SC1008 C and C++ Programming

Assistant Professor WANG Yong

yong-wang@ntu.edu.sg

CCDS, Nanyang Technological University

Week 11

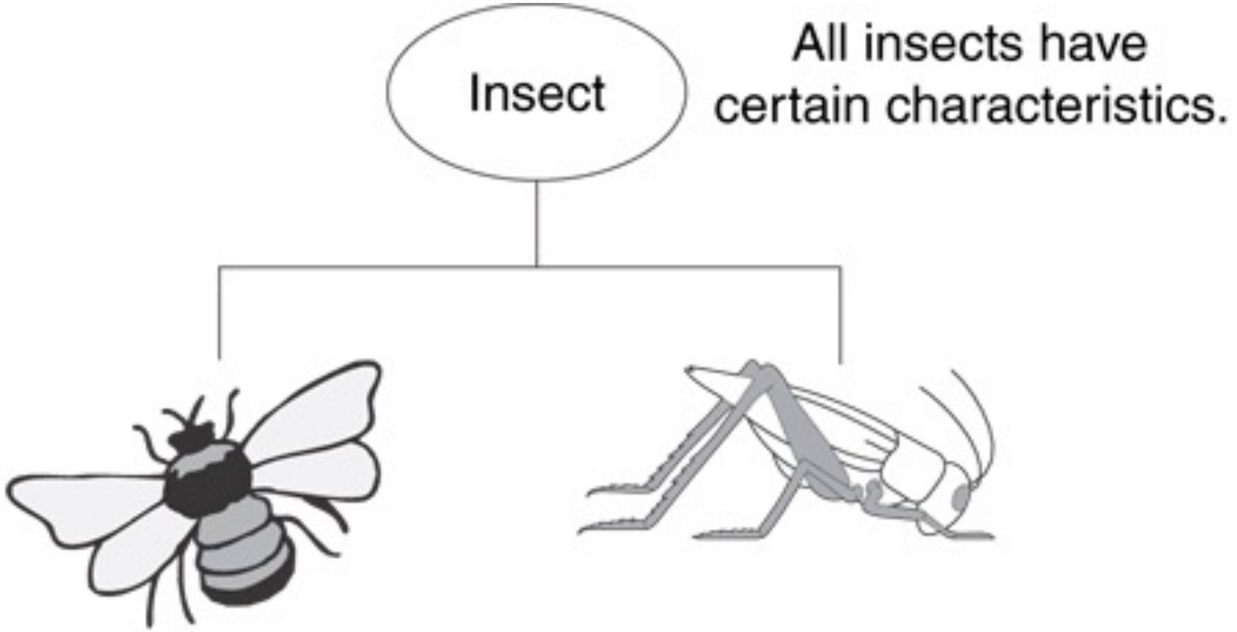
Inheritance and Polymorphism in C++

Outline

- Inheritance
- Constructors and Destructors in Base and Derived Classes
- Multi-level Inheritance vs. Multiple Inheritance
- Redefine Base Class Functions
- Polymorphism and Virtual Member Functions

Inheritance

- Example: Insect Taxonomy



Insect

All insects have certain characteristics.

In addition to the common insect characteristics, the bumble bee has its own unique characteristics such as the ability to sting.

In addition to the common insect characteristics, the grasshopper has its own unique characteristics such as the ability to jump.

Inheritance

- Inheritance is also called an ***is-a*** hierarchy
  - A poodle is a dog
  - A car is a vehicle
  - A flower is a plant
  - A football player is an athlete
- An object of child class (derived class) ***is an*** object of parent class (base class)
  - an `UnderGrad` is a `Student`
  - a `Mammal` is an `Animal`

Inheritance

Benefits:

- Code reuse: a derived class can automatically inherit code from base class
- Polymorphism: Ability to redefine existing behavior but preserve the interface; children can override the behavior of the parent
- Extensibility: Children can add behavior

```
class Student // base class
{
    . . .
};

class UnderGrad : public Student
{ // derived class
     . . .
};
```

What Does a Child Have?

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

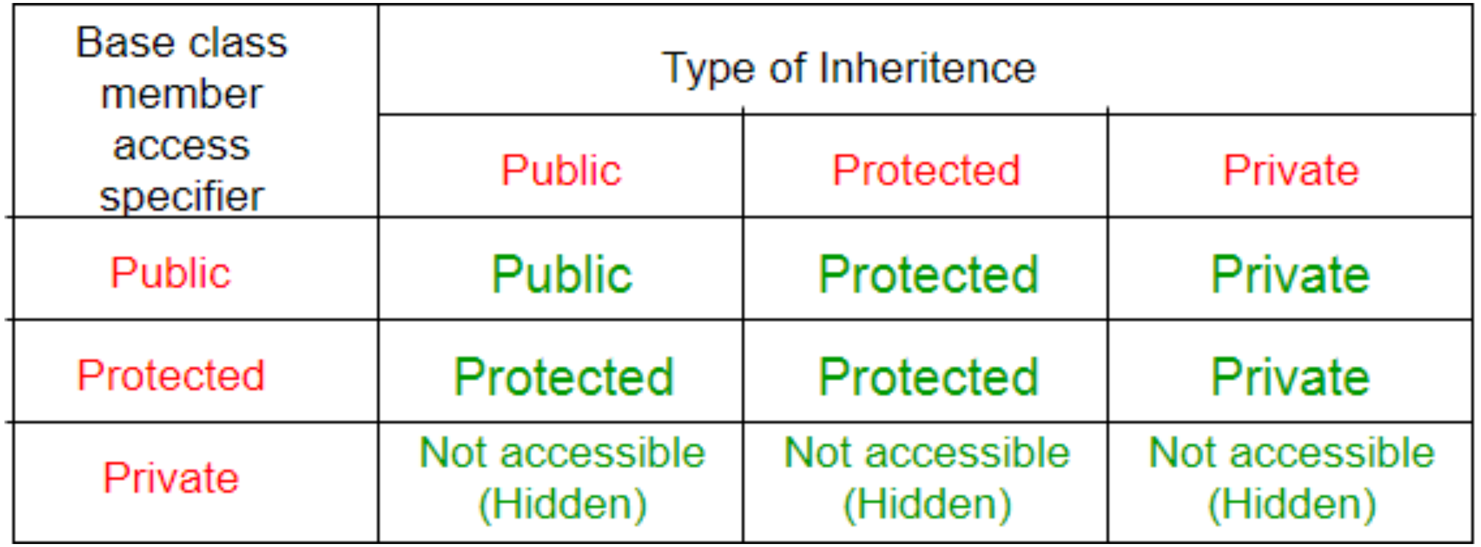
- An object of the derived class can use:
  - All `public` members defined in child class
  - All `public` members defined in parent class

Note: The following terms are often used interchangeably:

“child class” == “derived class”
“parent class” == “base class”

Class Access Specifiers

- `public` – object of derived class can be treated as object of base class (**not vice-versa**)
- `protected` – more restrictive than `public`, but allows derived classes to know details of parents
- `private` – prevents objects of derived class from being treated as objects of base class.

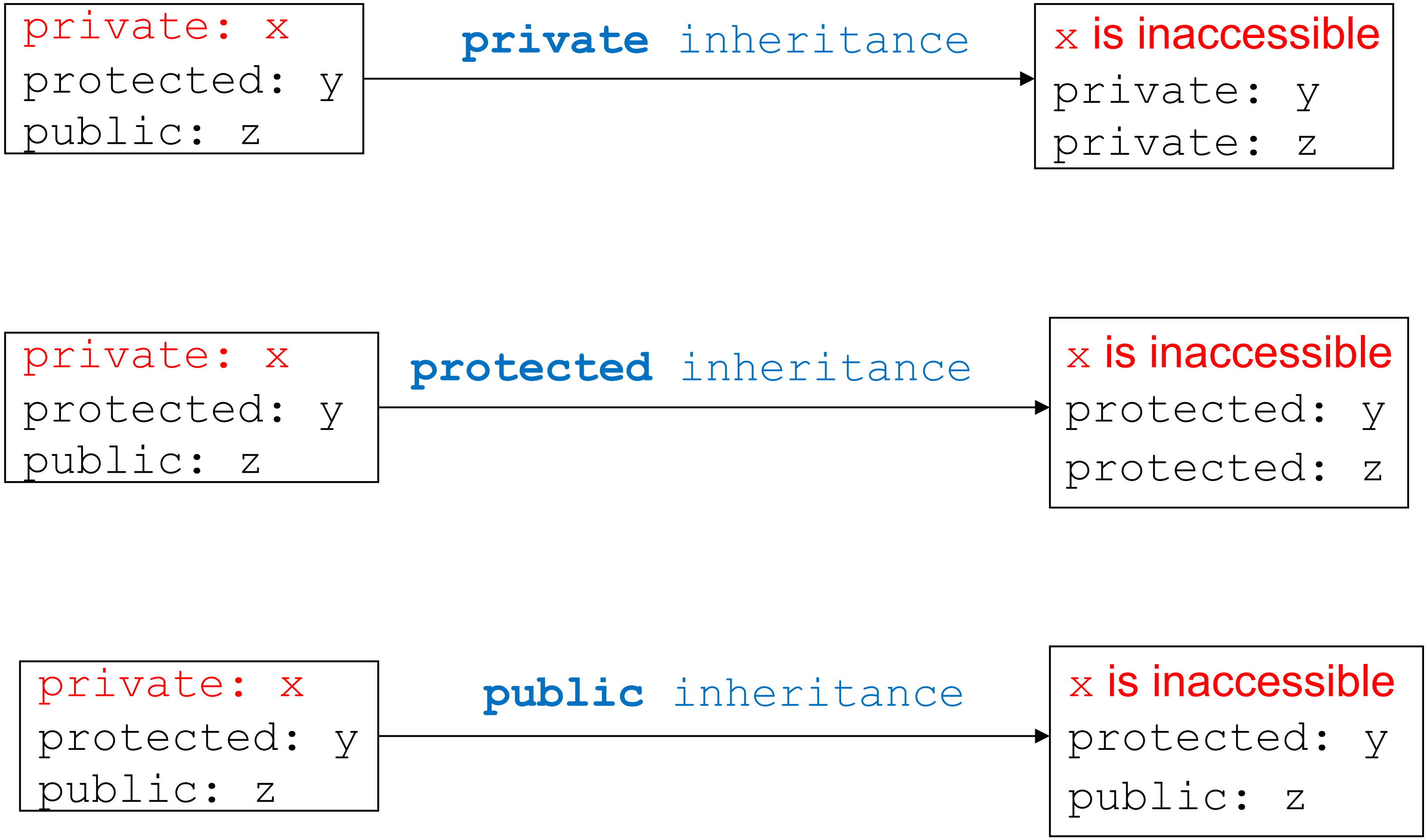


Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Inheritance vs. Access

Base class members

How inherited base class members appear in derived class



private: x
protected: y
public: z

private inheritance

x is inaccessible
private: y
private: z

private: x
protected: y
public: z

protected inheritance

x is inaccessible
protected: y
protected: z

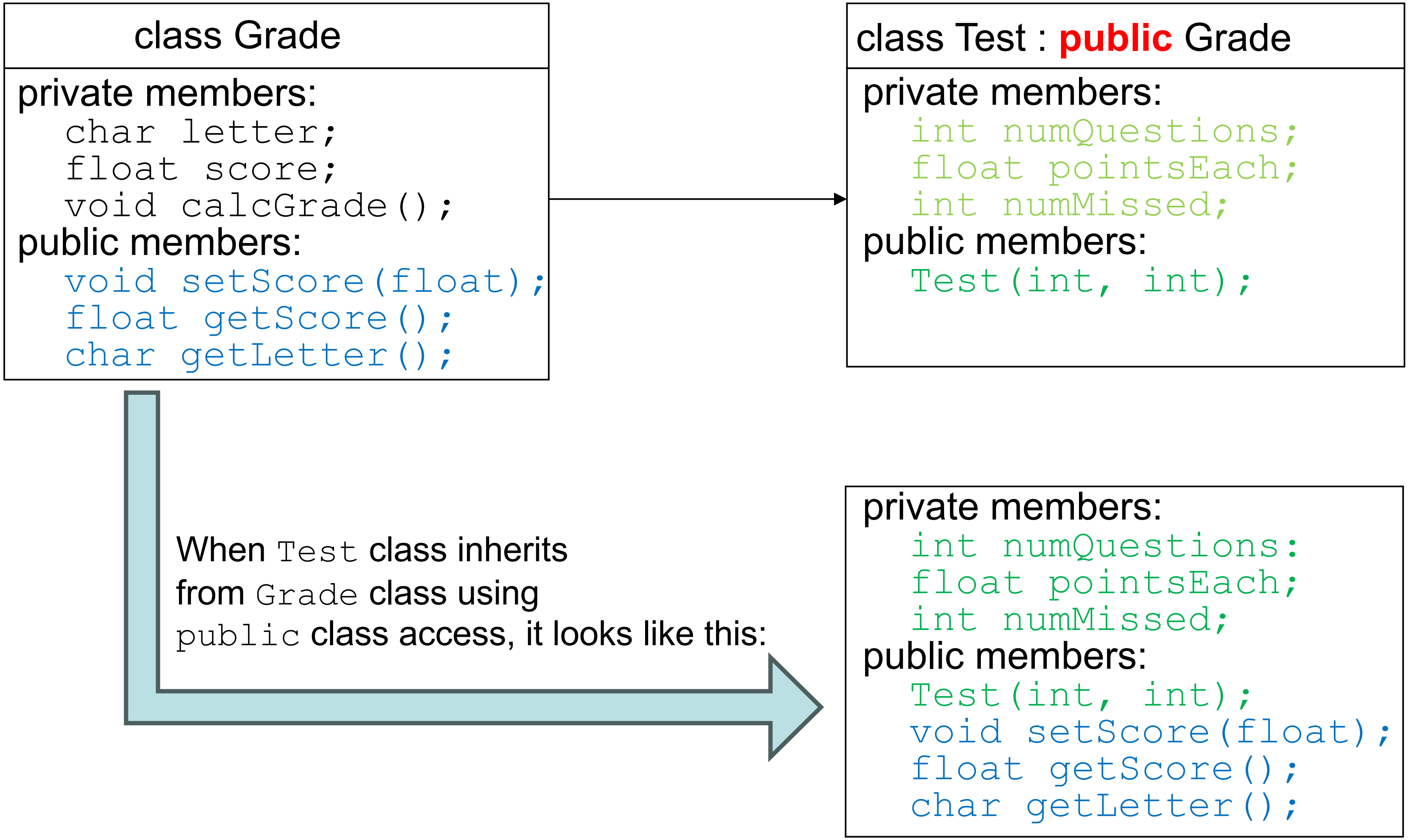
private: x
protected: y
public: z

public inheritance

x is inaccessible
protected: y
public: z

Almost always you will want public inheritance

Public Inheritance vs. Access



class Grade

```
private members:
   char letter;
   float score;
   void calcGrade();
public members:
   void setScore(float);
   float getScore();
   char getLetter();
```

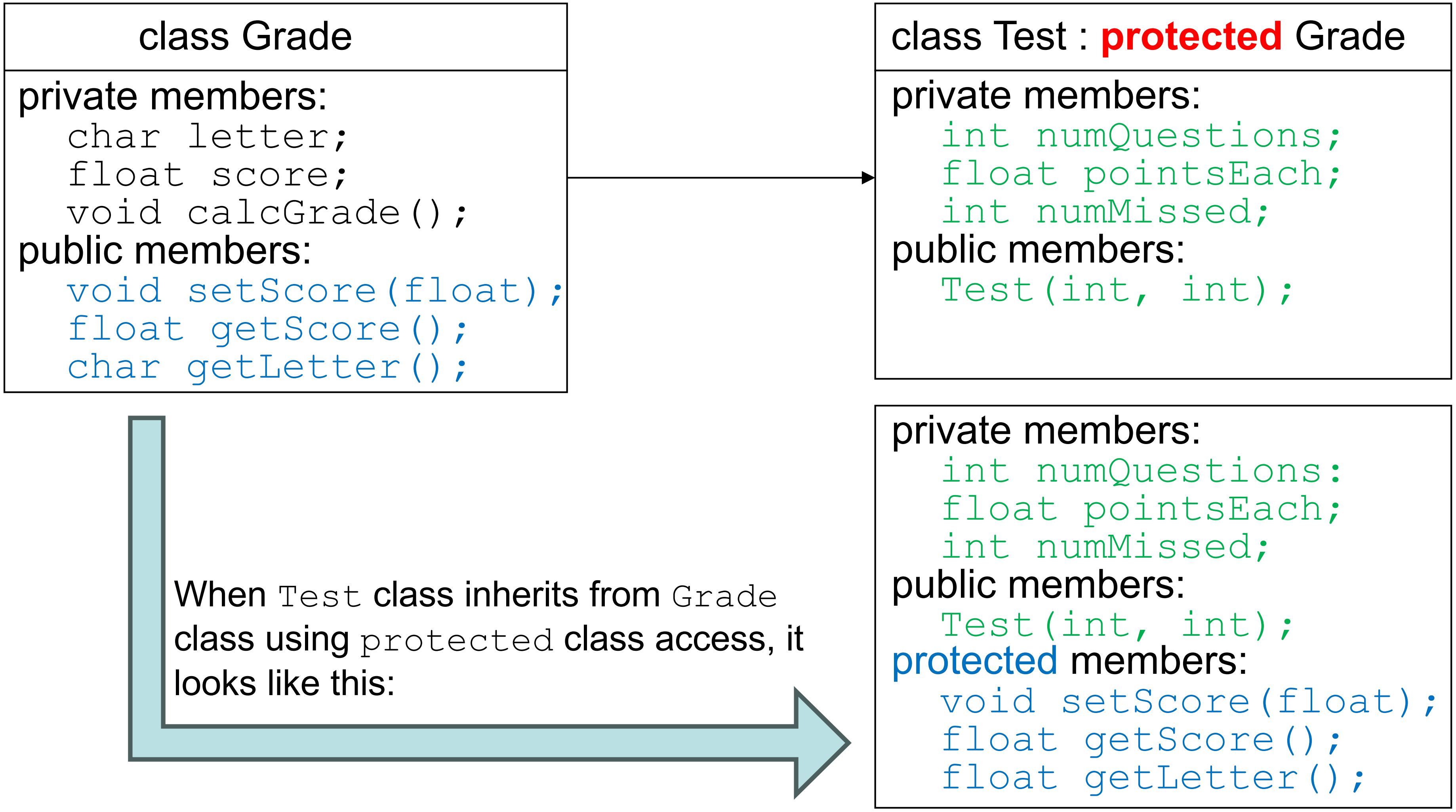
class Test : **public** Grade

```
private members:
   int numQuestions;
   float pointsEach;
   int numMissed;
public members:
   Test(int, int);
```

When `Test` class inherits from `Grade` class using `public` class access, it looks like this:

```
private members:
   int numQuestions:
   float pointsEach;
   int numMissed;
public members:
   Test(int, int);
   void setScore(float);
   float getScore();
   char getLetter();
```

Protected Inheritance vs. Access



class Grade

```
private members:
   char letter;
   float score;
   void calcGrade();
public members:
   void setScore(float);
   float getScore();
   char getLetter();
```

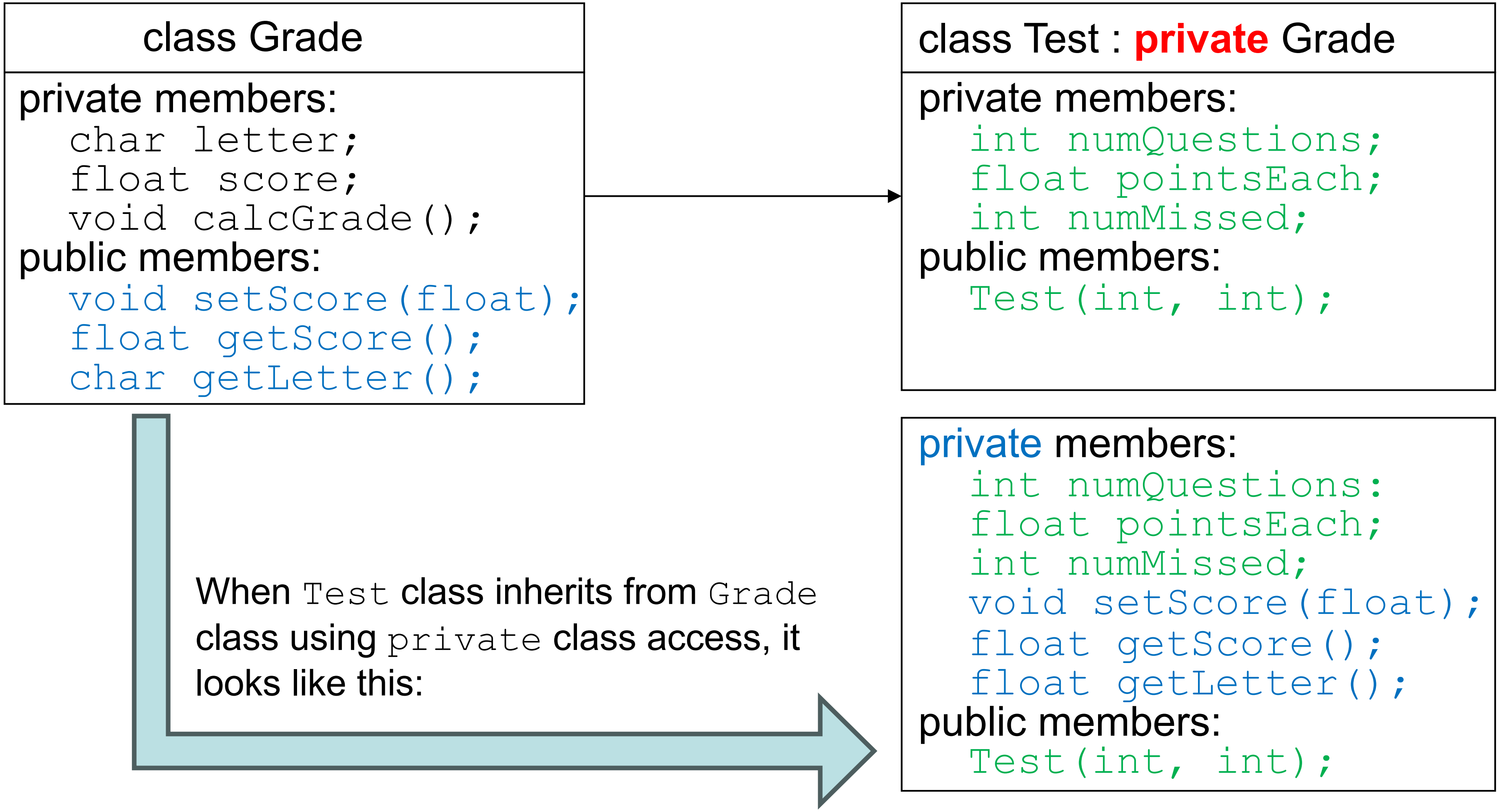
class Test : **protected** Grade

```
private members:
   int numQuestions;
   float pointsEach;
   int numMissed;
public members:
   Test(int, int);
```

When `Test` class inherits from `Grade` class using `protected` class access, it looks like this:

```
private members:
   int numQuestions:
   float pointsEach;
   int numMissed;
public members:
   Test(int, int);
protected members:
   void setScore(float);
   float getScore();
   float getLetter();
```

Private Inheritance vs. Access



```
class Grade
private members:
   char letter;
   float score;
   void calcGrade();
public members:
   void setScore(float);
   float getScore();
   char getLetter();
```

```
class Test : private Grade
private members:
   int numQuestions;
   float pointsEach;
   int numMissed;
public members:
   Test(int, int);
```

When `Test` class inherits from `Grade` class using `private` class access, it looks like this:

```
private members:
   int numQuestions:
   float pointsEach;
   int numMissed;
   void setScore(float);
   float getScore();
   float getLetter();
public members:
   Test(int, int);
```

Constructors and Destructors in Inheritance

- A derived class will **NOT** inherit the constructors, destructor or assignment operator from a base class

- Derived classes can have their own constructors and destructors

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

- When an object of a derived class is destroyed, its destructor is called first, then that of the base class

- Derived class constructors and assignment operators, however, **can call** base class constructors and assignment operators

Constructors and Destructors in Inheritance

- The execution order of constructors and destructors

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

class BaseClass {
public:
   BaseClass() { cout << "BaseClass Constructor\n"; }
   ~BaseClass() { cout << "BaseClass Destructor\n"; }
};

class DerivedClass : public BaseClass {
public:
   DerivedClass() { cout << "DerivedClass Constructor\n"; }
   ~DerivedClass() { cout << "DerivedClass Destructor\n"; }
};

int main() {
   cout << "Creating object...\n";
   DerivedClass object;
   cout << "Exiting program...\n";
   return 0;
}
```

Program output:

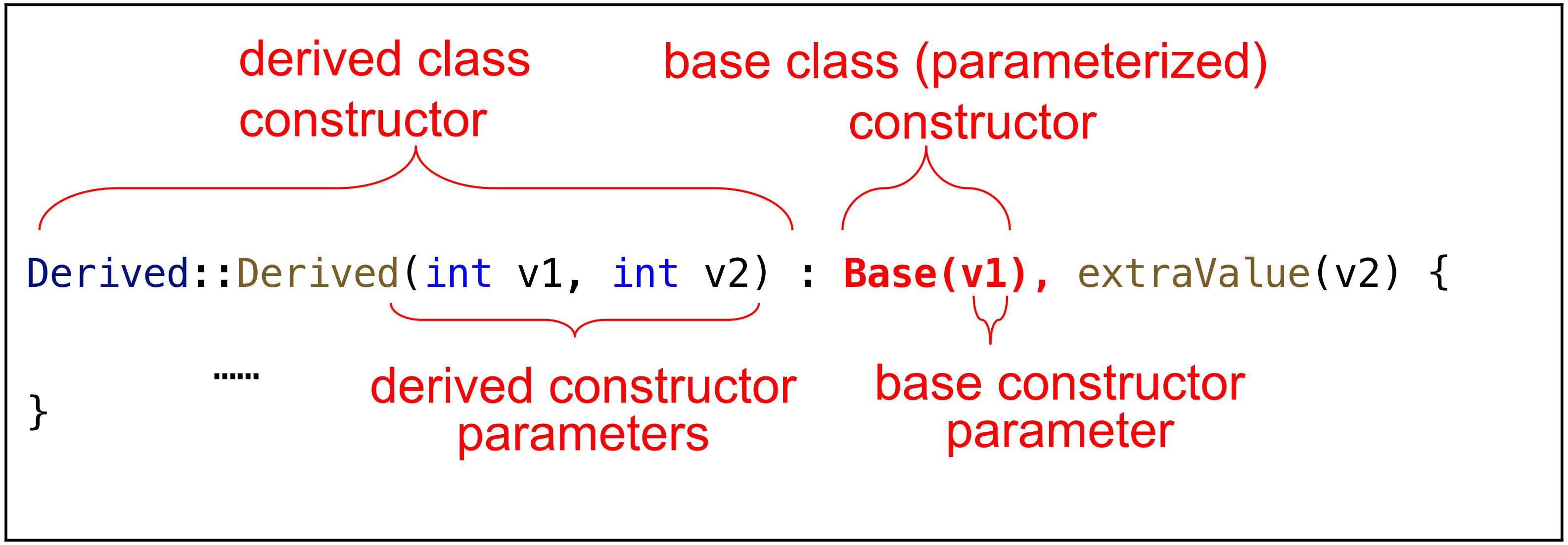
```
Creating object...
BaseClass Constructor
DerivedClass Constructor
Exiting program...
DerivedClass Destructor
BaseClass Destructor
```

Constructors and Destructors in Inheritance

Passing arguments to base class constructor

- Allow selection between multiple base class constructors
- Specify arguments to base constructor on derived constructor heading:

```
ClassName::ClassName(ParameterList) : BaseClassName(ArgumentList)
```



derived class constructor

base class (parameterized) constructor

```
Derived::Derived(int v1, int v2) : Base(v1), extraValue(v2) {
        ……
}
```

derived constructor parameters

base constructor parameter

- Must be done if base class has no default constructor

Constructors and Destructors in Inheritance

Passing arguments to base class constructor

Program output:

```
value: 10
Base Constructor
value: 10
extraValue: 20
Derived Constructor
```

```cpp
#include <iostream>
using namespace std;
class Base {
protected:
   int value;
public:
   Base(int v); // Constructor declaration
};

Base::Base(int v) : value(v) {
   cout<< "value: " << value << endl;
   cout << "Base Constructor\n";
}

class Derived : public Base {
private:
   int extraValue;
public:
   Derived(int v1, int v2);
};

// Derived constructor definition
Derived::Derived(int v1, int v2) : Base(v1), extraValue(v2) {
   cout<< "value: " << value << endl;
   cout<< "extraValue: " << extraValue << endl;
   cout << "Derived Constructor\n";
}

int main() {
   Derived d1(10, 20);
   return 0;
}
```

Constructors and Destructors in Inheritance

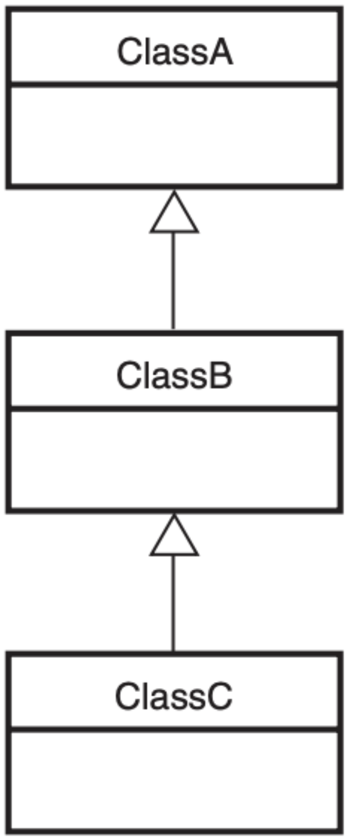
- If the base class has no default constructor, then the constructor of derived class must pass arguments to the parameterized constructor of the base class
- Otherwise, **the compiler does not know how to initialize the base class, as base class constructor must be executed first**

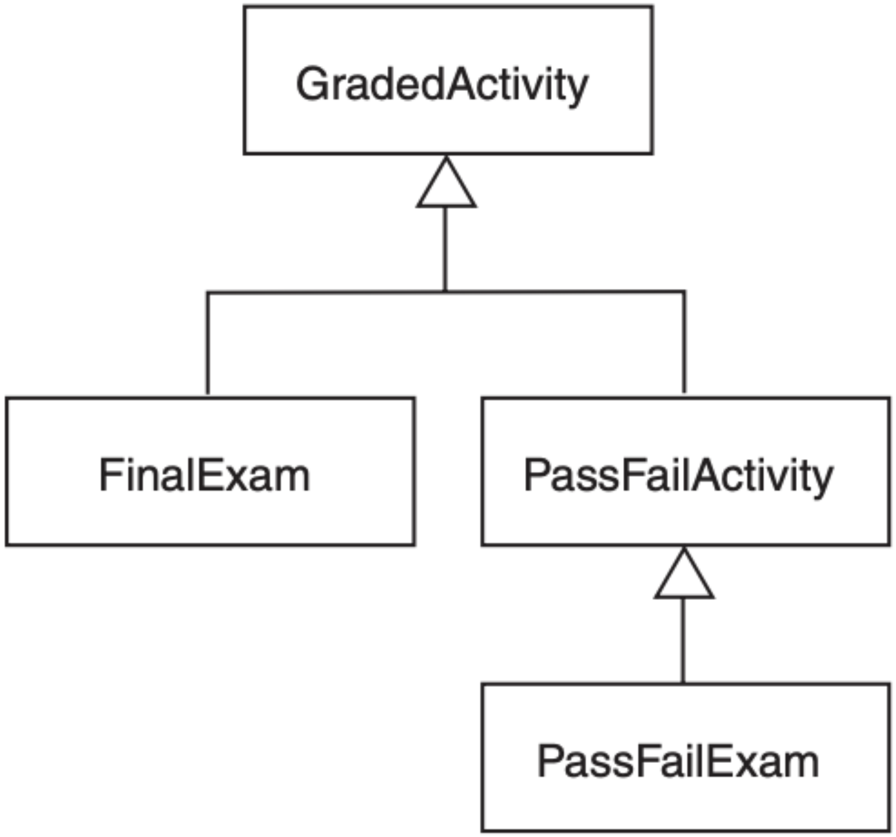
Error!!!

```
#include <iostream>
using namespace std;
class Base {
protected:
  int value;
public:
   Base(int v); // Constructor declaration
};

Base::Base(int v) : value(v) {
   cout << "Base Constructor\n";
}

class Derived : public Base {
private:
    int extraValue;
public:
    Derived(int v1, int v2);
};

// Derived constructor definition
Derived::Derived(int v1, int v2) {
   value = v1;
   extraValue = v2;
   cout << "Derived Constructor\n";
}

int main() {
Derived d1(10, 20);
return 0;
}
```

Multilevel Inheritance

- A base class can also be derived from another class





Multilevel Inheritance

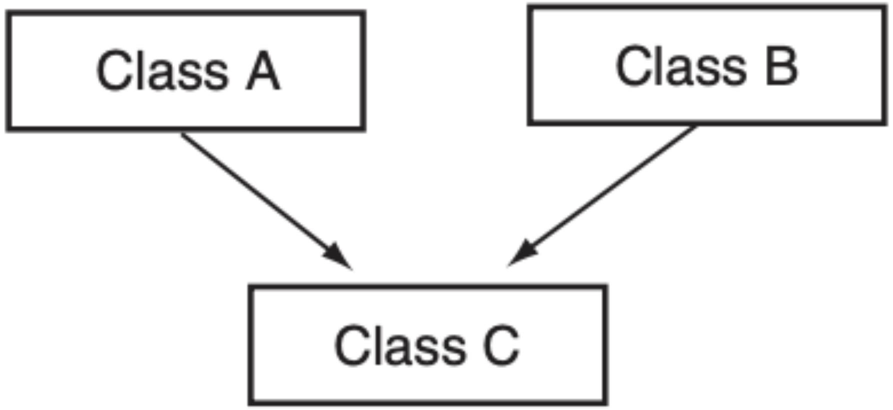
```cpp
#include <iostream>
using namespace std;

class Vehicle {
public:
   Vehicle() { cout << "Vehicle Constructor Called\n"; }
   void showVehicle() {
     cout << "This is a Vehicle\n";
   }
};

class FourWheeler : public Vehicle {
public:
   FourWheeler() { cout << "FourWheeler Constructor Called\n"; }
   void showFourWheeler() {
      cout << "This is a Four-Wheeler\n";
   }
};

class Car : public FourWheeler {
public:
    Car() { cout << "Car Constructor Called\n"; }
};

int main() {
   Car myCar;
   myCar.showVehicle(); // From Vehicle class
   myCar.showFourWheeler(); // From FourWheeler class

   return 0;
}
```

Program output:

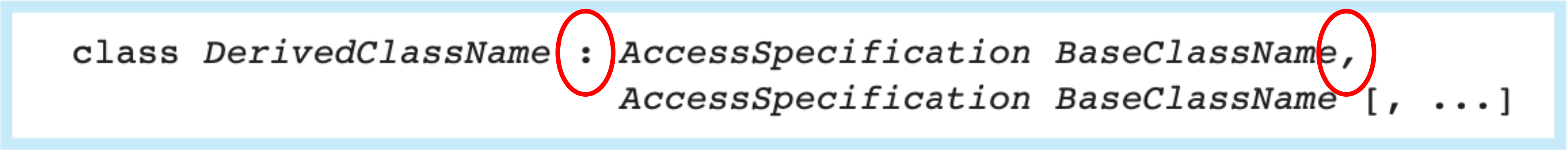
```
Vehicle Constructor Called
FourWheeler Constructor Called
Car Constructor Called
This is a Vehicle
This is a Four-Wheeler
```

Multiple Inheritance

- Multiple Inheritance is a feature of C++ where a class can **inherit from more than one base classes**



- General format of multiple inheritance in C++



```
class DerivedClassName : AccessSpecification BaseClassName,
                         AccessSpecification BaseClassName [, ...]
```

Base classes will be separated by a comma (",") and access mode for every base class must be specified

Multiple Inheritance

- The constructors of inherited classes are called in the same order in which they are inherited
- The destructors are called in reverse order of constructors

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

class Vehicle {
public:
  Vehicle() {  cout << "Vehicle Constructor Called\n"; }
  ~Vehicle() { cout << "Vehicle Destructor Called\n"; }
};

class FourWheeler {
public:
   FourWheeler() { cout << "FourWheeler Constructor Called\n"; }
   ~FourWheeler() { cout << "FourWheeler Destructor Called\n"; }
};

class Car : public Vehicle, public FourWheeler {
public:
   Car() { cout << "Car Constructor Called\n"; }
   ~Car() { cout << "Car Destructor Called\n"; }
};

int main() {
   Car myCar;
   return 0;
}
```

Program output:

```
Vehicle Constructor Called
FourWheeler Constructor Called
Car Constructor Called
Car Destructor Called
FourWheeler Destructor Called
Vehicle Destructor Called
```

Redefining Base Class Functions

- A base class member function may **be redefined** in a derived class

  - The function in a derived class has **the same name** and **same parameter list** as the function in the base class

  - Often used to replace a function in the base class with different actions in a derived class

  - It leads to **function hiding**: Objects of base class use base class version of function; objects of derived class use derived class version of function

Redefining Base Class Functions

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

class Base {
public:
  void display(int a = 10) {
     cout<< "Base class display(): " << a << endl;
  }
};

class Derived : public Base {
public:
   void display(int a = 100) {//It hides Base::display()
    cout << "Derived class display(): " << a << endl;
  }
};

int main() {
   Base b;
   Derived d;
   b.display();
   d.display();
   return 0;
}
```

Program output:

```
Base class display(): 10
Derived class display(): 100
```

Redefining Base Class Functions – Possible Problem

- Problem: **Static Binding -- function calls are bound at the compile time**

```
#include <iostream>
using namespace std;
class Base {
public:
   void display() {
      cout<< "Base class display() " << endl;
   }
};
class Derived : public Base {
public:
   void display() { //It hides Base::display()
     cout << "Derived class display() " << endl;
   }
};

int main() {
   Derived d;
   d.display();

   Base* ptr = &d;
   ptr->display();
   return 0;
}
```

It will call the display() function in the base class!

Program output:

```
Derived class display()
Base class display()
```

Redefining Base Class Functions

- **Redefining** vs. **overloading**

  - Overloaded functions have the same function name, but with a different parameter list

  - Overloading can take place inside the same class, i.e., multiple member functions of **the same class** have the same name

  - Overloading can also be enabled between base class and derived class, but should use

  ```
  using Base::functionName;
  ```

  to retain access to base function

Overloading a Base Class Function

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

class Base {
public:
   void show() {
      cout << "Base class show()" << endl;
   }
};

class Derived : public Base {
public:
    using Base::show; // This must be kept! Otherwise, Error
    void show(int x) { // Overloaded
      cout << "Derived class show() with int: " << x << endl;
    }
};

int main() {
   Derived d;
   d.show();   // Calls Base class show()
   d.show(10); // Calls Derived class show(int)
}
```

Program output:

```
Base class show()
Derived class show() with int: 10
```

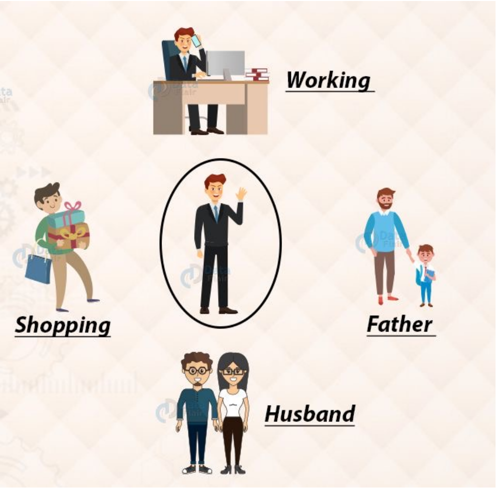
Polymorphism

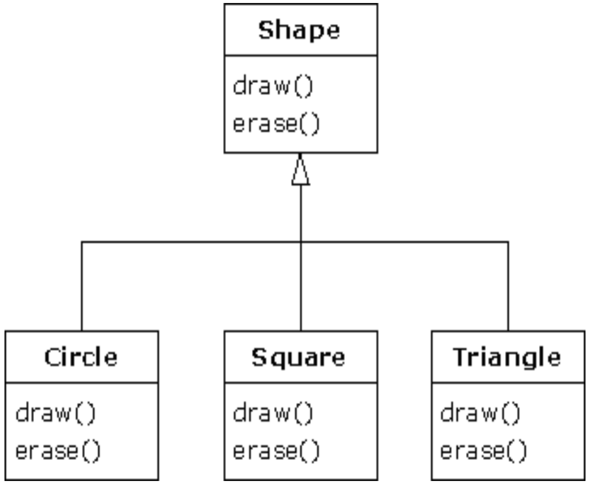
- *A real-life example of polymorphism* -- a person can have different characteristics:

  - A man can have different roles (father, husband, employee) at the same time

  - The same person can behave differently at different situations

- *Polymorphism in C++ programming:* Polymorphism allows ***an object reference variable*** or ***an object pointer*** to 1) **reference different types of objects** and 2) **call the correct member functions**, depending on the type of object being referenced.





Polymorphism

- *Example Recall:* Can we **use base class pointer to call the function display() in the derived class**?

```
#include <iostream>
using namespace std;
class Base {
public:
   void display() {
      cout<< "Base class display() " << endl;
   }
};
class Derived : public Base {
public:
   void display() { //It hides Base::display()
     cout << "Derived class display() " << endl;
   }
};

int main() {
   Derived d;
   d.display();

   Base* ptr = &d;
   ptr->display();
   return 0;
}
```

Base Class Pointers

- Base class pointers and references **only know members of the base classes**, so you cannot use a base class pointer to call a derived class function

- Redefined functions in the derived class will be ignored **unless base class declare the function as *a virtual function***

Virtual Functions

- Virtual (member) function: a function in base class that expects to be redefined in derived class and defined with the keyword `virtual`:

```
 virtual void Y() {...}
```

- Supports dynamic binding: The compiler will not bind the functions at the **compile time**. Instead, the program will bind them at the **runtime** and decide how they will actually behave

- Virtual functions are mainly used to achieve **runtime polymorphism**

- Without `virtual` member functions, C++ uses static binding, where functions are bound at the **compile time**

Polymorphism

- An example of using virtual function for runtime polymorphism

```
#include <iostream>
using namespace std;
class Base {
public:
   virtual void display() {
      cout<< "Base class display()" << endl;
   }
};
class Derived : public Base {
public:
   void display() {
     cout << “Derived class display()” << endl;
   }
};

int main() {
   Base a;
   Derived d;
   a.display();
   d.display();

   Base* ptr = &a;
   ptr->display();
   ptr = &d;
   ptr->display();
   return 0;
}
```

It will call the display() function in **the derived class** now!

Program output:

```
Base class display()
Derived class display()
Base class display()
Derived class display()
```

Polymorphism – Important Notes

- When a member function is declared virtual in a base class, any overridden versions of this function in the derived classes automatically become virtual
- But it is still good to declare the function as virtual in the derived class for documentation purposes

```
#include <iostream>
using namespace std;
class Base {
public:
   virtual void display() {
      cout<< "Base class display()" << endl;
   }
};
class Derived : public Base {
public:
   virtual void display() {
     cout << “Derived class display()” << endl;
   }
};

int main() {
   Base a;
   Derived d;
   a.display();
   d.display();

   Base* ptr = &a;
   ptr->display();
   ptr = &d;
   ptr->display();
   return 0;
}
```

Program output:

```
Base class display()
Derived class display()
Base class display()
Derived class display()
```

Polymorphism – Important Notes

- You place the `virtual` key word only in the function’s declaration or prototype
- If the function is defined outside the class, you do not place the `virtual` key word in the function header

```cpp
#include <iostream>
using namespace std;
class Base {
public:
   virtual void display();
};
void Base::display() { cout<< "Base class display()" << endl; }

class Derived : public Base {
public:
   void display() { cout << "Derived class display() "<< endl; }
};
int main() {
   Base a;
   Derived d;
   a.display();
   d.display();

   Base* ptr = &a;
   ptr->display();
   ptr = &d;
   ptr->display();
   return 0;
}
```

Program output:

```
Base class display()
Derived class display()
Base class display()
Derived class display()
```

Polymorphism – Important Notes

- C++ 11 introduces the keyword `override` in a derived class to explicitly indicate the purpose of overriding a virtual function in a base class
- The keyword override enhances the code clarity and enables compiler checking

A Coding Mistake

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

class Base {
public:
   virtual void display(int x) {
      cout << "Base display: " << x << endl;
   }
};

class Derived : public Base {
public:
   void display(double x) {
      cout << "Derived display: " << x << endl;
   }
};

int main() {
   Derived d;
   Base* ptr = &d;
   ptr->display(42);
   return 0;
}
```

Program output:

```
Base display: 42
```

?

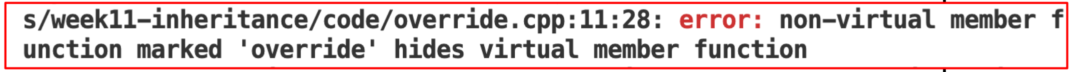
Polymorphism – Important Notes

- C++ 11 introduces the keyword `override` in a derived class to explicitly indicate the purpose of *overriding a virtual function in a base class*
- The keyword override enhances the code clarity and enables compiler checking

A Coding Mistake

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

class Base {
public:
   virtual void display(int x) {
      cout << "Base display: " << x << endl;
   }
};

class Derived : public Base {
public:
   void display(double x) override {
      cout << "Derived display: " << x << endl;
   }
};

int main() {
   Derived d;
   Base* ptr = &d;
   ptr->display(42);
   return 0;
}
```

With the keyword *override*, the compiler can **generate error message** to remind us of the code mistake here.



```
s/week11-inheritance/code/override.cpp:11:28: error: non-virtual member f
unction marked 'override' hides virtual member function
```

Virtual Destructor

- You should always declare a destructor **virtual**, if the class with a destructor could potentially become a base class

- *Reason:*

  - The compiler will **perform static binding on the destructor** if it is not declared virtual

  - This can lead to problems when a base class pointer or reference variable references a derived class object

Virtual Destructor

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

class Animal {
public:
   Animal() { cout << "Animal constructor executing.\n"; }
   ~Animal() { cout << "Animal destructor executing.\n"; }
};

class Dog : public Animal {
public:
   Dog() : Animal() {
     cout << "Dog constructor executing.\n";
   }
   ~Dog() {
     cout << "Dog destructor executing.\n";
   }
};

int main() {
   Animal* myAnimal = new Dog();
   delete myAnimal;
   myAnimal = nullptr;
   return 0;
}
```

Only the base class destructor is executed!

Program output:

```
Animal constructor executing.
Dog constructor executing.
Animal destructor executing.
```

Virtual Destructor

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

class Animal {
public:
   Animal() { cout << "Animal constructor executing.\n"; }
   virtual ~Animal() { cout << "Animal destructor executing.\n"; }
};

class Dog : public Animal {
public:
   Dog() : Animal() {
     cout << "Dog constructor executing.\n";
   }
   ~Dog() {
     cout << "Dog destructor executing.\n";
   }
};

int main() {
   Animal* myAnimal = new Dog();
   delete myAnimal;
   myAnimal = nullptr;
   return 0;
}
```

The destructors of both base and derived class are executed!

Program output:

```
Animal constructor executing.
Dog constructor executing.
Dog destructor executing.
Animal destructor executing.
```

Overriding vs. Redefining

- **Overriding**: a derived class ***overrides*** a *virtual* function of the base class
  -- Overridden functions are *dynamically bound*

- **Redefining**: a derived class ***redefines*** a *non-virtual* function of the base class
  -- Redefined non-virtual functions are *statically bound*

- A virtual function is ***overridden***, and a non-virtual function is ***redefined***

Taxonomy of Polymorphism in C++

- Compile-time (Static) Polymorphism:

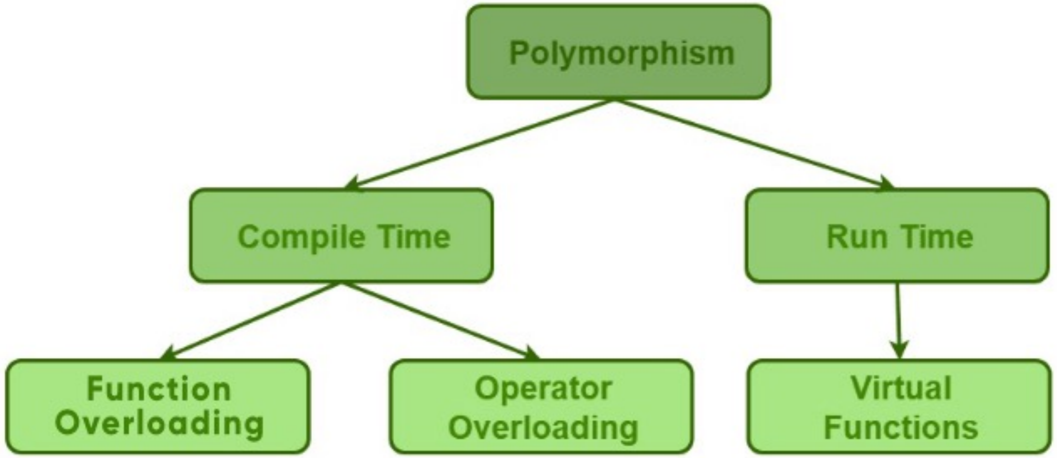
  - Function overloading

  - Operator overloading

  - Templates of functions or classes (a.k.a., generic programming)

- Run-time (Dynamic) Polymorphism:

  - Function overriding



https://www.geeksforgeeks.org/cpp-polymorphism/

Abstract Base Classes and Pure Virtual Functions

- Pure virtual function is a virtual member function declared in a base class like this:

```
    virtual void Y() = 0;
```

- Pure virtual function must have no function definition in the base class

- Pure virtual function must be overridden in a derived class that has objects

Abstract Base Classes and Pure Virtual Functions

- Abstract base class: a class that can have no objects, and serves as a basis for derived classes that may/will have objects

- A class becomes an abstract base class when **at least one of its member functions** is a pure virtual function

Abstract Base Classes and Pure Virtual Functions

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

class Shape {
public:
   // Pure virtual function
   virtual double area() const = 0;
   // Virtual destructor (best practice in abstract classes)
   virtual ~Shape() {}
};

class Circle : public Shape {
private:
   double radius;
public:
   Circle(double r) : radius(r) {}

   // Override the pure virtual function
   double area() const override {
      return M_PI * radius * radius;
   }
};

int main() {
    // Shape s; // ERROR: Cannot instantiate Shape directly
    Shape* shapePtr = new Circle(1.0);
    cout << "Circle area: " << shapePtr->area() << endl;
    delete shapePtr;
    shapePtr = nullptr;
    return 0;
}
```

Program output:

```
Circle area: 3.14159
```

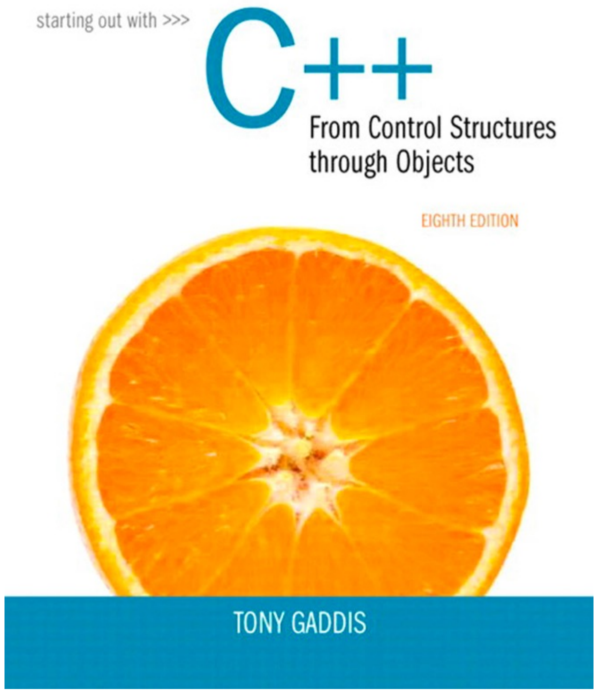
Abstract Base Classes and Pure Virtual Functions

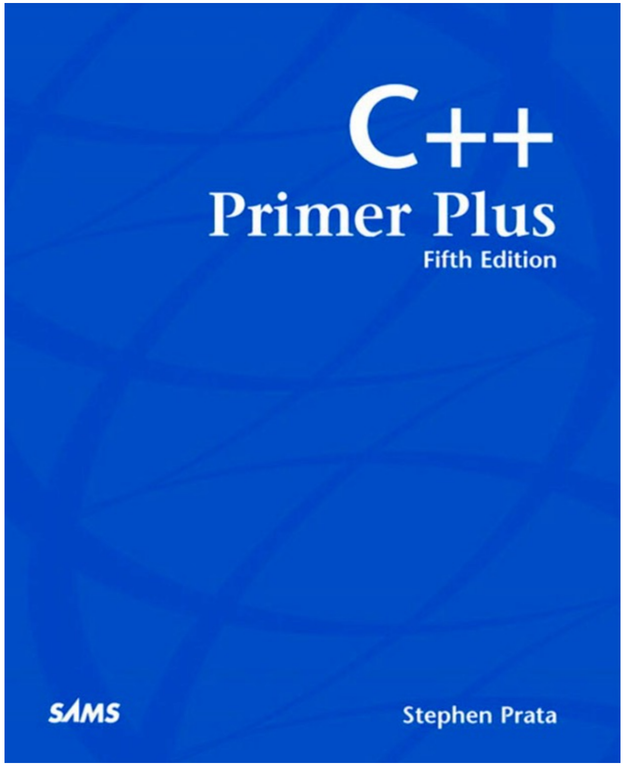
```cpp
#include <iostream>
#include <cmath>
using namespace std;

class Shape {
public:
   // Pure virtual function
   virtual double area() const = 0;
   // Virtual destructor (best practice in abstract classes)
   virtual ~Shape() {}
};

class Circle : public Shape {
private:
   double radius;
public:
   Circle(double r) : radius(r) {}

   // Override the pure virtual function
   double area() const override {
      return M_PI * radius * radius;
  }
};

int main() {
    // Shape s; // ERROR: Cannot instantiate Shape directly
    Shape* shapePtr = new Circle(1.0);
    cout << "Circle area: " << shapePtr->area() << endl;
    delete shapePtr;
    shapePtr = nullptr;
    return 0;
}
```

Program output:

```
Circle area: 3.14159
```

You can add the keyword "override" explicitly to indicate that you intend to do function overriding!





References:

[1] Tony Gaddis. Starting out with C++ from control structures through objects, 8th edition. Chapter 15.

[2] Prata, Stephen. C++ primer plus. Sams Publishing, 2002, 5th edition. Chapters 13.

Questions?

Thank You!