# ECS769P: Advanced Object Orientated Programming

*School of Electronic Engineering and Computer Science,*

*Queen Mary, University of London. 2017-2018*

**Dr Ling Ma (Module Organiser)**

ling.ma@qmul.ac.uk, Office E307, Tel 7343

# Topic: Inheritance – a deeper look

# Inheritance

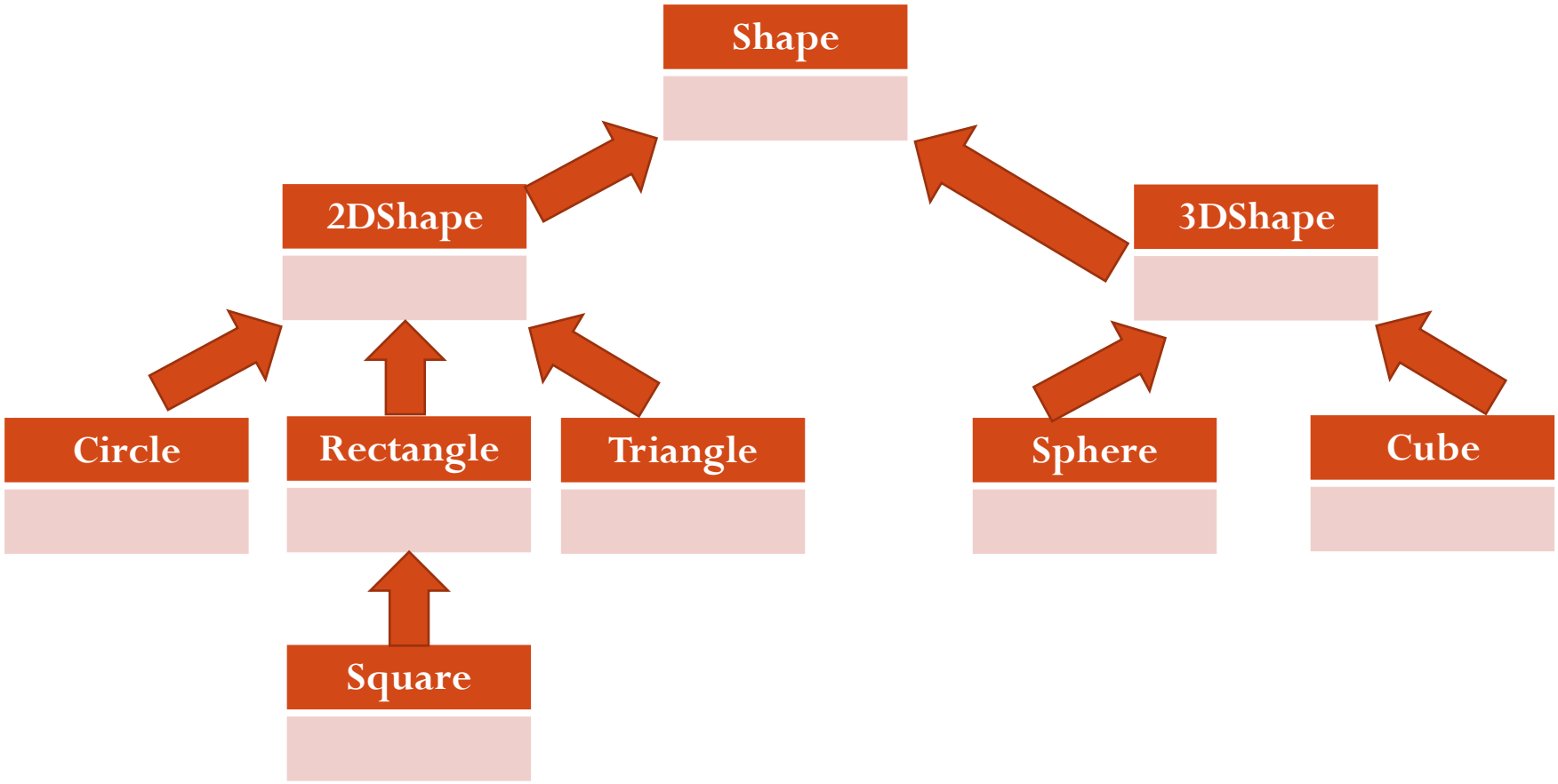
- A form of software reuse
  - create a class that absorbs an existing class's data and behaviours
  - enhances them with new capabilities
  - avoid duplicating code.
- The existing class is called the base class, and the new class is referred to as the derived class (a more specialised group of objects).
- It is the is-a relationship represents inheritance.
- C++ offers public, protected and private inheritance.

# Inheritance examples

| Base class | Derived classes |
|---|---|
| Student | UndergraduateStudent, PostgraduateStudent |
| Shape | Circle, Triangle, Square, Oval |
| Insurance | CarInsurance, HomeInsurance, TravelInsurance |
| Account | CurrentAccount, SavingAccount, StudentAccount |

- Base classes tend to be more general and derived classes tend to be more specific.
- Every derived-class object is an object of its base class

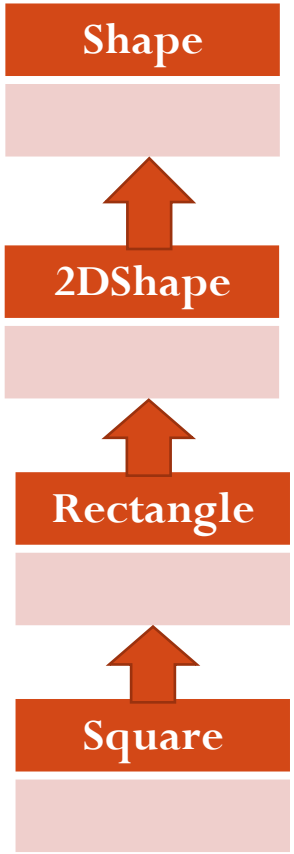
# Class hierarchies



# Class hierarchies

- If class B extends class A, class B is class A.
- If class C extends class B, then class C is class A and B.

works the way up only



# Design Process

1. Look for objects that have common attributes and behaviours.
2. Design the class (base class) that represents the common state and behaviour.
3. Decide if a derived class needs behaviours (functions implementation) that are specific to that particular derived class type.
4. Look for more opportunities to use abstraction, by finding two or more derived classes that might need common behaviour.
5. Finish the class hierarchy.

# C++ inheritance

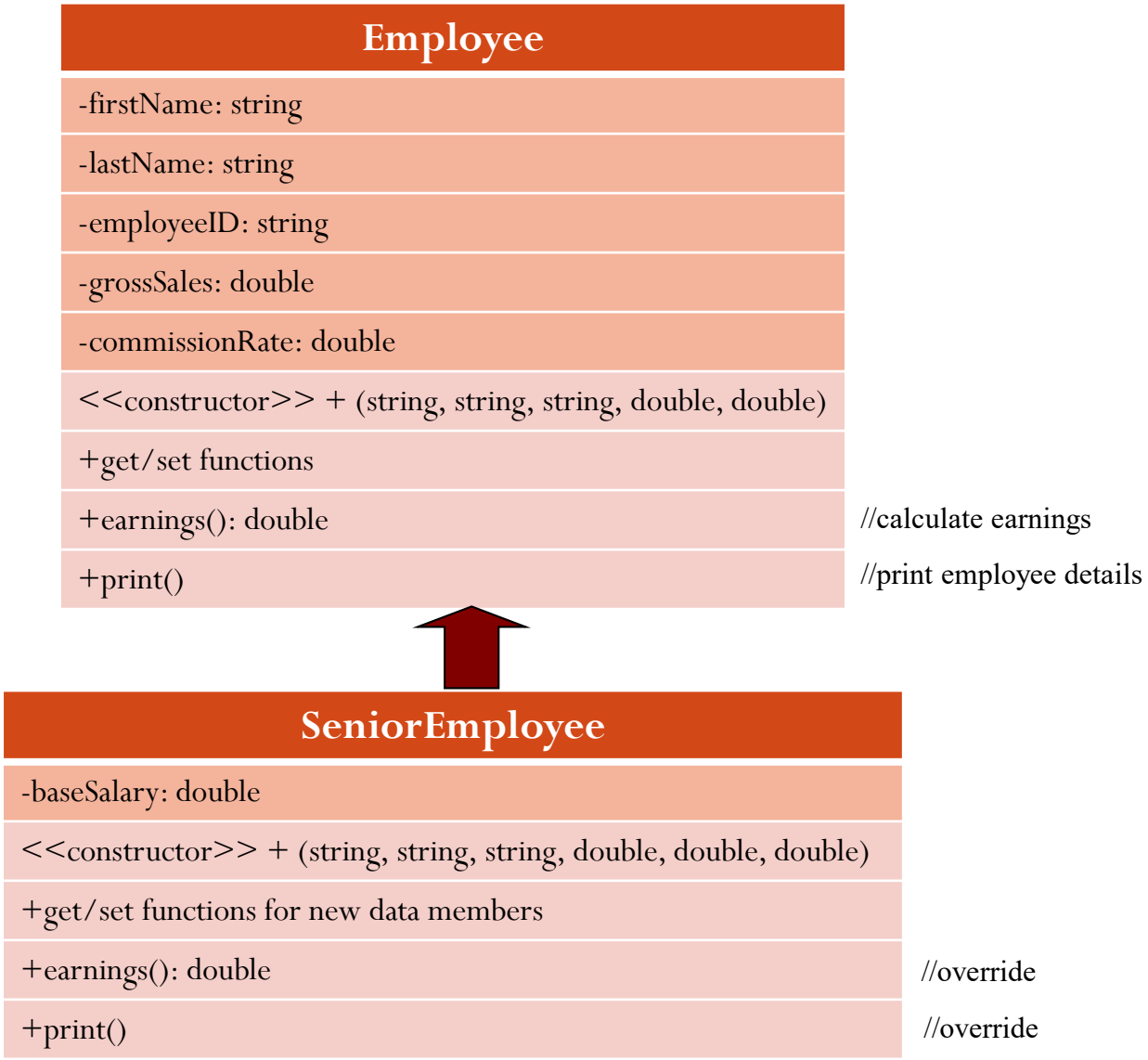
- Common data members and member functions are declared in a base class.
- Derived class inherits all the members of base class except for the constructor/destructor
  - each class provides its own constructors/destructors that are specific to the class.
  - C++ requires that a derived-class constructor call its base-class constructor to initialize the base-class data members that are inherited into the derived class.
- (:) indicates inheritance.
- A derived class cannot access the base class's private data.

# Employee example

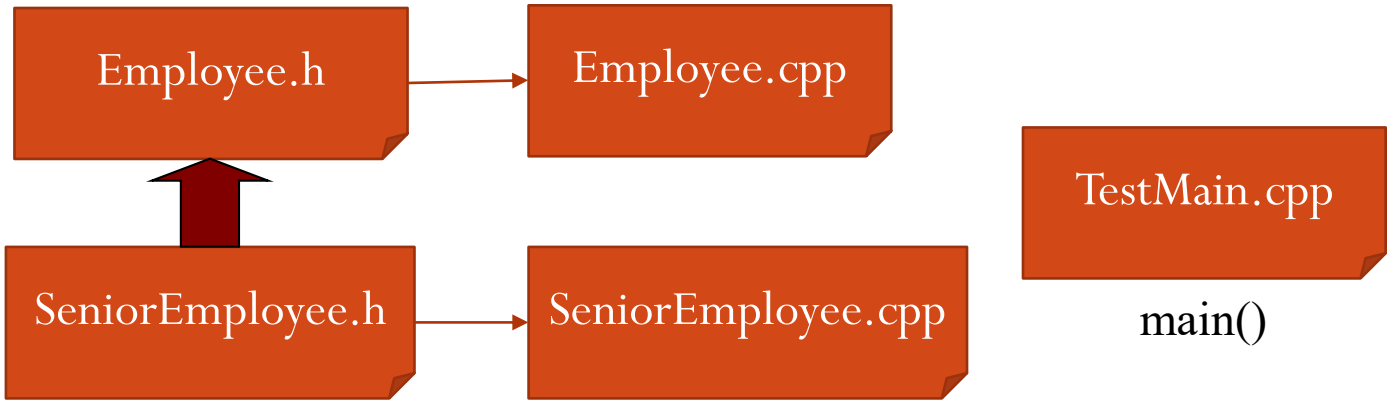
- Task

Write a program that calculates an employee's earning. The employee should have name, employee id and the earning should be calculated as (sale amount)*(commission rate). Senior employee's earning also includes a base salary.

# UML


Employee
-firstName: string
-lastName: string
-employeeID: string
-grossSales: double
-commissionRate: double
<<constructor>> + (string, string, string, double, double)
+get/set functions
+earnings(): double
//calculate earnings
+print()
//print employee details
SeniorEmployee
-baseSalary: double
<<constructor>> + (string, string, string, double, double, double)
+get/set functions for new data members
+earnings(): double
//override
+print()
//override

# C++ files


Employee.h
Employee.cpp
TestMain.cpp
SeniorEmployee.h
SeniorEmployee.cpp
main()

# Employee.h

```
#ifndef EMPLOYEE_H_
#define EMPLOYEE_H_

#include <string>

class Employee {
public:
   Employee(const std::string &, const std::string &, const std::string &,double=0.0,
double=0.0);

   void setFirstName(const std::string &);
   std::string getFirstName() const;
   void setLastName(const std::string &);
   std::string getLastName() const;
   void setEmployeeID(const std::string &);
   std::string getEmployeeID() const;
   void setGrossSales(double);
   double getGrossSales() const;
   void setCommissionRate(double); // set commission rate (percentage)
   double getCommissionRate() const;
   double earnings() const; // calculate earnings
   void print() const; // print Employee object

private:
   std::string firstName;
   std::string lastName;
   std::string employeeID;
   double grossSales; // gross weekly sales
   double commissionRate; // commission percentage
};

#endif /* EMPLOYEE_H_ */
```

Base class

private data members

# Employee.cpp (1/3)

```
#include <iostream>
#include <stdexcept>
#include "Employee.h" // Employee class definition
using namespace std;

Employee::Employee(const string &first, const string &last, const string &id,
   double sales, double rate) :
   firstName(first), lastName(last), employeeID(id) {
   setGrossSales(sales); // validate and store gross sales
   setCommissionRate(rate); // validate and store commission rate
}

void Employee::setFirstName(const string &first) {
   firstName = first; // should validate
}

string Employee::getFirstName() const {
   return firstName;
}

void Employee::setLastName(const string &last) {
   lastName = last; // should validate
}

string Employee::getLastName() const {
   return lastName;
}
```

member initialiser

call member functions rather than access private data directly.

# Employee.cpp (2/3)

```
void Employee::setEmployeeID(const string &id) {
   employeeID = id; // should validate
}

string Employee::getEmployeeID() const {
   return employeeID;
}

void Employee::setGrossSales(double sales) {
   if (sales >= 0.0)
      grossSales = sales;
   else
      throw invalid_argument("Gross sales must be >= 0.0");
}

double Employee::getGrossSales() const {
   return grossSales;
}

void Employee::setCommissionRate(double rate) {
   if (rate > 0.0 && rate < 1.0)
      commissionRate = rate;
   else
      throw invalid_argument("Commission rate must be > 0.0 and < 1.0");
}

double Employee::getCommissionRate() const {
   return commissionRate;
}
```

throw exceptions

# Employee.cpp (3/3)

```
double Employee::earnings() const {
   return getCommissionRate() * getGrossSales();
}

void Employee::print() const {
   cout << "Employee: " << getFirstName() << ' ' << getLastName()
      << "\nemployee ID: " << getEmployeeID()
      << "\ngross sales: " << getGrossSales() << "\ncommission rate: "
      << getCommissionRate();
}
```

call member functions rather than access private data directly.

# SeniorEmployee.h

```
#ifndef SENIOREMPLOYEE_H_
#define SENIOREMPLOYEE_H_

#include <string>
#include "Employee.h" // Employee class declaration

class SeniorEmployee: public Employee {
public:
   SeniorEmployee(const std::string &, const std::string &,
      const std::string &, double = 0.0, double = 0.0, double = 0.0);

   void setBaseSalary(double); // set base salary
   double getBaseSalary() const;

   double earnings() const; // calculate earnings
   void print() const;

private:
   double baseSalary; // base salary
};

#endif /* SENIOREMPLOYEE_H_ */
```

#include the base class's header in the derived class's header

: indicated inheritance relationship

derived class constructor

# SeniorEmployee.cpp

```
#include <iostream>
#include <stdexcept>
#include "SeniorEmployee.h"
using namespace std;

SeniorEmployee::SeniorEmployee(const string &first, const string &last,
   const string &id, double sales, double rate, double salary):
   Employee(first, last, id, sales, rate) {
   setBaseSalary(salary); // validate and store base salary
}

void SeniorEmployee::setBaseSalary(double salary) {
   if (salary >= 0.0)
      baseSalary = salary;
   else
      throw invalid_argument("Salary must be >= 0.0");
}

double SeniorEmployee::getBaseSalary() const {
   return baseSalary;
}

double SeniorEmployee::earnings() const {
   return getBaseSalary() + Employee::earnings();
}

void SeniorEmployee::print() const {
   cout << "Senior ";
   Employee::print(); // invoke Employee's print function
   cout << "\nbase salary: " << getBaseSalary();
}
```

explicitly call base-class constructor:

If SeniorEmployee's constructor did not invoke class Employee's constructor explicitly, C++ would attempt to invoke class Employee's default constructor—but the class does not have such a constructor, so the compiler would issue an error.

invoke base class's function, use :: careful about Infinite recursion!!

# TestMain.cpp

```
#include <iostream>
#include <iomanip>
#include "SeniorEmployee.h" // class definition
using namespace std;

int main() {
   SeniorEmployee se("John", "Smith", "0001210", 5000, .04, 300);
   cout << fixed << setprecision(2); // set floating-point output formatting

   /*get employee data*/
   cout << "Employee information: (calling individual functions)\n"
      << "\nFirst name: " << se.getFirstName()<< "\nLast name: "
      << se.getLastName()<< "\nEmployee ID: "<< se.getEmployeeID()
      << "\nGross sales: " << se.getGrossSales() << "\nCommission rate: "
      << se.getCommissionRate() << "\nBase salary: "
      << se.getBaseSalary() << endl;

   se.setBaseSalary(1000); // set base salary

   cout << "\nUpdated employee information: (calling print function)\n" << endl;

   se.print(); // display the new employee information

   cout << "\n\nEarnings: £" << se.earnings() << endl;
}
```

# Output

```
Employee information: (calling individual functions)

First name: John
Last name: Smith
Employee ID: 0001210
Gross sales: 5000.00
Commission rate: 0.04
Base salary: 300.00

Updated employee information: (calling print function)

Senior Employee: John Smith
employee ID: 0001210
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00

Earnings: £1200.00
```

call individual functions

call the print() function

# protected- who can access

- A base class's protected members can be accessed
  - within the body of that base class
  - by members and friends of that base class
  - by members and friends of any classes derived from that base class.
- Objects of a derived class also can access protected members in any of that derived class's indirect base classes.

# protected- performance

- Inheriting protected data members slightly increases performance, because we can directly access the members without incurring the overhead of calls to set or get member functions.

# protected- problems

- Using protected data members creates two serious problems.

1. The derived-class object does not have to use a member function to set the value of the base class's protected data member.
   - An invalid value can easily be assigned. – inconsistence state.
2. Derived-class member functions are more likely to be written so that *they depend on the base-class implementation*. (should depend only on the public service)
   - With protected data members in the base class, if the base-class implementation changes, we may need to modify all derived classes of that base class.

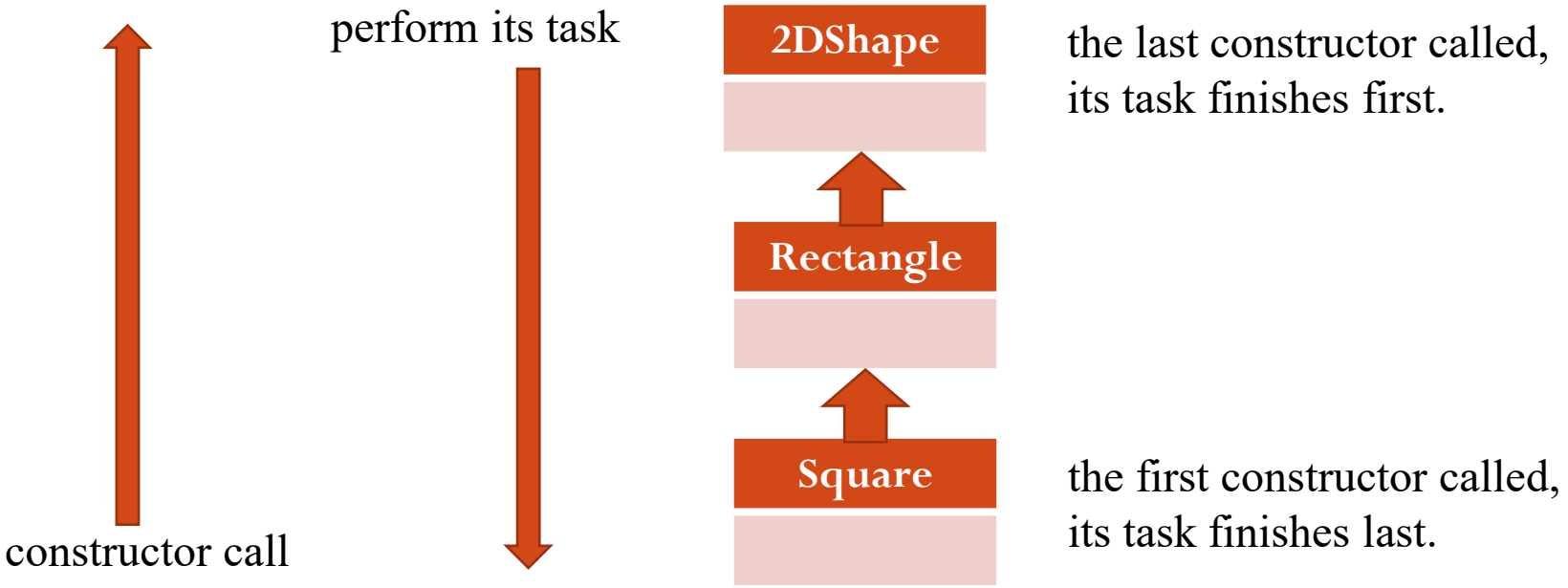
# protected – when to use

- It's appropriate to use the protected access specifier when a base class should provide a service only to its derived classes and friends.
- In most cases, better to use private data members to encourage proper software engineering, and leave code optimisation issues to the compiler.
- Declaring base class data members private enables you to change the base class implementation without having to change derived class implementations

# Constructors in Derived Classes

- Instantiating a derived-class object begins a chain of constructor calls
  - the derived-class constructor invokes its direct base class's constructor either explicitly or implicitly.
  - if the base class is derived from another class, the base-class constructor is required to invoke the constructor of the next class up in the hierarchy, and so on.
- Each base-class constructor initializes the base-class data members that the derived-class object inherits.

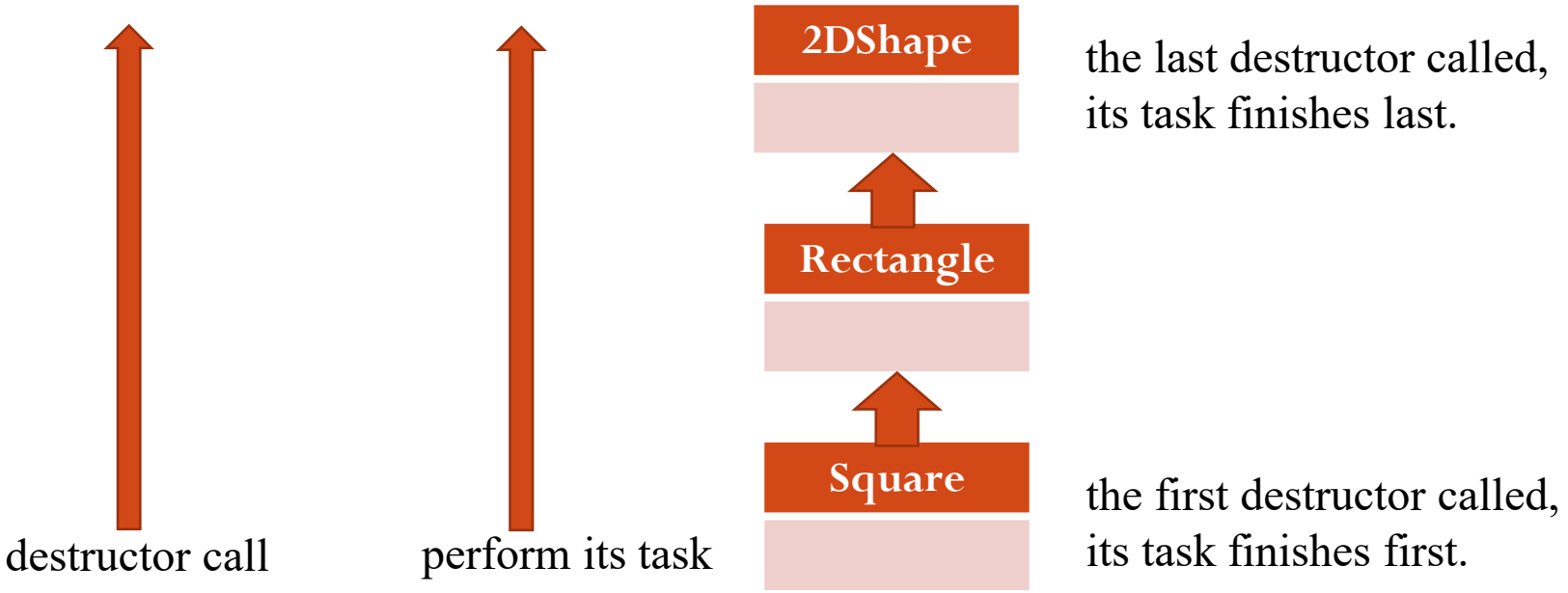
# Constructors in Derived Classes



# Destructors in Derived Classes

- When a derived-class object is destroyed, the program calls that object's destructor. This begins a chain of destructor calls.
- When a derived-class object's destructor is called, the destructor performs its task, then invokes the destructor of the next base class up the hierarchy.
- This process repeats until the destructor of the final base class at the top of the hierarchy is called.
- Then the object is removed from memory.

# Destructors in Derived Classes



# C++11 Inheriting Base Class Constructors

- Base-class constructors and destructors are not inherited by derived classes. However,
- In C++11: Inheriting Base Class Constructors is possible
  - by explicitly including a using declaration in the derived-class definition.

```
using BaseClass::BaseClass;
```

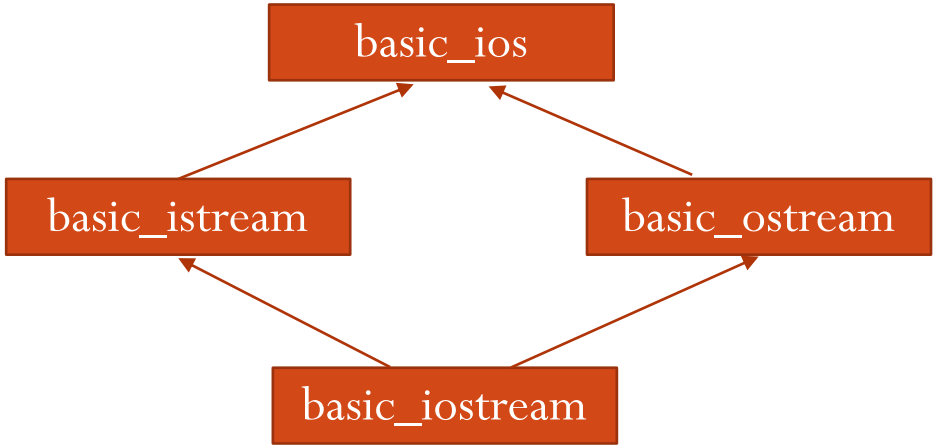
For example:

```
using Employee::Employee;
```

You may check more details online

# Multiple inheritance

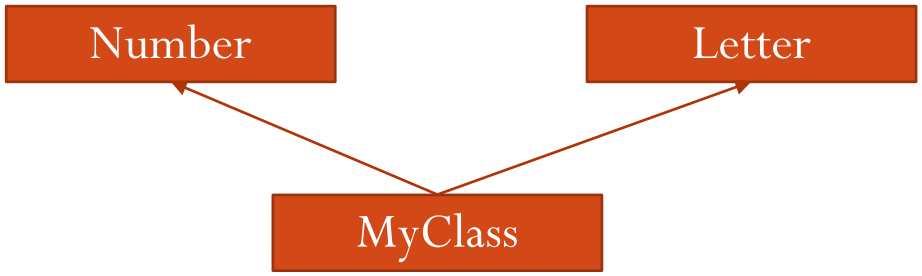
- C++ allows multiple inheritance
  - a class may be derived from more than one base class
- With multiple inheritance, a derived class inherits simultaneously from two or more (possibly unrelated) base classes.



# Multiple inheritance - caution!

- Multiple inheritance is generally discouraged.
  - Only for experienced programmers
  - Great care is required, it should not be used when single inheritance will do the job.
- This powerful capability encourages interesting forms of software reuse but can cause a variety of ambiguity problems.
  - E.g. each of the base classes might contain data members or member functions that have the same name.
- Newer programming languages, such as Java and C#, do not enable a class to derive from more than one base class.

# Example




- Number: represents one integer
- Letter: represents one letter
- MyClass: derived from both classes

# Number.h

```
#ifndef NUMBER_H_
#define NUMBER_H_

class Number {
   public:
   Number(int parameterValue) :
      value(parameterValue) {
   }

   int getData() const {
      return value;
   }

protected:
   int value;
};

#endif /* NUMBER_H_ */
```

# Letter.h

```
#ifndef LETTER_H_
#define LETTER_H_

class Letter {
   public:
   Letter(char characterData) :
      letter(characterData) {
   }

   char getData() const {
      return letter;
   }

protected:
   char letter;
};

#endif /* LETTER_H_ */
```

# MyClass.h

```
#ifndef MYCLASS_H_
#define MYCLASS_H_

#include <iostream>
#include "Number.h"
#include "Letter.h"
using namespace std;

class MyClass: public Number, public Letter {

public:
   MyClass(int, char, double);
   double getReal() const;
   void printData();

private:
   double real;
};

#endif /* MYCLASS_H_ */
```

indicates multiple inheritance

# MyClass.cpp

```
#include "MyClass.h"

MyClass::MyClass(int integer, char character, double double1) :
   Number(integer), Letter(character), real(double1) {
}

double MyClass::getReal() const {
   return real;
}

/*display all data members of MyClass*/
void MyClass::printData (){
   cout << "Integer: " << value << "  Character: " << letter << "  Real number: "
      << real <<endl;
}
```

explicitly calls base class constructors

the based constructors are called in the order that the inheritance is specified.

can access the protected data member directly

# TestMain.cpp

```
#include <iostream>
#include "Number.h"
#include "Letter.h"
#include "MyClass.h"
using namespace std;

int main(){
   Number n( 10 ); // create Number object
   Letter l( 'Z' ); // create Letter object
   MyClass m( 7, 'A', 3.5 ); // create MyClass object

   /* print data members of base-class objects*/
   cout << "Object n contains integer " << n.getData()
      << "\nObject l contains character " << l.getData()
      << "\nObject m contains: ";
   m.printData();

   /* print data members of derived-class object*/
   cout << "Data members of MyClass can be accessed individually:"
      << "\n    Integer: " << m.Number::getData()
      << "\n  Character: " << m.Letter::getData()
      << "\nReal number: " << m.getReal() << "\n\n";
   cout << "MyClass can be treated as an object of either base class:\n";

   /* treat MyClass as a Number object*/
   Number *nPtr = &m;
   cout << "nPtr->getData() yields " << nPtr->getData() << '\n';

   /* treat MyClass as a Letter object*/
   Letter *lPtr = &m;
   cout << "lPtr->getData() yields " << lPtr->getData() << endl;
}
```

Object n's getData()

Object l's getData()

Object m's printData()

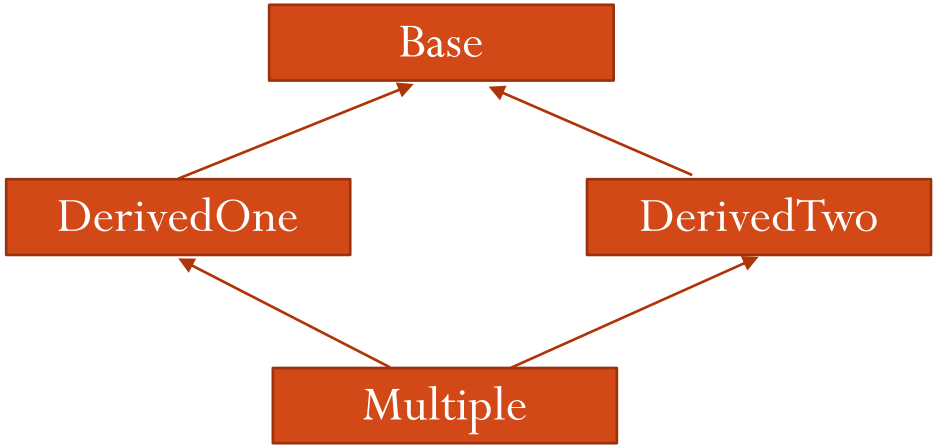
Invalid: m.getData()

# Output

```
Object n contains integer 10
Object l contains character Z
Object m contains: Integer: 7  Character: A  Real number: 3.5
Data members of MyClass can be accessed individually:
        Integer: 7
     Character: A
Real number: 3.5

MyClass can be treated as an object of either base class:
nPtr->getData() yields 7
lPtr->getData() yields A
```

# Diamond inheritance




- Class Multiple could contain two copies of the members of class Base : one inherited via class DerivedOne and one inherited via class DerivedTwo.
- Such a situation would be ambiguous and would result in a compilation error

# Virtual Base Classes

- Using virtual base classes can solve the problem of inheriting duplicate copies of an indirect base class.
- When a base class is inherited as virtual, only one subobject will appear in the derived class—a process called virtual base-class inheritance.

# Virtual Base Class example (1/2)

```
class Base {
public:
   virtual void print() const = 0; // pure virtual
};
```

```
class DerivedOne: virtual public Base {
public:
   /*override print function*/
   void print() const {
      cout << "DerivedOne\n";
 }
};
```

Use virtual base class

```
class DerivedTwo: virtual public Base {
public:
   /* override print function*/
   void print() const {
      cout << "DerivedTwo\n";
   }
};
```

Use virtual base class

both classes inherit from Base, they each contain a Base subobject.

# Virtual Base Class example (2/2)

```
class Multiple: public DerivedOne, public DerivedTwo {
public:
   /* qualify which version of function print*/
   void print() const {
      DerivedTwo::print();
   }
};
```

since each of the base classes used virtual inheritance to inherit class Base's members, the compiler ensures that only one Base subobject is inherited into class Multiple.

```
int main() {
   Multiple both; // instantiate Multiple object
   DerivedOne one; // instantiate DerivedOne object
   DerivedTwo two; // instantiate DerivedTwo object

   /*declare array of base-class pointers and initialize
    each element to a derived-class type*/
   Base *array[3];
   array[0] = &both;
   array[1] = &one;
   array[2] = &two;

   /* polymorphically invoke function print*/
   for (int i = 0; i < 3; ++i)
      array[i]->print();
}
```

```
DerivedTwo
DerivedOne
DerivedTwo
```

# Type of inheritance

- When deriving a class from a base class, the base class may be inherited through public, protected or private inheritance.
- Use of protected and private inheritance is rare.
- A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.

# public, protected or private inheritance

| Base-class member-access specifier | Type of inheritance | | |
|---|---|---|---|
| | `public` inheritance | `protected` inheritance | `private` inheritance |
| `public` | `public` in derived class.<br><br>Can be accessed directly by member functions, `friend` functions and nonmember functions. | `protected` in derived class.<br><br>Can be accessed directly by member functions and `friend` functions. | `private` in derived class.<br><br>Can be accessed directly by member functions and `friend` functions. |
| `protected` | `protected` in derived class.<br><br>Can be accessed directly by member functions and `friend` functions. | `protected` in derived class.<br><br>Can be accessed directly by member functions and `friend` functions. | `private` in derived class.<br><br>Can be accessed directly by member functions and `friend` functions. |
| `private` | Hidden in derived class.<br><br>Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class. | Hidden in derived class.<br><br>Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class. | Hidden in derived class.<br><br>Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class. |

# Software Engineering with Inheritance

- Software developers can develop proprietary classes for sale or license.
- The software developers need to supply the headers along with the object code.
- Users then can derive new classes from these library classes rapidly and without accessing the proprietary source code.
- The availability of substantial and useful class libraries delivers the maximum benefits of software reuse through inheritance.

# Summary

- Inheritance recap
- Design with inheritance
- Use protected
- Multiple inheritance
- Virtual base class