Software Engineering Concepts: Abstraction Parameter Passing Operator Overloading Silently Written & Called Functions

CS 311 Data Structures and Algorithms Lecture Slides Wednesday, September 9, 2009

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
CHAPPELLG@member.ams.org

© 2005–2009 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

(part)

Major Topics: S.E. Concepts
- Abstraction
- Invariants
- Testing
- Some principles
A **client** of a module is *code* that uses it.

Functions & classes have:
- **Declarations** (possibly many)
- A **definition** (just one)

**Type conversion**: take value and return value of another type.
- **Implicit**: `double d = 4.5 + 3;`
- **Explicit**: `double d = 4.5 + double(3);`
- No conversion: `double d = 4.5 + 3.0;`

**Conventions** for C++ packages:
- **Header** File (.h)
  - Intended to be included by other files.
  - Has `#ifndef` to avoid multiple inclusion.
  - Contains class definition(s).
- **Source** File (.cpp or other suffix)
  - Intended to be separately compiled.
  - Includes header.
  - Contains most member function definitions.
Abstraction: Separate the purpose of a module from its implementation.

- Functional abstraction
- Data 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.

Recall: Function, class, or other unit of code. Generally smaller than a package.
Software Engineering Concepts: Abstraction Example

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;
}

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 `chars` containing a blank and a null. After the loop, stream manipulator `endl` is inserted in `cout`. The function then terminates.

Describe this function, in detail.

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

Parameter Passing

C++ provides three ways to pass a parameter or return value:

- **By value.**
  
  ```cpp
  void byv1(Foo x); // Pass x by value
  ```
  ```cpp
  Foo byv2(); // Return by value
  ```

- **By reference.**
  
  ```cpp
  void byr1(Foo & x); // Pass x by reference
  ```
  ```cpp
  Foo & byr2(); // Return by reference
  ```

- **By reference-to-const.**
  
  ```cpp
  void byrc1(const Foo & x); // Pass x by reference-to-const
  ```
  ```cpp
  const Foo & byrc2(); // Return by reference-to-const
  ```
Parameter Passing [continued]  
Three Ways — By Reference

void byr1(Foo & x);  
Foo & byr2();

When passing by reference, no copy is made.
  - Instead, the original and the passed version are the **same object**.
  - Below, \( x \) (in `byr1`) is the same object as \( y \) (outside `byr1`). Modifying \( x \) will modify \( y \).

\[
\text{Foo } y;  
\text{byr1}(y);  
\]

Passing by reference allows for proper calling of virtual functions. Only non-const values can be passed by reference. Be careful when returning by reference.
  - Do not return a value that will be destroyed when the function returns.
  - In particular, never return a non-static local variable by reference.

```cpp
int & squareThis(int n)  
{ int square = n * n; return square; }  
BAD! 😞
```
Parameter Passing
Three Ways — By Reference-to-Const

```c
void byrc1(const Foo & x);
const Foo & byrc2();
```

When passing by reference-to-const, no copy is made.

- Instead, the original and the passed version are the same object ...
- ... **unless** they are of different types; implicit type conversions *may* be applied.

```c
void h(const double & x);
const double dd;
const int ii;
```

- `h(dd);` // x is dd
- `h(ii);` // Legal, but x is not ii

Passing by reference-to-const allows for proper calling of virtual functions. When passing this way, the passed version cannot be modified.
- Thus, const variables may be passed.

As before, be careful when returning by reference-to-const.
Parameter Passing  
Three Ways — Summary

<table>
<thead>
<tr>
<th></th>
<th>By value</th>
<th>By reference</th>
<th>By reference-to-const</th>
</tr>
</thead>
<tbody>
<tr>
<td>Makes a copy</td>
<td>YES ☹*</td>
<td>NO ☹</td>
<td>NO ☹</td>
</tr>
<tr>
<td>Allows for polymorphism</td>
<td>NO ☹*</td>
<td>YES ☺</td>
<td>YES ☺</td>
</tr>
<tr>
<td>Allows passing of const values</td>
<td>YES ☺</td>
<td>NO ☹**</td>
<td>YES ☺</td>
</tr>
<tr>
<td>Allows implicit type conversions</td>
<td>YES ☺</td>
<td>NO ☹</td>
<td>YES ☺</td>
</tr>
</tbody>
</table>

*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.
Parameter Passing
Rules of Thumb

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

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

Otherwise, we generally pass:

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

```c
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).

```c
int & arrayLookUp(int theArray[], int index);
const int & arrayLookUp(const int theArray[], int index);
```
C++ allows **overloading** of most of the standard operators.

- Define standard operators for new types.
- No new operators, no changes in **precedence**, **associativity**, or number of **operands**.
- An operator’s name, as a function, is “operator” plus its symbol, e.g., “operator-”.

Subtraction for a class **MyNum** (new numerical type) as a **global** function:

```cpp
MyNum operator-(const MyNum & a, const MyNum & b);
```

It could also be a **member** function; the first operand is the object (**this**):

```cpp
class MyNum {
public:
    MyNum operator-(const MyNum & b) const; // first operand is *this
};
```

```cpp
MyNum MyNum::operator-(const MyNum & b) const
{
    // Continue as usual
}
```

Which is better?
Operator Overloading
Global & Member [2/2]

Suppose there is an implicit conversion from \texttt{double} to \texttt{MyNum}.

- If we write \texttt{MyNum - MyNum} as a global, then we get, for free,
  - \texttt{MyNum - double}
  - \texttt{double - MyNum}

- But if it is a member, then we only get the first one above.

Rule: Implement an overloaded operator using a \texttt{member} function unless you have a good reason not to.

- Good Reason #1: To allow for implicit type conversions on the first operand, in a non-modifying arithmetic, comparison, or bitwise operator.

- Thus, the following should generally be implemented using global functions:
  - Arithmetic: $+$, $-$, $*$, $/$, $\%$
  - Comparison: $==$, $!=$, $<$, $<=$, $>$, $>=$
  - Bitwise: $\&$, $\mid$, $^$, $\sim$

But not $+=$, $*=\text{, etc.}$

Make these members!
Operator Overloading
Distinguishing

Operators with the same symbol are distinguished by their parameters.

\[
\begin{align*}
\text{MyNum} & \quad \text{operator-} (\text{const MyNum} & \ a, \ \text{const MyNum} & \ b); \ // \ a-b \\
\text{MyNum} & \quad \text{operator-} (\text{const MyNum} & \ a); \ // \ -a
\end{align*}
\]

Some cannot be distinguished by the parameters we would expect.
- In particular, "++a" and "a++". The latter gets a dummy \text{int} parameter.

\text{class MyNum} \{
\text{public:}
    \text{MyNum} & \ \text{operator++}(); \ // \ ++a \\
    \text{MyNum} \ \text{operator++}(\text{int}); \ // \ a++
\}

Note the different \text{return} methods.
- Why are they different?
To input or print our objects we use C++ standard-library streams.

- We will look at stream **insertion** (operator<<).
- Stream **extraction** (operator>>) is similar.

The stream insertion operator:

- Takes an output stream (std::ostream) and some object.
- Returns the output stream.

This all makes the following work:

```cpp
cout << a << b;
```

which is the same as

```cpp
(cout << a) << b;
```

Returns `cout`, which can then be reused with `b`. 

Stream insertion:

- **Must** be global.
  - Otherwise, member of `std::ostream`, which we cannot write.
  - This is “Good Reason #2”.
- Gets its stream by (non-const!) reference.
  - Because it modifies the stream (by outputting to it).
- Gets its object to be printed by reference-to-const.

```cpp
std::ostream & operator<<(std::ostream & theStream,
                          const MyClass & theObject)
{
    theStream << theObject.x << " , " << theObject.y;
    // An example;
    // in practice, do whatever is appropriate.
    return theStream;
}
```
Operator Overloading
Final Comments

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.
- More on this when we discuss efficiency.
Here is a simple class **Dog**:

```cpp
// 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.

```cpp
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.

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

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

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

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.
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:

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

- When you declare an object with no ctor parameters:

```cpp
Dog mutt;
```

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

```cpp
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.

```cpp
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.

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

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

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

- When you pass an object by value:

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

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

```cpp
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.

```cpp
class Dog {
public:
    Dog & operator=(const Dog & rhs) // Not a ctor; no initializers
    {
        if (this != &rhs) // Avoid self-assignment
        {
            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:

```cpp
mutt = purebred;
```
Silently Written & Called Functions
Destructor [1/2]

The destructor is the function called when an object is destroyed.
- The silently written version does nothing, except that destructors for all data members are automatically called.

```cpp
class Dog {
public:
    ~Dog() {}  // Destructors 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:

```cpp
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:

```cpp
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):

```cpp
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.

Done. See silent.cpp, on the web page.
Silently Written & Called Functions
TO BE CONTINUED ...