CPSC 427a: Object-Oriented Programming

Michael J. Fischer

Lecture 15
October 25, 2011

Runtime Tester

Demo: Hangman Game
- Game Rules
- Code Design
- Storage Management

Runtime Tester

Modularizing timing tests

The `14-StopWatch` demo presented last time illustrated the use of the `StopWatch` class to obtain the running time of some simple test programs.

All of the test code was thrown together in a monolithic `main()` function.

The code `PS2-timing`, which will be the basis of problem set 4, puts some structure on the test code.

It also illustrates some C++ features that we have not used much up to this point.

Structure of `class Tester`

Some design goals for class `Tester`:

1. Put the subject code to be timed into a separate function.
2. Separate the measurement code from the code block being tested.
3. Create a test manager `runAllTests()` that can be called by `main()` to carry out all tests, and delegate printing to it.
4. Provide a means for `main()` to control the parameters governing the test – in this case, the number of iterations and the seed for the random number generator.

Objective 1

`Tester` has two subject member functions:

- `test1()` exercises an array list.
- `test2()` exercises a linked list.

Objective 2

`Tester` has a function `measureRuntime()` that takes a member function as its argument and returns a `HirezTime` result.

This requires the use of **member function pointers**, which will be explained shortly.

Objective 3

Tests are managed by `runAllTests()` and `runOneTest()`.

Printing is handled by those two functions.

Objective 4

Optional parameters are passed to `Tester` through its constructor rather than being frozen in the `Tester` class.

Member function pointers

C++ supports pointers to class member functions through some new constructs:

1. The type declaration syntax is extended with the `::*` syntax.
2. The `&` operator is extended to apply to qualified member function names and return member function pointers.
3. Two new operators `.*` and `->*` are introduced to follow member function pointers and permit the referenced functions to be called.

Declaring member function pointers

The *type* of a member function must include the type of the implicit argument as well as the explicit argument types and return type. This is done by placing `::*` between the type of the explicit argument and the member function name.

Example:

```
    double (MyClass::*myPtr)(int)
```

declares `myPtr` to be a pointer to a member function of class `MyClass` that takes an explicit `int` argument and returns a `double`

If the implicit argument is `const`, the declaration becomes

```
    double (MyClass::*myPtr) const (int)
```

Using typedef with member function pointers

Many people find the member function pointer declaration syntax to be cumbersome and confusing.

I (and many others) highly recommend using `typedef` to give a simple descriptive name to the function pointer type.

This new type name is then used to declare function pointer parameters and function pointer variables.

Example:

```
typedef double (MyClass::*FunctionPtr)(int);
double someFunction(FunctionPtr f, int n) {...}
FunctionPtr myPtr;
```

Creating member function pointers

Function pointers are created using the & operator.

Example:

```
class MyClass {
public:
    double myFun( int n ) {
        return (double)n/3.0;
    }
};
...
myPtr = &MyClass::myFun;
```

Using member function pointers

Member function pointers are followed using one of the two C++ binary operators `.*` or `->*`.
Example:

1. `(obj.*myPtr)(17);`
2. `(objp->*myPtr)(17);`

Assuming `myPtr` currently points to member function `myFun`, these calls are equivalent to:

1. `obj.myFun(17);`
2. `(objp->myFun)(17);`

Demo: Hangman Game

Game Rules

Hangman game

Well-known letter-guessing game.

Start with a hidden *puzzle word*.

Player guesses a letter.

- If letter appears in puzzle word, matching letters are uncovered.
- If letter does not appear, it is shown in list of bad guesses.

Player wins when puzzle word is uncovered.

Player loses after 7 bad guesses

Code Design

Overall design

Game elements:

1. Puzzle word and letters found so far.
2. Bad guesses word.
3. Alphabet and letters left.
4. Vocabulary.
5. Game board display (viewer).
6. Game play (controller).

Use cases

Two levels.

1. Play one round of Hangman on a puzzle word.
    - Get input letter from user.
    - Classify input as good, bad, redundant, or not allowed.
    - Inform user and show updated board.
    - Announce termination and win/loss.
2. Repeated play
    - Choose unused word from vocabulary.
    - Play Hangman with that word.
    - Tally and announce win/loss.
    - Ask user whether to play again.

Code structure: Model

Model

1. `Alphabet` used to represent letters left.
2. `HangWord` used to represent puzzle word and bad guesses.
3. Both are derived from `BaseWord`
4. Common elements are a word and a visibility mask.
5. Variable elements:
   - How to print masked word.
   - Operations needed: `find` and `try`
6. Class `Board` data members store model state.

Code structure: Viewer and controller

Viewer Contained in class `Board`.

- `Board::print()` prints the puzzle, letters left, and bad guesses.
- `Board::move()` prints guess, outcome, and next board.
- `Board::play()` prints the win/loss message.

Controller Contained in class `Board`.

- `Board::play()` carries out turns and determines game termination.
- `Board::move()` prompts users for character and carries out turn.
- `Board::guess()` updates the model.

Class `Game`

Class `Game` is a top-level MVC design.

- **Model** contains alphabet, remaining vocabulary, and win/loss counters.
- **Viewer** is embedded in `Game::play()`.
- **Controller** is in `Game::playRound()` and `Game::play()`.

Storage Management

Storage management

Two storage management issues in Hangman:

1. How to store the vocabulary list?
2. How to store the words in the vocabulary?

Natural solutions are to store vocabulary as an array of pointers to strings.

Natural way to each string is to use `new` to allocate a character buffer of the appropriate length.

Design issues:

- How big should the vocabulary array be?
- Who owns the strings and takes responsibility for cleanup when they are no longer needed?

String store

A `StringStore` provide an alternative way to store words.

Instead of using `new` once for each string, allocate a big `char` array and copy strings into it.

When no longer needed, `~StringStore()` deletes entire array.

Advantages and disadvantages:

- *Much* more efficient—(each `new` consumes minimum of 32 bytes on modern machines).
- Simpler storage management—ownership of storage remains with `StringStore`.
- Downside: Can't reclaim storage from individual strings until the end.
- How big should the `char` array be?