Advanced Programming Concepts with C++

Generalization, Specialization, Inheritance, and Polymorphism

Fall 2021
CSI2372[A]

Amir EAMAN
Postdoc Researcher
Faculty of Engineering, uOttawa

Inheritance

- The main reason for inheritance is **reusing** *existing* code, and so doing less coding.
- By using inheritance, we can create classes based on existing classes.
- The newly created classes through inheritance can have new behaviors and data, or modify existing classes' behaviors.

Student "is-a" Person.

inheritance

Base class
(parent class)
(super class)

```
class Person
{
    // height, weight
    //run(),sleep()

};
```

properties

methods

Derived class
(sub class)
(child class)

```
class Student : public Person
{
    // grade, studentID
    // writeExam(), graduate()

}
```

Generalization

Specialization

Inheritance

By assuming that all members of the Person class have *Public* access, for the Student class we have:

student's
properties
height
weight
grade
studentID

methods
run()
sleep()
writeExam()
graduate()

Base

```
class Person
{
    // height, weight
    //run(),sleep()

};
```

Derived

```
class Student : public Person
{
    // grade, studentID
    // writeExam(), graduate()

}
```

access-specifier
1) public
2) private
3) protected

Inheritance

Access Specifier in inheritance:

- **Public**: (public inheritance)
  - **public** class members of base class are inherited and they are **public** in the derived class.
  - **Protected** class members of base class are inherited and they are **protected** in the derived class.
  - **Private** class members of bases class are ***not* accessible** from derived class.

- **Protected**: (private inheritance)
  - **public** class members of base class are inherited and they are **protected** in the derived class.
  - **Protected** class members of base class are inherited and they are **protected** in the derived class.
  - **Private** class members of bases class are ***not* accessible** from derived class.

- **Private**: (protected inheritance)
  - **public** class members of base class are inherited and they are **private** in the derived class.
  - **Protected** class members of base class are inherited and they are **private** in the derived class.
  - **Private** class members of bases class are ***not* accessible** from derived class.

Inheritance – Constructors and Destructors

- When an object from derived class is created first the constructor of the *base* class and then the constructor of the *derived* class are executed.

- When an object from derived class is destroyed first the destructor of the *derived* class and then the destructor of the *base* class are executed.

- The *constructor* and *destructor* of the base class are **not** inherited by the derived class. Likewise, *copy constructor* and *copy assignment operator* of the base class are **not** inherited by the derived class. However, we still can call these special methods of the base class from the derived class.

Inheritance – Constructors and Destructors

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

class Person
{
 private:
    int height;
    int weight;
public:
    Person() : height(10), weight(10) { std::cout << "This is the base constructor" << endl;  }
    void sleep() { std::cout << "sleeping zzZzZ"; }
    void run() { std::cout << "running ..."; }
    ~Person() { std::cout << "This is the base destructor" << endl; }
};

class Student : public Person
{
private:
    int grade;
    int studentID;
public:
    Student(): grade( 0), studentID(0) { std::cout << "This is the derived constructor" << endl; }
    void sleep() { std::cout << "sleeping zzZzZ in Derived"; }
    void run() { std::cout << "Running in Derived"; }
    void graduate() { std::cout << "graduating..Horaa..."; }
    void writeExam() { std::cout << "writing exam"; }
    ~Student() { std::cout << "This is the derived destructor" << endl; }
};

int main()
{
    Person person;
    std::cout << "-------------------" << std::endl;
    Student student;
    std::cout << "-------------------" << std::endl;
}
```

Name	Value	Type
person	{height=10 weight=10 }	Person
height	10	int
weight	10	int
student	{grade=0 studentID=0 }	Student
Person	{height=10 weight=10 }	Person
grade	0	int
studentID	0	int

```
This is the base constructor
-------------------
This is the base constructor
This is the derived constructor
-------------------
This is the derived destructor
This is the base destructor
This is the base destructor
```

Inheritance – Constructors and Destructors – calling the base class constructor

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

class Person
{
 private:
    int height;
    int weight;
public:
    Person() : height(10), weight(10) { std::cout << "This is the base constructor" << endl;  }
    Person(int arg) : height(arg), weight(arg) { std::cout << "This is the overloaded base constructor with one arg" << endl; }
    void sleep() { std::cout << "sleeping zzZzZ"; }
    void run() { std::cout << "running ..."; }
    ~Person() { std::cout << "This is the base destructor" << endl; }
};

class Student : public Person
{
private:
    int grade;
    int studentID;
public:
    Student(): grade(0), studentID(0) { std::cout << "This is the derived constructor" << endl; }
    Student(int ars) : Person{ ars } ,grade(0), studentID(0) { std::cout << "This is the overloaded derived constructor with one arg" << endl; }
    void sleep() { std::cout << "sleeping zzZzZ in Derived"; }
    void run() { std::cout << "Running in Derived"; }
    void graduate() { std::cout << "graduating..Horaa..."; }
    void writeExam() { std::cout << "writing exam"; }
    ~Student() { std::cout << "This is the derived destructor" << endl; }
};

int main()
{

    Student student(99);
    std::cout << "-------------------" << std::endl;
}
```

```
This is the overloaded base constructor with one arg
This is the overloaded derived constructor with one arg
-------------------
This is the derived destructor
This is the base destructor
```

Inheritance – calling the *copy constructor* of the base class

```cpp
class Person
{
private:
    int height;
    int weight;
public:
    Person() : height(10), weight(10) { std::cout << "This is the base constructor" << endl;  }

    Person(const Person& rhs)
    {
        std::cout << "This is the base Copy Constructor" << endl;
        height = rhs.height;
        weight = rhs.weight;
    }
};

class Student : public Person
{
private:
    int grade;
    int studentID;
public:
    Student(): grade(0), studentID(0) { std::cout << "This is the derived constructor" << endl; }

    Student(const Student& righths)
        : Person (righths)
    {
        std::cout << "This is the derived Copy Constructor" << endl;
        grade = righths.grade;
        studentID = righths.studentID;
    }
};

int main()
{
    Student st1;
    Student st2{ st1 };
}
```

```
This is the base constructor
This is the derived constructor
This is the base Copy Constructor
This is the derived Copy Constructor
```

Inheritance - (1) overriding the base class methods

(2) static binding of method calls --> *polymorphism* will be the solution

```cpp
class Person
{
private:
    int height;
    int weight;
public:
    Person() : height(10), weight(10) { }
    void run() { std::cout << "running .... "; }
};

class Student : public Person
{
private:
    int grade;
    int studentID;
public:
    Student(): grade(0), studentID(0) {  }
    void run() { Person::run();  std::cout << " and liseting music ..." << endl; }
};

int main()
{
    Student st1;
    st1.run();
    std::cout << "-----------------" << std::endl;
    Person* ptr = &st1; // ptr of type Person points to an object of type Student
    ptr->run();
}
```

redefining the run() method

but still it calls run() of person class (How to fix?)

```
running ....  and liseting music ...
-----------------
running ....
```

This is static binding to bind to run() of Person class

Polymorphism: Having multiple forms for a single action

- Polymorphism is the use of a single symbol to represent multiple different types.
- By polymorphism we can perform a single action in different ways (forms).
- Polymorphism occurs in the classes which are related to each other through inheritance, so there must be some relationships between involved classes.
- Polymorphism can happen during compile-time (early binding) or run-time (late binding).
- Polymorphism determines which methods to call.
- Overloading functions, overloading operators, and function overriding using virtual functions and base class pointers are three types of polymorphism.
- In Slide 9, we wanted to call the run() method of the Student class by the following code:

```
Person* ptr = &st1; // a base class pointers points to a derived class
ptr -> run(); //without polymorphism, run() method of base class is called
```

But because the example is non-polymorphic, the compiler binds this method call to the Person run() method. Making the run() method virtual in Person class fixes this issue.

Polymorphism – Virtual Functions

- By adding the `virtual` keyword to a method in a base class, the method becomes a `virtual function`.
- Any derived class which inherits from the base class with a virtual member function can override the virtual function (action). By doing this, compiler binds the method call at runtime (dynamic bind). The override function signature and return type in the derived classes must match exactly to the `virtual function` of the base class.
- As a result, we declare `virtual functions` on the base classes to override them in the derived classes.
- If a base class has a `virtual function`, any base class pointer pointing to a derived class will call the overridden function of the derived class. (see the next page example)

- The base classes with virtual functions must have *virtual destructors* because by deleting the base class pointer which points to a derived class, the destructor of the derived class get called (not the destructor of base class). In the example of next page, we add virtual destructors to classes.
- To the class Person: `virtual ~Person() { std::cout << "Person destructor" << std::endl; }`
- To the class Student: `virtual ~Student() { std::cout << "Student destructor" << std::endl; }`

Polymorphism – Virtual Functions - example

```cpp
using namespace std;
class Person
{
private:
    int height;
    int weight;
public:
    Person() : height(10), weight(10) { }
    virtual void run() { std::cout << "running .... "; }
};

class Student : public Person
{
private:
    int grade;
    int studentID;
public:
    Student(): grade(0), studentID(0) { }
    virtual void run() { Person::run();  std::cout << " and liseting music ..." << endl; }
};

int main()
{
    Student st1;
    st1.run();
    std::cout << "-----------------" << std::endl;
    Person* ptr = &st1; // ptr of type Person points to an object of type Student
    ptr->run();
}
```

overriding the run() virtual function

In the main method:

Person* ptr = &st1; // a base class pointers points to a derived class

ptr -> run(); // This is polymorphic behavior. The run() method of Student is called. (compare to the program in page 9)

```
running ....  and liseting music ...
-----------------
running ....  and liseting music ...
```

Polymorphism– override specifier

To make sure we are implementing the exact signature of the virtual function in a base class, we can add override keyword to a derived virtual function for overriding the virtual function.

```
class Student : public Person
{
private:
    int grade;
    int studentID;
public:
    Student(): grade(0), studentID(0) {  }
    virtual  void run() override { Person::run();  std::cout << " and liseting music ..." << endl; }
    virtual ~Student() { std::cout << "Student destructor" << std::endl; }
};
```

Polymorphism– final specifier for virtual functions

To make sure that a derived class cannot override the virtual function, we can add final to the virtual function.

Example)
If we apply final specifier to the run() virtual function of the Student class, any derived class from Student class cannot override run() method.

```
virtual void run() final
                  { Person::run();  std::cout << " and listening music ..." << endl; }
```

Abstract Class

- Abstract classes are *partially incomplete classes* which have some incomplete functions, and therefore inheriting children must specify their own implementation of the incomplete functions.
- If we know the general implementation for a class but not completely then we go with abstract class.
- Abstract class have the role of *bases classes* in inheritance.
- Abstract class means that we cannot define the class completely so we declare it as an abstract class.
- There is no way to make instances of abstract classes (because ***they are incomplete and generic and abstract***!), so **there is no object of type** abstract class.

- The kind of classes which we have studied so far are concrete classes. They have implementation for all their behaviors.
- Instances of concrete classes are *objects*.

Polymorphism – Pure Virtual Functions and Abstract Classes in C++

- The C++ classes which have ***at least one*** pure virtual function are abstract classes.
- Declaring a pure virtual function:

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

- It is possible to give implementation to pure virtual functions in abstract classes, but typically they don't have any implementation. This is because derived classes must define the proper behavior for pure virtual functions by overriding all of them.
- If a derived class fails to override all of the pure virtual functions of an abstract base class, the derived class is also abstract.
- Abstract classes make it mandatory for derived classes to implement specific functions (pure virtual functions)

Polymorphism – Pure Virtual Functions and Abstract Classes in C++

```
class Transportation    //Transporation is an abstract class
{
private:
    int source;
    int destination;
public:
    virtual void  move() = 0; // pure virtual function which makes Transporation abstract
    virtual ~Transportation() { cout << "this is destructor"; };
};
```

Polymorphism – Pure Virtual Functions

```
class Transportation   //Transporation is an abstract class
{
private:
    int source;
    int destination;
public:
    virtual void  move() = 0; // pure virtual function which makes Transporation abstract
    virtual ~Transportation() { cout << "this is destructor"; };
};

class Car : public Transportation {
private:
    int licensenumber;
public:
    virtual void move() override
    {
        cout << "overiding the pure virtual fucntion of the base abstract class Transporation";
    }
    virtual ~Car() { cout << "this is destructor"; };
};


int main()
{
    Transportaton tr; // Error :no object from an Abstract Class
    Car cr; // OK
    Transportation* ptr = &cr; //OK. a base class pointer points to the derived object
    ptr->move(); // this is polymorphism!
}
```

Polymorphism – Abstract Classes act as interfaces

- There is no language construct **interface** in C++.
- We can use an abstract class with just pure virtual functions as an **interface**.

Polymorphism – The benefits of Polymorphism - Example

```cpp
class Transportation   //Transporation is an abstract class
{
private:
    int source;
    int destination;
public:
    virtual void  move() = 0; // pure virtual function which makes Transporation abstract
    virtual ~Transportation() { cout << "this is destructor of Transportaion" <<endl; };
};

class Car : public Transportation {
private:
    int licensenumber;
public:
    virtual void move() override
    {
        cout << "Car is going on the street" << std::endl;
    }
    virtual ~Car() { cout << "this is destructor of Car"<<endl; };
};

class Airplane : public Transportation {
private:
    int licensenumber;
public:
    virtual void move() override
    {
        cout << "Airplane is in air" <<std::endl;
    }
    virtual ~Airplane() { cout << "this is destructor of Airplane"<<endl; };
};
```

Polymorphism – The benefits of Polymorphism - Example

```cpp
void print(Transportation& I_trp)
{
    I_trp.move();
}

int main()
{

    Car cr;
    Airplane ap;
    print(cr);
    print(ap);

}
```

```
Car is going on the street
Airplane is in air
this is destructor of Airplane
this is destructor of Transportaion
this is destructor of Car
this is destructor of Transportaion
```

Type Casting continue...

In previous lectures, we studied about two type-castings:

1- `static_cast`

2- `const_cast`

In this chapter, we study two other forms of type casting :

3- **`dynamic_cast`** is a casting operator which converts a base-class pointer into a derived-class pointer (in addition, it can cast a derived-class pointer to a base-class pointer). This type of conversion happens at runtime. To cast a pointer of type base class to a pointer of type derived class, the base class must have at least one virtual function, this is because dynamic_cast should be applied to *polymorphic types*. If the dynamic cast doesn't work, it returns 0 (returns nullptr).

- `dynamic_cast` is only used in ***polymorphism***.

`dynamic_cast` <new-pointer-type *> (pointer expression)

The dynamic_cast converts *pointer_expression* to a *new-type* pointer.

Type Casting Continue

Example for **dynamic_cast**

```
int main()
{

    Base bs; // The Base class has a virtual function virMethod
    Derived dv;
    AnotherDerived ad;
    Base* ptr = &ad;

    ptr->virMethod(); // call the virMethod in AnotherDerived class
    AnotherDerived* ptr_to_Anotherderived = dynamic_cast<AnotherDerived*> (ptr);
    //dynamic_cast works in the above statement becuase ptr points to AntherDerived class object
    ptr_to_Anotherderived->virMethod(); // call the virMethod in AnotherDerived class

    Derived* ptr_to_derived = dynamic_cast<Derived*> (ptr);
    //dynamic_cast doesn't work in the above statement becuase ptr points to AntherDerived class object
    //ptr_to_derived is Null because dynamic_cast returns 0
}
```

4- **reinterpret_cast**

This type casting works by reinterpreting the underlying bit pattern, and thus is considered a **dangerous** type casting operator. For example, it can be used to convert a pointer of particular type to another pointer of any type.

reinterpret_cast <new-type> (expression)

Example) char *ptr = reinterpret_cast <char *> (65); //caution!

//the above statement converts an int number to a memory address. No error but "unable to read memory" at runtime

int num = reinterpret_cast <int> (ptr); //num is equal to 65