CSCI 3110 Inheritance (2)

- **Static Binding (Early Binding)**
  - When a member function is called through an object or an object pointer, a decision needs to be made: which "version" of that function should be used – parent's version or its own version
  - If the decision is made at compile time, it is called static binding (earlier binding)

An example of early binding

```
class Sale
{
public:
        Sale();
        Sale(double thePrice);
        double GetPrice() const;
        void SetPrice(double newPrice);
        double  Bill () const;
        double Savings (const Sale & other)
        const;
protected:
        double price;
};

Sale::Sale() : price (0){}

Sale :: Sale(double thePrice){
        price = thePrice;
}

double Sale::Bill() {
        return price;
}

double Sale::GetPrice{} const {
        return price;
}

void Sale::SetPrice(double newPrice) {
        price = newPrice;
}

double Sale::Savings(const Sale & other) const
{       return (Bill() – other.Bill());
}
```

DiscountSale Class

```
class DiscountSale : public Sale
{
public:
        DiscountSale();
        DiscountSale(double thePrice, double
        theDiscount);

        double GetDiscount() const;
        void    GetDiscount(double
        newDiscount);
        double  Bill()  const;
private:
        double discount;
};

DiscountSale::DiscountSale() : Sale(0), discount
(0)
{}

DiscountSale::DiscountSale(double thePrice,
double theDiscount):Sale(thePrice),
discount(theDiscount){}

double DiscountSale::GetDiscount() const
{       return discount;
}

void DiscountSale::SetDiscount(double
newDiscount)
{       discount = newDiscount;
}

// overwrite base class Bill() definition
double DiscountSale::Bill() const
{       double fraction = discount / 100;
        return (1-fraction)*price;
                        // or getPrice();
}
```

client program:

```
bool operator < (const Sale& first, const Sale & second);

int main() {
        Sale simple(10.00);
        DiscountSale  disc(11.00, 10);

        if (disc < simple)    {
                cout << "Discounted item is cheaper";
                cout << "Saving is $" << simple.saving(disc) << endl;
        }
        else
                cout << "Discounted item is not cheaper."
}

bool operator < (const Sale& first, const Sale& second) {
        return (first.Bill() < second.Bill());
}
```

For overloaded operator < , when "dic < simple" is executed :

- "disc", an object of **DiscountedSale** class, is cast back to **Sale** class during compile time – **early binding (binding done at compile time**)
- As a result, the **Bill** method associated with parameter "first" is the **Bill** method defined for **Sale** class, not the **Bill** method for **DiscountSale** class.

How to make an object always use its own methods, e.g., not being bound by methods defined in base classes?
Answer: Dynamic binding using **virtual functions**
Virtual functions are member functions that are declared in the base class using the keyword virtual and can be overridden by the derived class. They are used to achieve **Runtime polymorphism** or say **late binding** or **dynamic binding.**

If a method is defined to be virtual, it is to tell the compiler, " I do not know how this function is implemented. Wait until it is used in a program, and then get the implementation from the object" -- **wait until run time to determine the implementation of a function – late binding (examples including overloaded function, function passed as parameter…)**

Dynamic Binding:

- The decision of which version of the member function should be used is made dynamically at run time
- Dynamic binding use object's "real type" to select the appropriate method that is invoked at run time
- To tell system to bind a member function call dynamically, define the member function as a virtual method
- Virtual member function is selected dynamically (at run-time) based on the type of the object, not the type of the pointer/reference to that object

Change the program to:

```
class Sale
{
public:
        Sale();
        Sale(double thePrice);
        double GetPrice() const;
        void SetPrice(double newPrice);
        virtual double  Bill () const;   // ← this is the only change necessary,
                                         /*  (a) if a method is defined to be virtual, then all
                                         definitions of the function in the derived classes
                                                 will automatically be virtual
                                            (b) virtual modifier is not necessary in the
                                                    implementation file.  */
        double Savings (const Sale & other) const;
private:
        double price;
};
```

No change in DiscountSale class header file and implementation file

…

```
// overwrite base class Bill() definition
double DiscountSale::Bill() const
{       double fraction = discount / 100;
        return (1-fraction)*price;      // or use getPrice();
}
…
```

Client program:

```
bool operator < (const Sale& first, const Sale & second);

int main()  {
        Sale simple(10.00);
        DiscountSale  disc(11.00, 10);

        if (disc < simple)          {
                cout << "Discounted item is cheaper";
                cout << "Saving is $" << simple.saving(disc) << endl;
        }
        else
                cout << "Discounted item is not cheaper."
}

bool operator < (const Sale& first, const Sale& second)  {
                                                // Note:  parameter passed by reference!!
                                                // What if use parameter passing by value?
        return (first.Bill() < second.Bill());
}
/* output:      Discounted item is cheaper
                Savings is $0.10                        */
```

<Re-Examine the Mammal-Dog example (before virtual keyword is used) >

```
#include <iostream>
#include "mam.h"
#include "dog.h"
int main()
{
        mammal Animal;
        dog MyDog;

        Animal.Speak();
        MyDog.Speak();
        mammal* mamptr=&Animal;
        mammal* dogptr=&MyDog;    // legal since MyDog "is a" mammal
        cout << "Mammal's pointer ";
        mamptr->Speak();
        cout <<"Dog's pointer ";
        dogptr->Speak();

        return 0;
}
```

Program Output:

```
Mammal's constructor
Mammal's constructor
Dog's constructor
Mammal is speaking
BOW-WOW
Mammal's pointer Mammal is speaking
Dog's pointer Mammal is speaking
```

← since "virtual" keyword was not used

- This is an example of **static or early binding** – ***appropriate version of a member function is determined at compilation time*** (compiler binds a particular object to a given function). MyDog.Speak() will execute the Speak() function found in the dog class.

- Since *dogptr* is declared as a pointer to a ***mammal,*** *dogptr->Speak()* will execute the Speak() function found in the mammal class.

Polymorphism:

a. Many forms : during run time, an object pointer or an object reference may take on many different forms of a class method depending on which underlying object class is used
b. C++ extends polymorphism to allow pointers to base classes to be assigned to derived class objects.
c. Polymorphism is supported using **dynamic binding** and **virtual member functions**.
d. The reserve word **virtual** should be used when ***declaring selected methods in a base class if it is possible that these methods may or should be overridden in a derived class.*** This notifies the compiler that these functions may be defined differently in a derived class and that the appropriate method should be determined at execution time.
e. Virtual function magic operates only on **pointers and references. ← IMPORTANT!!**

```
#ifndef MAM_H
#define MAM_H
class mammal {
        public:
                mammal();
                mammal (int W);
                ~mammal();
                int ReturnWeight();
                int ReturnHeight();
                virtual void Speak()const;
        protected:
                int Age;
                int Weight;
                int Height;
};
#endif

#include <iostream>
#include "mam.h"
#include "dog.h"
using namespace std;
int main(){
        mammal Animal;
        dog MyDog;
        Animal.Speak();
        MyDog.Speak();
        mammal* mamptr=&Animal;
        mammal* dogptr=&MyDog;
        cout <<"Mammal's pointer ";
        mamptr→Speak();
        cout << endl;
        cout <<"Dog's pointer ";
        dogptr→Speak();
        cout << endl;
        return 0;
}
```

Program Output:

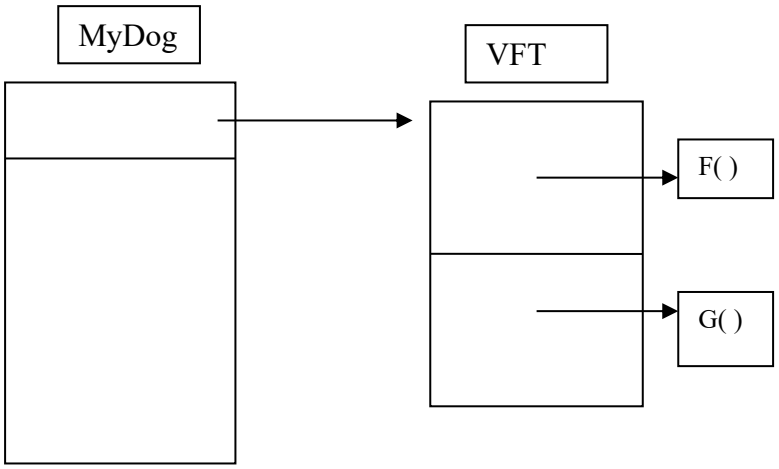
```
Mammal's constructor
Mammal's constructor
Dog's constructor
Mammal is speaking
BOW-WOW
Mammal's pointer Mammal is speaking

Dog's pointer BOW-WOW
```

Additional rules:

- **constructors can not be virtual**
- **destructor**
    a. should be virtual, ➔ guarantees that future descendants of the object can de-allocate themselves correctly.
    b. If any function in a class is virtual, destructor should be virtual as well.
    c. A class destructor must be defined when a class allocates dynamic memory.
- virtual function's return type can not be overridden
- **Virtual function table**
  having "virtual function" introduces overhead, e.g., more storage is needed. So, C++ gives this flexibility to the programmer to decide which functions should be made virtual. If a function does not need to take advantage of being virtual, then do not making it virtual makes the program runs more memory efficient.

    - Each object that has at least one virtual function contains a pointer to a run time ***virtual function table.*** The *virtual function table* contains the starting addresses of all virtual functions declared in the class.
    - When a virtual function is called through a pointer or reference, the run time system uses the object's address to access the pointer to the virtual function table, follows the pointer to the table, looks up the address of the function and calls the function.



Example:

```
class mammal
{public:
      virtual void F();
      virtual void G();
};
class dog : public mammal
{public:
      void F();
      void G();
};
dog MyDog;
```

What is the output of the following program?

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

class animal.  {
private:
        string animalName;
public:
        animal (string aAnimal)
        {
                animalName = aAnimal;
        }
        virtual void Identify ()
        {
                cout <<"I am a " << animalName
                << " animal" << endl;
        }
};

class cat: public animal {
private:
        string  catName;
public:

        cat (string nCat, string nAnimal):
        animal(nAnimal)
        {
                catName = nCat;
        }
        virtual void Identify ()
        {
                animal::Identify();
                cout <<"I am a " << catName
                << " cat" << endl;
        }
};

class tiger: public cat  {
private:
        string tigerName;
public:
        tiger (string nTiger, string nCat, string
        nAnimal):cat (ncat, nAnimal)
        {       tigerName = nTiger;     }
        virtual void Identify ()
        {
                cat::Identify();
                cout <<"I am a " << tigerName
                << " tiger" << endl;
        }
};
 void Announce1 (animal oneAnimal)  {
        cout <<"In Announce1, calling Identify:"
        << endl;
        oneAnimal.Identify();
        cout << endl;
}

void Announce2 (animal* aPtr) {
        cout <<"In Announce2, calling Identify: "
        << endl;
        aPtr→Identify();
        cout << endl;
}

int main()   {
        animal firstAnimal("reptile"), *p;
        cat secondAnimal("domestic",
        "warm blooded");
        tiger thirdAnimal("Bengal", "wild", "meat
        eating");

        firstAnimal=thirdAnimal;
        firstAnimal.Identify();

        // not allowed
        // thirdAnimal = firstAnimal;

        p=&secondAnimal;
        p→Identify();

        Announce1(thirdAnimal);

        Announce2(&firstAnimal);
        Announce2(&secondAnimal);
        Announce2(&thirdAnimal);

        return 0;
}

/* what if Annouce2 is changed to the following:
void Announce2 (animal& aPtr) {
        cout <<"In Announce2, calling Identify: "
        << endl;
        aPtr.Identify();
        cout << endl;
}

And call with:
        Announce2(firstAnimal);
        Announce2(secondAnimal);
        Announce2(thirdAnimal);

*/
```

What's the output of the following program?

```
class Animal
{
        public:
                virtual void eat() { cout << "I eat like a generic Animal. \n"; }
};
class Wolf : public Animal
{
        public:
                void eat() { cout << "I eat like a wolf!\n"; }
};
class Fish : public Animal
{
        public:
                void eat() { cout << "I eat like a fish!\n"; }
};
class OtherAnimal : public Animal { };

// client program
int main()
{
        Animal *anAnimal[4];

        anAnimal[0] = new Animal();
        anAnimal[1] = new Wolf();
        anAnimal[2] = new Fish();
        anAnimal[3] = new OtherAnimal();

        for(int i = 0; i < 4; i++)
                anAnimal[i]→eat();
}
```

1. A base class *Animal* could have a virtual function eat.
2. Subclass *Fish* would implement eat() differently than subclass *Wolf*.
3. But you can invoke eat() on any class instance referred to with pointer to *Animal*, and get the eat() behavior of the specific subclass.
4. This allows a programmer to process a list of objects of base and derived class from Animal class, telling each in turn to eat, **with no knowledge of what kind of animal may be in the list**.
5. You also do not need to have knowledge of how each animal eats, or what the complete set of possible animal types might be.

- **Virtual Destructor**
  - If a class has a virtual member function, then declare the destructor virtual.
  - The compiler will perform static binding on the destructor if it is not declared virtual. This can lead to problems such as:
    - When a base class pointer or reference variable refers a derived class object. If the derived class has its own destructor, it will not execute when the object is destroyed or goes out of scope. Only the base class destructor will execute.

- **Virtual Method**
  - Virtual methods **must** be class methods, i.e. they cannot be Ordinary "stand-alone" functions
  - Cannot be used for class Static methods
  - Supplying *virtual* keyword is optional when overriding a virtual method in derived classes.
  - You can declare a virtual method in any derived class.
  - Using virtual methods adds a small amount of time and space overhead to the class/object size and method invocation time.
  - Polymorphic behavior is not possible when an object is passed by value.

- **Static vs Dynamic Binding – A Comparison**
  - Static Binding
    - If you are sure that any derived classes will not want to override this operation dynamically (just redefine and hide)
    - Use mostly for reuse or to form "concrete data types"
  - Dynamic Binding
    - When derived classes may want to provide a different (more efficient, more functional) implementation that should be selected at run time.
    - Used to build dynamic type hierarchies and to form "abstract data types"
  - Efficiency vs flexibility are the primary tradeoffs between static and dynamic binding
  - Static binding is generally more efficient since
    - It has less time and space overhead
  - Dynamic binding is more flexible since it enables developers to extend the behavior of a system transparently (as seen in polymorphism behavior)
  - Both static and virtual functions can be inlined

- **Pure virtual Method**

```
class Shape{
public:
        virtual double area();
};
class Rectangle : public Shape {
        public:
                double area();
};
class Triangle : public Shape {
        public:
                double area();
};
class OtherShape : public Shape { };
```

Question: How to implement the function area() in class *Shape*?

- Define area() as a pure virtual method:

```
class Shape
{
public:
   virtual double area() = 0 ;
};
```

- Classes containing pure virtual methods are termed "abstract". Abstract classes cannot be instantiated directly.
- A pure virtual function is a virtual function that is required to be implemented by a derived class that is not abstract.
- A subclass of an abstract class can only be instantiated directly if all inherited pure virtual methods have been implemented by that class or a parent class.
- The only effect of declaring a pure virtual destructor is to cause the class being defined to be an abstract class.

- **Override Confusion**

  - **Overriding vs. Overloading**
    - When you redefine a virtual method, the new function definition given in the derived class has the same number and types of parameters.
    - On the other hand, a non-virtual function in the derived class will hide the functions with the same name defined in the base class, no matter if they have the same or different parameter lists.
    - To enable the overloading with functions from the base class, you need to use:
      using BaseClassName::FunctionName;
      in derived class declaration.
      (see replit example code)

  - Override keyword
    - No special keyword or annotation is needed for a function in a derived class to override a function in a base class. For example:

```
struct B {
        virtual void f();
        virtual void g() const;
        virtual void h(char);
        void k();                       // not virtual
};
struct D : B {
        void f();     // overrides B::f()
        void g();     // doesn't override B::g() (wrong type). It hides B::g() const
        virtual void h(char); // overrides B::h()
        void k();    // doesn't override B::k(), B::k() is not virtual
};
```

    - This can cause confusion and problems if a compiler doesn't warn against suspicious code.
      - Did the programmer mean to override **B::g()**? (almost certainly yes).
      - Did the programming mean to override **B::h(char)?** (probably not because of the redundant explicit virtual).
      - Did the programmer mean to override **B::k()**? (probably, but that's not possible).

    - Use the contextual keyword "**override**"
      - To allow the programmer to be more explicit about overriding, we now have the "contextual keyword" override:

```
struct D : B {
        void f() override; // OK: overrides B::f()
```

```
        void g() override; // error: wrong type
        virtual void h(char); // overrides B::h(); likely warning
        void k() override; // error: B::k() is not virtual
};
```

- A declaration marked override is only valid if there is a function to override.
- The problem with h() is not guaranteed to be caught (because it is not an error according to the language definition) but it is easily diagnosed.
- *override* is only a contextual keyword, so you can still use it as an identifier, but not recommended.

- Override control: **final**
  - Sometimes, a programmer wants to prevent a virtual function from being overridden. This can be achieved by adding the specifier final. For example:

```
struct B {
        virtual void f() const final; // do not override
        virtual void g();
};
struct D : B {
        void f() const; // error: D::f attempts to override final B::f
        void g(); // OK
};
```

  - Adding final closes the possibility of a future user of the class might provide a better implementation of the function for some class you haven't thought of.
  - So, if you feel the urge to add a final specifier, please double check that the reason is logical:
    - Would semantic errors be likely if someone defined a class that overwrote that virtual function?
    - If you don't want to keep that option open, why did you define the function to be virtual in the first place?
  - *final* is only a contextual keyword, so you can still use it as an identifier, but not recommended.

- **Upcast**
  1. An upcast moves an object or a object pointer up a class hierarchy, from a derived class to a class it is derived from.
  2. An upcast is an implicit conversion
  3. Reason: "is-a" relationship.
  4. Example: A hourly paid employee is an employee, and a salary paid employee is an employee.

```
// two function prototypes
void print(Employee  e);
void print(Employee*  p);
int main(  ) {
        Employee John("John", "111-11-1111");
        HourlyEmployee David("David", "222-22-2222", 13.0, 30);
        print(John);
        print(&John);
        print(David); // ✓ because of upcast
        print(&David); // ✓ because of upcast
        John = David; // ✓ because of upcast
```

```
        return 0;
}
```

Upcast Pitfall – Slicing Problem

- Assigning a derived class object to a base class object slices off data.
- Any data members in the derived class object that are not also in the base class will be lost in the assignment.
- Any member function that are not defined in the base class are similarly unavailable to the resulting base class object.

Example:

```
HourlyEmployee David("David", "222-22-2222", 13.0, 30);
Employee John;
John = David; // upcast
John.getRate(); // Compile Error: getRate sliced off
```