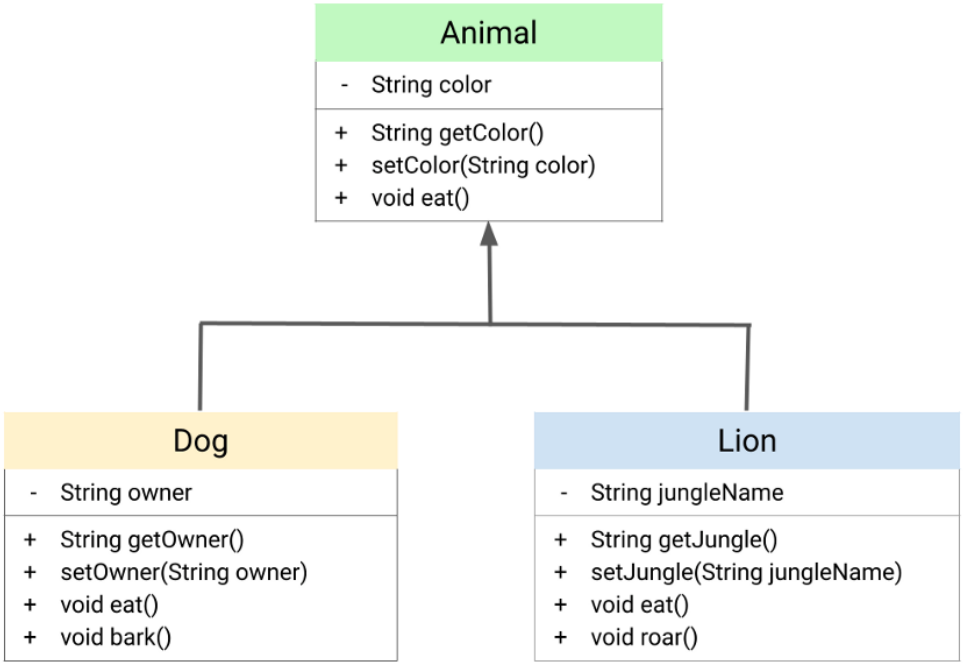
CSCI 3110 Inheritance (1)

- **Motivation**
  - Generalization and Specialization
    - In the real world, many objects are specialized versions of other more general objects.
    - Insect is a very general type of creature with numerous characteristics.
    - Since grasshoppers and bumble bees are insects, they have all the general characteristics of an insect.
    - Additionally, they have special characteristics of their own. The grasshopper has its jumping ability, and the bumble bee has its stinger.
    - Grasshoppers and bumble bees are specialized versions of an insect.
  - Inheritance and "Is-a" Relationship
    - When one object is a specialized version of another object, there is an "is-a" relationship between them.
      i. A grasshopper **is an** insect.
      ii. A poodle **is a** dog. A car **is a** vehicle.
      iii. A rectangle **is a** shape.
    - When an "is a" relationship exists between classes, it means that the specialized class has all of the characteristics of the general class, plus additional characteristics that make it special.
    - In object-oriented programming, *inheritance* is used to create "is a" relationship between classes.
  - **Purpose of inheritance** – inheritance provides a method for coping with the complexity of a problem by allowing the use of existing code.



- **Inheritance**
  - a relationship among classes whereby a class derives properties from a previously defined class (parent class).
  - Inheritance provides a means of deriving a new class from existing classes called base classes.
  - Inheritance describes the ability of a class to derive properties from a previously defined class.
  - **base class** – a class from which another class is derived.
  - **derived class** – a class that inherits the members of another class called the base class.
    - What does a derived class inherit?
      Every data member defined in the parent class (although such members may not always be accessible in the derived class)
      Every ordinary member function of the parent class (although such members may not always be accessible in the derived class!)

    - **What does a derived class NOT inherit**?
      1. The base class's constructors and destructor
      2. The base class's assignment operator
      3. The base class's friends

    - What can a derived class **add**?
      1. New data members
      2. New member functions
      3. New constructors and destructor
      4. New friends

- **Three data access types in a class**
  - All classes can contain a public section, private section and protected section.
  - **Public** portions of a class are accessible by anyone.
  - **Private** portions of a class are accessible only by members and friends of the class.
  - **Protected** portions of a class are accessible by members of the class, friends of the class and all derived classes.

```
class   MyType
{
public:
        // declarations of visible members, can be used everywhere, no restriction

protected:
         // declarations of protected members
        // only accessible by member functions of MyType, friends of MyType, and
        // member functions and friends of classes derived from MyType

private:
        // can only be used by the member functions of MyType and friends of
MyType
};
```

- **Three kinds of inheritance** – public inheritance, protected inheritance and private inheritance
  - **Public inheritance** – public and protected members of the base class remains to be public and protected members of the derived class. Public inheritance denotes a "is a" relationship.
    For example, a student "is a" person, a cat "is a" mammal.

```
class mammal {
……..
};
class cat : public mammal
{
…….
};
```

  - **Protected inheritance** – public and protected members of the base class are protected members of the derived class. Protected inheritance denotes a "has a" relationship. Also referred to as containment. For example, a pen "has a" ball. "has a" means a class has an object as a data member.

```
class engine            {
        …
};
class car : protected engine
{
        …
};
```

  - **Private inheritance** – public and protected members of the base class are private members of the derived class. Private inheritance denotes a "as a" relationship. For example, a stack can be implemented "as a" list. Private inheritance is used to implement one class in terms of another class.
    For example:

```
class list        {
        …
};
class stack: private list
{

};
```

    /* In this case stack can manipulate the items on the stack using list's method. Yet, the underlying list is hidden from the clients and descendants of the stack. */
    Private inheritance is useful when A class needs access to the protected members of another class, or If methods in a class need to be redefined

  - On all cases, *private* section of a base class cannot be accessed by a derived class

  - Protected inheritance is not often used.
  - Most software development uses public inheritance.

- **Rules of the visibility of base class data in the inherited class**

```
                                  // 1. private member of B are not visible anywhere outside
                                  //    class B, this is not affected by the inheritance method
class D : private B               // 2. all members of B that are protected and public will be
{ … };                            //    private in D

class D : protected B            // 3. all members of B that are public and protected will be
{ …};                             //    protected in D

class D : public B               // 4. public members of B remain public in D
{ …};                             //    protected members of B remain protected in D
```

- **Overriding member function and virtual function**

  a. Derived classes can revise any inherited member functions.
     - Used when a derived class requires a different implementation for an inherited virtual member function, the function can be redefined in the derived class, called overriding.

  b. A derived class inherits all members of its base class, except constructors and destructor, friend functions, overloaded = operator (other overloaded operators may be inherited)
     - A derived class's constructor executes **after** the base class's constructor.
     - A derived class's destructor executes **before** the base class's destructor.
     - When a base class constructor requires parameters, the derived class constructor must explicitly call the base constructor and pass necessary parameters.

  c. Virtual function
     - When a virtual member function is redefined, you must list its declaration in the definition of the derived class even though the declaration is the same as in the base class.
     - If you don't wish to redefine a virtual member function inherited from the base class, then it is not listed in the definition of the derived class.

```
class  Base
{
public:
        int a;
protected:
        int b;
private:
        int c;
};

class D1 : private Base
{
public:
        void f();
};

class D2 : protected Base
{
public:
        void g();
};

class D3 : public Base
{
public:
        void h();
};

class E1 : public D3
{
public:
        void Ef1();
};
```

Which of the following statements are correct? Which ones are incorrect?

```
void D1::f()
{       a=0;     // ok
        b=0;     // ok
        c=0;
}
void D2::g()
{       a=0;   // ok
        b=0;  //  ok
        c=0;
}
void D3::h()
{
        a=0;  // ok
        b=0;  // ok
        c=0;
}
void E1::Ef1()
{
        a=0; // ok
        b=0; //ok
        c=0;
}

int main()
{
        Base obj;
        obj.a = 0;          // ok
        obj.b=0;
        obj.c =0;

        D1  d1;
        d1.a = 0;
        d1.b = 0;
        d1.c = 0;

        D2  d2;
        d2.a = 0;
        d2.b = 0;
        d2.c = 0;

        D3 d3;
        d3.a = 0;    // ok
        d3.b = 0;
        d3.c = 0;

        E1 e1;
        e1.a = 0;   // ok
        e1.b = 0;
        e1.c = 0;

        return 0;
}
```

- **What happens when a derived-class object is created and destroyed?**

1. Space is allocated (on the stack or the heap) for the full object (that is, enough space to store the data members inherited from the base class plus the data members defined in the derived class itself)
2. The base class's constructor is called to initialize the data members inherited from the base class
3. The derived class's constructor is then called to initialize the data members added in the derived class
4. The derived-class object is then usable
5. When the object is destroyed (goes out of scope or is deleted) the derived class's destructor is called on the object first
6. Then the base class's destructor is called on the object
7. Finally the allocated space for the full object is reclaimed

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

class C1
{
public:
   C1 ();
   C1(int n);
   ~C1();
protected:
   int *pi;
   int intNum;
};

C1::C1() {
   intNum = 10;
   cout << "C1 default constructor, ";
   cout << intNum << " integers are allocated"
<< endl;
   pi = new int [intNum];

}
C1::C1(int n) : intNum(n)  {
   cout << intNum << " integers are allocated"
<< endl;
   pi = new int [intNum];
}

C1::~C1()  {
   cout << intNum << " integers are released"<<
endl;
  delete [] pi;
}

class C2 : public C1
{
public :
   C2 ();
   C2(int n);
   ~C2();
private:
   char *pc;
   int charNum=10;
};

C2::C2() : C1() {
   cout << "C2 default constructor, 10 characters
allocated" << endl;
   pc = new char [charNum];
}

C2::C2(int n) : C1(n), charNum (n)
{
   cout << charNum << " characters are
allocated."<< endl;
   pc = new char [charNum];
}

C2::~C2() {
   cout << charNum  << " characters are
released"<< endl;
   delete [] pc;
}

int main () {

   cout << "First object " << endl;
   C2   exampleObj;

   cout << endl << endl;
   cout << "Second object " << endl;

   C2   array(30);
   cout << endl << endl << "Objects exit the
scope ... " << endl;

   return 0;
}
```

Public Inheritance Example

```
//FILE: mam1.h
#ifndef MAM_H
#define MAM_H
class mammal
{
public:
        mammal();
        mammal (int W, int H, int A);
        ~mammal();
        int returnWeight() const;
        int returnHeight() const;
        void speak() const;
protected:
        int weight;
        int height;
        int age;
};
#endif

//FILE: mam1.cpp
#include "mam1.h"
#include <iostream>
using namespace std;

mammal::mammal(){
        weight=1;
        height=1;
        age=1;

        cout <<"Mammal's default
        constructor" << endl;
}

mammal::mammal(int W, int H, int A){
        weight=W;
        height=H;
        age = A;
        cout <<"Executing mammal's int
        constructor\n";
}

mammal::~mammal(){
        cout <<"Executing mammal's
                destructor\n";
}

int mammal:: returnWeight() const{
        return (weight);
}

int mammal::returnHeight() const{
        return (height);
}

void mammal::speak()const {
        cout <<"Mammal is speaking"
        << endl;
}

//FILE:  dog1.h
#include "mam1.h"
#ifndef DOG_H
#define DOG_H

enum BREED {LAB, COLLIE,
DACHOUND, COCKER};

class dog : public mammal
{
public:
        dog();
        dog(int H, int W, int A, BREED  B);

        ~dog();
        void printBreed() const;
        void speak()const;
private:
        BREED itsBreed;
};
#endif

//FILE: dog1.cpp
#include "dog1.h"
#include <iostream>
using namespace std;

dog::dog(){
        cout <<"Executing dog's default
        constructor\n";
        itsBreed=COLLIE;
}

dog::dog(int W, int H, int A, BREED B) :
mammal(W, H, A), itsBreed(B) {
        cout << "Executing dog's int
        constructor""<<endl;
}
```

```
dog::~dog()  {
        cout <<"Executing dog
destructor\n";
 }

void dog::printBreed() const  {
        cout <<"The breed is " ;
        switch (itsBreed)
        {
        case COLLIE:
                cout <<"COLLIE"<<endl;
                break;
        case DACHOUND:
                cout <<"Dachound"<<endl;
                break;
        case COCKER:
                cout <<"COCKER"
                <<endl;
                break;
        case LAB:
                cout <<"LABRADOR
                RETRIEVER"<<endl;
        }
}

void dog::speak()const {
        cout <<"BOW-WOW" << endl;
}
```

```
//FILE maindog1.cpp
#include <iostream>
#include "mam1.h"
#include "dog1.h"
using namespace std;

int main() {
        mammal Animal;
        dog MyDog(2, 10, 2, COLLIE);

        cout << My dog's weight is " <<
        MyDog.returnWeight() <<
        "pounds"<<endl;

        cout <<"My dog's height is " <<
        MyDog.returnHeight() << "
        inchese"<<endl;

        cout <<"The animal's weight is " <<
        Animal.returnWeight() <<
        "pounds"<<endl;

        cout <<"My dog says :\t";
        MyDog.speak();

        cout <<"Animal says :\t";
        Animal.speak();

        return 0;
}
```

==========

Program Output:

==========

```
Mammal's default constructor
Mammal's int constructor
Executing dog's int constructor
My dog's weight is 2  pounds
My dog's height is 10 inches
The animal's weight is 1 pounds
My dog says :   BOW-WOW
Animal says :   Mammal is speaking
Executing dog destructor
Executing mammal's destructor
Executing mammal's destructor
```

Developing an inherited class from a base class:
base class (rectangle class) → derived class (boxClass)

```
class rectangleClass
{
public:
        rectangleClass();
        rectangleClass(double l, double w);
        void setDeminsion(double l, double w);
        double getLength() const;
        double getWidth() const;
        double area () const;
        double perimeter() const;
        void print() const;
private :
        double length ;
        double width ;
} ;

class boxClass ::public rectangleClass
{
public :
        boxClass () ;
        boxClass(double l, double w, double h) ;
        void setDimension(double l, double w, double h) ;
        double getHeight() const ;
        double area() const ;
        double volume() const ;
        void print() const ;
private :
        double height ;
} ;
```

define in implementation file:

- default constructor
- value constructor
- setDimension method → redefinition
- area method → redefinition
- getHeight
- volume method
- print method → redefinition

Client program :

```
#include both header files
int main()
{
        rectangleClass myRect1;
        rectangleClass myRect2(8, 6);
        boxClass myBox1;
        boxClass myBox2(10, 7, 3);
```

```
        cout << “my rectangle 1:” << endl;
        myRect1.print();
        cout << endl;
        cout << “my rectangle 1’s area is : “ << myRect1.area() << endl;


        cout << “my rectangle 2:” << endl;
        myRect2.print();
        cout << endl;
        cout << “my rectangle 2’s area is : “ << myRect2.area() << endl;


        cout << “my box 1:” << endl;
        myBox1.print();
        cout << endl;
        cout << “my box 1’s area is : “ << myBox1.area() << endl;
        cout << “my box 1’s volume is : “ << myBox1.volume() << endl;

        cout << “my box 2:” << endl;
        myBox2.print();
        cout << endl;
        cout << “my box 2’s area is : “ << myBox2.area() << endl;
        cout << “my box 2’s volume is : “ << myBox2.volume() << endl;

        return 0;

}
```