Invisible Functions I continued
Simple Class Example

CS 311 Data Structures and Algorithms

Lecture Slides

Friday, August 28, 2020

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

`ggchappell@alaska.edu`

Some material contributed by Chris Hartman

Unit Overview
Advanced C++ & Software Engineering Concepts

Major Topics: Advanced C++

- ✓ Expressions
- ✓ Parameter passing I
- ✓ Operator overloading
- ✓ Parameter passing II
- (part) Invisible functions I
- Integer types
- Managing resources in a class
- Containers & iterators
- Invisible functions II
- Error handling
- Using exceptions
- A little about Linked Lists

Major Topics: S.E. Concepts

- Invariants
- Testing
- Abstraction

Review

Review
Operator Overloading

Operators can be implemented using global or member functions.

- Global: the parameters are the operands.
- Member: first operand is `*this`, the rest are parameters.

Function name is “`operator`” followed by the operator’s symbol.

Postfix increment & decrement (`n++`, `n--`) get a dummy `int` parameter, to distinguish from the prefix versions (`++n`, `--n`).

Use global functions for overloaded arithmetic, comparison, and bitwise operators that do not modify their first operand …

- `+` `-` binary `*` `/` `%` `==` `!=` `<` `>` `<=` `>=` `&` `|` `^`

… or if the first operand is a type you cannot add members to.

- Common examples: stream insertion <<, stream extraction >>.

Use member functions for other overloaded operators.

- `=` `[]` unary `*` `+=` `-=` `*=` `/=` `++` `--` etc.

Review
Parameter Passing I/II [1/4]

C++ provides four ways to pass a parameter or return value.

By value:

```
void p1(Foo x);          // Pass x by value
Foo r1();                // Return by value
```

By reference:

```
void p2(Foo & x);        // Pass x by reference
Foo & r2();              // Return by reference
```

By reference-to-const (some people say "const reference"):

```
void p3(const Foo & x); // Pass x by reference-to-const
const Foo & r3();        // Return by reference-to-const
```

By Rvalue reference—introduced in the 2011 C++ Standard:

```
void p4(Foo && x);       // Pass x by Rvalue reference
Foo && r4();             // Return by Rvalue reference
```

Review
Parameter Passing I/II [2/4]

	By Value	**By Reference**	**By Reference-to-Const**	**By Rvalue Reference**
Makes a copy	YES ☹	NO ☺	NO ☺	NO
Allows for polymorphism	NO ☹	YES ☺	YES ☺	YES
Allows implicit type conversions	YES ☺	NO ☹	YES ☺	YES
Allows passing of:	Any copyable value ☺	Non-const Lvalues ☹?	Any value* ☺	Non-const Rvalues*

For many purposes, *when we pass objects*, reference-to-const combines the best features of the first two methods.

*Rvalues *prefer* to be passed by Rvalue reference.

Review
Parameter Passing I/II [3/4]

For most parameter passing, we pass either by value or by reference-to-const.

- By value: simple types (`int`, `char`, etc.), pointers, iterators.
- By reference-to-const: larger objects, or things we are not sure of.

We normally return by value.

And then there are special cases …

We pass by reference, if we want to send the value of the parameter back to the caller.

We *might* return by reference or by reference-to-const, if we are returning a value that is not going away.

- The former if the caller gets to modify the value; the latter if not.

Review
Parameter Passing I/II [4/4]

We do not pass by Rvalue reference very often.

When we do so, we typically write two versions of a function.

```
void g(const Foo & p);   // Gets Lvalues, cannot modify
void g(Foo && p);        // Gets Rvalues, CAN modify
```

Since it is okay to “mess up” an Rvalue, the second version can often be written to be faster. (But if it cannot, then there is no point in writing it at all.)

Review
Invisible Functions I

A C++ compiler may automatically write a number of member functions. Here are six important ones:

```
Dog();                               // Default ctor
~Dog();                              // Dctor
Dog(const Dog & other);              // Copy ctor
Dog & operator=(const Dog & rhs);    // Copy assignment
Dog(Dog && other);                   // Move ctor*
Dog & operator=(Dog && rhs);         // Move assignment*
```

The **Big Five**

***Move** operations were added in C++11.

Invisible Functions I

continued

Invisible Functions I
The Six Functions — Overview

Now we look at each of the six functions (default ctor, dctor, copy & move ctors, copy & move assignment operators). For each function, we discuss:

- What it is.
- When it is called.

Invisible Functions I
The Six Functions — Default Ctor [1/2]

The **default constructor** is a ctor with no parameters.

```
class Dog {
public:
    Dog():_a(7), _b()
    {}
```

Member initializers specify how data members are constructed.

- These are called in the order data members are *declared*.
- A data member with no member initializer is default constructed.

```
Dog():_a(7) // _a is set ONCE, by its constructor
{}

Dog()       // _a is set TWICE: default ctor, assignment
{ _a = 7; }
```

Invisible Functions I
The Six Functions — Default Ctor [2/2]

The default ctor is called …

- When you declare an object with no ctor arguments:

```
Dog mutt;
```

- For each item in a (built-in) array:

```
Dog puppies[7];            // Default ctor called 7 times
Dog * dp = new Dog[8];     // Default ctor called 8 times
```

- When you call it explicitly.

```
myFunc(Dog());
```

Invisible Functions I
The Six Functions — Dctor [1/2]

The **destructor** is the function called when an object is destroyed.

The dctor is called …

- For an automatic object, when the object goes out of scope:

Automatic: ordinary local variable.

Static: global variable, static local variable, or static data member.

```
void func()
{
    Dog x;
}  // x.~Dog() is called when leaving
```

- For a static object, when the program ends.
- For a non-static member object, when the object it is a member of is destroyed.

Continued …

Invisible Functions I
The Six Functions — Dctor [2/2]

The dctor is called …

- For an object created with `new` (a **dynamic** object), when you `delete` a pointer to it:

```
Dog * p = new Dog;
delete p;           // Dctor called for *p
Dog * array = new Dog[42];
delete [] array;    // Dctor called 42 times
```

- When you call it explicitly (which does not happen much):

```
Dog * q = new Dog;
q->~Dog();  // Destroy *q without deallocating the
            //  memory that holds it.
            //  (This is a strange thing to do.)
```

Invisible Functions I
The Six Functions — Copy & Move Ctors [1/3]

The **copy ctor** and the **move ctor** are both ctors that take a value of the same type as that being constructed.

- In the **copy ctor** the parameter is passed by reference-to-const.

```
Dog(const Dog & other);  // Copy ctor
```

- In the **move ctor** the parameter is passed by Rvalue reference.

```
Dog(Dog && other);       // Move ctor
```

Recall how this works:

- If the move ctor exists, then it gets Rvalue arguments, while the copy ctor (if it exists) gets Lvalue arguments.
- If only the copy ctor exists, then it gets both Lvalues and Rvalues.

Invisible Functions I
The Six Functions — Copy & Move Ctors [2/3]

The copy or move ctor is called …

- When you pass an object by value—*maybe*.

"Maybe": a compiler is sometimes allowed to avoid copying. This is **copy elision**.

```
void myFunc2(Dog x);   // Parameter x is by-value
myFunc2(mutt);         // Copy/move ctor creates copy
                       //  of mutt
```

- When you return by value—*maybe*.

```
Dog myFunc3()
{ return Dog(); }      // Copy/move ctor is called here
```

Continued …

Invisible Functions I
The Six Functions — Copy & Move Ctors [3/3]

The copy or move ctor is called …

- When you declare an object with one ctor argument of the same type:

```
Dog mutt(purebred);
Dog mutt = purebred;  // Same as previous line
```

- When you call it explicitly.

```
myFunc(Dog(mutt));    // Make copy of mutt & pass to
                      //  myFunc
```

Invisible Functions I
The Six Functions — Copy & Move Assignment Operators

The **copy assignment** and **move assignment** operators are assignment operators (=) where both sides have the same type.

- **Copy assignment**: parameter passed by reference-to-const.

```
Dog & operator=(const Dog & other);  // Copy assignment
```

- **Move assignment**: parameter passed by Rvalue reference.

```
Dog & operator=(Dog && other);       // Move assignment
```

Copy/move assignment is called only when you call it explicitly:

```
mutt = purebred;
```

Invisible Functions I
Write or Not? [1/4]

Each of the six special member functions can be written for you by the compiler—**automatically generated**.

In all cases, the automatically generated version is public; it will call the corresponding member function on each data member.

For example, here are the automatically generated copy assignment operator and default ctor and for class `Dog`.

Code on this slide is not code for you to write. Rather, it shows what the compiler may write for you.

```
public:
    Dog & operator=(const Dog & rhs)
    {
        _a = rhs._a;
        _b = rhs._b;
        return *this;
    }
```

```
public:
    Dog():_a(), _b()
    {}
```

Invisible Functions I
Write or Not? [2/4]

For each of the six members, there are two special options (C++11 & later).

1. Go with the automatically generated version—even if it would not normally be written for you. Use "`= default;`".

```
Dog(Dog && other) = default;
```

`=default` and `=delete` make code easier to understand. Use them!

2. Eliminate the function. Use "`= delete;`".

```
Dog(Dog && other) = delete;
```

Invisible Functions I
Write or Not? [3/4]

So there are 4 ways we can handle each of the 6 special functions:

1. Define it explicitly.
2. =delete it.
3. =default it.
4. Do nothing.

```
Foo(const Foo & other)
        :_a(…), _b(…)
{ … }
```

```
Foo(const Foo & other) = delete;
```

```
Foo(const Foo & other) = default;
```

If we do nothing, then the function *might* be automatically generated. The C++ Standard specifies exactly when this happens, and shortly we will look at what it says. But there is an easier way: follow the **Rule of Five**. *See the next slide.*

Invisible Functions I
Write or Not? [4/4]

The default ctor is automatically generated when we declare no ctors.

For the other five—the Big Five—here is the rule we follow.

The **Rule of Five**: If you define one of the Big Five, then define or =delete all of them.

- We generally define/=delete these when an object directly manages some resource—like dynamically allocated memory—that needs clean-up.
- Otherwise, we usually =default them all. (Doing nothing for all five accomplishes the same thing.)

Recall: the **Big Five** are the dctor, copy & move ctors, and copy & move assignment operators.

Remember, **automatic generation is good**. We do not *want* to write these functions. But we sometimes *need* to do so.

Invisible Functions I
Details [1/2]

For completeness, here is when each of the six functions is automatically generated.

Each is only automatically generated if the corresponding function exists and is public for each data member. Given that, here is when each is written for you:

- Default ctor. When you declare no ctors.
- Dctor. When you do not declare it.
- Copy ctor, copy assignment. When you do not declare it and also declare neither *move* operation.
  - Before C++14: when you do not declare it.
- Move ctor, move assignment. When you do not declare any of the Big Five.

Know this one.
I will not require you
to know the rest.

Invisible Functions I
Details [2/2]

However, if we follow the Rule of Five, then we do not need to worry about the details of most of the above conditions. Either:

- For each of the Big Five, we do nothing or =default it (in this class, we require the latter), and they are all written for us, OR:
- For each of the Big Five, we explicitly define or =delete it, and none of them are written for us.

Again, we generally do the latter only when the class manages some resource that needs clean-up.

Simple Class Example

Simple Class Example Introduction

Next we write a simple C++ class that will involve many of the ideas we have covered. This class is similar to what you will write in Project 1.

Simple Class Example CODE

TO DO

- Write a C++ package consisting primarily of a class whose objects store and handle a **time of day**, in hours, minutes, and seconds.
  - Name the class `TimeOfDay`.
  - Give it reasonable ctors, etc.
    - Default ctor, ctor from hours/minutes/seconds, Big Five.
    - Can we use automatically generated functions?
      - We can =default the Big Five.
      - We will need to write our own default ctor and 3-parameter ctor.
  - Give it reasonable operators.
    - Pre & post ++, –– to make the time go forward & back by 1 second.
    - Stream insertion (<<) to print the time like "` 3:21:05`", 24-hr time.
    - Assignment (=) is included in the Big Five, above.
    - It might be nice to add more. We will not, due to time constraints.
  - Give it other reasonable member functions.
    - `getTime`: get hours/minutes/seconds from an object.
    - `setTime`: set an object's time, giving hours/minutes/seconds.
    - Again, time constraints prevent us from adding more.

Partially done. See `timeofday.h`*/*`.cpp`*.*
For a program that uses class `TimeOfDay`*, see* `timeofday_main.cpp`*.*

Simple Class Example
TO BE CONTINUED …

Simple Class Example will be continued next time.