Chapter 15: Inheritance, Polymorphism, and Virtual Functions
**What Is Inheritance?**

- **Class**: a user-defined type that binds (encapsulates) attributes and related functionality into a single unit.

- What if we write another class that shares some of the functionalities?

```plaintext
Person

string name

string getName()

void setName(string)

void sayHello()

Student

string name

int studentNumber

string getName()

void setName(string)

void sayHello()

int getNumber()
```
What Is Inheritance?

- **1\textsuperscript{st} option**: write two separate classes that have nothing to do with one another. Copy & paste the code.
- You might end up doing a lot of copy pasting...
- Have to debug twice now!

```
// Person
string name

string getName()

void setName(string)

void sayHello()

// Student
string name

int studentNumber

string getName()

void setName(string)

void sayHello()

int getNumber()
```
What Is Inheritance?

- 2nd option (smart choice): **inherit** the functionality/data of Person, add functionality specific to Student
- Now we are re-using the code conveniently and elegantly

```cpp
class Student : public Person 

"Student is a Person"
```

```cpp
Person

string name

string getName()

void setName(string)

void sayHello()


Student

int studentNumber

int getNumber()

```

© Pearson Education Limited 2015.
What Is Inheritance?

- **Inheritance** provides a way to create a **new class** from an **existing class**

- The new class is an extension, or a specialized version of the existing class

**Benefits:**

- Re-use previously written code!
- Code re-use means less time spent coding, and less debugging to do
- Implement elegant design patterns in your software (COS121)
**Example: Different Kinds of People**

- **Person**
  - string name
  - string getName()
  - void setName(string)
  - void party()

- **Lecturer**
  - int salary
  - int getSalary()
  - void giveClass()

- **Student**
  - int studentNumber
  - int getNumber()
  - void study()

"Lecturer is a Person"  
"Student is a Person"
THE "IS A" RELATIONSHIP

- **Inheritance** establishes an "is a" relationship between classes.
  - A poodle is a dog (class Poodle can inherit from class Dog)
  - A car is a vehicle
  - A flower is a plant
  - A football player is an athlete

- **Aggregation**: "has a"
  - Bicycle has a wheel
  - One class main contain objects of another class
Inheritance – Terminology and Notation

- **Base** class (or parent) – the class you inherited from
- **Derived** class (or child) – inherits from the base class
- **Notation:**

```cpp
class Person  // base class
{
    // (parent class)
    . . .
};
class Student: public Person
{
    // derived class
    // (child class)
    . . .
};
```
BACK TO THE ‘IS A’ RELATIONSHIP

- An object of a derived class 'is a(n)' object of the base class

- Example:
  - a **Student** is a **Person**
  - a **Mammal** is an **Animal**

- A derived object has all of the characteristics of the base class

- I.e., characteristics are **inherited**
WHAT DOES A CHILD HAVE?

An object of the **derived class** has:
- all members defined in **child** class
- all members declared in **parent** class

An object of the **derived class** can use:
- all **public** members defined in **child** class
- all **public** members defined in **parent** class

What about the **private** members?
- an object of derived class will contain the **private** members, but will only be able to access them through **public** functions defined in the parent class
### Protected Members of a Class

- **public** members are accessible outside of the class, and **private** members are not accessible outside of the class.
- **private** members of the parent class are not directly accessible from the child class.

What if we want to keep certain **parent class** members inaccessible from the outside, but allow **child class** to access them directly?

- Use the **protected** access modifier!

  ```cpp
  protected:
  double length;
  double width;
  ```
**PROTECTED MEMBERS OF A CLASS**

<table>
<thead>
<tr>
<th>Parent class access modifier</th>
<th>Access from the outside</th>
<th>Access from a child class</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Accessible</td>
<td>Accessible</td>
</tr>
<tr>
<td>private</td>
<td>Not accessible</td>
<td>Not accessible</td>
</tr>
<tr>
<td>protected</td>
<td>Not accessible</td>
<td>Accessible</td>
</tr>
</tbody>
</table>

**Protected members:**
- **public** for child classes
- **private** for the rest of the world

Inheritance can also be **public, protected, and private** (next slide)
INHERITANCE AS PUBLIC, PROTECTED, OR PRIVATE

// Inherit from parent publicly
class Child1: public Parent
{
};

// Inherit from parent privately
class Child2: private Parent
{
};

// Inherit from parent protectedly
class Child3: protected Parent
{
};

// Default: private inheritance
class Child4: Parent
{
};
# Inheritance as Public, Protected, or Private

<table>
<thead>
<tr>
<th>Parent Access Modifier</th>
<th>Public Inheritance</th>
<th>Protected Inheritance</th>
<th>Private Inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class Parent</td>
<td>Class Child: public Parent</td>
<td>Class Child: protected Parent</td>
<td>Class Child: private Parent</td>
</tr>
<tr>
<td>Public: x</td>
<td>Public: x</td>
<td>Protected: x</td>
<td>Private: x</td>
</tr>
<tr>
<td>Protected: x</td>
<td>Protected: x</td>
<td>Protected: x</td>
<td>Private: x</td>
</tr>
<tr>
<td>Private: x</td>
<td>Inaccessible</td>
<td>Inaccessible</td>
<td>Inaccessible</td>
</tr>
</tbody>
</table>
**Public inheritance**

class Parent
{
   public:
      int x;
   protected:
      int y;
   private:
      int z;
};
class Child: public Parent               // inherit publicly
{
   Child()
   {
      x = 1; // ALLOWED: x is public
      y = 2; // ALLOWED: y is protected
      z = 3; // COMPILe ERROR: z is private
   }
};
int main()
{
   Child obj;
   obj.x = 1; // ALLOWED: anybody can access public members
   obj.y = 2; // NOT OK: can not access protected members from outside
   obj.z = 3; // NOT OK: can not access private members from outside
}
## Protected inheritance

```cpp
class Parent {
    public:
        int x;
    protected:
        int y;
    private:
        int z;
};

class Child: protected Parent // inherit protectedly
{
    Child() {
        x = 1; // ALLOWED: x is protected in Child
        y = 2; // ALLOWED: y is protected in Child
        z = 3; // COMPILIE ERROR: z is private in Parent
    }
};

int main() {
    Child obj;
    obj.x = 1; // NOT OK: can not access protected members from outside
    obj.y = 2; // NOT OK: can not access protected members from outside
    obj.z = 3; // NOT OK: can not access private members from outside
}
```
**PRIVATE INHERITANCE**

class Parent {
    public:
        int x;
    protected:
        int y;
    private:
        int z;
};
class Child: private Parent // inherit privately
{
    Child() {
        x = 1; // ALLOWED: x is a private member of Child
        y = 2; // ALLOWED: y is a private member of Child
        z = 3; // COMPIL...
CLASS ACCESS SPECIFIERS

1) **public** – object of derived class can be treated as object of base class (not vice-versa)

2) **protected** – more restrictive than **public**, but allows derived classes to know details of parents

3) **private** – prevents objects of derived class from being treated as objects of base class.
CONSTRUCTORS AND DESTRUCTORS IN BASE AND DERIVED CLASSES

- Derived classes may have their own member variables – thus, derived classes will need their own constructors and destructors

- Constructors and destructors are not inherited, but private members of the parent class are inherited!

- How do we initialise them?

- Solution: call the constructor of the parent class from the constructor of the child class
CONSTRUCTOR INITIALIZATION LISTS

- What you’re used to (explicit assignment):

```cpp
Rectangle::Rectangle(int w, int l)
{
    width = w;
    length = l;
}
```

- Another way to do it (implicit assignment):

```cpp
Rectangle::Rectangle(int w, int l) : width(w), length(l)
{
} // known as member initialisation list
```
PASSING ARGUMENTS TO BASE CLASS CONSTRUCTOR

The same syntax can be used to pass arguments to a base class constructor

Suppose `Square` inherits from `Rectangle`:

- `class Square : public Rectangle`

Pass arguments to base constructor from the derived class constructor:

```cpp
Square::Square(int side)
    : Rectangle(side, side) {}
```

Must be done if base class has no default constructor
CONSTRUCTORS AND DESTRUCTORS IN BASE AND DERIVED CLASSES

- When an object of a derived class is created:
  - base class’s constructor is executed first
  - followed by the derived class’s constructor

- When an object of a derived class is destroyed:
  - derived class’s destructor is called first
  - followed by base class’s destructor

- When the destructor of the child class executes, it automatically invokes the destructor of the parent class

- No explicit reference from child destructor to parent destructor is necessary
PASSING ARGUMENTS TO BASE CLASS CONSTRUCTOR

Square::Square(int side):Rectangle(side,side)

derived class constructor

derived constructor parameter

base class constructor

base constructor parameters