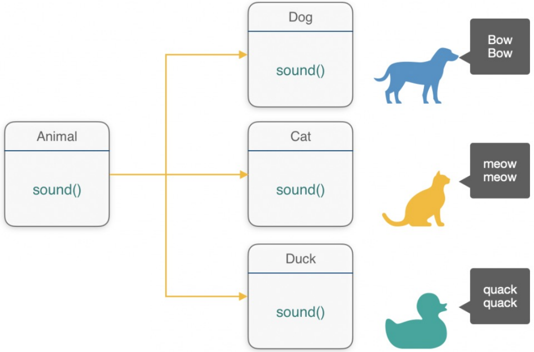
Inheritance

Prof. Seokin Hong

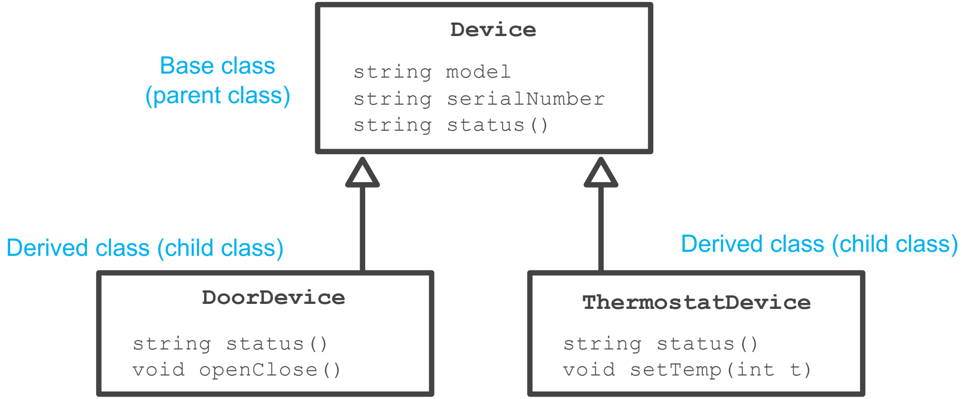
SUNGKYUNKWAN
UNIVERSITY

Inheritance Basics



Inheritance Basics

- **Inheritance is the process that creates a derived class from a base class**
  - Derived class automatically has all the member variables and functions of the base class
  - Derived class can have additional member variables and/or member functions
  - Derived class is a child of the base class (parent class)
- **Example**



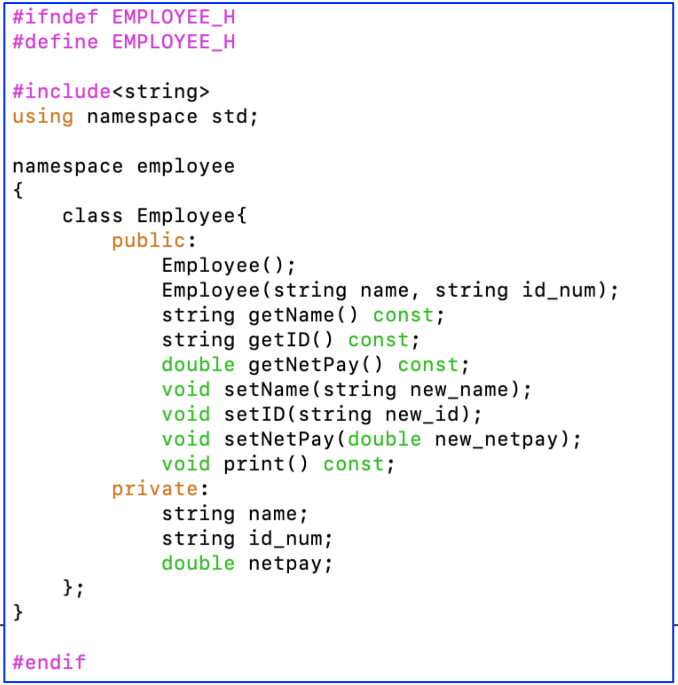
Example: Employee Classes

- **To design a program for maintaining employees records**
  - Employees belong to a class of people who share the property "employee”
    - **Salaried employee:** Employees with a fixed wage
    - **Hourly employee:** Employees with hourly wages

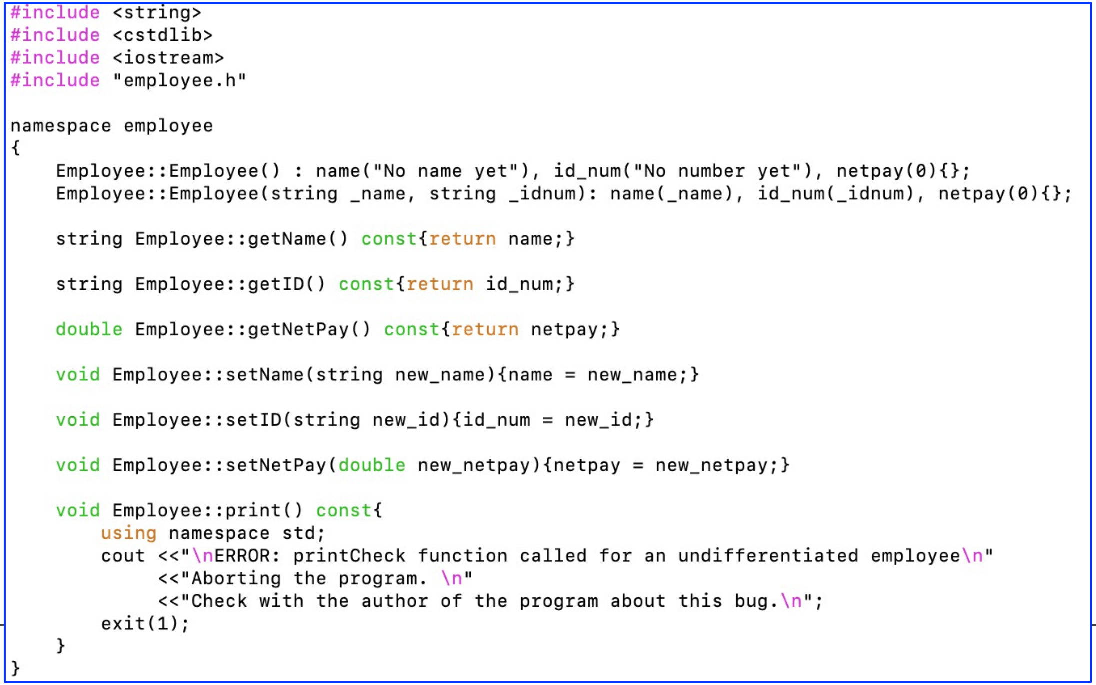
- **All employees have a name and ID number**
  - Functions to manipulate name and ID number are the same for hourly and salaried employees

Example: Employee Classes (cont.)

- **Base Class**
  - We will define a class called **Employee** for all employees
  - **Employee** class will be used to define classes for hourly and salaried employees



```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include<string>
using namespace std;

namespace employee
{
    class Employee{
        public:
            Employee();
            Employee(string name, string id_num);
            string getName() const;
            string getID() const;
            double getNetPay() const;
            void setName(string new_name);
            void setID(string new_id);
            void setNetPay(double new_netpay);
            void print() const;
        private:
            string name;
            string id_num;
            double netpay;
    };
}

#endif
```

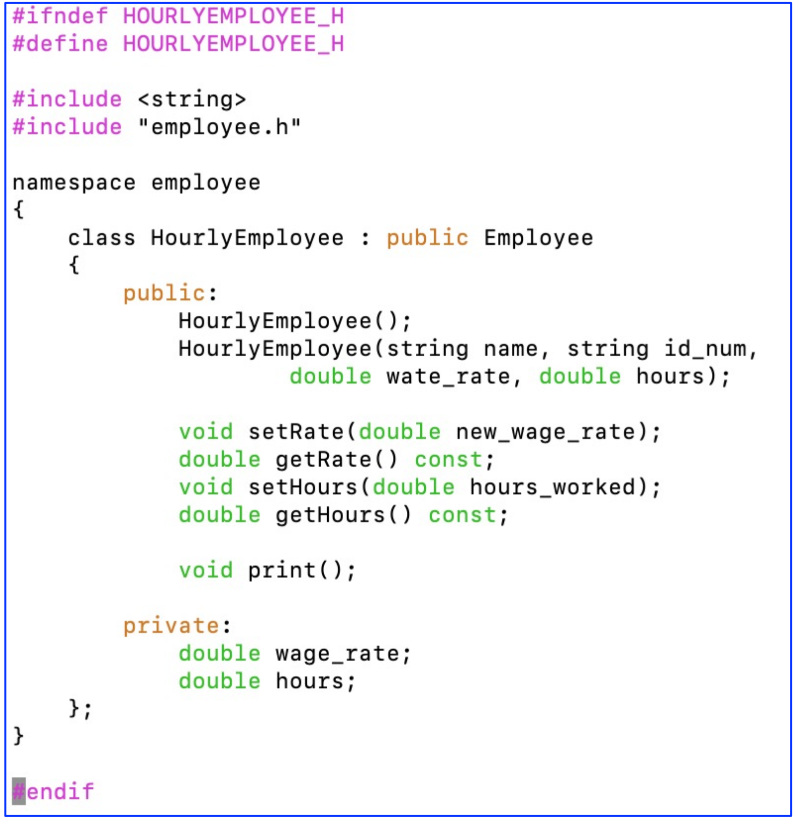


```
#include <string>
#include <cstdlib>
#include <iostream>
#include "employee.h"

namespace employee
{
    Employee::Employee() : name("No name yet"), id_num("No number yet"), netpay(0){};
    Employee::Employee(string _name, string _idnum): name(_name), id_num(_idnum), netpay(0){};

    string Employee::getName() const{return name;}

    string Employee::getID() const{return id_num;}

    double Employee::getNetPay() const{return netpay;}

    void Employee::setName(string new_name){name = new_name;}

    void Employee::setID(string new_id){id_num = new_id;}

    void Employee::setNetPay(double new_netpay){netpay = new_netpay;}

    void Employee::print() const{
        using namespace std;
        cout <<"\nERROR: printCheck function called for an undifferentiated employee\n"
             <<"Aborting the program. \n"
             <<"Check with the author of the program about this bug.\n";
        exit(1);
    }
}
```

Example: Employee Classes (cont.)

- **HourlyEmployee**
  - Derived from Class Employee
  - HourlyEmployee inherits all member functions and member variables of Employee
  - The class definition
    class HourlyEmployee : public Employee

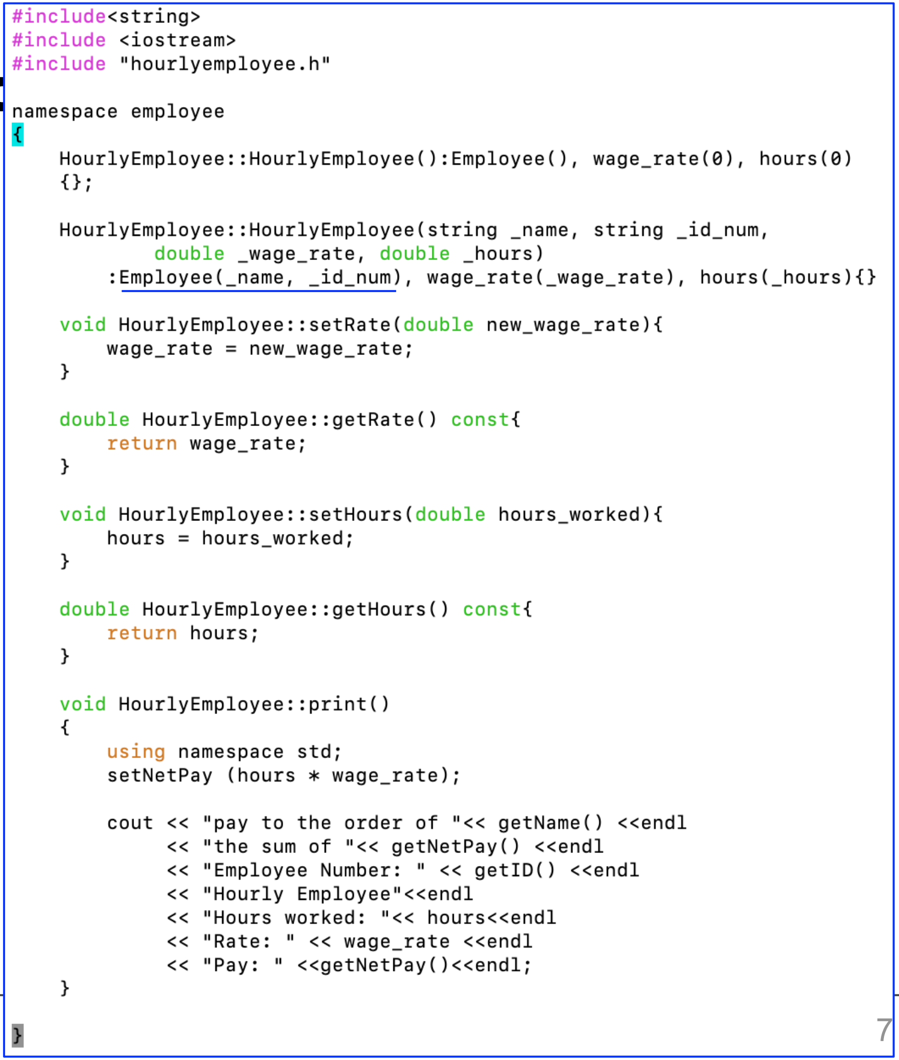
    shows that HourlyEmployee is derived from class Employee
  - HourlyEmployee declares additional member variables and functions



```cpp
#ifndef HOURLYEMPLOYEE_H
#define HOURLYEMPLOYEE_H

#include <string>
#include "employee.h"

namespace employee
{
    class HourlyEmployee : public Employee
    {
        public:
            HourlyEmployee();
            HourlyEmployee(string name, string id_num,
                    double wate_rate, double hours);

            void setRate(double new_wage_rate);
            double getRate() const;
            void setHours(double hours_worked);
            double getHours() const;

            void print();

        private:
            double wage_rate;
            double hours;
    };
}

#endif
```

Example: Employee Classes (c

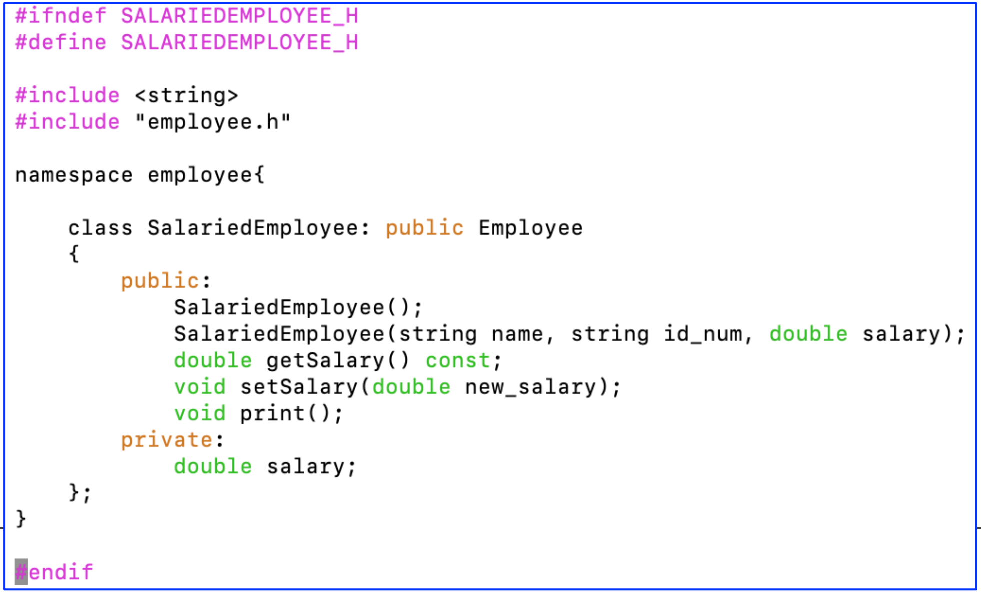
- **Implementing a Derived Class**
  - Any member functions added in the derived class are defined in the implementation file for the derived class
    - setRate()
    - getRate()
    - setHours()
    - getHours()
  - Some functions are redefined
    - print()
  - Definitions are not given for inherited functions that are not to be changed



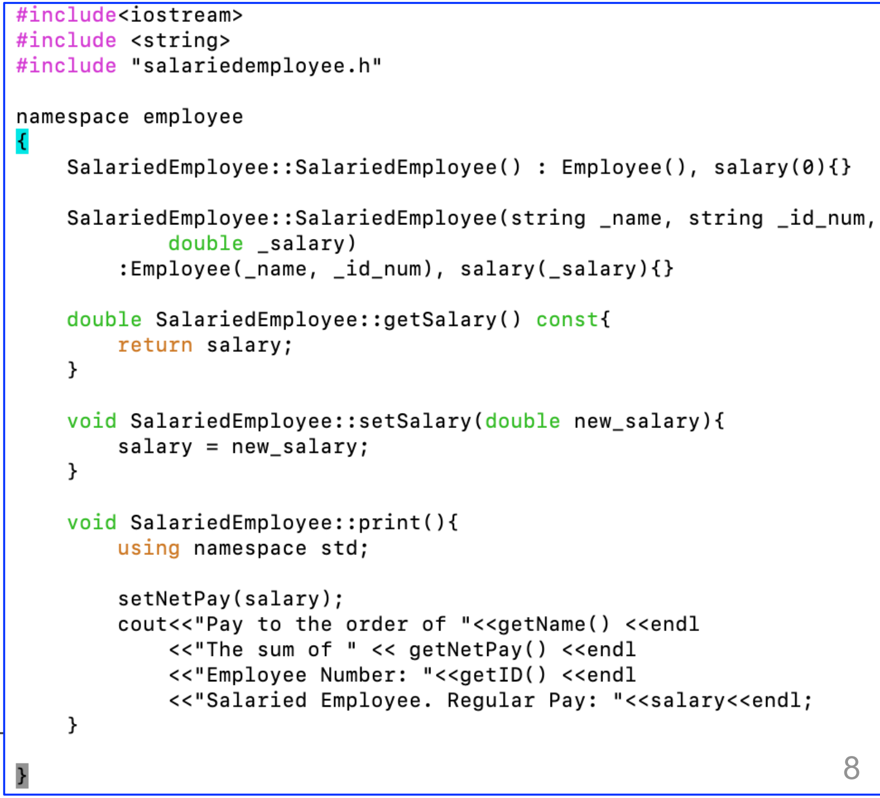
```cpp
#include<string>
#include <iostream>
#include "hourlyemployee.h"

namespace employee
{
    HourlyEmployee::HourlyEmployee():Employee(), wage_rate(0), hours(0)
    {};

    HourlyEmployee::HourlyEmployee(string _name, string _id_num,
            double _wage_rate, double _hours)
        :Employee(_name, _id_num), wage_rate(_wage_rate), hours(_hours){}

    void HourlyEmployee::setRate(double new_wage_rate){
        wage_rate = new_wage_rate;
    }

    double HourlyEmployee::getRate() const{
        return wage_rate;
    }

    void HourlyEmployee::setHours(double hours_worked){
        hours = hours_worked;
    }

    double HourlyEmployee::getHours() const{
        return hours;
    }

    void HourlyEmployee::print()
    {
        using namespace std;
        setNetPay (hours * wage_rate);

        cout << "pay to the order of "<< getName() <<endl
             << "the sum of "<< getNetPay() <<endl
             << "Employee Number: " << getID() <<endl
             << "Hourly Employee"<<endl
             << "Hours worked: "<< hours<<endl
             << "Rate: " << wage_rate <<endl
             << "Pay: " <<getNetPay()<<endl;
    }

}
```

Example: Employee Classes (cont.)

- **SalariedEmployee**
  - Also derived from Employee class
  - Function print() is redefined
  - Add member variables and functions



```
#ifndef SALARIEDEMPLOYEE_H
#define SALARIEDEMPLOYEE_H

#include <string>
#include "employee.h"

namespace employee{

    class SalariedEmployee: public Employee
    {
        public:
            SalariedEmployee();
            SalariedEmployee(string name, string id_num, double salary);
            double getSalary() const;
            void setSalary(double new_salary);
            void print();
        private:
            double salary;
    };
}

#endif
```



```
#include<iostream>
#include <string>
#include "salariedemployee.h"

namespace employee
{
    SalariedEmployee::SalariedEmployee() : Employee(), salary(0){}

    SalariedEmployee::SalariedEmployee(string _name, string _id_num,
            double _salary)
        :Employee(_name, _id_num), salary(_salary){}

    double SalariedEmployee::getSalary() const{
        return salary;
    }

    void SalariedEmployee::setSalary(double new_salary){
        salary = new_salary;
    }

    void SalariedEmployee::print(){
        using namespace std;

        setNetPay(salary);
        cout<<"Pay to the order of "<<getName() <<endl
            <<"The sum of " << getNetPay() <<endl
            <<"Employee Number: "<<getID() <<endl
            <<"Salaried Employee. Regular Pay: "<<salary<<endl;
    }

}
```

Parent and Child Classes

- **Child class automatically has all the members of the parent class**
- **Parent class contains all the code common to the child classes**
  - You do not have to re-write the code for each child

- **An object of a class type can be used wherever any of its ancestors can be used**
  - An object of type HourlyEmployee can be used where an object of type Employee can be used
- **An ancestor cannot be used wherever one of its descendents can be used**

Derived Class Constructors

- **A base class constructor is not inherited in a derived class**
  - The base class constructor can be invoked by the constructor of the derived class
  - The constructor of a derived class begins by invoking the constructor of the base class

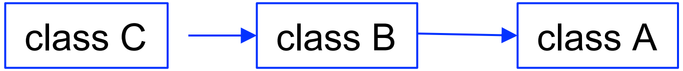
    - HourlyEmployee::HourlyEmployee : Employee( ), wageRate(0), hours( )
      { //no code needed }

Any Employee constructor could be invoked

Default Initialization

- **If a derived class constructor does not invoke a base class constructor explicitly, the base class default constructor will be used**

- **If class B is derived from class A and class C is derived from class B**
  - When a object of class C is created,
    - The base class A's constructor is the first invoked
    - Class B's constructor is invoked next
    - C's constructor completes execution



Private is Private

- **A member variable (or function) that is private in the parent class is not accessible to the child class**
  - Parent class member functions must be used to access the private members of parent
  - This code would be illegal:

```
void HourlyEmployee::print( )
    {
          netpay = hours * wage_rate;
           .........
```

→ netpay is a private member of Employee!

protected Qualifier

- **protected members of a class appear to be private outside the class, but are accessible by derived classes**
  - If variables name, netpay, and id_num are listed as protected in the Employee class
    - this code becomes legal:

```
    void HourlyEmployee::print( )
        {
                netpay = hours * wage_rate;
                 .........
```

Redefining (Overriding) vs Overloading

- **A function redefined (overrided) in a derived class has the same number and type of parameters**
  - The derived class has only one function with the same name as the base class

- **An overloaded function has a different number and/or type of parameters**
  - The derived class has two functions with the same name as the base class

Access to a Redefined Base Function

- **When a base class function is redefined in a derived class, the base class function can still be used**
  - To specify that you want to use the base class version of the redefined function:

```
HourlyEmployee sallyH;
sallyH.Employee::print( );
```

Inheritance Details

Inheritance Details

- **Some special functions are not inherited by a derived class**
  - Some of the special functions that are not effectively inherited by a derived class include
    - Destructors
    - Copy constructors
    - Assignment operator

Copy Constructors and Operator =

- **If a copy constructor is not defined in a derived class, C++ will generate a default copy constructor**
  - Default copy constructor copies only the contents of member variables and will not work with pointers and dynamic variables
  - The base class copy constructor is not used in the derived class

- **Operator =**
  - If a base class has a defined assignment operator = and the derived class does not,
    - C++ will use a default operator =
    - Assignment operator of the base class is not used
  - Usually use the assignment operator from the base class in the definition of the derived class's assignment operator

The Copy Constructor

- **Example**

```
Derived::Derived(const Derived& object)
                    :Base(object), <other initializing>
 {…}
```

  - Invoking the base class copy constructor sets up the inherited member variables

The Operator = Implementation

- Example

```
Derived& Derived::operator= (const Derived& rhs)
{
    Base::operator=(rhs)

    ……
```

  - This line handles the assignment of the inherited member variables by calling the base class assignment operator

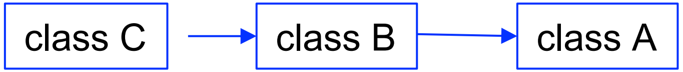
  - The remaining code would assign the member variables introduced in the derived class

Destructors in Derived Classes

- **The derived class should define its own destructor**
  - A destructor is not inherited by a derived class

- **When the destructor for a derived class is called, the destructor for the base class is automatically called**
  - Derived class destructor need only delete dynamic variables added in the derived class

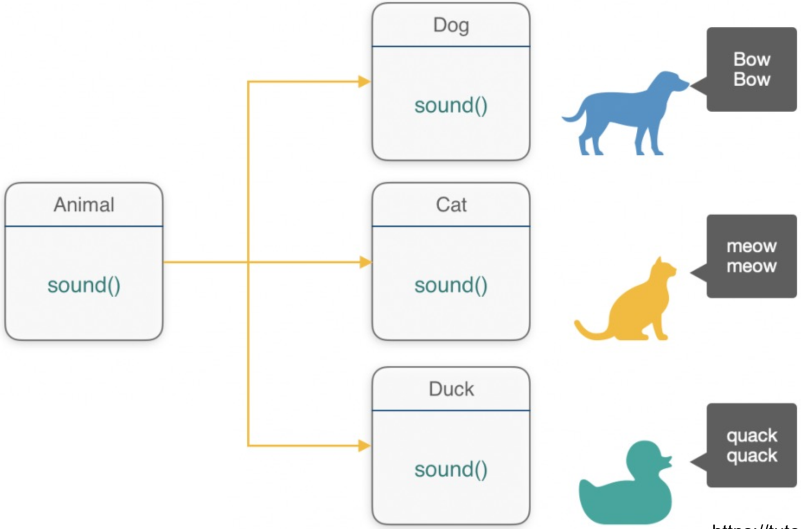
- **If class B is derived from class A and class C is derived from class B…**
  - When an object of class C goes out of scope
    - The destructor of class C is called
    - Then the destructor of class B
    - Then the destructor of class A



Polymorphism

Polymorphism

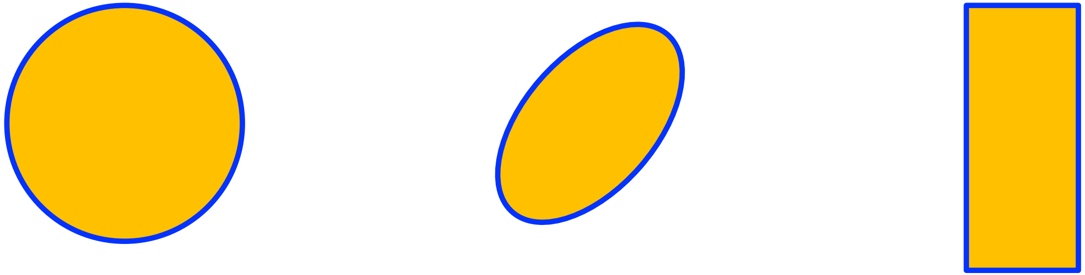
- Polymorphism means "many form"
- refers to the ability to associate multiple meanings with one function name using a mechanism called late binding



https://tutorial.eyehunts.com/java/java-polymorphism-definition-type-example/

A Late Binding Example

- **Imagine a graphics program with several types of figures**
  - Each figure may be an object of a different class, such as a Circle, Oval, Rectangle, etc.
  - Each is a descendant of a class Figure
  - Each has a function draw( ) implemented with code specific to each shape
  - Class Figure has functions common to all figures



A Late Binding Example

- **A Problem**
  - Suppose that class Figure has a function center()
    - center() moves a figure to the center of the screen by erasing the figure and redrawing it in the center of the screen
    - center() is inherited by each of the derived classes
      - center() uses each derived object's draw function to draw the figure
      - The Figure class does not know about its derived classes, so it cannot know how to draw each figure

```
Figure* A[2];

A[0] = new Circle();
A[1] = new Rectangle();

for (int i=0; i<2; i++)
                    A[i]->center();
```

Virtual Functions

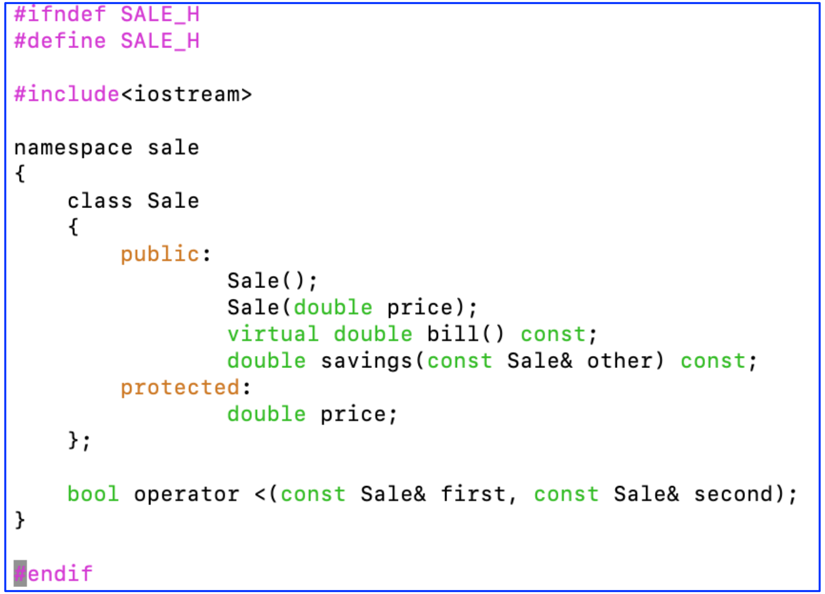
- **Because the Figure class includes a method to draw figures, but the Figure class cannot know how to draw the figures**

- **Making a function virtual tells the compiler that you don't know how the function is implemented and to wait until the function is used in a program, then get the implementation from the object.**

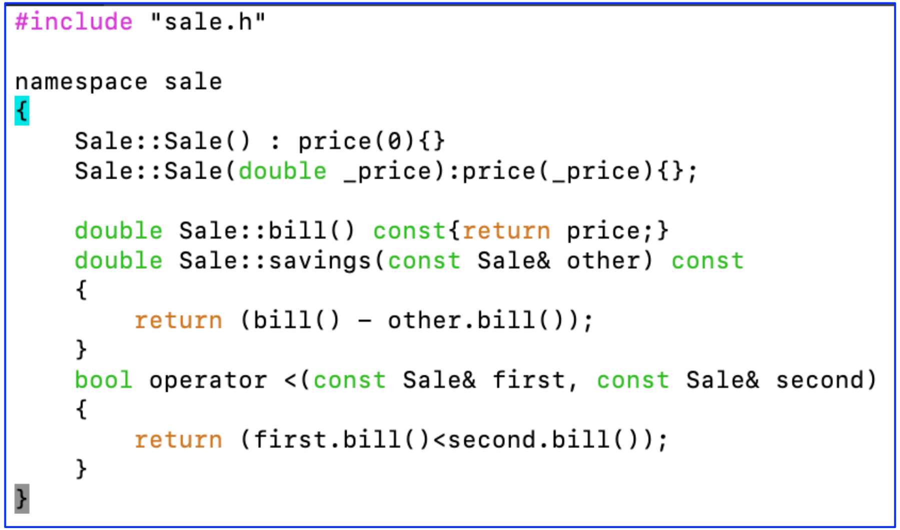
  → late binding

Example: Sale Classes

- **Class Sale**
  - Base class



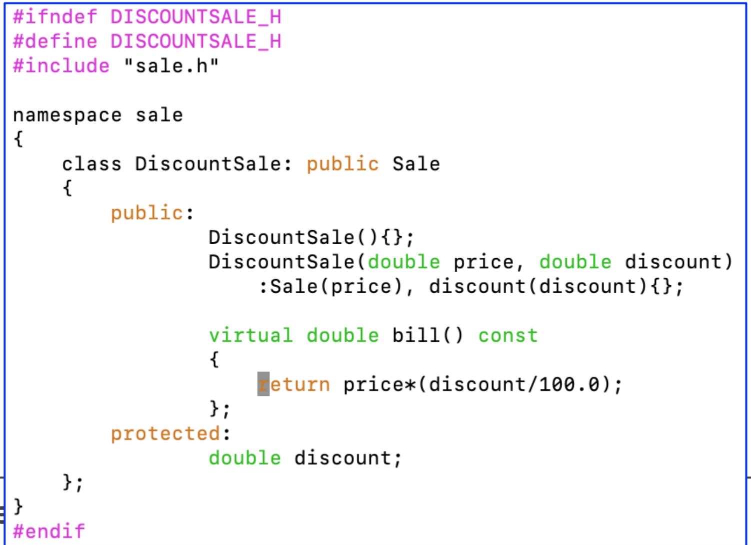
```
#ifndef SALE_H
#define SALE_H

#include<iostream>

namespace sale
{
    class Sale
    {
        public:
                Sale();
                Sale(double price);
                virtual double bill() const;
                double savings(const Sale& other) const;
        protected:
                double price;
    };

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

#endif
```



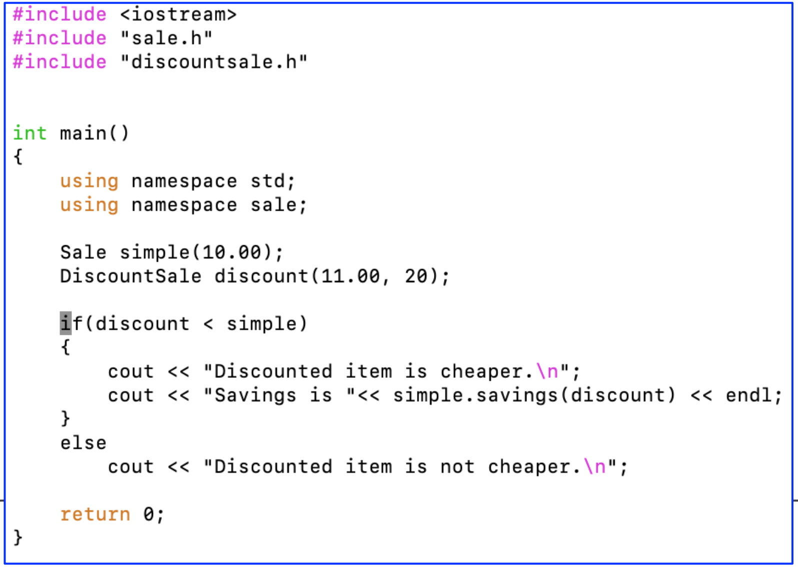
```
#include "sale.h"

namespace sale
{
    Sale::Sale() : price(0){}
    Sale::Sale(double _price):price(_price){};

    double Sale::bill() const{return price;}
    double Sale::savings(const Sale& other) const
    {
        return (bill() - other.bill());
    }
    bool operator <(const Sale& first, const Sale& second)
    {
        return (first.bill()<second.bill());
    }
}
```

Example: Sale Classes (cont.)

- **Class DiscountSale has its own version of virtual function bill()**
  - Even though class Sale is already compiled,
    Sale::savings( ) and Sale::operator< can still use function bill() from the DiscountSale class
  - The keyword virtual tells C++ to wait until bill() is used in a program to get the implementation of bill from the calling object



```
#ifndef DISCOUNTSALE_H
#define DISCOUNTSALE_H
#include "sale.h"

namespace sale
{
    class DiscountSale: public Sale
    {
        public:
                DiscountSale(){};
                DiscountSale(double price, double discount)
                    :Sale(price), discount(discount){};

                virtual double bill() const
                {
                    return price*(discount/100.0);
                };
        protected:
                double discount;
    };
}
#endif
```



```
#include <iostream>
#include "sale.h"
#include "discountsale.h"


int main()
{
    using namespace std;
    using namespace sale;

    Sale simple(10.00);
    DiscountSale discount(11.00, 20);

    if(discount < simple)
    {
        cout << "Discounted item is cheaper.\n";
        cout << "Savings is "<< simple.savings(discount) << endl;
    }
    else
        cout << "Discounted item is not cheaper.\n";

    return 0;
}
```

Example: Sale Class

- **Virtual Function bill()**
  - Because **bill()** is virtual in class Sale, **savings()** and **operator <**, defined only in the base class, can in turn use a version of bill found in a derived class
  - When a DiscountSale object calls **savings()** function, defined only in the base class, function **savings()** calls function **bill()**
  - Because **bill()** is a virtual function in class Sale, C++ uses the version of **bill()** defined in the object that called **savings()**

Virtual Details

- **To define a function differently in a derived class and to make it virtual**
  - Add keyword virtual to the function declaration in the base class
  - virtual is not needed for the function declaration in the derived class
  - virtual is not added to the function definition
  - virtual functions require considerable overhead so excessive use reduces program efficiency

Slicing Problem

- **Consider**

```
class Pet
{
      public:
           virtual void print();
           string name;
}


class Dog :public Pet
{
      public:
            virtual void print();
            string breed;
}
```

Slicing Problem (cont.)

- **Slicing Problem**
  - It is legal to assign a derived class object into a base class variable
    - This slices off data in the derived class that is not also part of the base class
    - Member functions and member variables are lost
  - Example
    - C++ allows the following assignments:

```
vdog.name = "Tiny";
vdog.breed = "Great Dane";
vpet = vdog;
```

    - However, vpet will loose the breed member of vdog since an object of class Pet has no breed member

cout << vpet.breed; ← This code would be illegal:

Extended Type Compatibility

- **It is possible in C++ to avoid the slicing problem**
  - Using pointers to dynamic variables we can assign objects of a derived class to variables of a base class without loosing members of the derived class object
  - If the domain type of the pointer p_ancestor is a base class for the domain type of pointer p_descendant, the following assignment of pointers is allowed

    p_ancestor = p_descendant;

    and no data members will be lost

Dynamic Variables and Derived Classes

- **Example:**

```
Pet   *ppet;
Dog *pdog;
pdog = new Dog;
pdog->name = "Tiny";
pdog->breed = "Great Dane";
ppet = pdog;
```

```
void Dog::print( )
{
   cout << "name: "
            <<  name << endl;
   cout << "breed: "
            << breed << endl;
}
```

  - ppet->print( );   is legal and produces:

```
name:  Tiny
breed:  Great Dane
```

Use Virtual Functions

- **The previous example:**

```
ppet->print( );
```

worked because print() was declared as a virtual function

- **This code would still produce an error:**

```
cout << "name: " << ppet->name
      << "breed: " << ppet->breed;
```

Why?

- **ppet->breed is still illegal because ppet is a pointer to a Pet object that has no breed member**

- **Function print( ) was declared virtual by class Pet**
  - When the computer sees ppet->print( ), it checks the virtual table(vtable) for classes Pet and Dog and finds that ppet points to an object of type Dog
    - Because ppet points to a Dog object, code for Dog::print( ) is used

Virtual Destructors

- **Destructors should be made virtual**
  - Consider Base *pBase = new Derived;

    ```
    …
    delete pBase;
    ```
  - If the destructor in Base is virtual, the destructor for Derived is invoked as pBase points to a Derived object, deallocating Derived members
    - The Derived destructor in turn calls the Base destructor

- **If the Base destructor is not virtual, only the Base destructor is invoked**
- **This leaves Derived members, not part of Base, in memory**

NEXT ?

Templates