C/C++ **PRIMER**

LECTURE 6: RESOURCE ACQUISITION (RAII) INHERITANCE AND OPERATOR OVERLOADING

Fabian Wermelinger
Harvard University

OUTLINE

- Resource acquisition (RAII)
- Operator overloading
- Inheritance and the `this` pointer

RESOURCE ACQUISITION IS INITIALIZATION (RAII)

Resource Acquisition Is Initialization (RAII), is a `C++` programming technique which binds the *life cycle of a resource* that must be acquired before use (allocated heap memory, thread of execution, open socket, open file, locked mutex, disk space, database connection—anything that exists in limited supply) to the lifetime of an object (https://en.cppreference.com/w/cpp/language/raii).

- Encapsulate each resource into a class.
- The class constructor acquires the resource and constructs other attributes or throws an exception if that can not be done.
- The destructor releases the resource and *never* throws an exception.
- The resource is then used through a class instance with *bounded* lifetime. E.g. automatic variable inside a function body.

This ensures that when something goes wrong in a later part of the code which depends on the acquired resource (e.g. something else throws an exception but we continue running the code), the resource will still be freed when the end of lifetime has been reached.

RESOURCE ACQUISITION IS INITIALIZATION (RAII)

Examples:

- Container classes in the Standard Template Library (STL)
- Anything that automatically allocates dynamic memory on the heap and automatically frees the resource again when the object reaches its end of life. *This prevents memory leaks.*
- In `C`, memory management is performed explicitly. It is sometimes not clear who is the *owner* of the resource and premature release of the resource leads to bugs in the code (sometimes hard to find).

RESOURCE ACQUISITION IS INITIALIZATION (RAII)

The rule of three:

- If your class follows the RAII principles (resource management) then the rule of three states that if you define one of the following, you should define all of them (the rule of three)
  1. Destructor
  2. Copy constructor
  3. Copy assignment operator

  We have seen the first two last time and will talk about the third today.
- With the `C++11` standard, move semantics have been introduced. Move semantics allow to *transfer ownership* of resources between instances of the class. They extend the rule of three to the *rule of five*.

RESOURCE ACQUISITION IS INITIALIZATION (RAII)

The rule of five:

- With the `C++11` standard, move semantics have been introduced. Move semantics allow to *transfer ownership* of resources between instances of the class. They extend the rule of three to the *rule of five*:
  1. Destructor
  2. Copy constructor
  3. Copy assignment operator
  4. Move constructor
  5. Move assignment operator

We will not talk about move semantics in this class.

OPERATOR OVERLOADING

In `C++` you can do things like this:

```
 1 class Foo
 2 {
 3     float data_[100];
 4
 5 public:
 6     float &get(int i) { return data_[i]; }
 7     float get(int i) const { return data_[i]; }
 8 };
 9
10 int main(void)
11 {
12     Foo f;
13     f.get(0) = 1;
14     return 0;
15 }
```

What do you think is happening in `line 12`? Which of the two methods in `line 6` and `line 7` is called? 🧐

OPERATOR OVERLOADING

The method returns a reference as an `lvalue` for which it is legal to assign another value to it. We will not go into depth of value categories in `C++`, a good blog post to read is given at the bottom of this slide.

An `lvalue` has *identity* and appears on the *left* side in an assignment. An `rvalue` has no identity (a temporary value) and appears on the right side of an assignment.

```
class Foo
{
    float data_[100];

public:
    float &get(int i) { return data_[i]; }
    float get(int i) const { return data_[i]; }
};
```

Is the return value of the highlighted method an `lvalue` or an `rvalue`?

https://blog.knatten.org/2018/03/09/lvalues-rvalues-glvalues-prvalues-xvalues-help

https://en.cppreference.com/w/cpp/language/value_category

OPERATOR OVERLOADING

Returning references from member functions has two use cases:

1. Change class state through `lvalue` assignment (what we just discussed).
2. Return a reference of something heavy that does not need to be copied. The following code illustrates this:

```
class Foo
{
    HeavyDataType heavy_;

public:
    const HeavyDataType &get_heavy() const { return heavy_; }
};
```

Note how the `const` type qualifier is used, *what does it express?*

References as return arguments in methods allow for an elegant way to overload operators.

OPERATOR OVERLOADING

Example: indexing `operator[]`

```cpp
#include <cassert>

class Foo
{
    float data_[100];

public:
    float &operator[](unsigned int i) // returns lvalue
    {
        assert(i < 100); // using assertions is good practice!
        return data_[i];
    }

    float operator[](unsigned int i) const // returns rvalue
    {
        assert(i < 100); // using assertions is good practice!
        return data_[i];
    }
};

int main(void)
{
    Foo f;
    f[0] = 1;       // calls float &operator[](unsigned int i)
    float a = f[0]; // calls float operator[](unsigned int i) const
    return 0;
}
```

OPERATOR OVERLOADING

Example: copy assignment `operator=` (this one is tricky)

Recall: the rule of three: destructor, copy constructor and copy assignment operator (`operator=`)

```
class Foo
{
    float data_[10];

public:
    Foo &operator=(const Foo &other)
    {
        if (&other == this) {
            return *this;
        }
        for (int i = 0; i < 10; ++i) {
            data_[i] = other.data_[i];
        }
        return *this;
    }
};
```

- The `other` parameter is passed a `const` reference.
- You must check for self assignment. (More recent `C++` idioms may use a `swap` operation instead. We do not discuss that here.)
- You must return a reference to the calling instance. (Why?)

OPERATOR OVERLOADING

Example: copy assignment *`operator=`* (this one is tricky)

```cpp
class Foo
{
    float data_[10];

public:
    Foo(const int v) { /* pass */ }

    Foo &operator=(const Foo &c)
    {
        if (&c == this) {
            return *this;
        }
        for (int i = 0; i < 10; ++i) {
            data_[i] = c.data_[i];
        }
        return *this;
    }
};
```

What do you think is being called in the following code?

```cpp
int main(void)
{
    Foo f = 1; // what is being called here?
    Foo g(2);  // what is being called here?
    f = g;     // what is being called here?
    return 0;
}
```

OPERATOR OVERLOADING

***Example:* copy assignment `operator=`** (this one is tricky)

- If you *do not* provide a destructor, copy constructor or copy assignment operator in your `class`, then the compiler will generate a default version for you.
- Most of the time the default version does not what you need it to do!
- ***Example:*** Dynamic memory on the heap. The default copy constructor and copy assignment operator will copy the ***value of the pointer*** and not the underlying data (shallow copy). In that case the copied object *shares* the data with the owner object. In most cases this will be a bug!
- For RAII, you ***must*** provide implementations for a destructor, copy constructor and copy assignment operator (and move constructor and move assignment operator in case you need move semantics).

See also: https://en.cppreference.com/w/cpp/language/operators

OPERATOR OVERLOADING

Hands-On: Copy assignment operator (`hands-on/01/README.md`)

The code in `copy_assignment_bug.cpp` generates the following output

```
6.3661e-42 0 9.18341e-41 1.4013e-45 -6.00393e-18 4.59121e-41 -179.948 3.0631e-41 2.8026e-45 0
3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3
```

while we would expect the following output

```
3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3
```

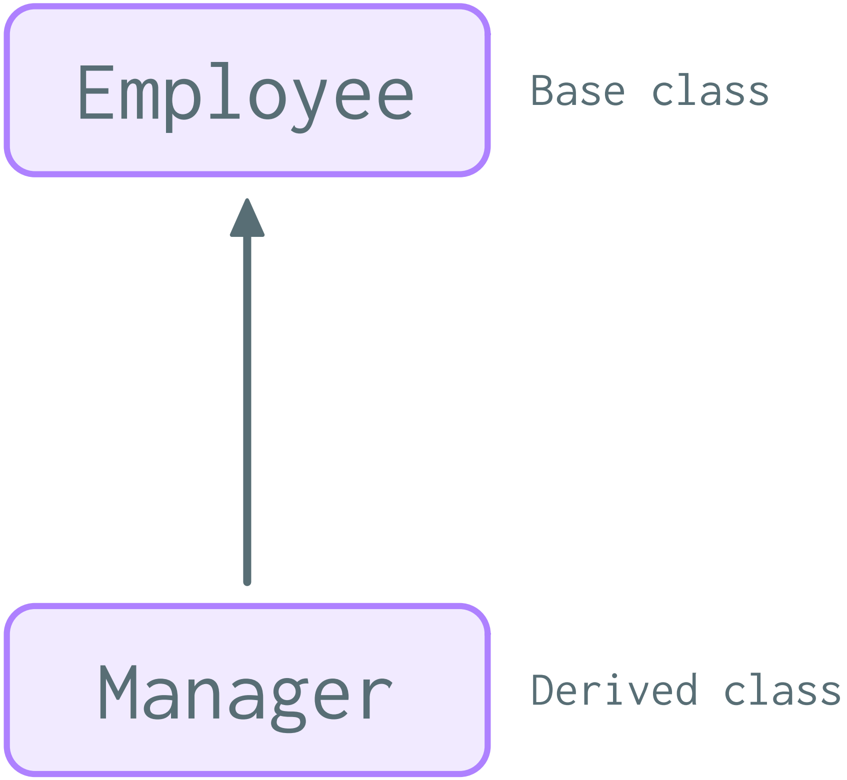
We will work on this problem together and we need a debugger to get a deeper insight. You can install for example gdb (https://www.gnu.org/software/gdb/) via `sudo apt-get install gdb`.

INHERITANCE AND THE `this` POINTER

- One of the key elements in OOP is inheritance: *a new class can inherit the data members and member functions of another already defined class.*
- The derived class may add new data members or methods, shadow members or extend functionality through polymorphism.
- ***Examples:***
  - Student, Assistant and Professor are University members
  - Triangles, Circles, Rectangles are shapes
  - Dogs, Cats, Mice and Elephants are animals
- Inheritance allows for a modular design by encapsulating data and sharing functionality that can further be extended or modify where needed. The extension of a feature *does not* duplicate code in the design hierarchy.

INHERITANCE AND THE `this` POINTER

- We work through an example with *base class* `Employee` and derived class `Manager`.
- Keep in mind that every `Manager` is an `Employee` but not every `Employee` is a `Manager`.
- A derived object can be treated the same as an object of its base class. We see what this means when we talk about base class pointers.
- A derived class can not access the private member of its base class.
- A derived class constructor has to call one of the constructors for its base class. Default constructor is called otherwise.



INHERITANCE AND THE `this` POINTER

The base class: `Employee`

```cpp
#include <iostream>
#include <string>

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

    void print_info() const { std::cout << "Name: " << name_ << std::endl; }
    std::string get_name() const { return name_; }

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

Recall:

- `private`: can not be accessed from outside the class.
- `public`: can be accessed from outside the class.

INHERITANCE AND THE `this` POINTER

The derived class: Manager

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

    void print_info() const {
        std::cout << "Name: " << name_ << std::endl;
    }
    std::string get_name() const { return name_; }

private:
    std::string name_;
};

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 " << department_ << std::endl;
    }

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

INHERITANCE AND THE `this` POINTER

The derived class: Manager

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

    void print_info() const {
        std::cout << "Name: " << name_ << std::endl;
    }
    std::string get_name() const { return name_; }

private:
    std::string name_;
};

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 " << department_ << std::endl;
    }

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

- `Manager` inherits `public` access rights. Recall that `public` and `private` allow or disallow access to the class members. The `protected` access modifier allows *derived* classes access to members that are *protected* from the public (similar to `private` but less strict).
- If you call a constructor for the base class, you must call it in the *initializer list*.
- Additional data members can be defined in derived classes with appropriate access modifiers.

INHERITANCE AND THE `this` POINTER

The derived class: Manager

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

    void print_info() const {
        std::cout << "Name: " << name_ << std::endl;
    }
    std::string get_name() const { return name_; }

private:
    std::string name_;
};

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 " << department_ << std::endl;
    }

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

- We can *overload* member functions in the derived class. (Note the emphasis on *overload*.)
- The derived class calls the `print_info()` explicitly via the scope operator "`::`", that is `Employee::print_info()`.
- After that call it adds one more line of output. `Manager` *extends* the functionality of `print_info()` in the `Employee` class.

INHERITANCE AND THE `this` POINTER

The derived class: Manager

```
int main(void)
{
    Employee e("Simpson");
    Manager m("Mr. Burns", "Nuclear Power Plant");
    e.print_info();
    m.print_info();
    return 0;
}
```

What output do you expect?

INHERITANCE AND THE `this` POINTER

Access modifiers revisited:

- The usage of `public`, `protected` and `private` within the class definition should now be clear.
- Naturally they are linked to inheritance:
  - The access modifiers keep their rights as they were.

```
class Derived : public Base
```

  - The inherited members with `public` access become `protected` in the derived class.

```
class Derived : protected Base
```

  - The inherited members with `public` and `protected` access become `private` in the derived class.

```
class Derived : private Base
```

INHERITANCE AND THE `this` POINTER

Access modifiers revisited: examples

Base class

```
class Base
{
public:
    void pub() {}
protected:
    void pro() {}
private:
    void pri() {}
};

int main(void)
{
    Base base;
    base.pub(); // OK
    base.pro(); // NOT OK
    base.pri(); // NOT OK

    return 0;
}
```

Public inheritance

```
class Deriv1 : public Base
{
    void do_things()
    {
        pub(); // OK
        pro(); // OK
        pri(); // NOT OK
    }
};

int main(void)
{
    Deriv1 dpub;
    dpub.pro(); // NOT OK

    return 0;
}
```

Protected inheritance

```
class Deriv2 :
    protected Base
{
    void do_things()
    {
        pub(); // OK
        pro(); // OK
        pri(); // NOT OK
    }
};


int main(void)
{
    Deriv2 dpro;
    dpro.pub(); // NOT OK

    return 0;
}
```

Private inheritance

```
class Deriv3 :
    private Base
{
    void do_things()
    {
        pub(); // OK
        pro(); // OK
        pri(); // NOT OK
    }
};

class Deriv4 :
    public Deriv3
{
    void do_things()
    {
        pub(); // NOT OK
        pro(); // NOT OK
        pri(); // NOT OK
    }
};
```

INHERITANCE AND THE `this` POINTER

The `this` pointer

- We have already seen the `this` pointer in an earlier lecture.
- It showed up again today in *operator overloading*.
- The value of the `this` pointer is the address of the *implicit object parameter* (the object on which non-static member functions are being called).
- It can appear in the following contexts:
    1. Within the body of non-static member functions.
    2. Within the declaration of non-static member functions after the type qualifier sequence.
    3. Within default member initialization.
- It is similar to the `self` reference in `python` except that `this` is implicit

INHERITANCE AND THE `this` POINTER

The `this` pointer: examples

```
class Foo
{
    int x;

    void bar()
    {
        x = 6;       // implicit use of this-> (same as this->x = 6;)
        this->x = 5; // explicit use of this->
    }

    void bar() const // cv-qualifier
    {
        x = 7; // NOT OK: *this is constant
    }

    void bar(int x) // parameter x shadows the member with same name
    {
        this->x = x; // this->x and x are not the same in this context!
    }

    int y;
    Foo(int x)
        : x(x),      // parameter x initializes member x
          y(this->x) // y is initialized using member x
    {
    }
};
```