Object-Oriented Principles and Practice / C++

Alice E. Fischer

June 3, 2013

Basic Concepts

Any class member, variable or parameter can be const.

- A const parameter is initialized during the process of calling the function.
- A const variable must be set = to a value in its declaration.
- A const class member is initialized by a ctor in the class constructor.

Const things cannot be modified by assignments after they are initialized.

Using Const

A const qualifier may appear in six different contexts in a program:

1. A class member can be const.
   It can be initialized by a ctor and later changed using a const-cast.
2. `retType funcName( const paramType& param );`
   `retType funcName( const paramType* param );`
   This says that the function does not modify its parameter.
   The actual argument does NOT need to be a const object.
3. `returnType funcName( paramType param ) const;`
   A const in this position says that the function does not modify its implied parameter (this). Used for documentation.
4. Inside the function, a local variable may be declared const.
   This follows the same rules for const as C.

More Const

5. `const retType& funcName( paramType param );`
   `const retType* funcName( paramType param );`
   This says that the function returns a pointer or reference to a const object.
   There is no purpose in using const this way if return value is NOT a pointer or reference. All primitive objects are const.
6. `retType* const funcName( paramType param );`
   The return value is a constant pointer that cannot be incremented.

When should you use const?

1. Const class members are usually static. They are used to create common knowledge among all the instances of a class.
2. Declare a parameter const to document your code if the function (and other functions it calls) do not change that parameter. This is good professional practice.
3. Declare *this* to be const for the same reasons.
4. Const local variables are useful if a value is calculated from the parameters and remains constant after that. Good practice.
5. Const return values are important. If a function returns a pointer or a reference, using const makes the value read-only.
6. If the return type of a function is a pointer, use const to prevent that pointer from pointing at any different object, or different slot of an array..

Global vs. member functions

A global function is one that is NOT defined inside a class. It takes zero or more *explicit* arguments.

- `f(a)` has one explicit argument, `a`.

A member function is defined inside a class. It always has an *implicit* argument along with zero or more *explicit* arguments.

- `c.g(a)` has explicit argument, `a` and implicit argument `c`.
- `d->g(a)` has explicit argument `a` and implicit argument `*d`.

An omitted implicit argument defaults to `(*this)`, which must make sense in the context.

- If `g` is a member function of class `MyClass`, then within `MyClass`, the call `g(a)` defaults to `this->.g(a)`

Defining Global and Member Functions

Placing a function declaration inside a class definition creates a member function.

There are three ways to define a global function.

1. Place the declaration inside a class definition, prefixed by the `static`. This creates a global function whose *name* is qualified by the class name. It's visibility is controlled by the visibility keywords `public`, `protected`, and `private`.
2. Place the declaration at the top level of your code, outside of any class declarations. Most functions in `C` are of this kind.
3. Place the declaration at the top level and prefix its name by `static`. Don't use this method; it is retained only for compatibility with `C`.

Operator Definition Syntax

- Each binary operator $\oplus$ corresponds to a function whose name is `operator`$\oplus$.
- All built-in operators can be extended as either global functions or member functions.
- The definition takes the form of a function definitions.
- Calls on the extended function can use EITHER operator syntax or function-call syntax.
- The extended operator has the same precedence and associativity as the built-in operator.

Operator syntax

- When compiling a program, the operator syntax $a \oplus b$ does not tell us whether to look for a global or a member function. Possibile meanings:
  - Global function: `operator⊕(a, b)`.
  - Member function: `a.operator⊕(b)`.
- It could mean either, and the compiler sees if either one matches. If both match, it reports an ambiguity.
- Given the choice, use a member operator function.
- We use a global form of `operator<<` because the left hand operator is of predefined type `ostream`, and we can't add member functions to that class.

Ambiguous operator extensions

```
class Bar {
public:
    int operator+(int y) { return y+2; }
};

int operator+(Bar& b, int y) { return y+3; }

int main() {
    Bar b;
    cout << b+5 << endl;
}
```

Compiler reports `error: ambiguous overload for 'operator+' in 'b + 5'`.

Operator Extensions

There are several kinds of operator extensions, each with its own syntax.

- You have been using extensions of the global output operator, <<. This is a 2-argument function; a definition looks like any 2-argument function in C.
- Operator extensions *inside* a class have an implied parameter in addition to the explicit parameters.
- Subscript is unusual because it returns a non-const reference.
- Prefix and postfix unary operator extensions.
- Constructor conversions.
- Type casts.

Operator extension as member function

Here's a sketch for defining a complex number class with operators.

```
class Complex {
private:
   double re;  // real part
   double im;  // imaginary part
public:
   Complex( double re, double im ) : re(re), im(im) {}
   Complex operator+(const Complex& b) const {
      return Complex( re+b.re, im+b.im );
   }
   Complex operator*(const Complex& b) const {
      return Complex(re*b.re-im*b.im, re*b.im+im*b.re);
   }
};
```

Extending Subscript

When defining a container class, it is normal to define `operator[]`

- This function will be used in the same ways as the built-in subscript for arrays, and have the same intuitive meaning.
- The implicit parameter is a data structure (such as a Flex-Array) that composes or aggregates an array.
- The explicit parameter is an integer, the slot number.
- Subscript normally returns a non-const reference, to allow reading and writing the array slot.
- Many subscript extensions do a bounds-check.

```
template <class T> inline T&
FlexArray<T>::operator[]( int k ) {
    if ( k >= N ) fatal( "Flex_array bounds error." );
    return Data[k];
}
```

Prefix unary operator extensions

C++ has a number of prefix unary operators
`*, -, ++, new, ...`

The corresponding operator functions are
`operator*(), operator-(), operator++(),`
`operator new(), ...`

The syntax for defining all the prefix operators is the same:

```
Complex operator ++ (){ rp++; return *this; }
```

Note that the only parameter is the implied parameter.

Postfix unary operator extensions

C++ also has two postfix unary operators, `++`, `--`.

The corresponding operator functions are `operator++(int)`, `operator--(int)`.

```
Complex operator ++ (int){
    Complex temp = *this; ++rp; return temp;
}
```

This is a special case that breaks all the normal rules of syntax, but it works because ++ and -- are not binary operators. The dummy `int` parameter should be ignored.

Type Casts

Suppose that the Cell class aggregates a pointer to an Item. Then you could define a cast from Cell to Item, in the Cell class:

```
operator Item() { return *data; }
```

- An extension of the type cast operator is used to convert a value from a class type to another type.
- This is also called "member conversion", because it converts a class member to some other type.
- Extensions of a type cast operator can be used explicitly, like get functions, but they are also used by the system to coerce operands and parameters.
- The presence of a cast operator in the class can contribute to code being hard to understand. Casts should be defined seldom and carefully!

Constructor Conversions

- A type cast operator in class C converts from type C to a different type, X.
- The opposite conversion, from type X to type C is done by defining a constructor with an explicit parameter of type X.
- Constructors of this form are called constructor conversions.
- A constructor conversion can be called explicitly, but it can also be used by the compiler to coerce a type-X value in a context where a type-C value is needed.

Polymorphism, Chapter 15

Basic Concepts

Purposes

Polymorphic Containers

Abstract Classes

Example: Linear

Basic Concepts

A class hierarchy becomes polymorphic when:

- The base class has one or more virtual functions.
- Derived classes may or may not redefine the virtual function. If so, the derived-class functions override the base-class function.
- The derived class implements the interface defined by the base class.

Compile Time Dispatching

The implied argument in a call on a class function can be an object or a pointer to an object.

- Suppose classs `B` and `C` are derived from class `A`, we have an object named `bb`, of class `B`, and we call `bb.print()`.
- Then the compiler will dispatch `B::print()`.
- If there is no print method in class `B`, the compiler will dispatch `A::print()`.
- If `A::print()` is not defined, compilation fails.
- All these tests and decisions are made at compile time.

Polymorphic Collections

Polymorphism becomes important when different objects are mixed in a container, and we use generic pointers to call functions.

- Let `A` be the base class for both `B` and `C`. All three classes define a `print()` function, which is virtual in `A`.
- Suppose we have an array named arr of type `A*`.
- We can store both `B*` and `C*` objects in this array because an instance of a derived type is also an instance of its base type.

Note: A class with even one virtual function must have a virtual destructor.

Virtual Dispatching

- If `A::print()` were NOT virtual, the compiler would dispatch `A::print()` to print all the items when we call `arr[k]->print();`.
- But because `A::print()` IS virtual, the compiler must dispatch the most specific method possible, which will often be in one of the derived classes.
- At compile time, we simply don't know what mixture of objects will be in this array.
- Thus, the dispatch decision must be made at run time.

Example: Run Time Dispatching

When we print the contents of `ary`

- We need to use the method that belongs to the actual type of the actual object in the array at run time.
- Every polymorphic object has a hidden type tag, at run time.
- To print an array value, the run-time system checks the type tag and dispatches the appropriate method. It will call `B::print()` for `B` objects and `C::print()` for `C::` objects.

This costs a little extra space on every instance and a little extra time on every function call.

We gladly pay this small price if the polymorphism is doing something useful.

Abstract Classes

An abstract class is incomplete

- In most ways, it is like a class, but one or more of its functions is not defined.
- These are called abstract functions.
- They are denoted by an `= 0;` at the end of the prototype in the class declaration.
- You cannot instantiate an abstract class. Its entire purpose is to be a base class for derivation.
- The abstract functions in the abstract class will be defined in some derived class, which can then be used to create objects.
- Abstract classes enforce a common interface on the derived classes.

Using polymorphism

Similar data structures:

- Linked list implementation of a stack of items.
- Linked list implementation of a queue of items.

Both support a common interface:

- `void put(Item*)`
- `Item* pop()`
- `Item* peek()`
- `ostream& print(ostream&)`

They differ only in where `put()` places a new item.

The demo `19-Virtual` (from Chapter 15 of textbook) shows how to exploit this commonality.

Interface file

We define this common interface by the abstract class.

```
class Container {
  public:
    virtual void     put(Item*)       =0;
    virtual Item*    pop()            =0;
    virtual Item*    peek()           =0;
    virtual ostream& print(ostream&) =0;
};
```

Any class derived from it is required to implement these four functions.

We could derive `Stack` and `Queue` directly from `Container`, but we instead exploit even more commonality between these two classes.

Class Linear

```
class Linear: public Container {
  protected:  Cell* head;
  private:    Cell* here; Cell* prior;
  protected:  Linear();
    virtual   ~Linear ();
              void     reset();
              bool     end() const;
              void     operator ++();
    virtual   void     insert( Cell* cp );
    virtual   void     focus() = 0;
              Cell*    remove();
              void     setPrior(Cell* cp);
  public:     void     put(Item * ep);
              Item*    pop();
              Item*    peek();
    virtual   ostream& print( ostream& out );
};
```

Example: Stack

```
class Stack : public Linear {
  public:
    Stack(){}
    ~Stack(){}
    void  insert( Cell* cp ) { reset(); Linear::insert(cp); }
    void  focus(){ reset(); }

    ostream& print( ostream& out ){
        out << "  The stack contains:\n";
        return Linear::print( out );
    }
};
```

Example: Queue

```
class Queue : public Linear {
  private:
    Cell*   tail;

  public:
    Queue() { tail = head; }
    ~Queue(){}

    void  insert( Cell* cp ) {
        setPrior(tail); Linear::insert(cp); tail=cp; }
    void  focus(){ reset(); }
};
```

Class structure

Class structure.

- `Container` specifies the common interface.
- `Linear` contains the bulk of the code. It is derived from `Container`.
- `Stack` and `Queue` are both derived from `Linear`.
- `Cell` is a "helper" class that is aggregated by `Linear`.
- `Item` is the base type for the container elements. It is defined by a `typedef` here but would normally be specified by a template.
- `Exam` is a non-trivial item type used by `main` to illustrate stacks and queues.

C++ features

The demo illustrates several C++ features.

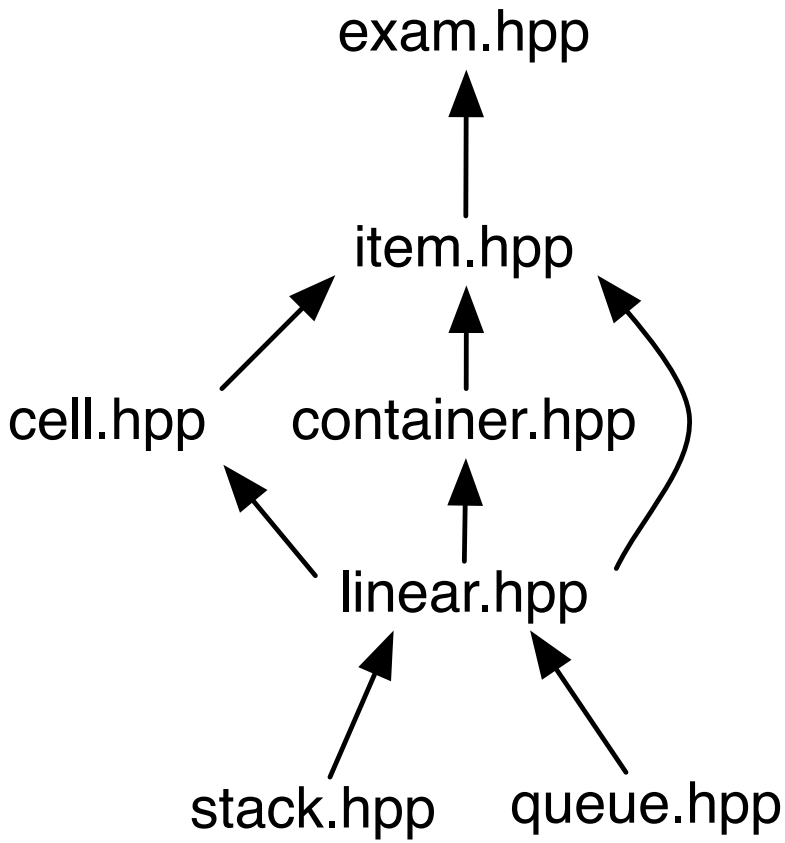
1. [`Container`] Pure abstract class.
2. [`Cell`] Friend functions.
3. [`Cell`] Printing a pointer in hex.
4. [`Cell`] Operator extension `operator Item*()`.
5. [`Linear`] Virtual functions and polymorphism.
6. [`Linear`] Scanner pairs (prior, here) for traversing a linked list.
7. [`Linear`] Operator extension `operator ++()`
8. [`Linear`, `Exam`] Use of `private`, `protected`, and `public` in same class.

#include structure

Getting #include's in the right order.

Problem: Making sure compiler sees symbol definitions before they are used.

Partial solution: Make dependency graph. If not cyclic, each .hpp file includes the .hpp files just above it.



Ordered Container

Demo `Multiple`

The purpose of demo `Multiple` is to generalize the linear containers of demo `Virtual` to support ordered lists of items.

It does this by adding `class Ordered`, creating two ordered containers of type `class List` and `class PQueue`, and extending the code appropriately.

`Ordered` base class

`Ordered` is an abstract class (interface) that promises items can be ordered based on an associated key.

It promises functions:

- A function `key()` that returns the key associated with an item.
- Comparison operators < and == that compare the derived item `*this` with an argument key.

Use:

```
class Item : public Exam, Ordered { ... };
```

Note: We can use private derivation because every function in `Ordered` is abstract and therefore must be overridden in `Item`.

Container base class

We saw the `Container` abstract class in demo `18c-Virtual`. It promises four functions: Colorblue

```
virtual void     put(Item*)       =0; // Put in Item
virtual Item*    pop()            =0; // Remove Item
virtual Item*    peek()           =0; // Look at Item
virtual ostream& print(ostream&) =0; // Print all Items
```

Use:

```
class Linear : Container { ...  };
```

class Item

Item is publicly derived from Exam, so it has access to Exam's public and protected members.

It fulfills the promises of Ordered by defining: Colorblue

```
bool
operator==(const KeyType& k) const { return key() == k; }
bool
operator< (const KeyType& k) const { return key() < k; }
bool
operator< (const Item& s)    const { return key() < s.key(); }
```

KeyType is defined with a typedef in exam.hpp to be int.

class Linear

`Linear` implements general lists through the use of a *cursor*, a pair of private `Cell` pointers `here` and `prior`.

Protected `insert()` inserts at the cursor.
Protected `focus()` is virtual and must be overridden in each derived class to set the cursor appropriately for insertion.

Cursors are accessed and manipulated through protected functions `reset()`, `end()`, and `operator ++()`.

Use:
`List::insert(Cell* cp) {reset(); Linear::insert(cp);}`
inserts at the beginning of the list.

class PQueue

PQueue inserts into a sorted list. Colorblue

```
void insert( Cell* cp ) {
    for (reset(); !end(); ++*this) {  // find  insertion spot.
        if ( !(*this < cp) )break;
    }
    Linear::insert( cp );             // do the insertion.
}
```

Note the use of the comparison between a PQueue and a Cell*.

This is defined in linear.hpp using the cursor:

```
bool operator< (Cell* cp) {
  return (*cp->data < *here->data); }
```