# Silently Written & Called Functions

CS 311 Data Structures and Algorithms
Lecture Slides
Friday, January 25, 2013

Chris Hartman
Department of Computer Science
University of Alaska Fairbanks
**cmhartman@alaska.edu**

Based on material by Glenn G. Chappell

# Unit Overview
# Advanced C++ & Software Engineering Concepts

## Major Topics: Advanced C++

- ✓ The structure of a package
- ✓ Parameter passing
- ✓ Operator overloading
- Silently written & called functions
- Pointers & dynamic allocation
- Managing resources in a class
- Templates
- Containers & iterators
- Error handling
- Introduction to exceptions
- Introduction to Linked Lists

## Major Topics: S.E. Concepts

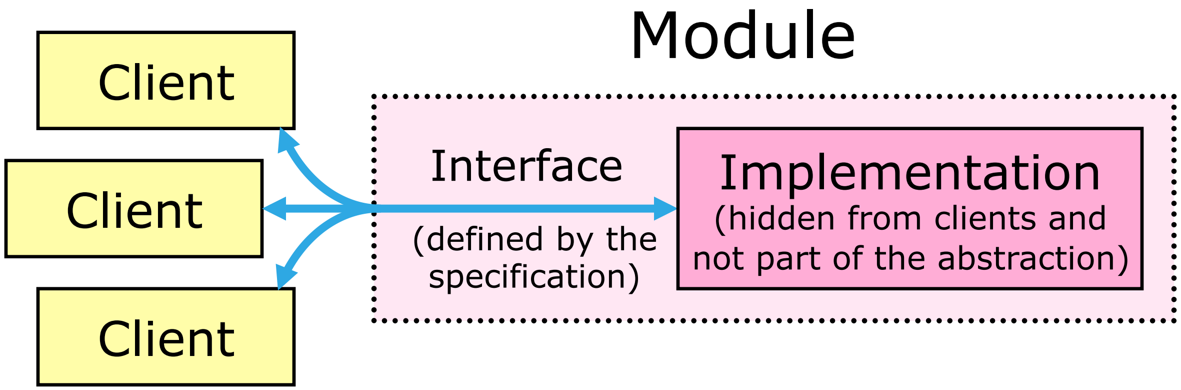
- ✓ Abstraction
- Invariants
- Testing
- Some principles

# Review
# Software Engineering Concepts: Abstraction [1/2]

**Abstraction**: Separate the purpose of a module from its implementation.

- **Functional abstraction**
- **Data abstraction**

Recall: Function, class, or other unit of code. Generally smaller than a *package*.


Module
Client
Client
Client
Interface
(defined by the specification)
Implementation
(hidden from clients and not part of the abstraction)

Key term: **Abstract Data Type**

- An *abstract data type* (ADT) is a collection of data and a set of operations on the data.
- The implementation is not specified.
- ADTs will be a major topic of this course.

# Review
# Software Engineering Concepts: Abstraction [2/2]

```
void printIntArray(const int arr[], std::size_t size)
{
    for (std::size_t i = 0; i < size; ++i)
        std::cout << arr[i] << " ";
    std::cout << std::endl;
}
```

Describe this function, in detail.

Function `printIntArray` is given an array of `int`s called “`arr`” and a `size_t` called “`size`”. It executes a `for` loop in which local `size_t` variable `i` is initialized to `0`, the loop continues as long as “`i < size`” evaluates to `true`, and `i` is pre-incremented after each loop iteration. Inside the loop, a reference to an item in array `arr` is retrieved using the bracket operator, with parameter `i`, and then inserted in `cout` (using overloaded `operator<<`), followed by an array of `char`s containing a blank and a null. After the loop, stream manipulator `endl` is inserted in `cout`. The function then terminates.

(Functional) abstraction

Function `printIntArray` prints an array of `int`s to `cout`, given the array and its size. Items are separated by blanks, and followed by a blank and a newline.

# Review
# Parameter Passing [1/2]

| | By value | By reference | By reference-to-const |
|---|---|---|---|
| Makes a copy | YES ☹* | NO ☺ | NO ☺ |
| Allows for polymorphism | NO ☹* | YES ☺ | YES ☺ |
| Allows passing of const values | YES ☺ | NO ☹** | YES ☺ |
| Allows implicit type conversions | YES ☺ | NO ☹ | YES ☺ |

*These are problems when we pass **objects**.

***Maybe* this is bad. When we want to send changes back to the client (which is a big reason for passing by reference), disallowing const values is a good thing.

So, for most purposes, *when we pass objects*, reference-to-const combines the best features of the other two methods.

# Review
# Parameter Passing [2/2]

We **pass parameters** by reference when we want to modify the client’ s copy.

```
void addThree(int & theInt)
{ theInt += 3; }
```

Otherwise, we generally pass:

- simple types by value.
- objects by reference-to-const.

```
void func(double d, const MyClass & q);
```

We usually **return** by value, unless we return an object not local to this function.

- Return by reference if we return a pre-existing object for the client to modify.
- Return by reference-to-const if we return a pre-existing object that the client should not modify (in particular, if the object is const).

```
int & arrayLookUp(int theArray[], int index);
const int & arrayLookUp(const int theArray[], int index);
```

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

Implement an operator using a member function, unless you have a good reason not to.

- Good Reason #1: To allow for implicit type conversions on the first argument. Applies to: non-modifying arithmetic, comparison, and bitwise operators.
  - For example: **`+ - * / % == != < <= > >=`**
- Good Reason #2: When you cannot make it a member, because it would have to be a member of a class you cannot modify.
  - Quintessential examples: stream insertion (**<<**) and extraction (**>>**).

We usually use operators only for operations that happen **quickly**.

- One exception: Assignment for container types.

# Silently Written & Called Functions Introduction [1/2]

Here is a simple class **`Dog`**:

```
// class Dog
// What member functions does this have?
// Invariants: None.
class Dog {

// ***** Dog: Data members *****
private:
    int a;
    double b;
    Cat c;
};  // End class Dog
```

How many member functions does class **`Dog`** have?

- Answer:

# Silently Written & Called Functions
# Introduction [1/2]

Here is a simple class **`Dog`**:

```
// class Dog
// What member functions does this have?
// Invariants: None.
class Dog {

// ***** Dog: Data members *****
private:
    int a;
    double b;
    Cat c;
};  // End class Dog
```

How many member functions does class **`Dog`** have?

- Answer: 6. *See the next slide …*

# Silently Written & Called Functions
# Introduction [2/2]

- Class **`Dog`** has 6 silently written member functions (prototypes below).
  - “Ctor” means constructor, and “dctor” means destructor.

```
class Dog {
public:
    Dog();                            // 1. Default ctor
    Dog(const Dog & other);           // 2. Copy ctor
    Dog & operator=(const Dog & rhs); // 3. Copy assignment
    ~Dog();                           // 4. Dctor
    const Dog * operator&() const;    // 5. Address-of (const)
    Dog * operator&();                // 6. Address-of
```

You **can** redefine the address-of operators, but don’t.

- The silently written versions do “**`return this;`**”. Anything else is confusing.

You may need to write the other four. Next we look closer at these.

# Silently Written & Called Functions
# Default Ctor [1/2]

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

- The silently written version calls the default ctor for all data members, as shown below.

```
class Dog {
public:
    Dog():a(), b(), c()
    {}
```

Initializer List

Empty function body (common)

Note: Every ctor has an **initializer list**.

- *Before* the function body, all data members are constructed. (Why?)
- Initializers give parameters for these ctors. They are called in the order *declared*.
- If a data member is left out of the initializer list, then it is default constructed.
- Using initializers properly leads to efficient code.

```
Dog():a(3) // a is modified once (constructor).
{}

Dog()      // a is modified twice (default constructor, assignment).
{ a = 3; }
```

# Silently Written & Called Functions
# Default Ctor [2/2]

The default ctor is **silently written** when you declare **no** ctors.

The default ctor is **called** …

- When you call it explicitly:

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

- When you declare an object with no ctor parameters:

```
Dog mutt;
```

- **Not** when you try to explicitly call it like this (why?):

```
Dog mutt(); // What does this declare?
```

- For each item in an array, when you declare the array:

```
Dog puppies[27];  // Default ctor called 27 times
```

# Silently Written & Called Functions
# Copy Ctor [1/2]

A **copy constructor** is a constructor that takes an object of the same type as that being constructed.

- The parameter should be passed by reference-to-const.
- The silently written version calls the copy ctor for all data members, as shown below.

```
class Dog {
public:
    Dog(const Dog & other)
        :a(other.a), b(other.b), c(other.c)
    {}
```

Note the initializer list and empty function body, as before.

# Silently Written & Called Functions
# Copy Ctor [2/2]

The copy ctor is **silently written** when you do not declare it.
The copy ctor is **called** …

- When you call it explicitly.

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

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

```
Dog mutt(purebred);
Dog mutt = purebred;  // Same as above
```

- When you pass an object by value:

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

- And *maybe* when we return by value (the call can be optimized away), see Return Value Optimization
  - Conclusion: your copy ctor had better to do a real copy (right?).

```
Dog myFunc3()
{ return Dog(); }     // MAYBE copy ctor is called here.
```

# Silently Written & Called Functions
# Copy Assignment

**Copy assignment** is assignment (“=”) in which both sides have the same type.

- The parameter should be passed by reference to const.
- The return value should be a reference to the object assigned to.
- The silently written version does copy assignment for all data members.

```
class Dog {
public:
    Dog & operator=(const Dog & rhs)   // Not a ctor; no initializers
    {
        a = rhs.a;
        b = rhs.b;
        c = rhs.c;

        return *this;
    }
```

Copy assignment is **silently written** when you do not declare it.
Copy assignment is **called** only when you call it explicitly:

```
mutt = purebred;
```

# Silently Written & Called Functions
# Dctor [1/2]

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

- The silently written version does nothing, except that dctors for all data members are automatically called.

```
class Dog {
public:
    ~Dog()
    {}   // Dctors for data members are called
         //  after the function body has executed.
```

# Silently Written & Called Functions
# Dctor [2/2]

The dctor is **silently written** when you do not declare it.
The dctor is **called** …

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

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

- For a static object, when the program ends.
- For a member object, when the object it is a member of is destroyed.
- For an object allocated with **`new`**, when you **`delete`** a pointer to it:

```
Dog * p = new Dog;
Dog * array = new Dog[27];
delete p;          // Dctor called for *p
delete [] array;   // Dctor called 27 times
```

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

```
Dog * q = new Dog;
q->~Dog();  // Destroy *q without deallocating memory.
```

# Silently Written & Called Functions
Summary

Silent Writing

- The default ctor is silently written when you declare no ctors.
- Each of the other three (copy ctor, copy assignment, dctor) is silently written when you do not declare it.
- For all four, the silently written versions are public; they call the corresponding functions for all data members.

Silent Calling

- The default ctor is called when you declare an object with no ctor parameters, and when you declare an array.
  - In general, to be able to put a type in a container, that type must be default constructable.
- The copy ctor is called when you pass by value, and *maybe* when you return by value.
- The dctor is called:
  - For an automatic object, when it goes out of scope.
  - For a static object, when the program ends.
  - For a member object, when the object it is a member of is destroyed.
  - For an object allocated with **`new`**, when you **`delete`** a pointer to it.

# Silently Written & Called Functions Example

TO DO

- Look at some code that does odd, unexpected things using silently written & silently called functions.

# Silently Written & Called Functions continued
# Commenting Them

Silently written functions are **good**.

- Do not waste effort. If the compiler will write a perfectly good function for you, then do not write it yourself.

So, use them often. And when you do, indicate this in a comment.

- This is a reminder that these functions exist and are part of the class design.

```
class Aardvark {
public:
    // Default ctor
    // Pre: None.
    // Post: None.
    Aardvark();

    // Compiler-generated copy ctor, copy assn, dctor are used.
```

# Silently Written & Called Functions
# When to Write Them?

When should you write these functions yourself?

- When you need them, but they are not written for you.
- When the silently written ones do not do what you want.

```
class Llama {
    …
private:
    int * p;
```

Should the copy ctor just copy `p` (**shallow** copy) or should it also copy the memory that `p` points to (**deep** copy)?

- The answer depends on what `p` is for.
- The silently written copy ctor does a shallow copy.

**The Law of the Big Three**

- **If you need to define one of the Big Three** (copy ctor, copy assignment, dctor), **then you probably need to define all of them.**
- This tends to happen when the class manages a resource (for example, dynamically allocated memory, an open file, etc.). More on this soon.

# Silently Written & Called Functions
# Eliminating Them [1/2]

We have covered:

- What the compiler writes for you.
- How & when to replace these with your own versions.

But sometimes we want to **eliminate** these functions.

Why would we want this?

- Most common reason: making objects uncopyable.
- This allows us to put strong controls on the creation and destruction of such objects.
- It also disallows passing by value.

So, how do we eliminate the copy ctor and copy assignment?

- If we do not write them, then the compiler will, right?
- If we do write them, then they exist, right?

# Silently Written & Called Functions
# Eliminating Them [1/2]

We have covered:

- What the compiler writes for you.
- How & when to replace these with your own versions.

But sometimes we want to **eliminate** these functions.

Why would we want this?

- Most common reason: making objects uncopyable.
- This allows us to put strong controls on the creation and destruction of such objects.
- It also disallows passing by value.

So, how do we eliminate the copy ctor and copy assignment?

- If we do not ~~write~~ them, then the compiler will, right?
- If we do ~~write~~ them, then they exist, right?

  define declare

- Thus: declare them, but do not define them.
- But what if someone else defines them …

# Silently Written & Called Functions
# Eliminating Them [2/2]

How do we eliminate the copy ctor and copy assignment?

- **Declare** the copy ctor and copy assignment **`private`**.
- Do not **define** them.

```
class Mule {
private:
    // Uncopyable class.
    // Private copy ctor, copy assn. Do not define these.
    Mule(const Mule &);
    Mule & operator=(const Mule &);
```

Now **no one** can call these functions.

- You (the class author) cannot accidentally call them, because you did not define them.
- Client code *can* define them, but that does not matter; they cannot call them, because they are private.

# Silently Written & Called Functions
# Eliminating Them: C++11

```
class Mule {
public:
    // Uncopyable class.
    Mule(const Mule &) = delete;
    Mule & operator=(const Mule &) = delete;
    Mule() = default;
}
```