OntarioTech Science

Kourosh Davoudi
kourosh@uoit.ca

Week 7: Inheritance
Functions in Hierarchy

CSCI 1061: Programming Workshop II

Learning Outcomes

In this week, we learn:

- What does inheritance mean?
- How to derive a class form the base class?
- What does protected mean in the class definition?
- Constructor/destructor in the derived class
- Shadowing versus overriding

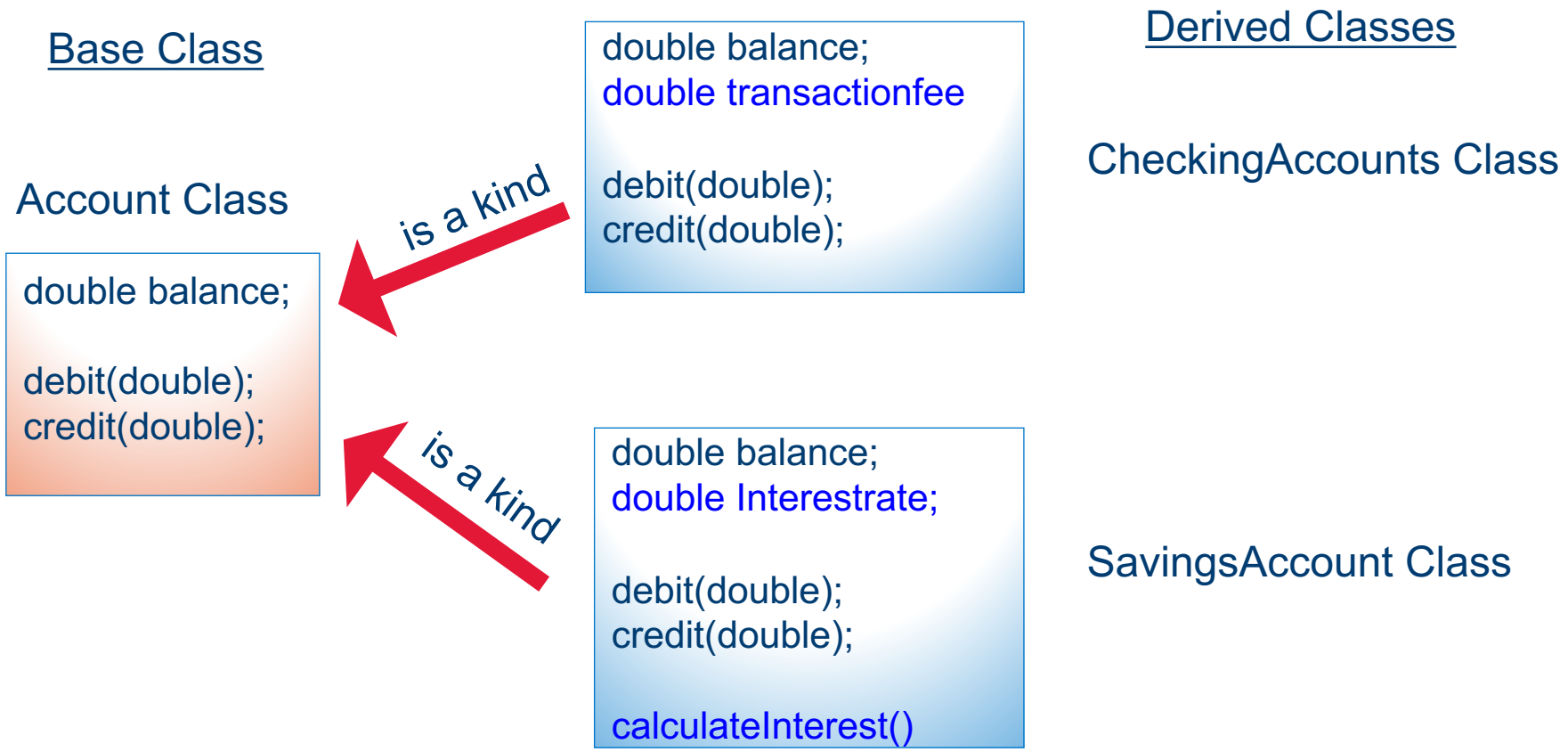
Inheritance and Hierarchy

- Inheritance is the second most prominent concept next to encapsulation.

- OOP Foundations:
  - Encapsulation
  - Inheritance
  - Polymorphism

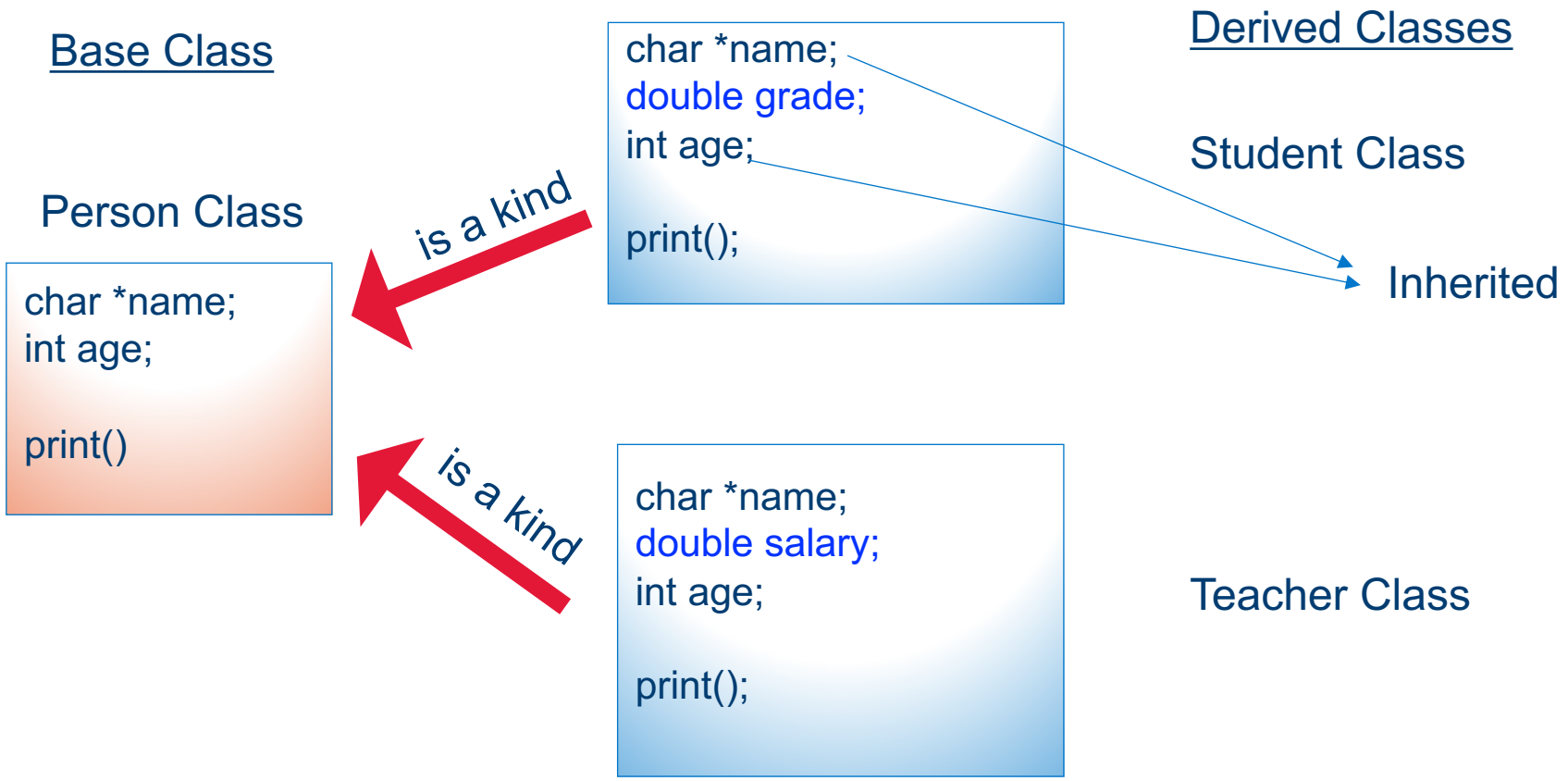
Why Inheritance is importance?

- Inheritance is a mechanism that allows us to use the existing code in class and add/change the functionality that we need.



Why Inheritance is importance?

- Inheritance is a mechanism that allows us to use the existing code in class and add/change the functionality that we need.



How to derive from a base class?

```
class Person{
    protected:
        char * name;

    public:
        void setname(char const *);
        void print();

        Person();                   // default constructor
        Person(char const *);
        Person(Person &);           // copy constructor

        ~Person();

};
```

Base

```
class Student : public Person{
    private:
        double grade;

    public:

        void setgrade(double);
        void printstudent();

        Student();     // default constructor
        Student(char const *, double);
        Student(Student &);  // copy constructor

        ~Student();
};
```

Derived Class

How to derive from a base class?

```
class Person{
    protected:
        char * name;

    public:
        void setname(char const *);
        void print();

        Person();                   // default constructor
        Person(char const *);
        Person(Person &);           // copy constructor

        ~Person();

};
```

Base

```
class Student : public Person{
    private:
        double grade;

    public:

        void setgrade(double);
        void printstudent();

        Student();     // default constructor
        Student(char const *, double);
        Student(Student &);  // copy constructor

        ~Student();
};
```

Derived Class

The **protected** members can be used by the members of derived class
These members cannot be accessed by non-members.

Access in Derived Class

class A: **public** B { ... }

- public members of the base class become public members of the derived class
- protected members of the base class become protected members of the derived

class A: **protected** B { ... }

- public and protected members of the base class become protected members of the derived class.

class A: **private** B { ... }

- public and Protected members of the base class become private members of the derived class.

(Note that there is no direct access to private members of base class)

How to derive from a base class?

```
class Person{
    protected:
        char * name;

    public:
        void setname(char const *);
        void print();

        Person();                   // default constructor
        Person(char const *);
        Person(Person &);           // copy constructor

        ~Person();

};
```

Base

```
class Student : public Person{
    private:
        double grade;

    public:

        void setgrade(double);
        void printstudent();

        Student();     // default constructor
        Student(char const *, double);
        Student(Student &);  // copy constructor

        ~Student();
};
```

Derived Class

A derived class does not by default inherit the constructors and destructors of the base class.

What is the order of calling constructor in base and derived class?

```
Student x;
```

1

```
Student::Student()
{
    cout << "This is default constructor for Student class !" << endl;
}
```

2

3

```
Person::Person()
{
    cout << "This is default constructor for Person class !" << endl;
    name = nullptr;
}
```

Derive class constructor automatically calls the based class constructor before starting the job.

What is the order of calling constructor in base and derived class?

```
Student y("Sarah",79.0);
```

1

You can pass parameter(s) to the base class constructor explicitly

```
Student::Student(char const *n, double g): Person(n)
{
    cout << "This is Student(char const *n, double g) constructor for Person class !" << endl;
    setgrade(g);
}
```

2

3

```
Person::Person(char const *n)
{
    cout << "This is Person(char const *n) constructor for Person class !" << endl;
    setname(n);
}
```

What is the order of calling destructor in base and derived class?

```
Student::~Student()
{
    // The based constructor will be called after the based constructor
    cout << "This is the Student destructor !" << endl;
}
```

Derive class destructor automatically calls the based class destructor after finishing the job.

```
Person::~Person()
{
    cout << "This is the Person destructor !" << endl;
    if(name)
        delete [] name;
}
```

What is the order of calling destructor in base and derived class?

```
int main()
{
    Person x("John");
    x.print();

    Student y("Sarah",79.0);
    y.printstudent();

    return 0;
}
```

```
This is Person(char const *n) constructor for Person class !
```

```
name: John
```

```
This is Person(char const *n) constructor for Person class !
This is Student(char const *n, double g) constructor for Person class !
```

```
name: Sarah
grade: 79
```

```
This is the Student destructor !
This is the Person destructor !
```

```
This is the Person destructor !
```

Central Question

- What will happen if we define a function with the same name (identifier) that already exists in the base class?
  - Shadowing (default)
  - Overriding (when this function is virtual in base class) => next week

Polymorphism

Shadowing

- A member function of a derived class shadows the base class member function with the same identifier.

```
int main()
{
    Person x("John");
    x.print();

    Student y("Sarah",79.0);
    y.print();

    return 0;
}
```

```
void Person::print()
{
    if(name)
        cout << "\tname: " << name << endl;
    else
        cout << "\tThis is an empty object !" << endl;

}
```

Shadow

```
void Student::print()
{
    Person::print();
    if(name)
        cout << "\tgrade: " << grade << endl;
}
```

in Shadowing: The C++ compiler binds a call to the member function of the derived class.

Important Property

- The pointer to base class can points to the object of derived class:

```
Student x("Sarah", 82.6);
Person *p = & x ;
```

Or

```
Person *p = new Student("Jessi", 87.7);
```

- You can access members using pointers:

```
p->print();
```

or

```
(*p).print();
```

Shadowing

- A member function of a derived class shadows the base class member function with the same identifier.

```
int main()
{
    Person x("John");
    x.print();

    Student y("Sarah",79.0);
    y.print();

    Person * p = new Student("Jessi",87.8);
    p-> print();

    return 0;
}
```

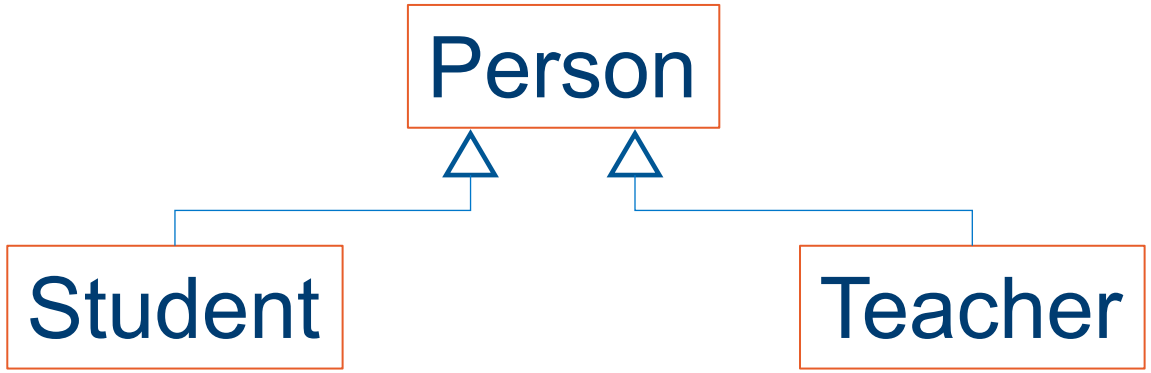
```
void Person::print()
{
    if(name)
        cout << "\tname: " << name << endl;
    else
        cout << "\tThis is an empty object !" << endl;

}
```

Shadow

```
void Student::print()
{
    Person::print();
    if(name)
        cout << "\tgrade: " << grade << endl;
}
```

Exercise

- Derive the Teacher class for the `Person`. `Teacher` class has a salary member instead of grade.
  - Define constructors/destructor
  - Define similar useful member function similar to `Student` class



Shadowing

```
int main()
{
    Person x("John");
    x.print();

    Student y("Sarah",79.0);
    y.print();

    Person * p = new Student("Jessi",87.8);
    p-> print();

    delete p;

    return 0;
}
```

```
This is Person(char const *n) constructor for Person class !
This is Student(char const *n, double g) constructor for Person class !
```

```
name: Jessi
```

```
This is the Person destructor !
```

```
This is the Student destructor !
This is the Person destructor !
```

```
This is the Person destructor !
```

Overriding

```
class Person{
    protected:
        char * name;

    public:
        void setname(char const *);
        virtual void print();

        Person();                   // default constructor
        Person(char const *);
        Person(Person &);           // copy constructor

        virtual ~Person();
};
```

Solution: **virtual** functions can be override by the derived class

Overriding

```
int main()
{
    Person x("John");
    x.print();

    Student y("Sarah",79.0);
    y.print();

    Person * p = new Student("Jessi",87.8);
    p-> print();

    delete p;

    return 0;
}
```

```
This is Person(char const *n) constructor for Person class !
This is Student(char const *n, double g) constructor for Person class !
```

```
name: Jessi
grade: 87.8
```

```
This is the Student destructor !
This is the Person destructor !
```

```
This is the Student destructor !
This is the Person destructor !
```

```
This is the Person destructor !
```

Overriding

```
int main()
{
    Person x("John");
    x.print();

    Student y("Sarah",79.0);
    y.print();

    Person * p = new Student("Jessi",87.8);
    p-> print();

    delete p;

    return 0;
}
```

```
This is Person(char const *n) constructor for Person class !
This is Student(char const *n, double g) constructor for Person class !
```

```
name: Jessi
grade: 87.8
```

```
This is the Student destructor !
This is the Person destructor !
```

```
This is the Student destructor !
This is the Person destructor !
```

```
This is the Person destructor !
```

Polymorphism

```
int main()
{
    Person x("John");
    x.print();

    Student y("Sarah",79.0);
    y.print();

    Person * p = new Person("Alex");
    p-> print();

    delete p;

    p = new Student("Jessi",87.8);
    p-> print();

    delete p;

    return 0;
}
```

This is an example of polymorphism

```
name: Alex
```

```
name: Jessi
grade: 87.8
```