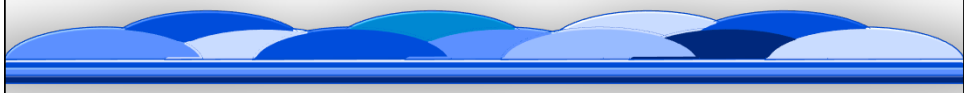
Topic 13 – Object Oriented Programming (OOP)



Different Styles of Programming

- Unstructured
- Structured
  - Procedural
  - Modular
- Object Oriented

Unstructured Programming

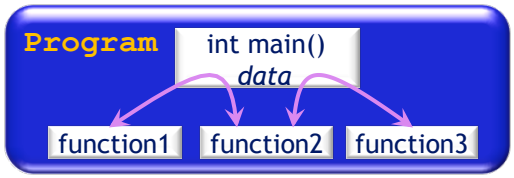
- Everything is in Main

- Data and code all in one file

Downside: Repetitive code
Reuse is difficult

Structured Programming

- Procedural Programming

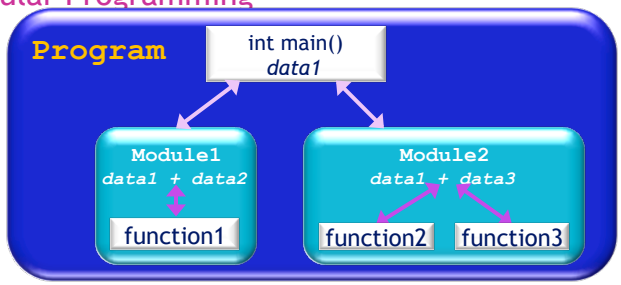


- Program is viewed as a series of procedure calls
  - Main is responsible for passing data to procedures
  - Procedures are responsible for processing data
    - **Procedures (a.k.a. functions, subprograms)**
- Eliminates the need to replicate code
- Easier to debug (isolates segments of code)
- Once they are correct they can be reused
  - If dependence on the main is reduced

Downside: To solve a general problem groups of procedures must be separately available.

Structured Programming

- Modular Programming

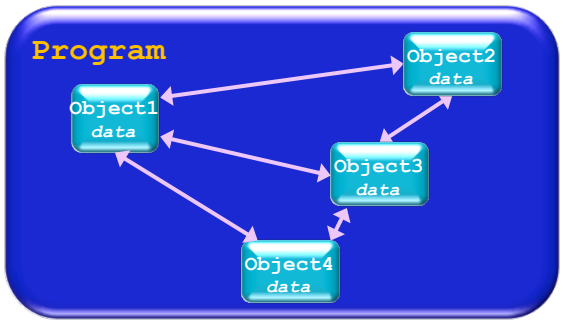


  - Procedures with common functionality are grouped together into separate *modules*
    - Main coordinates calls to procedures in separate modules
    - Each module can have its own data
    - Modules manage an internal state – data is modified by functions
  - Program divided into several smaller parts

Downside: To solve a general problem groups of procedures must be separately available.

Object Oriented Programming

- The structure is oriented around the data
  - Rather than the operations
  - Choose data representations which best fit your requirements
  - Each object manages its own data
  - Objects are (to an extent) self-contained



3 fundamental concepts in OOP

1 – Encapsulation

- *the grouping of related ideas into one unit, which can thereafter be referred to by a single name (Page-Jones)*
- In procedural programming we do this with functions
- In OOP it means, keeping the data and operations on the data together → we do this with objects

2 – Inheritance

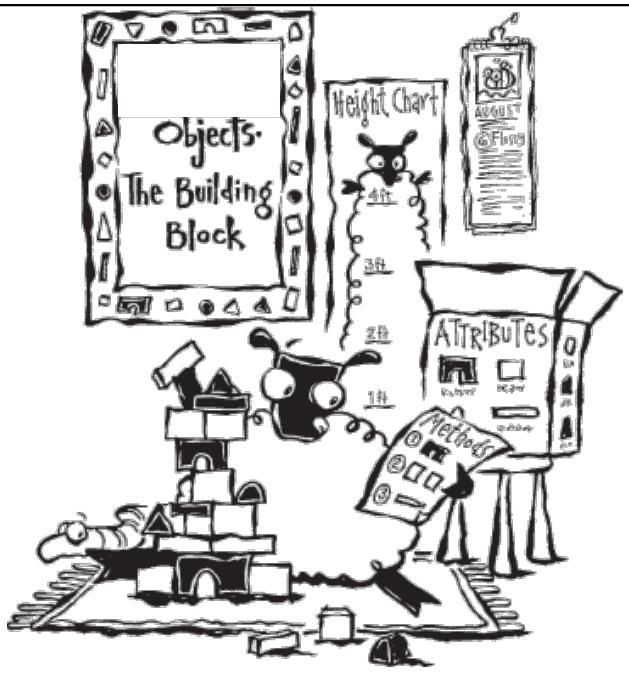
We don't view everything as unique – we view one thing as being *like* something else, but with differences or additions.

- In OOP *inheritance* means we can build upon an object/class that already exists and add or modify attributes of that object/class

3 – Polymorphism

- *polymorphism* means that some code or operations or objects behave differently in different contexts.

Using these 3 concepts we can improve the design, structure and reusability of code.



Objects

- Object-Oriented languages use *objects* as building blocks for code
- Why?
  - Reusability
  - Encourages good style
  - Objects can be portable
- Why reuse code?
  - Faster development
  - Code is already tested
  - Lower costs

What are objects?

- An object is often thought of as
  - Something of interest in the "real world"
- Objects consist of:
  - Attributes
    - Values that describe the object (identifiers)
    - Properties that characterize the object
    - States that the object might be in
  - Methods
    - Actions that can be performed on or be the object
    - Procedures or functions that perform *calculations* or *processing*

Object Types

- We define an object – that is its type
- We then can create instances of an object
- Object type corresponds to a group of object in the real world

For example

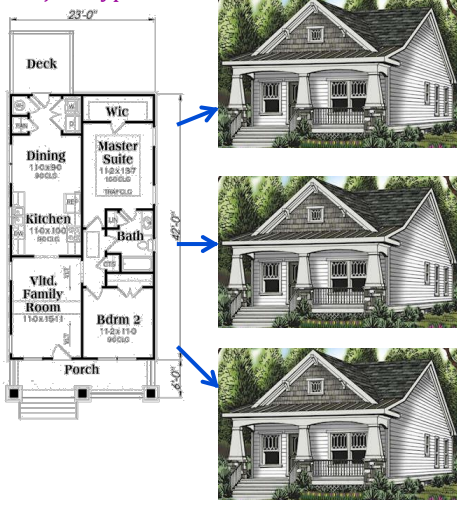
- Objects exists all around us
  - Books
  - People
  - Pens
  - Sheep

Think of an Object Type as a blueprint

- The object type describes the object in terms of attributes and actions

Object Type

Objects or Instances of the type



Exercise

- Observe the real-world objects that are in your immediate area!
- For each object that you see, ask yourself two questions:
  - "What possible *attributes* characterize this object?"
  - "What possible *methods* (or actions) are associated to this object?"
- Make sure to write down your observations
  - As you do, you'll notice that real-world objects vary in complexity
  - In addition, some objects may contain other objects

Attributes

Features that uniquely describe an object

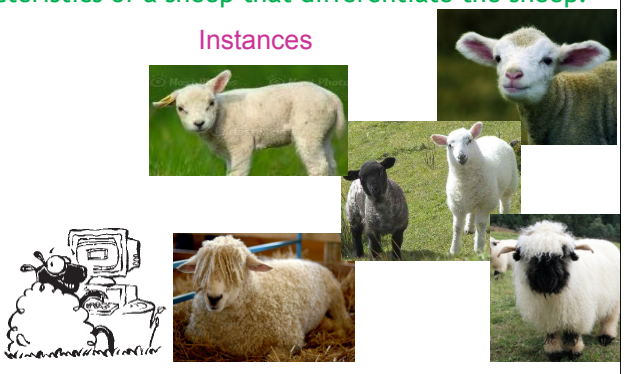
For example

We can have a sheep class

What are characteristics of a sheep that differentiate the sheep?

Attributes

Instances



Attributes for a sheep

- Each attribute will also have an affiliated type

ATTRIBUTE	DATATYPE

Date as a string has limitations
Format is fixed
Can't do calculations (such as age)

Methods

- It could also be an object
  - What would the attributes be?

Date

Attribute	Type
day	Integer
month	Integer
year	Integer

- Date is now stored in a simple neutral form
- Displaying it is an operation that uses the object (or the data)
  - This operation is closely liked to the object in a what it is also an attribute of an object
  - We call these *Methods*

Method
-A procedure that uses the attributes of an object to produce some useful result or effect

Methods

- Operations on attributes

For example:

What possible methods do you think a Date object type might be (identify what it would return and the return type as well)?

METHOD	RETURN VALUE	RETURN TYPE
Find the next date		
Find the day of the week		
Find the next date & change the values of the object's attributes to the values of the next date		
Is the date valid		
Display as DD/MM/YYYY		
Display as 2nd January 2001		

This list is incomplete – but a good start

Putting it Together

- Complete Definition of Date object Type

DATE OBJECT

ATTRIBUTE	TYPE
day	Integer
month	Integer
year	Integer

METHOD	RETURN VALUE	RETURN TYPE
Find the next date	The next date	Date
Find the day of the week	The day of the week	String
Find the next date & change the values of the object's attributes to the values of the next date	Nothing	Null
Is the date valid	True / False	Boolean
Display as DD/MM/YYYY	Nothing	Null
Display as 2nd January 2001	Nothing	Null

- A programmer can see what the data is (attributes) and what procedures/functions are available on these attributes (methods)

Object Oriented Programming

- First, identify the types of objects you will need
- Next, specify the object types (*class definition*)
  - Attributes and
  - Methods
- Then the programs use instances of these object types (called *objects*)
- Now these object types can be reused in many programs

LIBRARIES
-Groups of working classes that can be used over and over again in different programs

Objects in C++

- First, we must define our object types
  - In C++ we call them *classes*
- Defining a class is a two step process
  1 – Specify the class in a header file
  2 – Implement the class
- Then we can use it

Defining Classes

- Classes are defined in the header file
- Classes are defined by
  - Attributes (equivalent to variables)
  - Methods (equivalent to functions)
  - Interfaces
- The class definition provides an *interface* to the class
  - It describes…
    - what methods are available
    - What the methods do
    - What types of values they process and
    - What they produce

How interfaces can be used

- Once the interface is defined programmers can…
  - Implement the class
  - Implement code that uses this class
  - Both can happen in parallel
- If a programmer is using a class this is all they need
  - They don't need to know *how* it is implemented
  - The process of providing a public interface while hiding the implementation details is called *abstraction*
  - ADVANTAGE: If the programmer chooses to make changes to the code (for example, make it more efficient) it doesn't impact the objects using them

Public and Private Data

- The header file specifies data in two categories *public* and *private*
- *Public Data* is available to any program that uses the class
  - Anyone using this class sees this part
  - This provides the interface and should not be changed
  - Changing this means that all code using the class will have to be changed as well
- *Private Data* is available only to the class itself
  - The private part can be modified
  - Changing this won't effect the code using the class
  - This section allows for a powerful mechanism called *information hiding*
- Methods are typically defined in the Public part
- Attributes are defined in the Private Part

Information hiding is the ability of one program part to hide its data from other parts; thus avoiding improper addressing or name collisions of data.

Defining a class

Syntax

```
class className
{
        public:
            public parts go here
        private:
            private parts go here
};
```

- Attributes are defined similarly to variables but you can't initialize them in the declaration; and memory is not allocated
- Note: the data type can also be a class

Syntax

```
datatype attributeName;
```

```
class Sheep
{
        public:
                // methods go here

        private:
                string name; // the sheep's name
                int age;     // the sheep's age in years
                int x, y;    // the sheep's position in the field
};
```

Adding Methods

- The syntax for adding methods is similar to declaring functions

Syntax

```
returnDatatype MethodName (datatype name, datatype name) const;
```

Example:

```
bool GetPosition(int xCoord, int yCoord) const;
```

This indicates that the data values in the object won't be modified

There are two types of methods:

1. Mutators - functions that modify data
2. Accessors - query the object for data - but don't modify anything

Accessors have the word const after them as shown above

To access a public method, use the dot (.) operator

Syntax

```
variable = classObjectName.MethodName (datatype name, ...);
```

Mutators and Accessors

Which of the following are mutators and which are accessors?

```
int   GetAge ()                                    // return the sheep's age
void  SetName(string sheepName)                    // sets the sheep's name
float DistanceFrom (int xCoord, int yCoord)        // return distance from
                                                   //   a point
bool  GetPosition (int xCoord, int yCoord)         // true if sheep is at
                                                   //   position provided
void  ChangePosition(int xCoord, int yCoord)       // changes the sheep's
                                                   //   position
void  SetAge(int sheepAge)                         // sets the sheep's age
```

Constructors

- Next, we need a constructor
- A **constructor** is a special method that allows an instance of the class to be created
- The *default constructor* will have the same name as the class with no parameters

  For Example

  ```
  Sheep();
  ```

- We don't have to call this method explicitly
  - This happens automatically when we declare a variable (object) of our class type

    For Example

    ```
    Sheep flossy;
    ```

  - This would call the constructor to create an instance of the class - allocates memory for our variable (object) flossy

Destructor

- Lastly, we need a destructor
- A **destructor** is a special method that allows an instance of the class to be terminated
  - it can be very useful for releasing resources before coming out of the program like closing files, releasing memories, etc.
- The destructor will have the same name as the class prefixed with a tilde (~) with no parameters
  - For Example

```
~Sheep();
```

- We don't have to call this method explicitly
  - This happens automatically whenever an class object goes out of scope or whenever the delete expression is applied to a pointer to the class object

The Sheep Class

```
class Sheep
{
   public:
     Sheep ();                                        // constructor
     ~Sheep ();                                       // destructor
     void  SetAge(int sheepAge);                      // sets the sheep's age
     void  SetName(string sheepName);                 // sets the sheep's name
     void  ChangePosition(int xCoord, int yCoord);// changes the location
                                                      //    of the sheep
     int   GetAge () const;                           // return the sheep's age
     void  PrintNeatly () const;                      // print sheep details
     float DistanceFrom (int xCoord, int yCoord) const;// return distance
                                                       //   from a point
     bool  GetPosition (int xCoord, int yCoord) const; // true if sheep is
                                                       //   at position provided
   private:
     string name; // the sheep's name
     int    age;  // the sheep's age in years
     int    x, y; // the sheep's position in the field

};
```

Defining the Methods

Syntax
returnType ClassName::*MethodName*(*type parameterName…*)

Example:

```
void Sheep::SetAge(int sheepAge)
{
    age = sheepAge;
}

int Sheep::GetAge() const
{
    return age;
}
```

CONSTRUCTOR - you can initialize your attributes

```
Sheep::Sheep()
{
    age = 0;
    name.clear();
    x = 0;
    y = 0;
}
```

** These would go in a source file – be sure to include the header

Built-in Operations on Classes

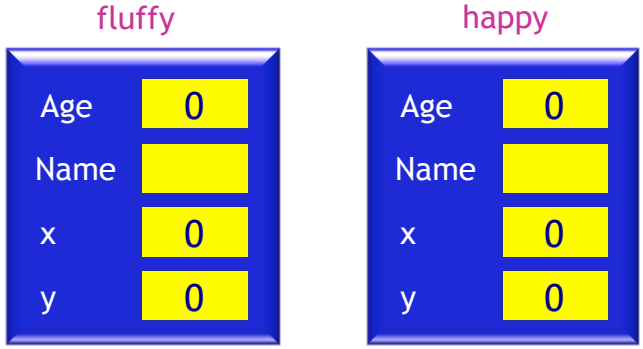
- Most of C++'s built-in operations do not apply to classes
  - Arithmetic operators cannot be used on class objects
  - Cannot use relational operators to compare two class objects for equality
- Built-in operations that are valid for class objects:
  - Method access (.)
    - You can access a public method from a class object
  - Assignment (=)
    - You can assign a class object to another class object (instance of the same class)

Declaring an Object

- We declare an object just like we would declare anything in C++
  - Sheep is our Class name and is our defined type
- Every time we declare an object:
  - The constructor method is called
  - An *instance* of our Class is created which we call an object

For Example:

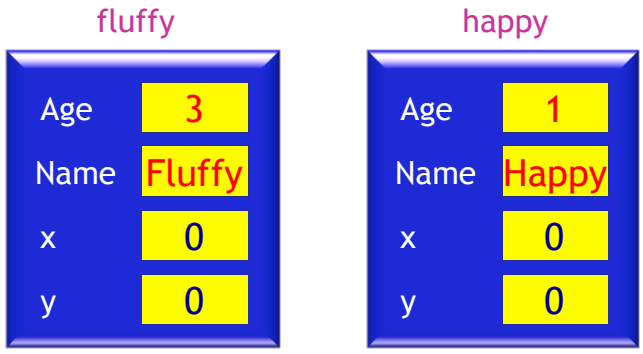
```
Sheep fluffy;  // This will create a sheep object we will refer to as "fluffy"
Sheep happy; // This will create a sheep object we will refer to as "happy"
```



Accessing Methods

Now we use the . operator to access the other methods

```
fluffy.SetAge(3);
happy.SetAge(1);
fluffy.SetName("Fluffy");
happy.SetName("Happy");
```



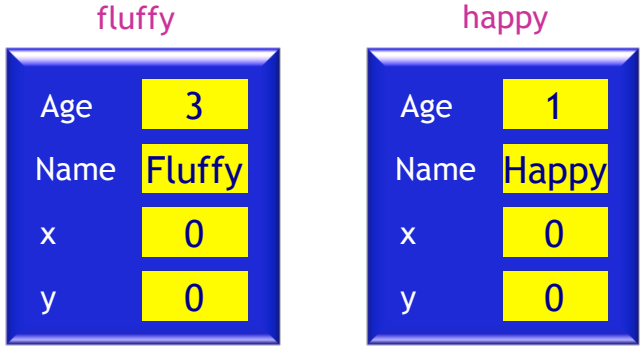
Using Value Returning Methods

Just like value returning functions

→ if it is a value returning method the value has to go somewhere

```
int fluffyAge;
int happyAge;

fluffyAge = fluffy.GetAge();    // This will return the value 3
happyAge = happy.GetAge();      // This will return the value 1
```

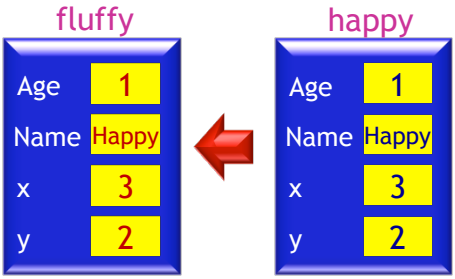


Assigning Objects

You can assign a class object into another class object

fluffy = happy;

After



More in Using Objects

You can have a Class use another Class type

```
private:
    Date birthday;
```

When you define a method for a class you can call that method just don't use the class name

When you define a method you can call another classes method

Exercise

- Given the sheep Class, write a program that creates two instances (class objects) from Sheep Class: black and white. Then perform the following tasks
  - Set black's name as "Black"
  - Input (from console) and set black sheep age and position
  - Set white's name as "White"
  - Input (from console) and set white sheep age and position
  - Output black and white details to the console
  - Output (to console) the distance between black and white sheep to farmer's house (coordinate=0,0)

The Sheep Class

```
class Sheep
{
   public:
     Sheep ();                                          // constructor
     ~Sheep ();                                         // destructor

     void  SetAge(int sheepAge);                       // sets the sheep's age
     void  SetName(string sheepName);                  // sets the sheep's name
     void  ChangePosition(int xCoord, int yCoord);     // changes the location of
                                                       //   the sheep
     int   GetAge () const;                             // return the sheep's age
     void  PrintNeatly () const;                        // print sheep details
     float DistanceFrom (int xCoord, int yCoord) const; // return distance from
                                                       //   a point
     bool  GetPosition (int xCoord, int yCoord) const;  // true if sheep is at
                                                       //   position provided
   private:
     string name;  // the sheep's name
     int    age;   // the sheep's age in years
     int    x, y; // the sheep's position in the field

};
```

```
int main()
{









return 0;
}
```

Documenting Classes

- Recall
  - Classes are defined in the header file
  - Classes are defined by
    - Attributes (equivalent to variables)
    - Methods (equivalent to functions)
    - Interfaces
  - The class definition provides an *interface* to the class
    - It describes…
      - **What methods are available**
      - **What the methods do**
      - **What types of values they process and**
      - **What they produce**

Components of a Class

- Attributes
  - They are equivalent as variables
  - So, we will document attributes as variables using a Data Table
    - For each attribute we will:
      - **State the use of the attribute**
      - **How its value is obtained/used**
- Methods
  - They are equivalent as functions
  - So, we will document methods similarly as functions

Documenting Class Definition

- Attributes
  - They will be document with a Data Table
- Methods
  - They will be grouped as:
    - Constructor and Destructor
    - Mutators
    - Accessors
  - They will be documented as functions prototypes but listed right after the class definition

```
class Sheep
{
   public:
      /*****************************
       ** CONSTRUCTOR & DESTRUCTOR **
       *****************************/

     Sheep ();                          // constructor
     ~Sheep ();                         // destructor

     /***************
     /** MUTATORS  **
      ***************/

     void  SetAge (int age);
     void  SetName(string name);
     void  ChangePosition(int xCoord,
                          int yCoord);

     /***************
      ** ACCESSORS **
      ***************/

     int   GetAge       () const;
     void  PrintNeatly  () const;
     float DistanceFrom (int xCoord,
                         int yCoord) const;
     bool  GetPosition  (int xCoord,
                         int yCoord) const;

   private:
     string name;                       // IN/OUT - the sheep's name
     int    age;                        // IN/OUT - the sheep's age in years
     int    x, y;                       // IN/OUT - the sheep's position in the field

};
```

The Sheep Class (documented)

```
/*****************************************************************
 * Sheep Class
 *   This class represents a sheep object. It manages 4 attributes,
 *   name, age and position (x and y coordinate)
 *****************************************************************/


/*****************************
 ** CONSTRUCTOR & DESTRUCTOR **
 *****************************/

/*****************************************************************
 * Sheep ();
 *   Constructor; Initialize class attributes
 *   Parameters: none
 *   Return: none
 *****************************************************************/

/*****************************************************************
 * ~Sheep ();
 *   Destructor; does not perform any specific function
 *   Parameters: none
 *   Return: none
 *****************************************************************/
```

```
 /**************
  ** MUTATORS **
  **************/

/*****************************************************************
 * void SetAge (int age);
 *
 *   Mutator; This method will update the age attribute to the
 *             parameter age value
 *---------------------------------------------------------------
 *   Parameter: age (integer) // IN - the age for the new attribute
 *---------------------------------------------------------------
 *   Return: none
 *****************************************************************/


…
 /***************
  ** ACCESSORS **
  ***************/
/*****************************************************************
 * int GetAge () const;
 *
 *   Accessor; This method will return the age attribute
 * ---------------------------------------------------------------
 *   Parameters: none
 * ---------------------------------------------------------------
 *   Return: age (integer)
```

Documenting Method Definition

- Methods are defined in a .cpp file and should be documented similarly as a function above the definition
- Documentation should include:
  - Method name
  - Class that method belongs to
  - Description
  - Pre-conditions including input parameters
  - Post-conditions including return value and any other values changed (by reference)

Sample Method Definition Documentation
Methods should be presented in the same order as the interface

```
/*********************************************************
 *
 * Method ChangePosition: Class Sheep
 *_______________________________________________________
 * This method receives a x and y coordinate representing
 *   a new position for a sheep object and update the sheep's
 *   position – returns nothing.
 *_______________________________________________________
 * PRE-CONDITIONS
 *   The following need previously defined values:
 *      xCoord: New sheep's x coordinate
 *      yCoord: New sheep's y coordinate
 *
 * POST-CONDITIONS
 *     This function will update x and y coordinate attribute
 *     in the sheep object. There is no return value.
 *     <Post-conditions are how the program execution is
 *      affected by this method -  Did it output something
 *      is it modifying reference parameters – does it return
 *      something? – state that here  >
*********************************************************/
 void Sheep::ChangePosition(int xCoord, // IN – New sheep's x coordinate
                            int yCoord} // IN – New sheep's y coordinate
{
```