C/C++ **PRIMER**

LECTURE 5: BASICS OF C++ CLASSES

Fabian Wermelinger
Harvard University

OUTLINE

- Object versus data oriented designs
- `C++` classes:
  - Constructors
  - Destructors
  - Initializer lists
  - Member functions and attributes
- Access modifiers

OBJECT VERSUS DATA ORIENTED DESIGNS

OBJECT VERSUS DATA ORIENTED DESIGNS

We will not discuss in much depth what object oriented programming is. We will see how these concepts are exploited in the `C++` programming language.

The main concepts of object oriented programming are the following:

- Data encapsulation and interfaces (API)
- Inheritance
- Polymorphism

"Objects" are meant to be owners of data and the *interface* defined for that object dictates how the data can be accessed or transformed.

OBJECT VERSUS DATA ORIENTED DESIGNS

Object oriented design ***is not always a good choice*** if you care about performance.

But software design is not all about performance. You also want code that is *modular*, *extensible* and *maintainable*. These are high-level requirements, while performance is low-level.

The main intention of any program is ***transformation of data***. An input representation of data is transformed through *algorithms* to another representation of output data.

Obviously, the efficiency of such transformations highly depends on how you *interpret or read* the data in your program when you process it.

OBJECT VERSUS DATA ORIENTED DESIGNS

Usually you have many objects of the same type that is stored in arrays. Examples are: particles, computational cells in a structured grid or pixels of images.

- ***Array of structures:*** object oriented design.

```
struct Particle {
    double x, y, z, u, v, w; // position and velocities in 3D
};

Particle particles[100]; // array of structures
```

- ***Structure of arrays:*** data oriented design.

```
struct Particles {
    double x[100];
    double y[100];
    double z[100];
    double u[100];
    double v[100];
    double w[100];
};

Particles particles; // structure of arrays
```

C++ CLASSES

C++ CLASSES

A `C++` class called `Foo`:

```cpp
class Foo
{
    int i1_; // private attribute

public:
    int i2;                    // public attribute
    int g() { return i1_; }    // public member function
    void s(int i) { i1_ = i; } // public member function
};

int main(void)
{
    Foo foo;
    foo.i1_ = 1; // Compile time error! Attribute is private
    foo.i2 = 2;  // OK, public attribute
    foo.s(3);    // call a public method to set i1_ internally
    return 0;
}
```

C++ CLASSES

A C++ class called Foo (dynamic allocation):

```
 1 class Foo
 2 {
 3     int i1_; // private attribute
 4
 5 public:
 6     int i2;                    // public attribute
 7     int g() { return i1_; }    // public member function
 8     void s(int i) { i1_ = i; } // public member function
 9 };
10
11 int main(void)
12 {
13     // dynamic allocation on heap
14     Foo *foo = new Foo;
15     foo->i2 = 2;   // class member access through '->' operator (pointer only)
16     (*foo).i2 = 2; // dereference pointer first followed by normal '.' access
17     foo->s(3);     // does not matter if attribute or member function
18     delete foo;    // clean up when you are done
19     return 0;
20 }
```

Are the parenthesis required in line 16?

C++ CLASSES

A `C++` class called `Foo`:

```
class Foo
{
    int i1_; // private attribute

public:
    int i2;                      // public attribute
    int g() { return i1_; }     // public member function
    void s(int i) { i1_ = i; } // public member function
};
```

- A class is defined with the `class` keyword.
- The class definition is contained within `{}` (same as for function definitions).
- The class definition must be terminated with a `;` (semi-colon).

C++ CLASSES

A `C++` class called `Foo`:

```
class Foo
{
    int i1_; // private attribute

public:
    int i2;                      // public attribute
    int g() { return i1_; }     // public member function
    void s(int i) { i1_ = i; } // public member function
};
```

- Anything inside the class definition is *private* by default (data encapsulation).
- You can use the `public` access specifier to allow direct data access from outside.
- There are `public`, `protected` and `private` access specifiers (more in the last part).

C++ CLASSES

A `C++` class called `Foo`:

Every instance of a class in `C++` has a pointer to *itself*. The name of this pointer is called `this`.

```
1  class Foo
2  {
3      int value_;
4
5  public:
6      Foo(int v) : value_(v) {}
7
8      // method to return whoever has a larger value
9      Foo *getMaximum(Foo *other)
10     {
11         if (this->value_ > other->value_) {
12             return this;
13         } else {
14             return other;
15         }
16     }
17 };
18
19 int main(void)
20 {
21     Foo foo_small = 1;
22     Foo foo_large = 100;
23     Foo *foo_max = foo_small->getMaximum(foo_large);
24     // foo_max points to foo_large, the following is the same
25     foo_max = foo_large->getMaximum(foo_small);
26     return 0;
27 }
```

- Writing `line 11` like this is also valid:

```
if (value_ > other->value_) {
```

- The `this` pointer can be used to express meaning more explicitly but is often omitted when clear from the context.
- Line 12 returns a pointer of the current class instance:

```
return this;
```

- The `this` pointer exists *implicitly*.

C++ CLASSES

Class constructors:

- Constructors are used to initialize data in a class.
- It may include dynamic memory allocation on the heap. Data encapsulation means that the managing class is responsible for the allocation of resources.
- If you *do not define a constructor* in your class, the compiler will create a default constructor.
- You can define as many constructors as you need.

C++ CLASSES

Class constructors:

Simple class, default constructor generated by compiler but `size_` is *uninitialized*:

```
class Foo
{
    int size_;
};
```

The compiler generated default constructor will only make sure the object is constructed correctly, it will not initialize your data.

Same with a constructor:

```
class Foo
{
    int size_;
public:
    Foo(int N) { size_ = N; }
};
```

You should avoid initializing attributes in the constructor body. If you allocate resources *dynamically* then you must initialize the data in the body.

With constructor and initializer list:

```
class Foo
{
    int size_;
public:
    Foo(int N) : size_(N) { }
};
```

You should always favor the initializer list for data initialization. If your attribute is `const` you must use it.

C++ CLASSES

Class constructors:

- Default constructors have no arguments:

```cpp
class Foo
{
    int size_;
public:
    Foo() : size_(0) { } // default constructor
    Foo(int N) : size_(N) { } // not a default constructor
};
```

- If you write constructors that are not default constructors, then those *must* be used. Default construction is not possible in this example:

```cpp
class Foo
{
    int size_;
public:
    Foo(int N) : size_(N) { } // not a default constructor
};

int main(void)
{
    Foo foo; // Compile-time error! No default constructor available
    return 0;
}
```

C++ **CLASSES**

Class destructors:

- Classes are used to encapsulate data in OOP
- If the data needs to be allocated dynamically on the heap, it is usually done in the ***constructor***.
- Because a class is responsible for the memory management, you need a way to free up the claimed resources when it is no longer needed.
- A ***destructor*** is called when a class instance is destroyed (has reached the end of scope) and is used mainly to free allocated resources.

C++ CLASSES

Class destructors:

A destructor takes no arguments and has the same form as a default constructor with a tilde " ~ " pre-pended to its name (which must be the name of the class):

```cpp
class Foo
{
    int size_;
public:
    Foo() : size_(0) { } // default constructor
    ~Foo() { /* body of the destructor */ }
};
```

C++ CLASSES

Class destructors:

If there is an inheritance hierarchy, destructors will be called from the derived class(es) upwards to the base class ***only if*** they are declared as `virtual`:

```
class Foo
{
    int size_;
public:
    Foo() : size_(0) { } // default constructor
    virtual ~Foo() { /* body of the destructor */ }
};
```

If you forget that your destructors should be `virtual`, your implementation will be leaking memory, a common bug of novice `C++` programmers.

C++ CLASSES

Class destructors:

Example with dynamic memory allocation:

```
class Foo
{
    int size_;
    double *data_;

public:
    Foo(int N) : size_(N) { data_ = new double[N]; }
    ~Foo() { delete[] data_; }
};
```

When is a destructor called: when the instance reaches an end of scope (for example when a function body finishes):

```
1 int main(void)
2 {
3     Foo f(10); // constructor is being called for this f
4     { // start of scope
5         Foo f(10); // constructor call for another f (shadows previous f)
6     } // end of scope for f: destructor is called here
7     return 0;
8 } // end of scope for f instantiated in line 3
```

C++ CLASSES

Copy constructors:

Copy constructors are special constructors that create new instances based on an existing instance. Calling a copy constructor can be *expensive* in certain cases and you should be clear about when they are called. ***A simple example:***

```
int main(void)
{
    int i0 = 1;
    int i1(i0); // copy constructor of built-in int type
    return 0;
}
```

C++ CLASSES

Copy constructors:

Copy constructors in classes have the following form:

```
class Foo
{
    int size_;

public:
    Foo() : size_(0) {} // default constructor
    virtual ~Foo() { /* body of the destructor */ }

    Foo(const Foo &c) : size_(c.size_) { /* nothing else to do in this case */ }
};
```

C++ CLASSES

Member functions:

Member functions of classes (also called *methods*) are just functions defined inside the class itself. Member functions can access private attributes in a class:

```cpp
class Foo
{
    int size_;

    void private_method_() { std::cout << "I am a private method\n"; }

public:
    Foo() : size_(0) {} // default constructor
    virtual ~Foo() { /* body of the destructor */ }

    Foo(const Foo &c) : size_(c.size_) { /* nothing else to do in this case */ }

    void public_method() { std::cout << "I am a public method\n"; }
};
```

C++ CLASSES

Member functions:

Member functions can be `const` qualified. This expresses that calling the member function does not modify the state of the instance:

```
class Foo
{
public:
    void hello() { std::cout << "Hi there!\n"; }
    void hello_const() const { std::cout << "Hi there!\n"; }
};

int main(void)
{
    Foo f;
    f.hello();       // OK
    f.hello_const(); // OK
    const Foo cf;
    cf.hello();       // NOT OK
    cf.hello_const(); // OK
    return 0;
}
```

C++ CLASSES

Member functions:

You can not modify attributes with a *`const` qualified* member function:

```
class Foo
{
    int a_
public:
    void const_qualified() const { a_ = 1; } // WILL NOT COMPILE!
};
```

By qualifying member functions as `const` you are explicit about your intention and you should always favor this habit. It will help others reading your code and more importantly reduce the attack surface for bugs.

ACCESS MODIFIERS

There are *3* access modifiers in `C++` that are used to define access to the encapsulated data of a class. These access right involve inheritance obviously and you will see them again when we talk about inheriting from classes.

Access modifier	Explanation
`public`	Access to the attributes and methods in this section is granted to anyone. This is similar to a plain old `struct` in `C`. No data encapsulation
`protected`	Access is only granted via methods in this class *as well as* via methods in derived classes.
`private`	Access is only granted via methods in this class.

ACCESS MODIFIERS

Examples:

```
class Foo
{ // default class access is private
private: // I would not have to write this (default is private)
    // class private sections are not inherited
    int size_; // I am a private class attribute
    void private_method_() const { std::cout << "I am a private method\n"; }
protected:
    // attributes and methods in here are inherited but are not accessible
    // by the public
    int another_size_; // I am a protected class attribute (like private)
    void protected_method_() const { std::cout << "I am a protected method\n"; }
public:
    // attributes and methods in here are inherited as public by default
    int size; // I am a public class attribute, everybody can access me
    void public_method() const { std::cout << "I am a public method\n"; }
};
```

HANDS-ON: REVERSE ENGINEER A CLASS

You are given a `main.cpp` file that you can compile (once you fixed a small issue). The code includes a header `Ghost.h` for which you are given a skeleton code for a class `Ghost` and one that derives from it called `Rider`. Reverse engineer the classes `Ghost` and `Rider` such that the following output is reproduced when you execute the code in `main.cpp`:

```
$ ./a.out
Construct A
Construct A
Construct B
Copy A
Ghost BOO!
Rider BOO!
Destruct A
Destruct B
Destruct A
Destruct A
```

The description above is also contained in `hands-on/01/README.md`.