CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 18
November 5, 2018

The Many Uses of Classes

Virtue Demo

Linear Data Structure Demo

Functions Revisited

Operator Extensions

The Many Uses of Classes

What is a class?

- A collection of things that belong together.
- A struct with associated functions.
- A way to encapsulate behavior: public interface, private implementation.
- A way to protect data integrity, providing world with functions that provide a read-only view of the data.
- A data type from which objects (instances) can be formed. We say the instances belong to the class.
- A way to organize and automate allocation, initialization, and deallocation of storage.
- A way to break a complex problem into manageable, semi-independent pieces, each with a defined interface.
- A reusable module.

Virtue Demo

Virtual virtue

```
class Basic {
public:
    virtual void print(){cout <<"I am basic.  "; }
};
class Virtue : public Basic {
public:
    virtual void print(){cout <<"I have virtue.  "; }
};
class Question : public Virtue {
public:
    void print(){cout <<"I am questing.  "; }
};
```

Main virtue

What does this do?

```
int main (void) {
    cout << "Searching for Virtue\n";
    Basic* array[3];
    array[0] = new Basic();
    array[1] = new Virtue();
    array[2] = new Question();
    array[0]->print();
    array[1]->print();
    array[2]->print();
   return 0;
}
```

See demo `18a-Virtue`!

Linear Data Structure Demo

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 `18b-Virtual` (from Chapter 15 of textbook) shows how to exploit this commonality.

Interface file

We define this common interface by the pure abstract class.

```
class Container {
  public:
    virtual ~Container() {}
    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.

`Stack` and `Queue` could be derived directly from `Container`. Instead we exploit additional commonality between them.

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.

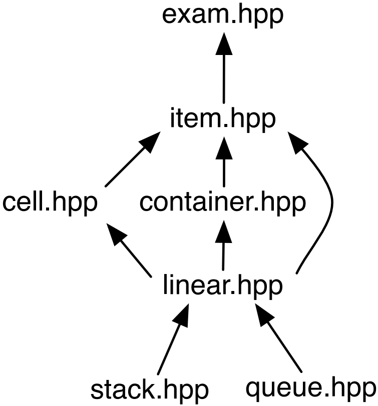
1. [`Container`] Pure abstract class.
2. [`Cell`] Friend functions.
3. [`Cell`] Printing a pointer in hex.
4. [`Cell`] Operator extension `operator Item*()`.
5. [`Linear`] Virtual functions and polymorphism.
6. [`Linear`] Scanner pairs (prior, here) for traversing a linked list.
7. [`Linear`] Operator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



Functions Revisited

Global vs. member functions

A **global** function is one that takes zero or more *explicit* arguments.
Example: `f(a, b)` has two explicit arguments `a` and `b`.

A **member** function is one that takes an *implicit* argument along with zero or more *explicit* arguments.
Example: `c.g(a, b)` has two explicit arguments `a` and `b` and implicit argument `c`.
Example: `d->g(a, b)` has two explicit arguments `a` and `b` and implicit argument `*d`.

Note that an omitted implicit argument defaults to `(*this)`, which must make sense in the context.
Example: If `g` is a member function of class `MyClass`, then within `MyClass`, the call `g(a, b)` defaults to `(*this).g(a,b)` (or equivalently `this->g(a,b)`).

Defining global functions

There are three ways to define a global function.

1. Place the declaration at the top level of your code, outside of any class declarations. Most functions in C are of this kind.
2. Place the declaration inside a class definition, prefixed by the keyword `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`.
3. Place the declaration at the top level and prefix its name by `static`. This creates a C-style static function whose name is visible only within the one compile module. Classes and static member functions provide a better way to provide modularity and control name visibility, so this should not be used in C++. It is retained only for compatibility with C.

Defining member functions

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

Its definition is considered to be “inside” the class, whether or not it appears in the class or as an out-of-line function in a `.cpp` file.

Example:

```
class MyClass {
protected:
    double g(const int* a, unsigned b) const;
};
```

This defines a member function `g` with explicit parameters of type `const int*` and `unsigned` and implicit parameter of type `const MyClass&`.

Operator Extensions

Operator syntax

We have seen the `operator` keyword used to extend the meaning of operators.

Each binary operator $\oplus$ corresponds to a function whose name is `operator`$\oplus$, but 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`$\oplus$`(a, b)`.
- Member function: `a.operator`$\oplus$`(b)`.

It could mean either, and the compiler sees if either one matches. If both match, it reports an ambiguity.

Operator extension as member function

Here's a sketch for how one might go about defining a complex number class.

```
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 );
   }
};
```

Operator extension as global function

We have seen one important example of a global operator extension when we define the output operator on a new class.

Given the choice, it is preferable to 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.

Prefix unary operator extensions

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

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

Postfix unary operator extensions

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

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

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

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'`.

Summary: How to define operator extensions

Unary operator `op` is shorthand for `operator op ()`.

Binary operator `op` is shorthand for `operator op (T arg2)`.

Some exceptions: Pre-increment and post-increment.

To define meaning of `++x` on type `T`, define `operator ++()`.

To define meaning of `x++` on type `T`, define `operator ++(int)` (a function of one argument). The argument is ignored.

Special case operator extensions

Some special cases.

- Subscript: `T& operator [](S index)`.
- Arrow: `X* operator ->()` returns pointer to a class `X` to which the selector is then applied.
- Function call; `T2 operator ()(`*`arg list`*`)`.
- Cast: `operator T()` defines a cast to type `T`.

Can also extend the `new`, `delete`, and `,` (comma) operators.