CPSC 427a: Object-Oriented Programming

Michael J. Fischer

Lecture 15
October 28, 2010

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 15a-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 push(Item*)`
- `Item* pop()`
- `Item* peek()`
- `ostream& print(ostream&)`

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

The demo `15b-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.

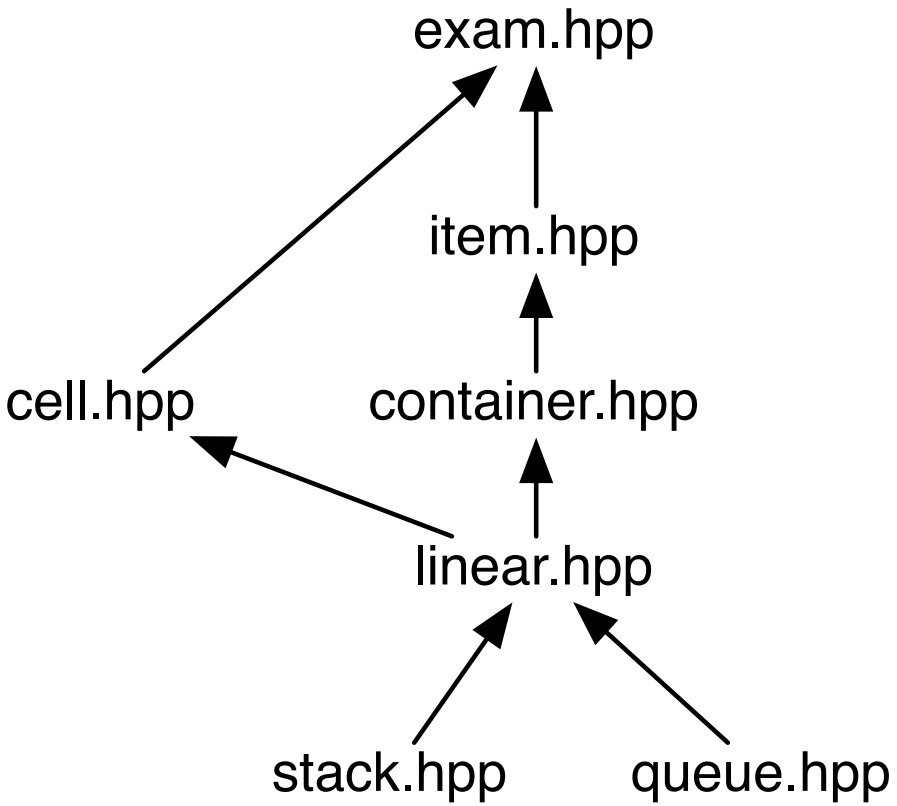
1. [`Container`] Pure abstract class.
2. [`Cell`] Friend functions.
3. [`Cell`] Printing a pointer in hex.
4. [`Cell`] Operator extension `operator Item*()`.
5. [`Linear`] Virtual functions and polymorphism.
6. [`Linear`] Scanner pairs (prior, here) for traversing a linked list.
7. [`Linear`] Operator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



Templates

Template overview

Templates are instructions for generating code.

Are type-safe replacement for C macros.

Can be applied to functions or classes.

Allow for type variability.

Example:

```
template <class T>
class FlexArray { ... };
```

Later, can instantiate

```
class RandString : FlexArray<const char*> { ... };
```

and use

```
FlexArray<const char*>::put(store.put(s, len));
```

Template functions

Definition:

```
template <class X> void swapargs(X& a, X& b) {
  X temp;
  temp = a;
  a = b;
  b = temp;
}
```

Use:

```
  int i,j;
  double x,y;
  char a, b;
  swapargs(i,j);
  swapargs(x,y);
  swapargs(a,b);
```

Specialization

Definition:

```
template <> void swapargs(int& a, int& b) {
  // different code
}
```

This overrides the template body for `int` arguments.

Template classes

Like functions, classes can be made into templates.

```
template <class T>
class FlexArray { ... };
```

makes `FlexArray` into a template class.

When instantiated, it can be used just like any other class.

For a flex array of ints, the name is `FlexArray<int>`.

No implicit instantiation, unlike functions.

Compilation issues

Remote (non-inline) template functions must be compiled and linked for each instantiation.

Two possible solutions:

1. Put all template function definitions in the `.hpp` file along with the class definition.
2. Put template function definitions in a `.cpp` file as usual but explicitly instantiate.
   E.g., `template class FlexArray(int);` forces compilation of the `int` instantiation of `FlexArray`.

Template parameters

Templates can have multiple parameters.

Example:
`template<class T, int size>` declares a template with two parameters, a type parameter `T` and an int parameter `size`.

Template parameters can also have default values.
Used when parameter is omitted.

Example:
`template<class T=int, int size=100> class A { ... }`.

`A<double>` instantiates `A` to type `A<double, 100>`.
`A<50>` instantiates `A` to type `A<int, 50>`.

Using template classes

Demo `15c-Evaluate` implements a simple expression evaluator based on a precedence parser.

It derives a template class `Stack<T>` from the template class `FlexArray<T>` introduced in `13-Hangman-full`.

The precedence parser makes uses of two instantiations of `Stack<T>`:

1. `Stack<double> Ands;`
2. `Stack<Operator> Ators;`