Invisible Functions I continued
Simple Class Example
Unit Overview
Advanced C++ & Software Engineering Concepts

Major Topics: Advanced C++
- Expressions
- Parameter passing I
- Operator overloading
- Parameter passing II
- 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.
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
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.*
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.
We do not pass by Rvalue reference very often.

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

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

```cpp
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*
```

*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() // _a is set ONCE, by its constructor
    {}

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

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

**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.
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 — Destructor [1/2]

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

The destructor is called ...

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

```cpp
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:

```cpp
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):

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

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

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

```cpp
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.
The copy or move ctor is called ...

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

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

- When you return by value—*maybe*.

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

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

Continued ...
The copy or move ctor is called ...

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

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

- When you call it explicitly.

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

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

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

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

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

```cpp
mutt = purebred;
```
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.

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

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

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

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

2. Eliminate the function. Use “= delete;”.

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

2. =delete it.
   
   ```cpp
   Foo(const Foo & other) = delete;
   ```

3. =default it.
   
   ```cpp
   Foo(const Foo & other) = default;
   ```

4. Do nothing.

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.**
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.)

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

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