Haotao Lai (Eric)
h_lai@encs.concordia.ca

Lab Instructor

Section: B-X 9999 --W---- 20:30 22:20 H929

Name: Haotao Lai (Eric)

Office: EV 8.241

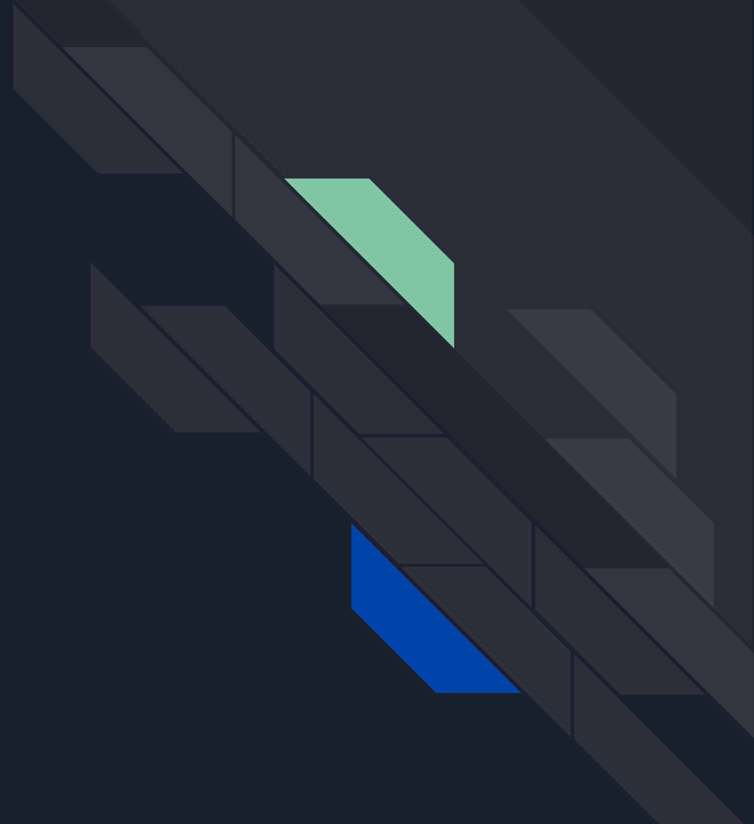
Email: h_lai@encs.concordia

Website: http://laihaotao.me/ta

Content

- class, struct
- constructors, destructors
- inheritance

Class vs. Struct

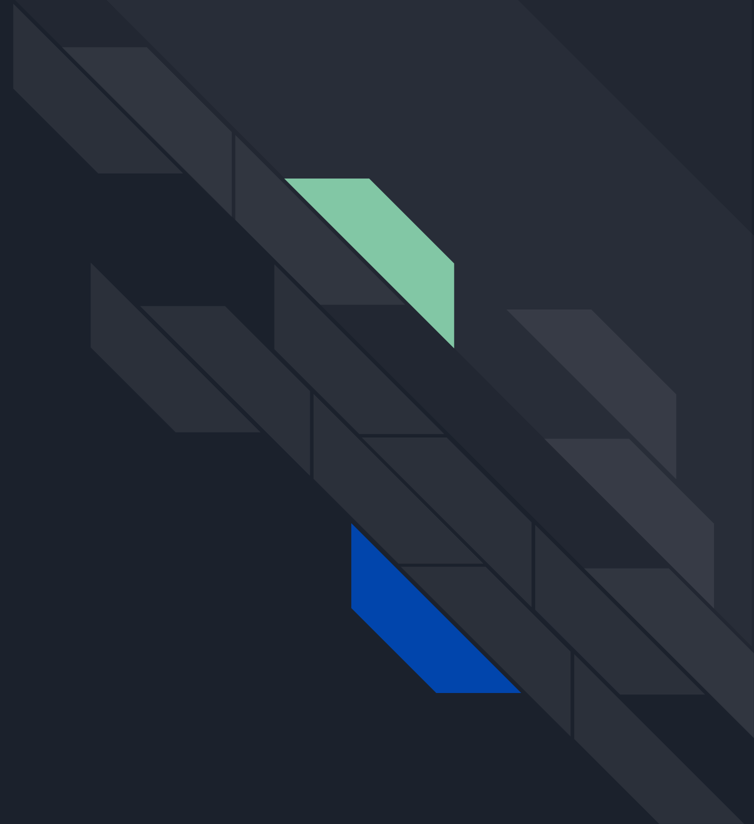


The only difference of struct and class is that all the members inside a struct by default the accessibility to them is public while inside the class is private.

Some “pure C++ programmers” said when you write your new code, you should not use struct anymore, use class instead. Struct is just the legacy from C.

Also some other “neutral” programmers say “I would recommend using structs as plain-old-data structures without any class-like features, and using classes as aggregate data structures with private data and member functions.”

Constructor



Three type of constructor

- Default constructor
- Parameterized Constructor
- Copy constructor

Just like the default constructor in Java, if you don't provide one, the C++ compiler will generate one for you. Basically, it is just an empty constructor. It will initialize all the primitive type variables to the default value and pointer variable to nullptr.

```
class Node {
public:
    Node(); // default constructor
    Node(float x, float y, Node *anotherNode);
    Node(Node &anotherNode);

private:
    float x, y;
    Node *next;

    // some other code
    // ...
};
```

Parameterized Constructor

Also a common used constructor, basically takes all the parameters the class need to create an object and store them as member variables. But the compiler will not generate it for you automatically.

```
class Node {
public:
    Node();
    Node(float x, float y, Node *anotherNode); // parameterized constructor
    Node(Node &anotherNode);

private:
    float x, y;
    Node *next;

    // some other code
    // ...
};
```

Constructor with “explicit” keyword

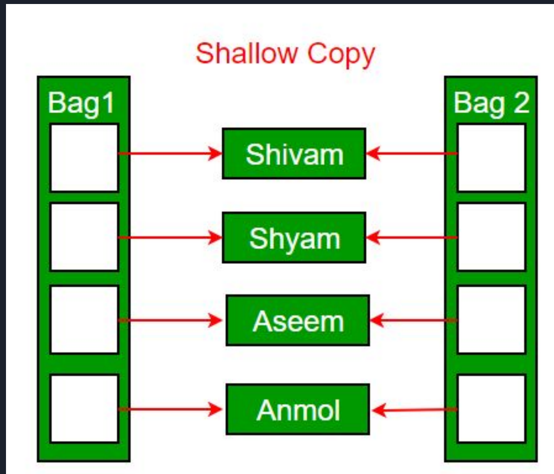
The compiler is allowed to make one implicit conversion to resolve the parameters to a function. What this means is that the compiler can use constructors callable with a single parameter to convert from one type to another in order to get the right type for a parameter.

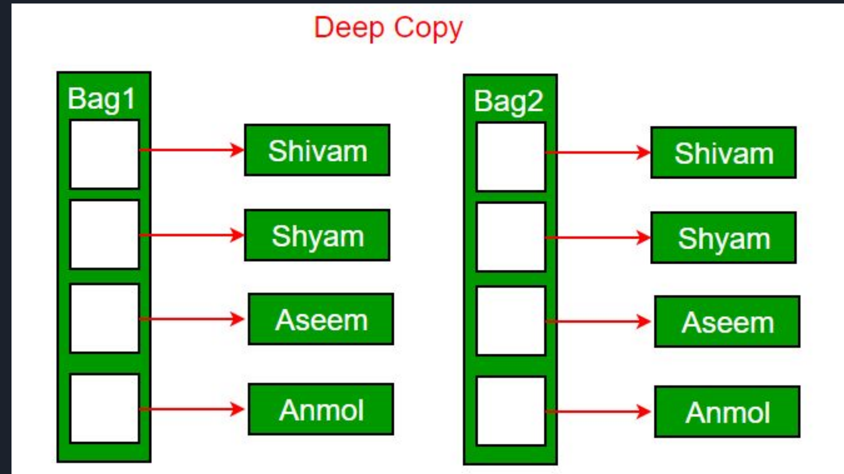
```
class Foo {
public:
    /* explicit */ Foo(int foo) : m_foo(foo) {
        cout << "Foo single parameter constructor" << endl;
    }
    int getFoo() { return m_foo; }
private:
    int m_foo;
};
int doBar(Foo foo) {
    return foo.getFoo();
}
int main() {
    int i = doBar(42);
    cout << "result from doBar: " << i << endl;
    return 0;
}
```

Copy constructor, the compiler will generate one for you if you don’t declare one. If your class has pointer variable or dynamic memory allocation, you have to provide your own copy constructor.

```
class Node {
public:
    Node();
    Node(float x, float y, Node *anotherNode);
    Node(Node &anotherNode); // copy constructor

private:
    float x, y;
    Node *next;

    // some other code
    // ...
};
```

Shallow Copy vs. Deep Copy





The copy constructor is used to:

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

Copy Constructor

```
int main() {
    // case 1
    Node node1;
    Node node2(node1);

    // case 2
    foo(node1);

    // case 3
    Node node3 = getNode();

    return 0;
}
```

```
void foo(Node node) {
    // do something with node
}

Node getNode() {
    Node node;
    // do something with node
    return node;
}
```

Assignment Operator

When an object has been created and you would like to assign it to some variables, then the assignment operator will be called.

If you try to use one object to create another object then the copy constructor will be invoked.

The Rule of Three

There's a well-established C++ principle called the "rule of three" that almost always identifies the spots where you'll need to write your own copy constructor and assignment operator. If this were a math textbook, you'd probably see the rule of three written out like this:

Theorem: If a class has any of the following three member functions:

- Destructor
- Copy Constructor
- Assignment Operator

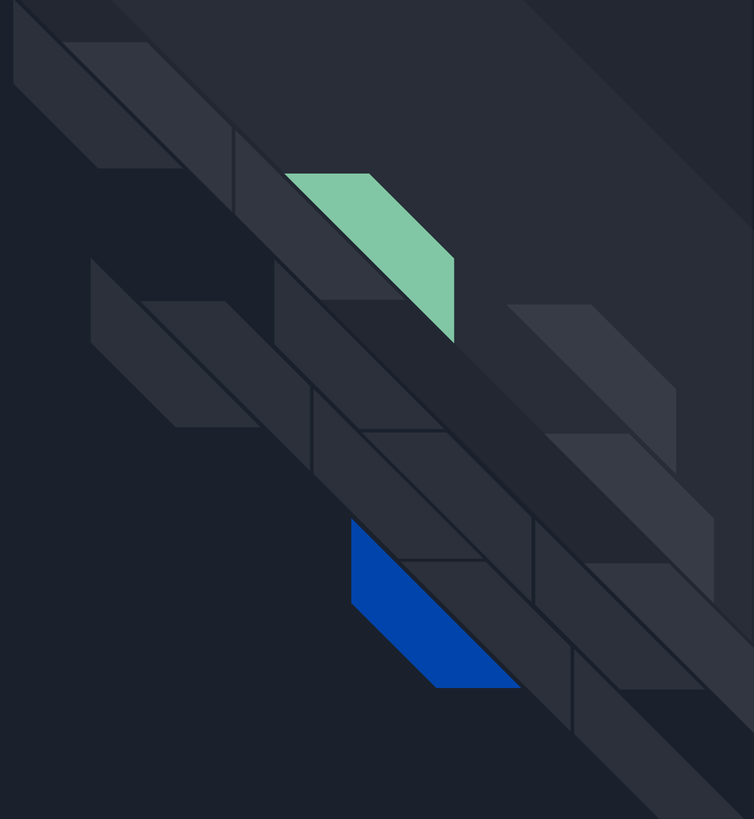
Then that class should have all three of those functions.

Corollary: If a class has a destructor, it should also have a copy constructor and assignment operator.

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

class Foo {
public:
    Foo() : val(0) { cout << "default constructor" << endl; }
    explicit Foo(int val=0) : val(val) { }
    Foo(Foo &foo) {
        cout << "copy constructor" << endl;
        val = foo.val;
    }
    ~Foo() { cout << "destructor" << endl; }
    Foo& operator = (Foo &foo) {
        cout << "assignment operator" << endl;
        val = foo.val;
        return *this;
    }

    int val;
};

int main(int argc, char const *argv[]) {
    Foo f1(1), f2(2);
    f1 = f2;
    cout << "f1 val: " << f1.val << endl;

    Foo f3(3);
    Foo f4 = f3;
    cout << "f4 val: " << f4.val << endl;
    return 0;
}
```

```
[-MacBook-Pro comp345 ]$ g++ -o assignment_operator assignment_operator.cpp
[-MacBook-Pro comp345 ]$ ./assignment_operator
assignment operator
f1 val: 2
copy constructor
f4 val: 3
destructor
destructor
destructor
destructor
```

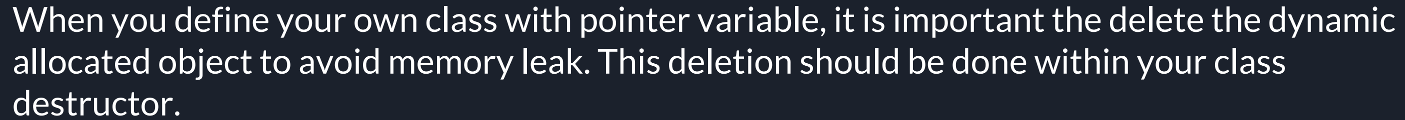
Destructor



- When an object go out of scope
- Programmer using the keyword delete explicitly

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

class Foo {
public:
    Foo() { cout << "Foo constructor" << endl;}
    ~Foo() { cout << "Foo destructor" << endl;}
};

int main() {
    Foo foo;
    return 0;
}
```

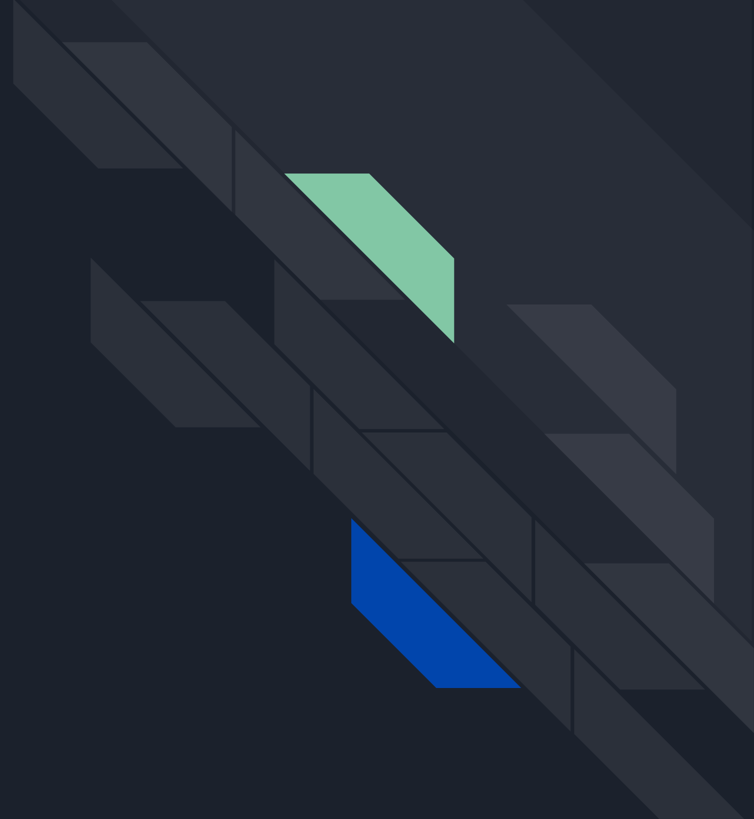
```cpp
#include <iostream>
using namespace std;

class Foo {
public:
    Foo() { cout << "Foo constructor" << endl;}
    ~Foo() { cout << "Foo destructor" << endl;}
};

int main() {
    Foo *foo = new Foo;
    delete foo;
    return 0;
}
```



When you define your own class with pointer variable, it is important the delete the dynamic allocated object to avoid memory leak. This deletion should be done within your class destructor.

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

class Bar;

class Foo {
public:
    Foo() {
        cout << "Foo constructor" << endl;
        pBar = new Bar;
    }
    ~Foo() {
        cout << "Foo destructor" << endl;
        delete pBar;
    }
private:
    Bar *pBar;
};

int main() {
    Foo foo;
    return 0;
}
```

Inheritance



Question?