G52CPP
C++ Programming
Lecture 15

Dr Jason Atkin

Last lecture

- Coursework framework

This lecture

- **`this`** and **`static`** members
- Inheritance and constructors
- Friends

The **`this`** pointer

The `this` pointer

- An object is a collection of data (its state)
- A class defines the ***structure*** of the object and what you can do with it (a design for an object)
  - e.g. Clothing, cars, programs, etc
- **For functions to actually do something to an object, they need to know which object to affect**
- (Non-static) member functions have an **implicit** extra parameter saying which object to act on
  - Parameter ***type*** is a ***pointer to object*** (of correct class)
  - And the parameter ***name*** is `this`
- Note: `this` exists in Java too, as you know
  - As an object reference to the current object

The `this` pointer

```
class DemoClass
{
public:
   int GetValue()
   {
      return m_iValue;
   }

   void SetValue(int iValue)
   {
      m_iValue = iValue;
   }

private:
   int m_iValue;
};
```

- **`GetValue()`** is effectively:

```
int GetValue(DemoClass* this)
{
   return m_iValue;
}
```

- **`SetValue(int)`** is effectively:

```
void SetValue(DemoClass* this,
      int iValue )
{
   m_iValue = iValue;
}
```

- i.e. you can refer to `m_iValue` as

  `this->m_iValue`

- Not always obvious because you can miss out the `this->`

Static methods and attributes

- **`static`** members are shared between **all** objects of that class
- **NOT** associated with a specific object
  - Same as **`static`** in Java
- Static member functions **do not** have a **`this`** pointer
- **Both static and non-static member data and functions are class members**
  - i.e. They **all** have access to **`private`** members

```
class MyClass
{
public:
  static int var;
  static void foo();
};

int MyClass::var = 25;

void MyClass::foo()
{
   var = 32;
}

int main()
{
   MyClass::var = 15;
   MyClass::foo();
}
```

Static methods/functions

- **Declaration** of static member function:
  `static void foo();`
- **Usually in .h file**

- **Definition** of static member function

```
void MyClass::foo()
{
    var = 32;
}
```

- **Usually in .cpp file**
- No ‘static’ keyword in cpp file

- Call static function
  `MyClass::foo();`

```
class MyClass
{
public:
  static int var;
  static void foo();
};

int MyClass::var = 25;

void MyClass::foo()
{
   var = 32;
}

int main()
{
   MyClass::var = 15;
   MyClass::foo();
}
```

Static data members / attributes

- Declaration of static data member:
  `static int var;`
- Usually in a header file

- Definition and initialisation of static member
  `int MyClass::var = 25;`
- Usually in .cpp file
- Done ONCE

- Use of static member
  `var = 32; // Within class`
  `MyClass::var = 15;`

```
class MyClass
{
public:
  static int var;
  static void foo();
};

int MyClass::var = 25;

void MyClass::foo()
{
   var = 32;
}

int main()
{
   MyClass::var = 15;
   MyClass::foo();
}
```

Inheritance and constructors

Construction and destruction (1)

```
struct Base
{
   Base()
   { printf("Base constructed\n"); }

   ~Base()
   { printf("Base destroyed\n"); }
};

struct Derived : public Base
{
   Derived()
   { printf("Derived constructed\n"); }

   ~Derived()
   { printf("Derived destroyed\n"); }
};

int main()
{
    Derived d;
}
```

Base class with constructor and destructor

Sub-class of Base

Create object of Derived/Sub-class

Construction and destruction (1)

```
struct Base
{
   Base()
   { printf( "Base constructed\n" ); }

   ~Base()
   { printf( "Base destroyed\n" ); }
};

struct Derived : public Base
{
   Derived()
   { printf("Derived constructed\n"); }

   ~Derived()
   { printf("Derived destroyed\n"); }
};
```

Source Code:

```
{ Derived d; }
```

Purpose:

Create object d, allow it to be destroyed as stack frame exits.

Output:

?

Construction and destruction (1)

```
struct Base
{
   Base()
   { printf( "Base constructed\n" ); }

   ~Base()
   { printf( "Base destroyed\n" ); }
};

struct Derived : public Base
{
   Derived()
   { printf("Derived constructed\n"); }

   ~Derived()
   { printf("Derived destroyed\n"); }
};
```

Source Code:

```
{ Derived d; }
```

Purpose:

Create object d, allow it to be destroyed as stack frame exits.

Output:

```
Base constructed
Derived constructed

Derived destroyed
Base destroyed
```

Construction and destruction (2)

```
struct Base
{
   Base() { printf( "Base constructed\n" ); }

   ~Base() { printf( "Base destroyed\n" ); }
};

struct Derived : public Base
{
   Derived()  { printf("Derived constructed\n"); }

   ~Derived() { printf("Derived destroyed\n"); }
};

int main()
{
        Derived* pD = new Derived;
        delete pD;
}
```

Created on the heap instead of the stack

Construction and destruction (2)

```
struct Base
{
   Base()
   { printf( "Base constructed\n" ); }

   ~Base()
   { printf( "Base destroyed\n" ); }
};

struct Derived : public Base
{
   Derived()
   { printf("Derived constructed\n"); }

   ~Derived()
   { printf("Derived destroyed\n"); }
};
```

Source Code:

```
Derived* pD =
       new Derived;
delete pD;
```

Purpose:

Create object d, then destroy it

Output:

?

Construction and destruction (2)

```
struct Base
{
   Base()
   { printf( "Base constructed\n" ); }

   ~Base()
   { printf( "Base destroyed\n" ); }
};

struct Derived : public Base
{
   Derived()
   { printf("Derived constructed\n"); }

   ~Derived()
   { printf("Derived destroyed\n"); }
};
```

Source Code:

```
Derived* pD =
       new Derived;
delete pD;
```

Purpose:

Create object d, then destroy it

Output:

```
Base constructed
Derived constructed

Derived destroyed
Base destroyed
```

Constructors and destructors

- **Construction occurs in the order:**
  - **Base class first, then derived class**
- **Destruction occurs in the order:**
  - **Derived class first, then base class**
- Effects:
  - **Derived class part of the object can always assume that base class part exists**
    - Derived class can assume that the base class has been constructed when the derived class is constructed
    - Derived class can assume that the base class has not yet been destroyed at the point the derived destructor is used
  - Derived class will NOT exist/be initialised when the base class constructor/destructor is called, so:
  - **Do not call virtual functions from the constructor or destructor**

Construction and destruction (3)

```
struct Base
{
   Base()
   { printf( "Base constructed\n" ); }

   ~Base()
   { printf( "Base destroyed\n" ); }
};

struct Derived : public Base
{
   Derived()
   { printf("Derived constructed\n"); }

   ~Derived()
   { printf("Derived destroyed\n"); }
};
```

Source Code:

```
Base* pD =
       new Derived;
delete pD;
```

Purpose:

Create object d, then destroy it through a **base class pointer**

Output:

?

Construction and destruction (3)

```
struct Base
{
   Base()
   { printf( "Base constructed\n" ); }

   ~Base()
   { printf( "Base destroyed\n" ); }
};

struct Derived : public Base
{
   Derived()
   { printf("Derived constructed\n"); }

   ~Derived()
   { printf("Derived destroyed\n"); }
};
```

Source Code:

```
Base* pD =
       new Derived;
delete pD;
```

Purpose:

Create object d, then destroy it through a **base class pointer**

Output:

```
Base constructed
Derived constructed
Base destroyed
```

NOT Derived destroyed

Construction and destruction (4)

```
struct VirtualBase
{
   VirtualBase()
   {
        printf("Base constructed\n");
   }

   virtual ~VirtualBase()
   {
        printf("Base destroyed\n");
   }
};

struct VirtualDerived : public VirtualBase
{
   …
}
```

Virtual Destructor

VirtualBase* pD = new VirtualDerived;
delete pD;

Construction and destruction (4)

```
struct VirtualBase
{
   VirtualBase()
   { printf("Base constructed\n"); }

   virtual ~VirtualBase()
   { printf("Base destroyed\n"); }
};

struct VirtualDerived
   : public VirtualBase
{
   VirtualDerived()
   {printf("Derived constructed\n");}

   ~VirtualDerived()
   { printf("Derived destroyed\n");}
};
```

Source Code:

```
VirtualBase* pD =
      new VirtualDerived;
delete pD;
```

Purpose:

Create object d, then destroy it through base class pointer.

Output:

?

Construction and destruction (4)

```
struct VirtualBase
{
   VirtualBase()
   { printf("Base constructed\n"); }

   virtual ~VirtualBase()
   { printf("Base destroyed\n"); }
};

struct VirtualDerived
   : public VirtualBase
{
   VirtualDerived()
   {printf("Derived constructed\n");}

   ~VirtualDerived()
   { printf("Derived destroyed\n");}
};
```

Source Code:

```
VirtualBase* pD =
      new VirtualDerived;
delete pD;
```

Purpose:

Create object d, then destroy it through base class pointer.

Output:

```
Base constructed
Derived constructed
Derived destroyed
Base destroyed
```

Virtual destructors

- **If destructor is NOT `virtual` then it will NOT be called if the object is destroyed through a base class pointer, reference or function**
  - **Since type of pointer/reference/function will determine the destructor to call**

- But, if you make destructor `virtual` then the objects of that class will have a (hidden) vtable pointer (or equivalent)
  - i.e. they grow

Virtual destructors: Question

- **Do we make the destructor virtual or not?**
- My advice: (only advice!!!)
  - Make it virtual if and only if there are **ANY other** virtual functions
    - No loss since vtable pointer already exists anyway
    - Probably using object through a base class pointer/reference, so object potentially COULD be destroyed that way too
  - If there are no other virtual functions
    AND you do **not** expect the object to be **`delete`**d through a pointer or reference to the base class
    THEN do **not** make your destructor **`virtual`**
    - Otherwise you add an unnecessary vtable pointer (or equivalent) to objects

Friends

`friend`s

- Classes can grant access to their `private` member data and functions to their `friends`
- The class still maintains control over which classes and functions have access
- The `friends` of a class are treated as class members **for access purposes** – although they are **not** members
- **Declare** your friends within your class body and use the keyword `friend`

friend function

```
class Friendly
{
// Make function a friend
friend void FriendFunc( const char* msg, const Friendly& ob );

public:
   Friendly(int i=4) : _i(i)
   {}

private:
   int _i;
};
```

```
int main()
{
   Friendly d1(2), d2;
   FriendFunc( "d1", d1 );
   FriendFunc( "d2", d2 );
}
```

```
void FriendFunc( const char* msg, const Friendly& ob )
{
   printf( "%s : _i = %d\n", msg, ob._i );
}
```

friend class

.h file:

```
class Friendly;

class TheFriend
{
public:
   void DoSomething(
        Friendly& dest,
        const Friendly& source);
};
```

Forward declaration of class

```
class Friendly
{
friend class TheFriend;
public:
   Friendly(int i=4) : _i(i){}
private:
   int _i;
};
```

.cpp file:

```
void TheFriend::DoSomething(
   Friendly& dest,
   const Friendly& source )
{
   dest._i = source._i + 1;
}
```

Note: Could make this a static member function since it does not need to access or alter any member data

```
int main()
{
   Friendly d1(2), d2;
   TheFriend f;
   f.DoSomething( d2, d1);
}
```

Next Lecture

- Function pointers
- Virtual and non-virtual functions
  - v-tables