C/C++ **PRIMER**

LECTURE 7: BASE CLASS POINTERS, POLYMORPHISM AND `virtual` METHODS

Fabian Wermelinger
Harvard University

OUTLINE

- Base class pointers
- Polymorphism and interfaces
- `virtual` methods

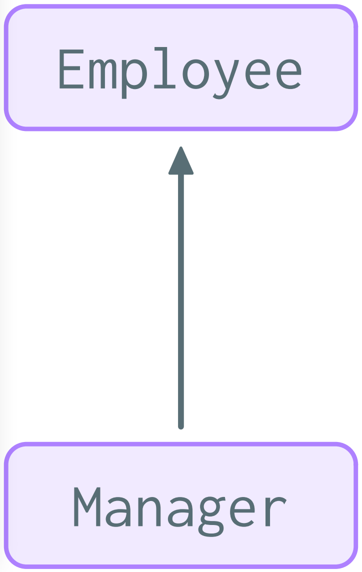
BASE CLASS POINTERS

Recall our small code from last time:

```cpp
class Employee
{
public:
    Employee(std::string name)
        : name_(name) {}

    void print_info() const
    {
        std::cout << "Name: " << name_;
        std::cout << '\n';
    }

    std::string get_name() const
    {
        return name_;
    }

private:
    std::string name_;
};
```

```cpp
class Manager : public Employee
{
public:
    Manager(std::string name,
            std::string department)
          : Employee(name),
            department_(department)
    {
    }

    void print_info() const
    {
        Employee::print_info();
        std::cout << "Head of the ";
        std::cout << department_ << '\n'
    }

private:
    std::string department_;
};
```

Simple inheritance




```cpp
int main(void)
{
    Employee employee("Simpson");
    Manager manager("Mr. Burns", "Nuclear Power Plant");
    Employee *base_ptr = &manager;
    base_ptr->print_info(); // what will this output?
    return 0;
}
```

- Because `Manager` is an `Employee`, it is valid to use a pointer of type `Employee` and assign it an address of an object with type `Manager`.
- What will `base_ptr->print_info()` output?

BASE CLASS POINTERS

Recall our small code from last time:

```
int main(void)
{
    Employee employee("Simpson");
    Manager manager("Mr. Burns", "Nuclear Power Plant");
    Employee *base_ptr = &manager;
    base_ptr->print_info(); // what will this output?
    return 0;
}
```

- Because `Manager` is an `Employee`, it is valid to use a pointer of type `Employee` and assign it an address of an object with type `Manager`.
- What will `base_ptr->print_info()` output?

Base class pointer: `base_ptr` above is a pointer to an object of class `Employee` and its value is the address of the `Manager` object. Because of the inheritance hierarchy, this is correct code. Every object of type `Manager` is an `Employee` too!

Recall the notation: the following two are identical:

- `base_ptr->print_info();`
- `(*base_ptr).print_info();`

Why is `base_ptr` in parenthesis in the second statement?

BASE CLASS POINTERS

Which method is being called?

```cpp
class Employee
{
public:
    void print_info() const;
    std::string get_name() const;
};

class Manager : public Employee
{
public:
    void print_info() const;
};
```

```cpp
int main(void)
{
    Employee employee;
    employee.print_info();

    Manager manager;
    std::cout << manager.get_name();
    manager.print_info();
    manager.Employee::print_info();

    Employee *e_ptr = &manager;
    e_ptr->print_info();

    return 0;
}
```

The *type of the pointer* determines which object's member function is being called ***and not*** the type of the object the pointer is pointing to.

BASE CLASS POINTERS

Which method is being called?

```cpp
class Employee
{
public:
    void print_info() const;
    std::string get_name() const;
};

class Manager : public Employee
{
public:
    void print_info() const;
};
```

```cpp
int main(void)
{
    Employee employee;
    employee.print_info();

    Manager manager;
    std::cout << manager.get_name();
    manager.print_info();
    manager.Employee::print_info();

    Employee *e_ptr = &manager;
    e_ptr->print_info();

    return 0;
}
```

The *type of the pointer* determines which object's member function is being called ***and not*** the type of the object the pointer is pointing to.

BASE CLASS POINTERS

Which method is being called?

```cpp
class Employee
{
public:
    void print_info() const;
    std::string get_name() const;
};

class Manager : public Employee
{
public:
    void print_info() const;
};
```

```cpp
int main(void)
{
    Employee employee;
    employee.print_info();

    Manager manager;
    std::cout << manager.get_name();
    manager.print_info();
    manager.Employee::print_info();

    Employee *e_ptr = &manager;
    e_ptr->print_info();

    return 0;
}
```

The *type of the pointer* determines which object's member function is being called ***and not*** the type of the object the pointer is pointing to.

BASE CLASS POINTERS

Which method is being called?

```cpp
class Employee
{
public:
    void print_info() const;
    std::string get_name() const;
};

class Manager : public Employee
{
public:
    void print_info() const;
};
```

```cpp
int main(void)
{
    Employee employee;
    employee.print_info();

    Manager manager;
    std::cout << manager.get_name();
    manager.print_info();
    manager.Employee::print_info();

    Employee *e_ptr = &manager;
    e_ptr->print_info();

    return 0;
}
```

The *type of the pointer* determines which object's member function is being called ***and not*** the type of the object the pointer is pointing to.

The base class pointer calls the method from its type!

BASE CLASS POINTERS

- Base class pointers can point to any object of a derived class
- Only the ***interface defined in the base class*** is accessible through the base class pointer. *This is foundational to object oriented programming.*
- Derived class pointers ***cannot*** point to a base class object.
- Don't do pointer arithmetic with base pointers! For example, don't do things like this:

```
DerivedClass darray[10]; // array of derived classes
BaseClass *ptr = darray; // base pointer to the beginning of array
ptr[4].some_method();
```

Instead write code like this:

```
DerivedClass darray[10]; // array of derived classes
BaseClass *ptr = darray[4];
ptr->some_method();
```

BASE CLASS POINTERS

```cpp
#include <iostream>

class Base
{
public:
    void who_am_i() const { std::cout << "Base\n"; }
};

class Derived : public Base
{
public:
    void who_am_i() const { std::cout << "Derived\n"; }
    void only_in_derived() const { std::cout << "Not in base interface\n"; }
};
```

Base class pointer

```cpp
int main(void)
{
    Base b, *bp;
    Derived d;

    bp = &b;                // point to base object
    bp->who_am_i();         // OK
    bp = &d;                // point to derived obje
    bp->who_am_i();         // OK
    bp->only_in_derived();  // NOT OK
    return 0;
}
```

Derived class pointer

```cpp
int main(void)
{
    Base b;
    Derived d, *dp;

    dp = &d;                // point to derived obj
    dp->who_am_i();         // OK
    dp->only_in_derived();  // OK
    dp = &b;                // NOT OK
    return 0;
}
```

POLYMORPHISM AND INTERFACES

- In OOP software design, you have a hierarchy of derived classes which you must handle dynamically at runtime.
- ***Example:*** you develop a drawing software where the user can choose from a large number of shapes like rectangles, ellipses, polygons, etc. You only know ***at runtime*** which shape(s) it will be.
- You therefore need to work with base pointers that can handle all cases independently.
- The interaction with the derived class via the base pointer is ***through a common interface*** defined in the base class.
- But we just found out that the base pointer calls the method in the class of its type and not the one defined in the derived class. ***The solution to this problem is called polymorphism.***

POLYMORPHISM AND INTERFACES

- In `C++` there are different forms of polymorphism divided into *compile time polymorphism* and *runtime polymorphism*. Examples are:
  - Function overloading (runtime)
  - Overriding of virtual functions (runtime)
  - Template meta-programming (compile time)
  - Abstract base classes (runtime)
- Recall the difference between overloading and overriding:
  - ***Overload:*** same function/method name with different function signature (e.g. `sqrt(float)` versus `sqrt(int)`)
  - ***Override:*** override what a method of a class is doing while its signature remains unchanged. This is what makes an interface polymorphic.
- To enable `C++` runtime polymorphism of a method that belongs to an interface, the method must be declared `virtual`.

BASE CLASS POINTERS

```
#include <iostream>

class Base
{
public:
    void who_am_i() const { std::cout << "Base\n"; }
};

class Derived : public Base
{
public:
    void who_am_i() const { std::cout << "Derived\n"; }
};
```

Access by base class pointer

```
int main(void)
{
    Base b, *bp;
    Derived d;

    bp = &b;         // point to base object b
    bp->who_am_i(); // OK
    bp = &d;         // point to derived object d
    bp->who_am_i(); // OK
    return 0;
}
```

Result

```
$ ./a.out
Base
Base
```

BASE CLASS POINTERS

```cpp
#include <iostream>

class Base
{
public:
    virtual void who_am_i() const { std::cout << "Base\n"; }
};

class Derived : public Base
{
public:
    void who_am_i() const override { std::cout << "Derived\n"; }
};
```

Access by base class pointer

```cpp
int main(void)
{
    Base b, *bp;
    Derived d;

    bp = &b;          // point to base object b
    bp->who_am_i(); // OK
    bp = &d;          // point to derived object d
    bp->who_am_i(); // OK
    return 0;
}
```

Result

```
$ ./a.out
Base
Derived
```

BASE CLASS POINTERS

```
#include <iostream>

class Base
{
public:
    virtual void who_am_i() const { std::cout << "Base\n"; }
};

class Derived : public Base
{
public:
    void who_am_i() const override { std::cout << "Derived\n"; }
};
```

- The `virtual` keyword enables runtime polymorphism and must be specified in the base class for a *consistent* interface throughout the inheritance hierarchy.
- Once a method is declared `virtual` the property is inherited in derived classes. When you *override* a `virtual` method in a derived class, you should mark it as `override` to make your code more readable and your intentions more obvious. See also https://stackoverflow.com/a/13880342.

`virtual` METHODS

- If a method is declared `virtual` each derived method can `override` the implementation of its inherited version, allowing for runtime polymorphism.
- Resolving polymorphism at runtime comes at a cost. Performance critical code should not be called repeatedly by `virtual` functions. See this post to dig deeper: https://stackoverflow.com/a/3004555.
- The signature of a `virtual` method must not change across inheritance. The `override` keyword can protect you against silent errors related to different signatures:

```
class Base {
public:
    virtual void print() const { }
};

class Derived : public Base {
public:
    void print() { } // this is not the same method as in the base class!
    // void print() override { } // The above compiles, this will not
};
```

`virtual` METHODS

- Constructors ***cannot*** be `virtual.`
- A method in a class template can not be `virtual.`
- Destructors ***can*** be `virtual` and should be if you inherit from a class. For example:

```
1 Base *bp = new Derived;
2 bp->who_am_i();
3 delete bp; // which destructor should be called here?
```

In `line 3` above, a correct implementation calls the destructor of the `Derived` class first, followed by the destructor of the `Base` class. *Recall the `Ghost` class we reverse engineered in lecture 5, its destructor must be virtual to capture the correct behavior.*

> If your destructor is not `virtual` in a class hierarchy with RAII, your code will leak memory. This is a common bug among novices, watch out!

virtual METHODS

Constructor/destructor revisited:

```
class Base
{
public:
    Base() : b_(new double[10]) {}
    Base(const int n) : b_(new double[n]) {}
    ~Base() { delete[] b_; }

private:
    double *b_;
};
```

```
class Derived : public Base
{
public:
    Derived() : d_(new double[10]) {}
    Derived(const int n) : d_(new double[n]) {}
    ~Derived() { delete[] d_; }

private:
    double *d_;
};
```

```
int main(void)
{
    // case 1
    {Derived d0;}

    // case 2
    {Derived d1(20);}

    // case 3
    Base *p = new Derived;
    delete p;

    return 0;
}
```

- This is our current code of interest
- What constructors/destructors are being called for the individual cases in the `main` function?

virtual METHODS

Constructor/destructor revisited:

```
class Base
{
public:
    Base() : b_(new double[10]) {}
    Base(const int n) : b_(new double[n]) {}
    ~Base() { delete[] b_; }

private:
    double *b_;
};
```

```
class Derived : public Base
{
public:
    Derived() : d_(new double[10]) {}
    Derived(const int n) : d_(new double[n]) {}
    ~Derived() { delete[] d_; }

private:
    double *d_;
};
```

```
int main(void)
{
    // case 1
    {Derived d0;}

    // case 2
    {Derived d1(20);}

    // case 3
    Base *p = new Derived;
    delete p;

    return 0;
}
```

- Here we first construct the object `d0` and then we destruct it when we leave the scope (curly braces).
- Is this code correct?

virtual METHODS

Constructor/destructor revisited:

```
class Base
{
public:
    Base() : b_(new double[10]) {}
    Base(const int n) : b_(new double[n]) {}
    ~Base() { delete[] b_; }

private:
    double *b_;
};
```

```
class Derived : public Base
{
public:
    Derived() : d_(new double[10]) {}
    Derived(const int n) : d_(new double[n]) {}
    ~Derived() { delete[] d_; }

private:
    double *d_;
};
```

```
int main(void)
{
    // case 1
    {Derived d0;}

    // case 2
    {Derived d1(20);}

    // case 3
    Base *p = new Derived;
    delete p;

    return 0;
}
```

- Which of the two highlighted base class constructors is being called here?
- ***Recall:*** you must call the base class constructor in the initializer list, otherwise the *default* constructor is called, e.g.:

```
Derived(const int n) : Base(n), d_(new double[n]) {}
```

`virtual` METHODS

Constructor/destructor revisited:

```cpp
class Base
{
public:
    Base() : b_(new double[10]) {}
    Base(const int n) : b_(new double[n]) {}
    ~Base() { delete[] b_; }

private:
    double *b_;
};
```

```cpp
class Derived : public Base
{
public:
    Derived() : d_(new double[10]) {}
    Derived(const int n) : d_(new double[n]) {}
    ~Derived() { delete[] d_; }

private:
    double *d_;
};
```

```cpp
1  int main(void)
2  {
3      // case 1
4      {Derived d0;}
5
6      // case 2
7      {Derived d1(20);}
8
9      // case 3
10     Base *p = new Derived;
11     delete p;
12
13     return 0;
14 }
```

- We now use a base class pointer and perform the same steps as in case 1 before.
- How many and in what order are the two highlighted destructors above called when we execute `line 11` on the left? Is this code correct?

`virtual` METHODS

Constructor/destructor revisited:

```cpp
class Base
{
public:
    Base() : b_(new double[10]) {}
    Base(const int n) : b_(new double[n]) {}
    virtual ~Base() { delete[] b_; }

private:
    double *b_;
};
```

```cpp
class Derived : public Base
{
public:
    Derived() : d_(new double[10]) {}
    Derived(const int n) : d_(new double[n]) {}
    ~Derived() override { delete[] d_; }

private:
    double *d_;
};
```

```cpp
int main(void)
{
    // case 1
    {Derived d0;}

    // case 2
    {Derived d1(20);}

    // case 3
    Base *p = new Derived;
    delete p;

    return 0;
}
```

- This code is correct. It calls the destructors in the correct sequence upon object destruction.
- It is important that your destructors are `virtual` if you inherit from a class which you use as a base pointer type later in your code.

HANDS-ON: (`virtual`) INHERITANCE

Hands-On: `virtual` inheritance (`hands-on/01/README.md`)

Define three classes `A`, `B` and `C`, where `B` inherits from `A` and `C` inherits from `B`. The public interface of all classes should consist of a function `f()` that is supposed to print a different message to `stdout` depending on the class it is implemented. Work through the tasks outlined in the `README.md` file.