Globant
we are ready

We create **innovative software products** that appeal to **global audiences**

C++ Course

Reviewed and Updated by:

- Bertoa Nicolas, 2013 / 2015
- Rodriguez Matias, 2013 / 2014
- Quesada Manuela, 2013 / 2014

Class 7 – Object Oriented Features

Access Control

Classes and Structures

Constructors and Destructors

const, static

Class

```
class Character {
    size_t mHealth;
    size_t mMana;

public:
    size_t health() const;
    size_t mana() const;
};
```

Member Variables

Member Functions

Access Control

- private (classes by default):

  A member can only be accessed by member functions and friends of that class

- protected:

  A member can only be accessed by member functions and friends of that class and derived classes

- public (structs by default):

  A member can be accessed by anyone

Constructor

- Member function that initializes a class instance
- Same name as the class itself
- If we do not implement it, the compiler will TRY to implement one for us

Constructor

```
class Character {
    size_t mHealth;
    size_t mMana;
public:
    Character(const size_t health, const size_t mana);
};
```

Initialization List

- In the body of a Constructor, each member variable class was already constructed.
- By default, each member variable will use its default constructor
- In the initialization list we can specify what arguments should be used for each Constructor of each member variable.
- Member variables are initialized in the order they were declared in the class body

Initialization List

```
Character::Character(const size_t health, const size_t mana)
    : mHealth(health)
    , mMana(mana)
{

}
```

What should get printed in the program below?

```
class Foo {
public:
    Foo() : z(x+1), y(2), x(3) {
        std::cout << "z: " << z << std::endl;
    }
private:
    int x;
    int y;
    int z;
};
int main() {
    Foo f;
}
```

What should get printed in the program below?

```
class Foo {
public:
    Foo() : z(x+1), y(2), x(3) {
        std::cout << "z: " << z << std::endl;
    }
private:
    int x;
    int y;
    int z;
};
int main() {
    Foo f;
}
```

4

What should get printed in the program below?

```
class Foo {
public:
    Foo() : z(x+1), y(2), x(3) {
        std::cout << "z: " << z << std::endl;
    }
private:
    int x;
    int y;
    int z;
};
int main() {
    Foo f;
}
```

4

According the C++ standard: non-static data members shall be initialized in the order they were declared in the class definition, regardless of the order of the mem-initializers.

Destructor

- It is called once a class/struct instance is destroyed (out of scope or manually deletion)

- Its purpose is to properly release resources, deallocate memory, etc.

- Compiler generates a default destructor if user did not implement one. It calls variable member destructors in inverse construction order.

Destructor

```
class Character {
    size_t* mHealth;
    size_t mMana;
public:
    Character::Character(const size_t health, const size_t mana)
    : mHealth(new size_t(health))
    , mMana(mana)
    {
    }
    ~Character() { delete mHealth; }
};
```

Copy Constructor

- Member function that initializes a class instance according other instance

- By default, the copy of a class object is a shallow copy of each member

Copy Constructor

```
class Character {
    size_t mHealth;
    size_t mMana;
public:
    Character(const Character& copy);
};

Character::Character(const Character& copy)
: mHealth(copy.mHealth)
, mMana(copy.mMana)
{}
```

Is copy constructor properly implemented?

```
class Foo {
    size_t*  mMember;
public:
    Foo() : mMember(new size_t) { *mMember = 9; }
    ~Foo() { delete mMember; }
    Foo(const Foo& copy);
};

Foo::Foo(const Foo& copy)
: mMember(copy.mMember)
{}
```

Is copy constructor properly implemented?

```
class Foo {
    size_t*  mMember;
public:
    Foo() : mMember(new size_t) { *mMember = 9; }
    ~Foo() { delete mMember; }
    Foo(const Foo& copy);
};

Foo::Foo(const Foo& copy)
: mMember(copy.mMember)
{}
```

No

Is copy constructor properly implemented?

```
class Foo {
    size_t*  mMember;
public:
    Foo() : mMember(new size_t) { *mMember = 9; }
    ~Foo() { delete mMember; }
    Foo(const Foo& copy);
};

Foo::Foo(const Foo& copy)
: mMember(copy.mMember)
{}
```

No

mMember allocated memory will be shared by copy constructed instance.

If one instance frees that memory, the other one will be reading/writing in an invalid memory section.

Which lines below should not compile?

```
struct A {
    A(int x) : n(x) {}
    int n;
};

int main() {
    A a1;
    A a2(2);
    A a3(a2);
}
```

Which lines below should not compile?

```
struct A {
    A(const int x) : n(x) {}
    int n;
};

int main() {
    A a1;
    A a2(2);
    A a3(a2);
}
```

A a1;

Which lines below should not compile?

```
struct A {
    A(const int x) : n(x) {}
    int n;
};

int main() {
    A a1;
    A a2(2);
    A a3(a2);
}
```

A a1;

If any user-declared constructor is present in the class, then no default constructor will be created implicitly.

What is the máximum number of implicitly defined constructs that this struct will have?

```
struct A {
    A(A& a) { }
    A(const double d) {}
    int val;
}
```

What is the máximum number of implicitly defined constructs that this struct will have?

```
struct A {
    A(A& a) { }
    A(const double d) {}
    int val;
}
```

0

What is the máximum number of implicitly defined constructs that this struct will have?

```
struct A {
    A(A& a) { }
    A(const double d) {}
    int val;
}
```

0

There will be no implicitly defined constructors for this struct

What gets printed?

```
struct A {
    A() : val(0) {}
    A(const int v) : val(v) {}
    A(A& a) : val(a.val) {}

    int val;
};
int main() {
    const A a1;
    const A a2(5);
    const A a3 = a2;
    std::cout << a1.val + a2.val + a3.val << std::endl;
}
```

What gets printed?

```
struct A {
    A() : val(0) {}
    A(const int v) : val(v) {}
    A(A& a) : val(a.val) {}

    int val;
};
int main() {
    const A a1;
    const A a2(5);
    const A a3 = a2;
    std::cout << a1.val + a2.val + a3.val << std::endl;
}
```

Ill-formed

What gets printed?

```
struct A {
    A() : val(0) {}
    A(const int v) : val(v) {}
    A(A& a) : val(a.val) {}

    int val;
};
int main() {
    const A a1;
    const A a2(5);
    const A a3 = a2;
    std::cout << a1.val + a2.val + a3.val << std::endl;
}
```

Ill-formed

The third line of main tries to initialize a3 with a2, but A's copy constructor takes a non-const reference which violates a2's const declaration

What gets printed?

```
struct A {
    A() : val() {}
    A(const int v) : val(v) {}
    A(A a) : val(a.val) {}

    int val;
};

int main() {
    A a1(5);
    A a2(a1);
    std::cout << a1.val + a2.val << std::endl;
}
```

What gets printed?

```
struct A {
    A() : val() {}
    A(const int v) : val(v) {}
    A(A a) : val(a.val) {}

    int val;
};

int main() {
    A a1(5);
    A a2(a1);
    std::cout << a1.val + a2.val << std::endl;
}
```

Ill-formed

What gets printed?

```
struct A {
    A() : val() {}
    A(const int v) : val(v) {}
    A(A a) : val(a.val) {}

    int val;
};

int main() {
    A a1(5);
    A a2(a1);
    std::cout << a1.val + a2.val << std::endl;
}
```

Ill-formed

It is illegal to have a constructor whose first and only non-default argument is a value parameter for the class type

Const Member Function

- Cannot modify members of a class unless they have mutable keyword
- A const member function can be invoked for both const and non-const objects
- A non-const member function can be invoked only for non-const objects

Const Member Function

```
class Character {
    size_t mHealth;
    size_t mMana;
public:
    size_t health() const;
    size_t mana() const;
};
```

What is the output of the program?

```
struct Foo {
    void go() {
        std::cout << "Foo" << std::endl;
    }
};
struct Bar : public Foo {
    void go() {
        std::cout << "Bar" << std::endl;
    }
};

int main() {
    Bar b;
    const Foo f = b;
    f.go();
}
```

What is the output of the program?

```
struct Foo {
    void go() {
        std::cout << "Foo" << std::endl;
    }
};
struct Bar : public Foo {
    void go() {
        std::cout << "Bar" << std::endl;
    }
};

int main() {
    Bar b;
    const Foo f = b;
    f.go();
}
```

ill-formed

What is the output of the program?

```
struct Foo {
    void go() {
        std::cout << "Foo" << std::endl;
    }
};
struct Bar : public Foo {
    void go() {
        std::cout << "Bar" << std::endl;
    }
};

int main() {
    Bar b;
    const Foo f = b;
    f.go();
}
```

ill-formed

Non-const member functions cannot be called on const objects

Static Member Variable

- Variable that is part of a class, but is not part of an instance of that class
- Can be referred to without mentioning an object. Instead, its name is qualified by the name of its class.

Static Member Variable

```
class Character {
    // Other data

public:
    static size_t sNumCreatedCharacters;
};

Character::sNumCreatedCharacters = 0;
```

Static Member Variable Definition

- Must be defined in exactly one translation unit

A.h

```
struct Character {
   static std::string name;
};
```

A.cpp

```
std::string Character::name = “Albert”
```

- *name* would be defined in each translation unit that #includes this header file
- Non-const static member variables must be defined outside class declaration

Static Member Variable Definition

- C++ allows to define integral static members within the declaration

  The expression is const integral or enumeration

  The expression can be evaluated at compile time

  There is not a definition somewhere that violates one definition rule.

Static Member Variable Definition

```
class Character {
public:
    static const int sMember = 10;
};
```

OK

Static Member Variable Definition

```
class Character {
public:
    static int sMember = 10;
};
```

Not const

Static Member Variable Definition

```
class Character {
public:
    const int sMember = 10;
};
```

Not static

Static Member Variable Definition

```
class Character {
public:
    static const int sMember = f(17);
};
```

Initializer
non const

Static Member Variable Definition

```
class Character {
public:
    static const int sMember = 7.0;
};
```

Initializer not integral

Which, if any, of the member function definitions below are ill-formed?

```
static int gX = 44;

struct Foo {
    int mX;
    static int sX;

    Foo(int x) : mX(x) {}

    int a(int x = gX)  {
        return x + 1;
    }
    int b(int x = mX) {
        return x + 1;
    }
    int c(int x = sX) {
        return x + 1;
    }
};
int Foo::sX = 22;
```

Which, if any, of the member function definitions below are ill-formed?

```
static int gX = 44;

struct Foo {
   int mX;
   static int sX;

   Foo(int x) : mX(x) {}

   int a(int x = gX)  {
      return x + 1;
   }
   int b(int x = mX) {
      return x + 1;
   }
   int c(int x = sX) {
      return x + 1;
   }
};
int Foo::sX = 22;
```

b

Which, if any, of the member function definitions below are ill-formed?

```
static int gX = 44;

struct Foo {
   int mX;
   static int sX;

   Foo(int x) : mX(x) {}

   int a(int x = gX)  {
      return x + 1;
   }
   int b(int x = mX) {
      return x + 1;
   }
   int c(int x = sX) {
      return x + 1;
   }
};
int Foo::sX = 22;
```

b

Non-static members can not be used as default arguments

Static Class/Struct method

- It can be called without class/struct instantiation
- It only can read/write static member variables

Static Class/Struct method

A.h

```
class Character {
public:
   Character() { ++mNumInstances; }
   static size_t numInstances() { return mNumInstances; }
private:
   static size_t mNumInstances;
};
```

A.cpp

```
size_t Character::mNumInstances = 0;
```

main.cpp

```
#include A.h
std::cout << Character::numInstances() << std::endl;
```

Q&A