;
```

Memory assoc. with intObject1
Memory assoc. with intObejct2
Memory assoc. with other vars



* Memory that is accessible from current statement

The First C++ Class, p.11

EECS 402

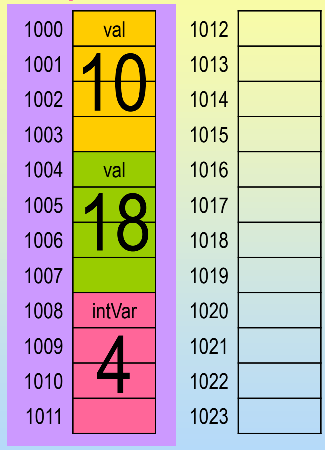
```
class IntClass
{
  public:
    int val;  //Member variable/attribute

    //Member function/method.  Is able to
    //access attributes of the class.
    void add(int addVal)
    {
      val = val + addVal;
    }
};
```

In the main function:

```
  IntClass intObject1;
  IntClass intObject2;
  int      intVar;
  intObject1.val = 6;
  intObject2.val = 14;
  intVar = 4;
  intObject1.add(intVar);
  intObject2.add(intVar);
  cout << "1. Value is: " << intObject1.val << endl;
  cout << "2. Value is: " << intObject2.val << endl;
```

1. Value is: 10
2. Value is: 18

Memory assoc. with intObject1
Memory assoc. with intObejct2
Memory assoc. with other vars



* Memory that is accessible from current statement

Common "Roles"

EECS 402

- Programmer: The person who implements and tests a class (or a function, etc.)
  - The "Programmer" may be specifically responsible for his/her class (or function, etc)
  - Develops component modules and classes that others could use when needed
- User: The person who is implements a program that is meant to be used to solve a certain problem.
  - To write the program, the "User" may use classes and/or functions written by the "Programmers".
  - Notice that this role is distinct from the "End User" (described next)
- End User: The person who executes the program written by the "User" in order to solve the certain problem.
  - This person is often not a coder at all, but is someone who utilizes pre-built programs from others – sometimes this person is the "customer"

- Note: In an academic setting like this, a student often plays all of these roles simultaneously
  - That can make it hard to understand the differences of the roles
  - Much of software development is done the way it is to separate these roles though, so its worth thinking about

Intro To Scope Resolution

- Previous example: class included interface and implementation to member function
- One advantage of encapsulation is the ability to just provide an interface to the class
- Put prototypes in class definition, put function implementation elsewhere
  - Accomplish this using the scope resolution operator, ::
- Allows you to bind the implementation of a function to a class
  - Differentiates it from a global function
- Often read as "belongs to"

Use Of Scope Resolution


```
class IntClass
{
  public:
    int val;

    void add(int addVal);
};
```

The add function is still a member function, but only the prototype is provided in the class

```
void IntClass::add(int addVal)
{
  val = val + addVal;
}
```

This is the implementation of the add function that "belongs to" the class IntClass. In other words, the definition is for IntClass' member function called add.

Implementation is outside of class definition, use scope resolution is required.

Methods And Object Modification

- Many methods are developed specifically to modify the state of the object they are operating on
  - For example, "myCircleObj.setRadius(10.0);"
    - Will change the state of myCircleObj by changing its radius from its current value to the value 10
- Some methods are not expected to modify the object though
  - For example, "myCircleObj.printAttributes();"
    - Will print the object's attributes to the screen, but would not be expected to change "myCircleObj" in ANY way
  - Can enforce this by specifying the method to be a "const" method!
- Examples:

```
void CircleClass::setRadius(const double inRadius)
{
  radiusAttr = inRadius;
}

void CircleClass::printAttributes() const
{
  cout << "Radius: " << radiusAttr << endl;
}
```

This "const" means the function will not change the value of the parameter you pass in in any way

This "const" means the function will not change the state of the object it is operating on in any way

Access To Member Variables

- Class IntClass contained the keyword public
- Member variables and functions may also be kept "private"
- Private member variables can ***only*** be accessed by member functions of the class to which they belong
- Private member functions can ***only*** be called by member functions of the class to which they belong
- In an object-oriented sense, when you want to change a member variable, you should always do so using a member function from the interface of the class
  - Having the member variables be private ensures this restriction

Example Class With Private - Definition


```
class AccessClass
{
  public:
    //Set the attribute "intAttr", enforcing rule that intAttr
    //must always be greater than 20.
    void setInt(const int inVal);

    //Return the value of the "intAttr" attribute
    int getInt() const;

  private:
    int intAttr;
};
```

All AccessClass member functions can access the private data member "intAttr" since they are member functions of the class that intAttr is a member variable of.

intAttr can not be accessed from within any function that is not a member function of AccessClass, however.

Example Class With Private - Implementation


```
//Set the attribute "intAttr", enforcing rule that intAttr
//must always be greater than 20.
void AccessClass::setInt(const int inVal)
{
  if (inVal > 20)
  {
    intAttr = inVal;
  }
  else
  {
    cout << "Val out of range!" << endl;
  }
}

//Return the value of the "intAttr" attribute
int AccessClass::getInt() const
{
  return intAttr;
}
```

Using The AccessClass


```
//This function is a "global function" - not a member of
//the class AccessClass
void printACInt(const AccessClass acParam)
{
  //cout << acParam.intAttr << endl; //ILLEGAL! intAttr is private!
  cout << acParam.getInt() << endl;  //Have to use public interface!
}

int main(void)
{
  AccessClass acObj;

  //acObj.intAttr = 18; //ILLEGAL - again intAttr is private!
  acObj.setInt(18);     //Use the interface to set intAttr to 18
  printACInt(acObj); //Since 18 is not a valid value, the
                     //member variable is not updated
  acObj.setInt(22);  //22 is in range, so intAttr will be set
  printACInt(acObj);

  return 0;
}
```

```
Val out of range!
0
22
```


Abstract Data Type

- Data Type: A collection of values *and* the operations that can be performed on those values
- Abstract Data Type (ADT): A data type, which has its implementation details hidden from the programmer using it
  - Programmer using ADT may not know what algorithms were used to implement the functions making up the interface of the ADT
  - All that really matters is that, when a member function from the interface is called, it results in the expected result
- In C++, developing classes that have their member function implementations hidden outside the class definition results in an ADT
- Programmer is provided with the member function prototypes (interface), but not the implementations

Example Of An ADT

- Consider the following ADT:

```
class RemoteControlledCarClass
{
  public:
    //Turns the car "numDegrees" to the right
    void turnRight(int numDegrees);

    //Turns the car "numDegrees" to the left
    void turnLeft(int numDegrees);

    //Sets the car's speed to newSpeed, as long as newSpeed
    //is not out of range of the car's capabilities
    void changeSpeed(int newSpeed);

    ...  //More functions as necessary
};
```

- If given this ADT and asked to write a program to steer a car through a maze in a set amount of time, this is all you would need
  - Details of *how* the car manages to turn or accelerate are unimportant, as long as when you call the functions, it does what it is supposed to