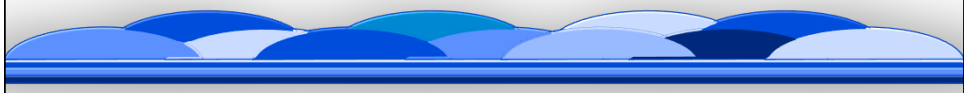
Topic 15 – Polymorphism



Overloading
Redefining
Hiding methods
Virtual functions
Overriding
Dynamic binding

Quick - Review

Suppose we have a class we've developed called Pet

For our pets we will track:

- name,
- age,
- weight,
- typeOfPet

We want to be able to…

- set all values,
- change age & weight
- retrieve name, age, weight
- display all attributes of our pet
- Output when our pet is speaking

What would the interface look like?

```
#include <string>
#include <iostream>
using namespace std;
class Pet
{    public:
          Pet();     // constructor
          ~Pet();    // destructor

          /******************
           **** MUTATORS ****
           ******************/
          void   SetValues(string name,
                           int    age,
                           float  weight,
                           string typeOfPet);
          void   ChangeAge   (int age);
          void   ChangeWeight(float  weight);

          /*******************
           **** ACCESSORS ****
           *******************/
          string GetName  () const;
          int    GetAge   () const;
          float  GetWeight() const;
          void   Display  () const;
          void   Speak    () const;
     protected:
          string petName;
          int    petAge;
          float  petWeight;
          string petType;
     private::
};
```

```
void Pet::Speak() const
{
     cout << "Pet::Speak\n\t"
          << petName << " is speaking..."
          << endl;
}
```

Polymorphism

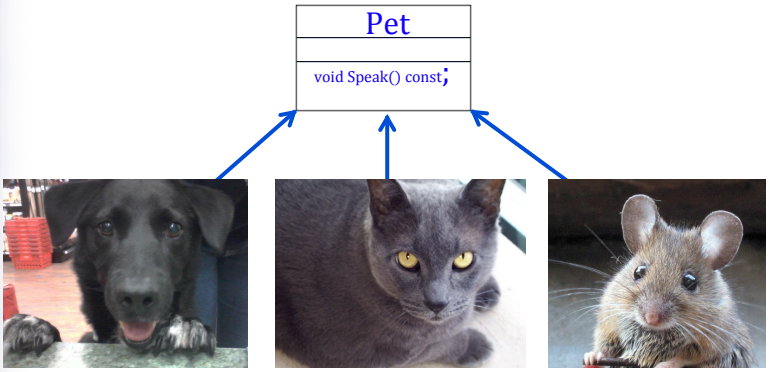
Poly = many

Morphism = forms

- Polymorphism refers to a way that a single name can have multiple meanings → in the context of inheritance
  - Ability to use the same expression to denote different operations or
  - provision of a single interface for entities of different types
- C++ supports different types of polymorphism
  - Ad-hoc → compile time (also known as overloading)
  - Parametric → compile time (using templates)
  - Subtype → run time (what is commonly known as polymorphism)

Polymorphism – Basic Idea

One method – many forms



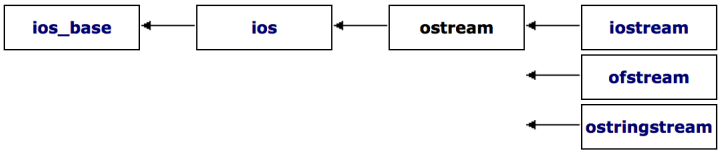
They all speak… but do they all speak the same language?

Need for Polymorphism

- Assume that you want to have a function to handle different data types
  - For instance, our validate input function – you want to input a variable (either int or float) and validate for the correct type and range
  - Now you have to create two complete different functions and remember each function name when call them
  - Is there a better way?
- Yes! Different types of polymorphism can help us make our code more re-usable

Example of Polymorphism Output Stream

- Output stream datatype can be used to represent different types of output objects such as files, console and output string



- We can use an ostream datatype to allow a function to output either to a file or to console (cout)

Polymorphism

- *Ad-hoc*

  An example of *function (or method) overloading* is a class with multiple Methods, all with the same name but different list of parameters
  - When a class object is created based on the number of parameters used, the correct constructor will be executed

- *Parametric Polymorphism*
  - It occurs when data types are left unspecified providing means to execute the same code for any data type
    - In C++ parametric polymorphism is implemented via templates, which allow functions and classes to operate with generic types

- *SubType Polymorphism: Virtual Function (or method)*
  - Used with an inheritance hierarchy
    - the appropriate method is selected during run-time execution of the code
      - **Based on the object instantiation**
    - implement by defining a method as *virtual* in the base class and redefining the same method in the derived classes

Overloading: Ad-Hoc Polymorphism

Overloading is when you have the same name for a method (or function), but different parameters

The method name is the same, but *signatures* are different

- A signature is what we've been calling the prototype (not including the return type)

```
float ValidateInput(const float MIN_VALUE, const float MAX_VALUE);
int   ValidateInput(const int MIN_VALUE, const int MAX_VALUE);
char  ValidateInput(const char VALID_CHARS[], const int NUM_CHARS);
```

The compiler can determine which one to execute based on the parameters being sent

Overloading

Constructor with Parameters

- So far we have learned about the class *default constructor* that has no parameters
- Any class can have other constructors with parameters - in this way we are overloading the constructor
  - They can be used to initialized the class attributes when the class object is created
  - In the Pet Class, we could create another constructor
    Pet(string name, int age, float weight, string breed);
  - And initialize a class object named bear as
    Pet bear("Bear", 2, 20.5, "Cat");
  - The constructor to be executed depends on the number and types of values passed when object is created

We can overload any method

Overloading is resolved at compile time

Overloading Methods - Example

This is Overloading -
Ad-Hoc polymorphism
2 methods →
- Same Method name
- Different parameters
- Different implementations
- Same class

```
// THIS IS OVERLOADING the constructor
Pet();
Pet(string name,
    int    age,
    float  weight,
    string breed);
Pet(string name);
~Pet();                    // destructor

/******************
 **** MUTATORS ****
 ******************/
// THIS IS OVERLOADING
void   SetValues(string name,
                 int    age,
                 float  weight);
void   SetValues(int    age,
                 float  weight);
// THIS IS OVERLOADING
void   ChangeValue(int    age);
void   ChangeValue(float  weight);
```

This is also Overloading

This is also Overloading

Implement & Call

```
// THIS IS OVERLOADING the constructor
Pet::Pet()
{
        petName.clear();
        petAge    = 0;
        petWeight = 0;
        petBreed.clear();
}
Pet::Pet(string name,
        int    age,
        float  weight,
        string breed)
{
        petName   = name;
        petAge    = age;
        petWeight = weight;
        petBreed  = breed;
}
Pet::Pet(string name)
{
        petName   = name;
        petAge    = 0;
        petWeight = 0;
        petBreed  = "Breed undefined!";
}
```

```
#include "OOP Header.h"
#include "pet.h"


int main()
{
    Pet  bear("Bear", 2, 20.2, "Cat");
    Pet  othello("Othello");
    Pet  rusty;

    •••
```

The compiler chooses the correct method based on the parameters

```
// THIS IS OVERLOADING the SetValues method
void Pet::SetValues(string name,
                    int    age,
                    float  weight)
{
        petName   = name;
        petAge    = age;
        petWeight = weight;
}
void Pet::SetValues(int    age,
                    float  weight)
{
        petAge    = age;
        petWeight = weight;
}

// THIS IS OVERLOADING ChangeValue

void Pet::ChangeValue(int age)
{
        petAge = age;
}
void Pet::ChangeValue(float weight)
{
        petWeight = weight;
}
```

```
int main()

{...
    float weight;
    int   age;

    rusty.SetValues("Rusty", 3, 25.0);
    othello.SetValues(4, 12.5);



    age = 5;
    rusty.ChangeValue(age);

    weight = 23.5;
    othello.ChangeValue(weight);

    ...
}
```

NOTE: for ChangeValue we have the same number of parameters just different types

Rest of code…. and output

```
int main()
{

    ...

    bear.Speak();

    cout << endl << endl;

    bear.Display();
    cout << endl;

    othello.Display();
    cout << endl;

    rusty.Display();
    cout << endl;

    ….
    return 0;
}
```

OUTPUT

```
Bear is speaking...

Bear     2    20.20    Cat
Othello  4    23.50    Breed undefined!
Rusty    5    25.00
```

We can overload operators too

- It occurs when the function is called and the operator is evaluated according to the arguments used.

- For instance, the '+' operator

```
4 + 5          <-- integer addition
3.14 + 2.0     <-- floating point addition
s1 + "bar"     <-- string concatenation
```

Redefining

Redefining occurs when you are using inheritance

- you create a method in a *derived class* with the *same name* and *same signature* a method in the base class

Example

- Let's say we have a Dog class and a Cat class that inherit from the Pet class
- We may want to implement speak for dog and cat differently
- So the signature is the same, but the implementation is different

This is redefining
2 methods: on in Base class one in Derived →
Same Method name
Same Method parameters
Different implementations
Different classes (BASE & DERIVED)

All interfaces will have this method

```
void Speak() const;
```

IMPLEMENTATIONS

```
void Pet::Speak() const
{
   cout << petName << " is speaking...";
}
```

```
void Dog::Speak() const
{
   cout << petName << " says: Woof Woof!";
}
```

```
void Cat::Speak() const
{
   cout << petName << " says: Meoowww!";
}
```

Hiding

Hiding a method occurs in two cases

1. when you redefine an overloaded method, but you don't redefine all versions.
   - This is problematic because if you try to use one of the overloaded methods in the base class and you've created an instance of an object from the derived class you will get a compile-time error

Example

OVERLOADED in the BASE CLASS Pet

```
void   SetValues(string name,
                 int    age,
                 float  weight);

void   SetValues(int    age,
                 float  weight);
```

REDEFINED in the DERIVED CLASS Dog

```
void   SetValues(string breed);
```

Note the different signatures

HIDING IS PROBLEMATIC
Objects of type Dog can't access the overloaded methods Pet
Dog buddy;
buddy.SetValues("Buddy", 6, 28.4);
Will result in a compiler error!

Hiding – Case #2

- Parameters are the same, but the signatures are still different.

OVERLOADED in the BASE CLASS Pet

```
void GetValues() const;

void GetValues(bool justName) const;
```

REDEFINED in the DERIVED CLASS Dog

```
void GetValues();
void GetValues(bool justName);
```

What happened here?

Summary
Overloading vs. Redefining

- Overloading
  - Same Method Name
  - Same class
  - Different Parameter Signatures
- Redefining
  - Same Method Name
  - Different classes
  - Same Parameter Signatures

Which Method to bind to?

How does it know which method to invoke?

- Overloading – the compiler knows which method to bind to because the signatures are different
- Redefining the signatures are the same
  - The compiler knows which to bind to tell by how you declare the object

Templates: Parametric Polymorphism

It occurs when data types are left unspecified providing means to execute the same code for any data type

There are two types of Templates

- Function Template
  - single code body for a set of related functions
- Class Template
  - single code body for a set of related classes

The compiler is allowed to generate multiple versions of a class (or function) due to the parameterized data types

- One version for each actual data type used (e.g., int, float, etc.)

Function Template

Syntax
template <class Type>
function definition and function prototype

- Type is formal parameter for the template (as variables are parameters for the function)
  - It can represent any user-defined type or built-in type
- Type can be used to:
  - Specify type of parameters to the function
  - Specify return type of the function
  - Declare variables within the function
- Type will be resolved during compiler time based on the actual use of the function

Using Function Template

```
template <class Type>
Type larger (Type x, Type y)
{
    if (x >= y)
        return x;
    else
        return y;
}
```

- Type could be any user-defined or built-in variable type

Sample Function Template

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

template <class Type>
Type larger (Type x, Type y);

int main ()
{
   string str1;
   string str2;

   cout << "Line 1: Larger of 5 and 6 =" << larger (5, 6) << endl;
   cout << "Line 2: Larger of A and B =" << larger ("A", "B") << endl;
   cout << "Line 3: Larger of 5.6 and 3.2 =" << larger (5.6, 3.2) << endl;

   str1 = "Hello";
   str2 = "Happy";

   cout << "Line 4: Larger of " << str1 << " and " << str2 << larger (str1, str2) << endl;

   return 0;
}

template <class Type>
Type larger (Type x, Type y)              // use template parameter as a data type
{
   if (x >= y)
      return x;
   else
      return y;
}
```

Output

```
Line 1: Larger of 5 and 6 = 6
Line 2: Larger of A and B = B
Line 3: Larger of 5.6 and 3.2 = 5.6
Line 4: Larger of Hello and Happy = Hello
```

Class Template

Syntax
template <class Type>
class declaration or method definition

- Type is formal parameter for the template (as variables are parameters for the methods)
  - It can represent any user-defined type or built-in type
- Type can be used to:
  - Specify type of parameters to the methods
  - Specify return type of the methods
  - Declare attribute within the class
- Type will be resolved during compiler time based on the actual use of the class

Sample Class Template

```
// StackList Class Template Definition

template <class StackType>
Class StackList
{
public:
   StackList ();                        // constructor - initialize the stack as empty
   ~StackList ();                       // destructor - delete all items still in stack

   void Push(StackType newItem);        // create a StackNode, add a StackNode in the stack,
                                        // by adding to the front of the linked List
   StackType Pop();                     // return the StackNode at the top of the stack, remove the StackNode
                                        // from the stack and delete the StackNode
   bool IsEmpty() const;                // check if stack is empty

   StackType Peek() const;              // return the StackNode at the top of the stack

   int Size() const;                    // return the number of items in the linked list


private:
   StackNode Struct {
      StackType   item;                 // item to store
      StackNode   *next;                // linked list next pointer
   };
   StackNode      *head;                // head pointer for Stack
   int            stackCount;           // total number of items in the stack

};
```

Class Template
Creating Objects

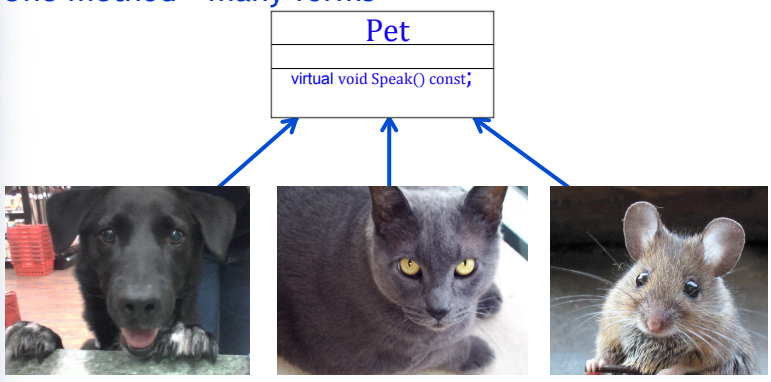
Syntax
ClassName <data-type> objectName;

- Template parameter (data type) for classes must be explicit
- data-type has to be any user-defined or built-in data type
- Examples:

```
StackList<int> myStack;
StackList<double> newStack;
StackList<WoolType> woolStack;
(where WoolType is an enum data type)
```

Polymorphism – Let's go back to this example

One method - many forms



They all speak… but do they all speak the same language?

We can override the method - but what if we want to be able to create new types of pets later and minimize modifying the code that is utilizing them

Static binding with pointers

```
Pet *myPet;
Dog *myDog;
Cat *myCat

myPet = new Pet;
myDog = new Dog;
myCat = new Cat;
```

This binds to the Pet implementations

This binds to the Dog implementations

This binds to the Cat implementations

This is Overriding
Dog inherits everything in Pet, but also contains attributes and methods that are extended by Dog

Issues with Redefining

Let's say we want to write a method to make our pets speak?

```
void PetTalk(Pet *myPet)
{
        myPet->Speak();
}
```

We would need a function for each pet.

```
void DogTalk(Dog *myDog)
…
```

We could do this, but what if we know in the future we'll have different types of pets - but we don't know what they will be… this can't be handled with static overriding.

Also, this can be tedious if we have many different types of pets.

Ideally, we could have one function to handle all pets… but… which method would it bind to and how would it know?

Need for Virtual Methods

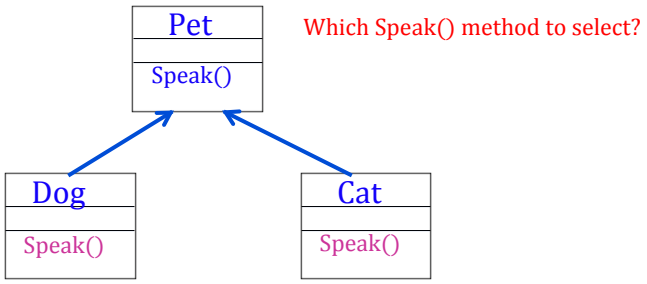
Static (Early) Binding -> redefining

- For normal function and method calls
  - the compiler will determine the correct function or method address and *binds* to it generating a direct call at compile time
- Limited support for dynamic inheritance
  - when methods in the base class are overridden in the derived classes
    - Compiler will not know the correct method to select and binding will be created during run time

Need for Virtual Methods

Assume that speak() method in Dog and Cat classes have been overridden from the Pet class.

How can it support dynamic objects?



- Dynamic binding and virtual methods address this limitation

SubType Polymorphism: Virtual Functions or Method

- In terms of functions or methods
  - Ability to assign multiple functions/methods with the same name using *virtual functions* and *late binding or dynamic binding*
    - We implement this by creating a pointer to an object of the base class
    - and then instantiate them as the derived class ("*new*" the derived class)
  - We use it with inheritance so each derived class can implement the function in their own way.
  - Also, it allows the interface for the parent class to be more complete
    - The virtual functions don't have to be implemented until later

Polymorphism

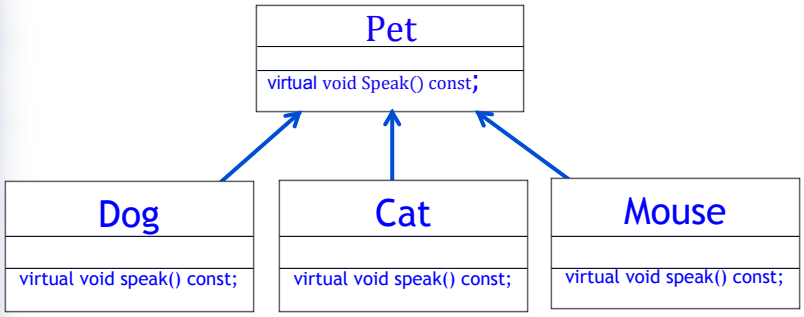
- *Virtual Function (or method)*
  - With an inheritance hierarchy
    - the appropriate method is selected during run-time execution of the code
      - **Based on the object instantiation**
    - implement by defining a method as *virtual* in the base class and redefining the same method in the derived classes
- Why?
  - Use of the *virtual method* makes it easy to extend the code
  - increases code reusability
  - Simplifies maintenance
  - We don't always know what the future derived classes will be… we just know we will need them

Virtual functions

- *Redefined methods* are linked at compile time
  - The compiler knows what the methods are at compile time
    - Based on which object is instantiated
- When would you use a *virtual function*?
  - Let's say method ***a*** uses method ***b***
  - Each derived class implements ***b*** differently
  - This is when you would use a virtual method because the compiler does not know ahead of time how ***b*** will be implemented
  - You may want to derive classes later that obviously don't have an implementation for ***b*** – so how can method a use it?
- Virtual functions → we don't know at compile time what all possible implementations are going to be – so virtual functions determine this at run-time - dynamically

Polymorphism

Dog, Cat, & Mouse are all derived from Pet



Do we know all possible pets at this point? We know they all will speak

Let's say later down the road we want to implement horse.

- We don't have the implementation of speak for a horse yet… so we use a *virtual function* which will be implemented later
- This will "bind" the new method during runtime – we call that *dynamic binding*

Virtual Methods

Syntax
virtual returnDatatype MethodName (datatype name, datatype name) const;

- Once a method is declared as virtual in the base class, it is virtual in all derived classes too
  - However, it is a good style explicitly label the method as virtual in derived classes too as a reminder
- Pointers of the base class type will be able to access virtual methods in derived class
  - In dynamic object creation, we may not know which derived class will be created; so we need to use base class type pointers
- When overriding virtual methods you can alter the data access state (public, protected or private) of the method in the derived class

Virtual Function Example

Make the destructor virtual too

```
class Pet
{   public:
       Pet();              // constructor
       virtual ~Pet();     // destructor
       …
       virtual void  Speak() const;
    protected:
       string petName;
       int    petAge;
       float  petWeight;
       string petBreed;
    private:
};
```

DERIVED CLASS

```
class Mouse : public Pet
{
    public:
      // constructors
      Mouse(); Mouse(string name,
                     int    age,
                     float  weight,
                     string typeOfPet);
      virtual ~Mouse();  // destructor

      /**** ACCESSORS ****/
      virtual void Speak() const;
};
```

IMPLEMENTATIONS

```
void Pet::Speak() const
{
   cout << petName << " is speaking...";
}
```

```
void Dog::Speak() const
{
   cout << petName << " says Woof Woof!";
}
```

```
void Cat::Speak() const
{
   cout << petName << " says Meoowww!";
}
```

```
void Mouse::Speak() const
{
   cout << petName << " says squeak!";
}
```

It isn't necessary to make the method virtual the derived class - but it is good practice

Example Main & Output

```
#include "OOP Header.h"
#include "pet.h"
#include "dog.h"
#include "cat.h"
#include "mouse.h"
int main()
{
    Pet *buddy;
    Pet *bear;
    Pet *ben;
    Pet *fluffy;

    fluffy = new Pet  ("Fluffy");
    buddy  = new Dog  ("Buddy",  6, 32.5, "Dog");
    bear   = new Cat  ("Bear" , 18, 20.5, "Cat");
    ben    = new Mouse("Ben"  ,  1,  0.5, "Mouse");

    PetTalk(buddy);
    PetTalk(bear);
    PetTalk(ben);
    PetTalk(fluffy);

    Display(buddy);
    Display(bear);
    Display(ben);
    Display(fluffy);

    return 0;
}
```

These all need to be Pointers
→ for Dynamic binding
This enables sub-type polymorphism

```
void PetTalk(Pet* myPet)
{
        myPet->Speak();
}
```

OUTPUT

```
Buddy says Woof Woof!
Bear says meoowww!
Ben says squeak!
Fluffy is speaking...

Fluffy     0     0.00
Buddy      6    32.50   Dog
Bear      18    20.50   Cat
Ben        1     0.50   Mouse
```

Dynamic Binding and Virtual Methods

- In dynamic (late) binding the correct method is selected only after the object type is known
  - implementing the concept of *polymorphism*
- Compiler does not generate code to call a specific method
  - it generates information to enable the run-time system to generate specific code for the method call
- Polymorphic methods are declared as *virtual* in the base class
  - *virtual method* "expect" to be overridden

The method for each object is tracked in a virtual table.

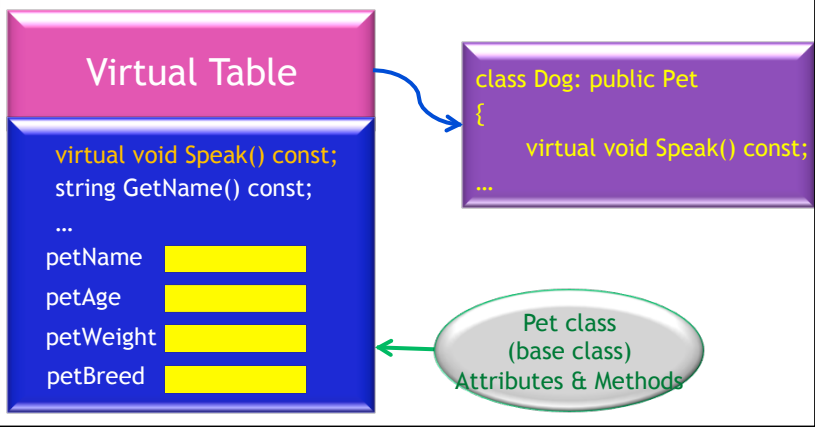
Virtual Table

Dynamic Binding works by means of a virtual table

```
Pet *myPet;
myPet = new Dog;
```

It knows to bind to the Dog implementations here

The virtual table keeps track of which methods to bind to dynamically at the "new" statement



Slicing problem

- If we have a pointer to a base class (Pet) and we instantiate it as a derived class

```
Pet *myPet;
myPet = new Dog;
```

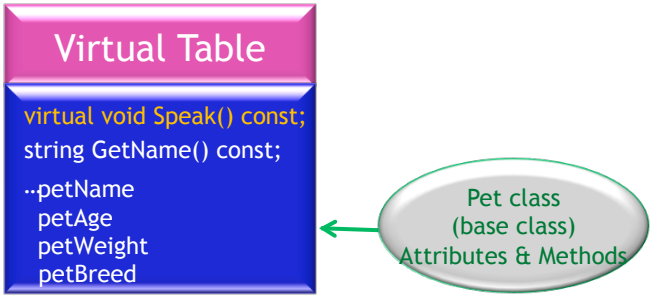
  - Only the attributes/methods of a base class (Pet) are created
    - → none of the extended methods/attributes are available
  - Let's say we extended Dog to have a wagTail attribute and methods to set it
    - They would not be accessible
  - Only the inherited attributes/Methods from Pet are available

Slicing Example

```
Pet *myPet;
myPet = new Dog;
```

My Pet is still type Pet - this just allows it to Bind to Dog's virtual methods

- Dogs extended methods/attributes are not available

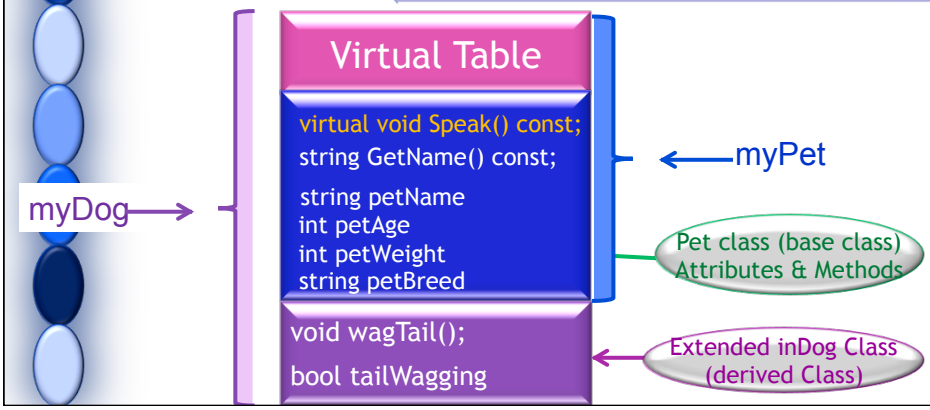


Avoiding the Slicing Problem

```
Pet *myPet;
Dog *myDog;

myDog = new Dog;
myPet = myDog
```

myDog instantiates all attributes of Pet & Dog Because it is of type Dog - not Pet

myPet is now pointing to the same object ast Dog except it cannot access anything extended by Dog

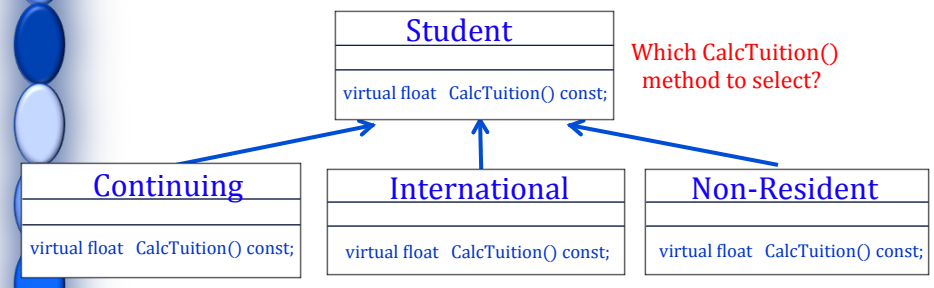


Some final notes

- Can't have a virtual constructor
  - The virtual table has not been created before the constructor has run
- Should have a virtual destructor if you have a least 1 method that is virtual
  - If you don't you risk a memory leak
- Don't make it virtual unless you are going to derive classes from the class
  - There is an overhead in having a v-table

Another Example of Using Virtual Function

Let's assume different types of college students: Continuing, International, & NonResident which derive from Student



Do we know all possible students at this point? We know they all will pay tuition

Let's say later down the road we want to implement a High School Student who is co-enrolled at a local High School & Saddleback

Updating Animal Class to define Display() and Speak() methods as virtual

```
class Animal                                    // base class
{
        public:
                Animal();                       // constructor
                virtual  ~Animal();             // destructor
                void   SetInitialValues(string  animalName,
                                        int     animalAge,
                                        float   animalValue);
                void   ChangeAge(int animalAge);
                void   ChangeValue(float animalValue);
                string GetName() const;         // return the animal's name
                int    GetAge() const;          // return the animal's age
                float  GetValue() const;        // return the animal's value
                virtual void   Speak() const;   // Output Animal's sound
                virtual void   Display() const; // print all animal's details
        private:                                // animal details
                string name;                    // animal's name
                int    age;                     // animal's age in years
                float  value;                   // animal's value in dollars
};
```

Updated Sheep Class

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

                void        SetWool(WoolType typeOfWool,
                                    string   woolColor);
                WoolType    GetWool() const;    // return the sheep's wool type
                string      GetColor() const;   // return the sheep's wool color
                void        DisplayWool() const; // print sheep's wool details
                virtual void Display() const;   // print sheep's details
                virtual void Speak() const;     // Speak for sheep


        private:
                // sheep wool details
                WoolType    wool;               // sheep's wool type (enum.)
                string      Color;              // sheep's wool color
};
```

Using Virtual Methods

```
…
int main()
{
        Animal *ptrA;          // create an Animal Class pointer
        Animal *ptrS;          // create another Animal Class pointer
        …
        ptrA = new Animal;     // create a dynamic Animal object
        ptrS = new Sheep;      // create a dynamic Sheep object
        …                      // derived from Animal Class
        ptrA -> Display();     // invoke the virtual Display()
                               // method from Animal Class
        ptrS -> Display();     // invoke the virtual Display()
        …                      // method from Sheep Class
        return 0;
}
```

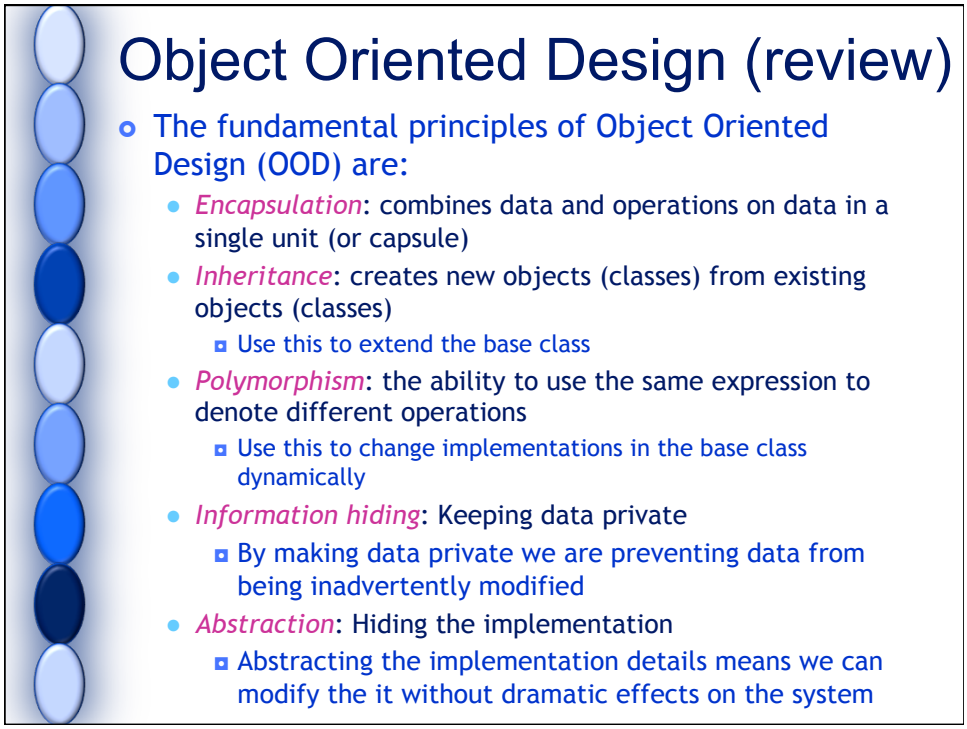
Exercise - Virtual Functions

Indicate which method will be called

```
int main()
{
   Parent *ptrP1;          // Parent Ptr will point to Parent Object
   Parent *ptrP2;          // Parent Ptr will point to Child Object
   Child  *ptrC;           // Child pointer

   ptrP1 = new Parent;
   ptrP2 = new Child;
   ptrC  = new Child;

   ptrP1 -> method1();
   ptrP1 -> method2();
   ptrP1 -> method3();

   ptrP2 -> method1();
   ptrP2 -> method2();
   ptrP2 -> method3();

   ptrC  -> method1();
   ptrC  -> method2();
   ptrC  -> method3();
}`
```

```
class Parent
{
public:
   void method1();
   virtual void method2();
   void method3();
…
};
```

```
class Child : public Parent
{
public:
   void method1();
   virtual void method2();
…
};
```


Object Oriented Design (review)
The fundamental principles of Object Oriented Design (OOD) are:
Encapsulation: combines data and operations on data in a single unit (or capsule)
Inheritance: creates new objects (classes) from existing objects (classes)
Use this to extend the base class
Polymorphism: the ability to use the same expression to denote different operations
Use this to change implementations in the base class dynamically
Information hiding: Keeping data private
By making data private we are preventing data from being inadvertently modified
Abstraction: Hiding the implementation
Abstracting the implementation details means we can modify the it without dramatic effects on the system