CSC 211: Computer Programming

Class Inheritance and Polymorphism

Michael Conti

Department of Computer Science and Statistics
University of Rhode Island

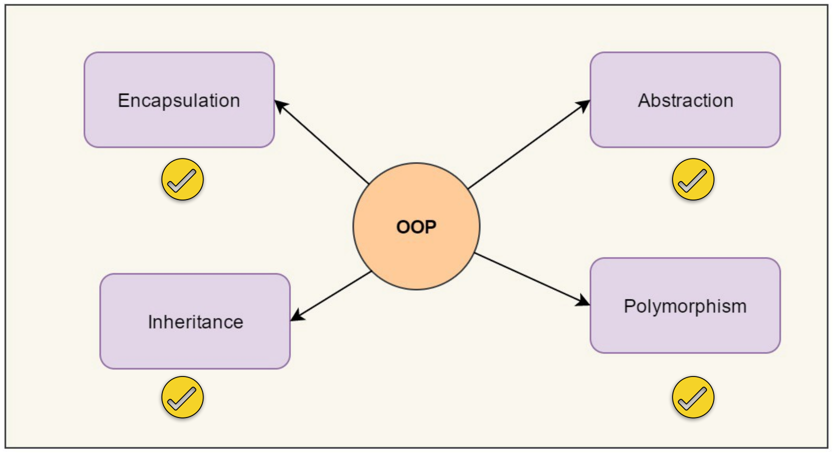
Spring 2025

THINK BIG WE DO℠

Administrative notes

Administrative notes

- Last lecture optional
  - ✓ Free forum
    - Answer any questions you have on any topic we've covered.
- Final Exam - 05/06 @ 3p - 5p



Four Pillars of Object Oriented Programming

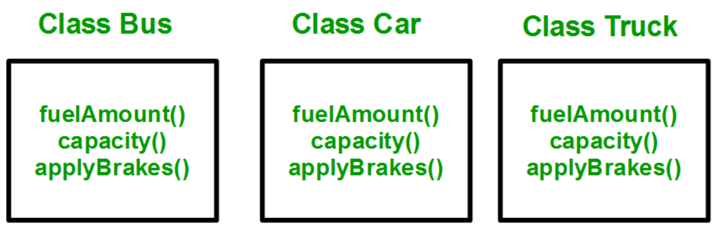
Inheritance in C++

- The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important feature of Object Oriented Programming.
- **Derived Class:** The class that inherits properties from another class is called Sub class / Derived Class / Child Class.
- **Base Class:**The class whose properties are inherited by sub class is called Base Class / Super class / Parent Class.
- Derived class is a *superset* of the base class.

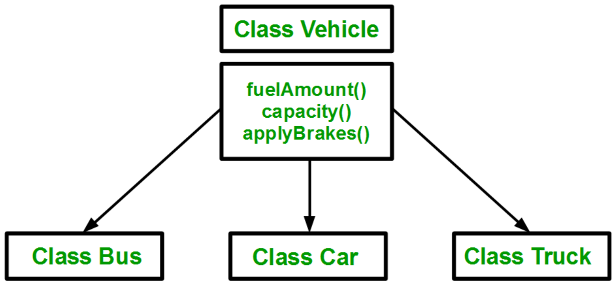
Inheritance in C++

- What if we create a stand alone function that accepts an object from the base class as an argument. Could we pass in a derived class object instead?
- Yes! The derived class object has everything a base class object would have (and maybe more)!

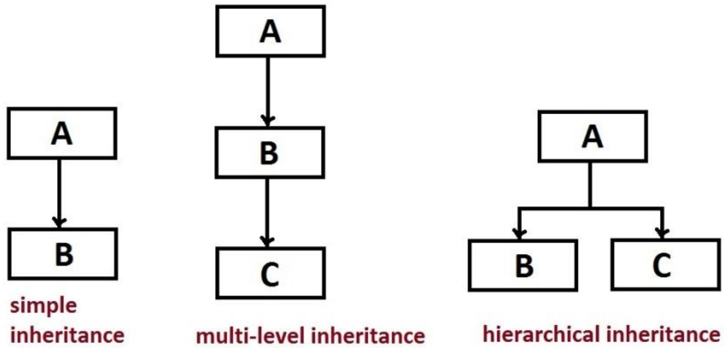
Why and When to use Inheritance?



Note: duplication of same code 3 times



Note: by using inheritance, we can avoid the duplication of data and increase re-usability of code



Implementing Inheritance

Syntax

```
class subclass_name : access_mode base_class_name
{
  //body of subclass
};
```

Modes of inheritance

Modes of inheritance

- **Public mode**: If we derive a sub class from a public base class then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.
- **Protected mode**: If we derive a sub class from a Protected base class then both public member and protected members of the base class will become protected in derived class.
- **Private mode**: If we derive a sub class from a Private base class then both public member and protected members of the base class will become Private in derived class.

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

- Protected members act like private members, but are able to be inherited

Example

```
// C++ Implementation to show that a derived class
// doesn’t inherit access to private data members.
// However, it does inherit a full parent object
class A {

public:
    int x;

protected:
    int y;

private:
    int z;

};

class B : public A {

    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A {

    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A {    // 'private' is default for classes

    // x is private
    // y is private
    // z is not accessible from D
};
```

Order of Constructor Call with Inheritance

Order of Constructor Call

- Base class constructors are **always called** in the derived class constructors.
- Whenever you create a derived class object, **first** the base class default constructor is executed and **then** the derived class's constructor finishes execution.

Order of Constructor Call

```cpp
class Base
{
    public:

    int x;
    // default constructor
    Base()
    {
        std::cout << "Base default constructor\n";
    }
};

class Derived : public Base
{

    public:

    int y;
    // default constructor
    Derived()
    {
        std::cout << "Derived default constructor\n";
    }
    // parameterized constructor
    Derived(int i)
    {
        std::cout << "Derived parameterized constructor\n";
    }
};
```

Order of Constructor / Destructor Call

```cpp
int main()
{
    Base b;
    Derived d1;
    Derived d2(10);
}
```

OUTPUT

```
Base default constructor
Base default constructor
Derived default constructor
Base default constructor
Derived parameterized constructor

Derived default destructor
Base default destructor
Derived default destructor
Base default destructor
Base default destructor
```

Inheritance Example

Example

```
class Entity{
    public:
       float x, y;

       void move(float xa, float ya){
           x += xa;
           y += ya;
       }

       void printLoc(){
           std::cout << "x = " << x << std::endl;
           std::cout << "y = " << y << std::endl;
       }
};
```

Example

```
class Player{

    public:
       const char* name;
       float x, y;

    void move(float xa, float ya){
           x += xa;
           y += ya;
       }

    void printLoc(){
           std::cout << "x = " << x << std::endl;
           std::cout << "y = " << y << std::endl;
       }

    void printName(){

        std::cout << name << std::endl;
    }

};
```

Example

```
class Player : public Entity{

    public:
       const char* name;

    void printName(){

        std::cout << name << std::endl;
    }
};
```

Polymorphism

Polymorphism

- Polymorphism in C++ means, the same entity (function or object) behaves differently in different scenarios.
- Consider this example:
- The " +" operator in c++ can perform two specific functions at two different scenarios

Example

```
int a = 6;
int b = 6;
int sum = a + b;   // sum =12
```

v.s

```
string firstName = "John";
string lastName = "Doe";

// name = "John Doe "
string name = firstName + lastName;
```

Types of Polymorphism in C++

- Polymorphism in C++ is categorized into two types
- **Compile Time Polymorphism**
  - ✓ Function overloading
  - ✓ Operator overloading
- **Runtime (Dynamic) Polymorphism**
  - ✓ Function overriding
  - ✓ Virtual functions

Compile Time Polymorphism

- Function overloading
  - ✓ One function can perform many tasks.
  - ✓ A single function is used to perform many tasks with the same name and different types of arguments.
  - ✓ Correct overloaded function will be called at compile time based on argument type

Function overloading

```
class Addition {
public:
    int ADD(int X,int Y)   // Function with parameter
    {
        return X+Y;     // this function is performing addition of  two int values
    }

    int ADD() {              // Function with same name but without parameter
        string a= "HELLO";
        string b="SAM";   // in this function concatenation is performed
       string c= a+b;
       cout<<c<<endl;

    }
};

int main() {
    Addition obj;   // Object is created
    cout<<obj.ADD(128, 15)<<endl; //first method is called
    obj.ADD();  // second method is called
    return 0;
}
```

Compile Time Polymorphism

- **Operator Overloading**
  - ✓ Defining additional semantic behavior to operators
  - ✓ The purpose of operator overloading is to provide a special meaning to the user-defined data types.
  - ✓ The advantage of operator overloading is to perform different operations on the same operand.

Operator overloading

The parameter 'A a' represents the second operand in the addition (the right-hand side of the + operator).

```
#include <iostream>
using namespace std;
class A
{
  public:
    string x;

    A(){}
    A(string i)
    {
       x=i;
    }
    void operator+(A);
    void display();
};
```

```
void A::operator+(A a)
{

    string m = x+a.x;
    cout<<"The result of the addition
                of two objects is : "<<m;

}
int main()
{
    A a1("Welcome");
    A a2("back");
    a1+a2;
    return 0;
}
```

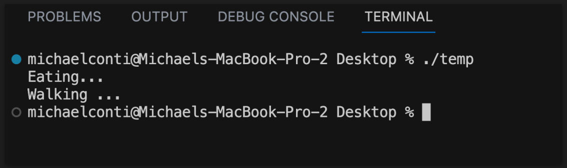
Run Time (Dynamic) Polymorphism

- **Run Time (Dynamic) Polymorphism**
- Function overriding
  - ✓ Give the new definition to base class function in the derived class. It can be only possible to override a function from the 'derived class'.
  - ✓ Have two definitions of the same function, one in the superclass and one in the derived class. The decision about which function definition requires calling happens at runtime.
  - ✓ That is the reason we call it 'Runtime polymorphism'.

Function overriding

```
class Animal {
    public:
    void function()
        {cout<<"Eating..."<<endl;}
};


class Man: public Animal
{
 public:
    void function() override //override keyword optional
        { cout<<"Walking ..."<<endl; }
};
```

```
int main() {

        Animal A;
        A.function();//parent class object

        Man m;
        m.function();// child class object

    return 0;
}
```



```
michaelconti@Michaels-MacBook-Pro-2 Desktop % ./temp
Eating...
Walking ...
michaelconti@Michaels-MacBook-Pro-2 Desktop % 
```

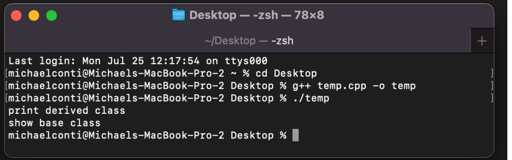
Run Time Polymorphism

- Virtual Functions
  - ✓ Virtual functions ensure that the correct function is called for an object, **regardless of the type of reference (or pointer) used for function call**.
  - ✓ The actual function that gets executed is based on the object the pointer or reference points to.
  - ✓ Functions are declared with a virtual keyword in base class.
  - ✓ The resolving of function call is done at runtime.

Virtual Functions

```
class base {
public:
    virtual void print()
    {
        cout << "print base class\n";
    }

    void show()
    {
        cout << "show base class\n";
    }
};


class derived : public base {
public:
    void print()
    {
        cout << "print derived class\n";
    }

    void show()
    {
        cout << "show derived class\n";
    }
};
```

```
int main()
{
    base *basePtr;
    derived d;
    basePtr = &d;

    // Virtual function bypassed, binded at runtime
    // Object pointed to is of type 'derived'
    basePtr->print();

    // Non-virtual function, binded at compile time
    // Pointer is of type 'base'
    basePtr->show();

    return 0;
}
```



```
Last login: Mon Jul 25 12:17:54 on ttys000
michaelconti@Michaels-MacBook-Pro-2 ~ % cd Desktop
michaelconti@Michaels-MacBook-Pro-2 Desktop % g++ temp.cpp -o temp
michaelconti@Michaels-MacBook-Pro-2 Desktop % ./temp
print derived class
show base class
michaelconti@Michaels-MacBook-Pro-2 Desktop % 
```

Practice

- Write a base Person class with following properties and methods
- Person (base):
  Member Variables: name, age, favorite color, birthday
- **Derive a Student from person** and an **Employee class derives student** with the respective additional attributes
- Student: GPA, Major, Year, StudentID
  Employee: Job Title, Salary, Years Employed
  - Print for employee class

https://www.geeksforgeeks.org/inheritance-in-c/