More on C++

CS A250 – C++ Programming II

TOPICS

- A couple of topics we need to cover…
  - Static variables and functions
  - Virtual functions

Static Variables and Functions

TOPIC 1: STATIC CLASS MEMBERS

- A **static member** of a **class** can be shared by **any object** the class
  - Represents a class-wide information shared by
    - all instances (*objects*) of the class
    - <u>not</u> just a specific instance (*object*)

STATIC MEMBER VARIABLE

- A **static member variable** is a variable that contains data shared by ***all*** objects of a class
  - This is different from ***member variables***, which have **distinct** data for **each** object
- All objects of class "share" one copy
  - If one object changes it → all other objects will see the change
- Useful for "tracking"
  - How many objects exist at a given time
  - How often a member function is called

Static Variables - Example

- Suppose you have a **video game** that contains a class that creates **spaceships** (objects of the class)
  - When your class creates a spaceship, the spaceship appears on the screen
  - Every time there are more than 20 spaceships, your weapon changes from a one-directional laser to a radial laser
  - How does your game keep track of how many spaceships (objects) have been created?
    - We can use a **static variable**!

Static Variables – Example (cont.)

- If we have a **static variable** that stores the number of objects created
  - Every time we create a spaceship, we can increment the static variable.
- This requires **less memory**
  - Instead of having each object keeping track of how many objects were created
    - *Think about how redundant this implementation would be!*

Declaring a Static Variable

- How and where do you implement them?
  - **Declaration** is in the **private** section of the class definition, preceded by the keyword **static**

```
class MyClass
{
public:
      MyClass(); //default constructor
      //member functions
private:
      static int count;
      //other member variables and/or functions
};
```

Declaring a Static Variable

- How and where do you implement them?
  - **Initialization** is in the **implementation** file, ***before*** all functions

```
#include "MyClass.h"

int MyClass::count = 0;


MyClass::MyClass()   //default constructor
{
        ...
}

//other member functions
```

NO need to write the keyword **static**

MUST re-declare the variable

Scope of Static Variables

- Although they may seem like *global variables*, a class's **static variable** has **class scope**

- Can be declared as
  - **`private`**
  - **`public`**
  - **`protected`**

- Needs to be **updated** in the **destructor**
  - And in any other function where it is required.

Static Constant

- A **constant** can also be **static**

```
const static double INTEREST = 0.3;
```

- Can be **declared** *and* **initialized** in the **class interface** (.h file)

ACCESSING STATIC CLASS MEMBERS

- From **inside** the **class**
  - ***All*** **static members** can be accessed directly (just like any other member variable)

- From **outside** the **class**
  - You will need an **static accessor function**

Static Member Functions

- A **member functions** that **returns** a static variable must be **static**
  - Declaration

```
static int getCount();
```

  - Definition

```
int ClassName::getCount()
{
        return count;
}
```

STATIC MEMBER FUNCTIONS

- A **member functions** that **returns** a static variable must be **static**
  - Declaration

Need to have keyword “static”

```
static int getCount();
```

  - Definition

```
int ClassName::getCount()
{
        return count;
}
```

No keyword “static” needed

Static Member Functions

- A **member functions** that **returns** a static variable must be **static**
  - **Declaration**

Need to have keyword "static"

```
static int getCount();
```

Cannot be a const function

  - **Definition**

```
int ClassName::getCount()
{
        return count;
}
```

No keyword "static" needed

Static Member Functions

- A **static member functions** can be called ***outside*** the class in **two different ways**:
  - If **no** objects were created

```
ClassName::staticFunctionName()
```

  - If objects were created, you can use ***any*** object

```
objName.staticFunctionName()
```

Example 1

- Project: Spaceship Class

Virtual Functions

Virtual Functions Basics

- **Polymorphism**
  - Associating many meanings to one function
    - Values of different data types handled by using a uniform interface
  - Fundamental principle of **object-oriented programming**
  - Virtual functions provide this capability

- **Virtual Function**
  - Can be "used" before it is "defined"

A Simple Example (cont.)

- Assume you create a class named **MyClass**.

```
class MyClass
{
public:
        MyClass();
        MyClass(int newNum);
        int getNum() const;
        void print() const;
        //other member functions
private:
        int num;
};
```

```
void MyClass::print() const
{
    cout << num << endl;
}
```

A Simple Example

- Assume you use the class somewhere else.

```
int main
{
        MyClass obj(1);
        myFunction(obj);

        return 0;
}

void myFunction(const MyClass& obj)
{
        // do something
        obj.print();
}
```

A Simple Example

- No problem so far.

```
int main
{
        MyClass obj(1);
        myFunction(obj);

        return 0;
}

void myFunction(const MyClass& obj)
{
        // do something
        obj.print();
}
```

Output:

1

A Simple Example (cont.)

- Now assume you create the class **MyClassChild**.

```
class MyClassChild : public MyClass
{
public:
        MyClassChild();
        MyClassChild(int num, const string& newStr);
        void print() const;
        //other member functions
private:
        string str;
};
```

A Simple Example (cont.)

- Now assume you create the class **MyClassChild**.

```
class MyClassChild : public MyClass
{
public:
        MyClassChild();
        MyClassChild(int num, const string& newStr);
        void print() const;
        //other member functions
private:
        string str;
};
```

```
//redefinition of function print
void MyClassChild::print() const
{
        cout << getNum() << endl;
        cout << str << endl;
}
```

A Simple Example (cont.)

- What if we want to create an object of **MyClassChild**?

```
int main
{
        ~~MyClass obj(1);~~ MyClassChild obj(1,“one”);
        myFunction(obj);

        return 0;
}

void myFunction(const MyClass& obj)
{
        // do something
        obj.print();
}
```

A Simple Example (cont.)

- What if we want to create an object of **MyClassChild**?

```
int main
{
        MyClass obj(1); MyClassChild obj(1,"one");
        myFunction(obj);

        return 0;
}

void myFunction(const MyClass& obj)
{
        // do something
        obj.print();
}
```

Can we use **myFunction**?
What would be the output?

Function **myFunction** is passing a parameter of class **MyClass**.

A Simple Example (cont.)

- What if we want to create an object of **MyClassChild**?

```
int main
{
        MyClass obj(1); MyClassChild obj(1,"one");
        myFunction(obj);

        return 0;
}

void myFunction(const MyClass& obj)
{
        // do something
        obj.print();
}
```

Output:

1

A Simple Example (cont.)

- What if we want to create an object of **MyClassChild**?

```
int main
{
        MyClass obj(1); MyClassChild obj(1,"one");
        myFunction(obj);

        return 0;
}

void myFunction(const MyClass& obj)
{
        // do something
        obj.print();
}
```

Output:

1

The string “one” will **not** be printed, because function **print** from **MyClass** was called.

A Simple Example (cont.)

- What if we want to create an object of **MyClassChild**?

```
int main
{
        MyClass obj(1); MyClassChild obj(1,"one");
        myFunction(obj);

        return 0;
}

void myFunction(const MyClass& obj)
{
        // do something
        obj.print();
}
```

Output:

1

We would **not** be able to get the correct output from **myFunction**, because we did **not** pass an object of the class **MyClassChild**.

A SIMPLE EXAMPLE (CONT.)

- There is a way to get the correct output from function **myFunction**

  - Make the **redefined function** of the **parent** class a **virtual function**

- Even if the **parameter** is an **object** of the **parent** class, the **virtual** function will re-direct the call to the **print** function of the **child** class.

A Simple Example (cont.)

- Make the **parent** **print** function a **virtual** function.

```
class MyClass
{
public:
        MyClass();
        MyClass(int newNum);
        int getNum() const;
        virtual void print() const;
        //other member functions
private:
        int num;
};
```

A Simple Example (cont.)

- Definition does not need the keyword "**virtual**"

```
#include "MyClass.h"

...
void MyClass::print() const
{
        cout << num << endl;
}
```

The function **definition** stays the **same**.

A Simple Example (cont.)

- Now the output will be correct.

```
int main
{
        ~~MyClass obj(1);~~ MyClassChild obj(1,"one");
        func(obj);

        return 0;
}

void myFunction(const MyClass& obj)
{
        // do something
        obj.print();
}
```

Output:

1

one

OVERRIDING

- When a **virtual function** ***definition*** is changed in a **derived class**
  - We say it is been "**overridden**"
  - Similar to ***redefined***

- So:
  - Virtual functions are ***overridden***
  - **Non**-virtual functions are ***redefined***

Virtual Functions: Why Not All?

- One major ***disadvantage***: **overhead**
  - Uses ***more*** storage
  - Late binding is "on the fly", so programs run slower.

- So if virtual functions are not needed, they should not be used.

Virtual Destructors

- Recall:
  - **Destructors** are automatically executed when the class object goes out of scope.
- Now consider:
  - We pass a **child** object to the **non-member** function **print**
  - The parameter is still an object of the **parent** class
  - The parameter is **passed by value**
    - A **copy** of the object will be made
  - The **child** object goes out of scope when function ends
  - The **destructor** of the **parent** class will be called.

  - Will the **destructor** of the **child** class be also called?

Virtual Destructors (cont.)

- No.
  - The **destructor** of the **child** class will ***not*** be called.

- To correct the problem:
  - The **destructor** of the **parent** class must be **virtual**.
  - The **virtual destructor** of a **parent** class automatically makes the destructor of a **child** class be **virtual** so that it can also be called when the object is out of scope.
    - The **child** class destructor will be called first, then the **parent** class destructor will be called.

VIRTUAL DESTRUCTORS

- Any class that includes ***at least one*** **virtual member function** should define a **virtual destructor**

- If you are using inheritance,
  it is a good idea to have the **destructor** of the **base class** declared as **virtual**

MORE ON C++ (END)