Object-Oriented Principles and Practice / C++

Alice E. Fischer

June 1, 2015

A Multi-Class Example: Hangman

This program illustrates several things:

- A top-down design exercise for the class.
- Derive a template from a template.
- Simultaneously derive from and instantiate a template.
- Derived classes without polymorphism.
- Managing your own dynamic memory with a StringStore.
- Random selection from a list and removal from further use.
- A masked array.
- Various documentation techniques.
- Use of a command-line argument.
- Use of a friend class.

Hangman

This is a well-known letter-guessing game.

Start with a hidden *puzzle word*. You are given the length of the word in the form of a series of dashes, one per letter.

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

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 an unused puzzle word.
   - Play Hangman with that word.
   - Tally and announce win/loss.
   - Ask user whether to play again.

The Game Interface

What should be displayed?

1. The letters of the puzzle word that have been correctly guessed. Something must be displayed for letters that have not yet been guessed.
2. A list of wrong letters that have been guessed.
3. The number of remaining bad guesses before you die.
4. The remaining unguessed letters in the alphabet.

What should NOT be displayed?

1. A hidden vocabulary list from which a puzzle word can be randomly chosen.
2. A hidden puzzle word, chosen from the vocabulary.

How will we represent this data in our program?

Game Operation

The first three Game elements are displayed, the fourth is concealed:

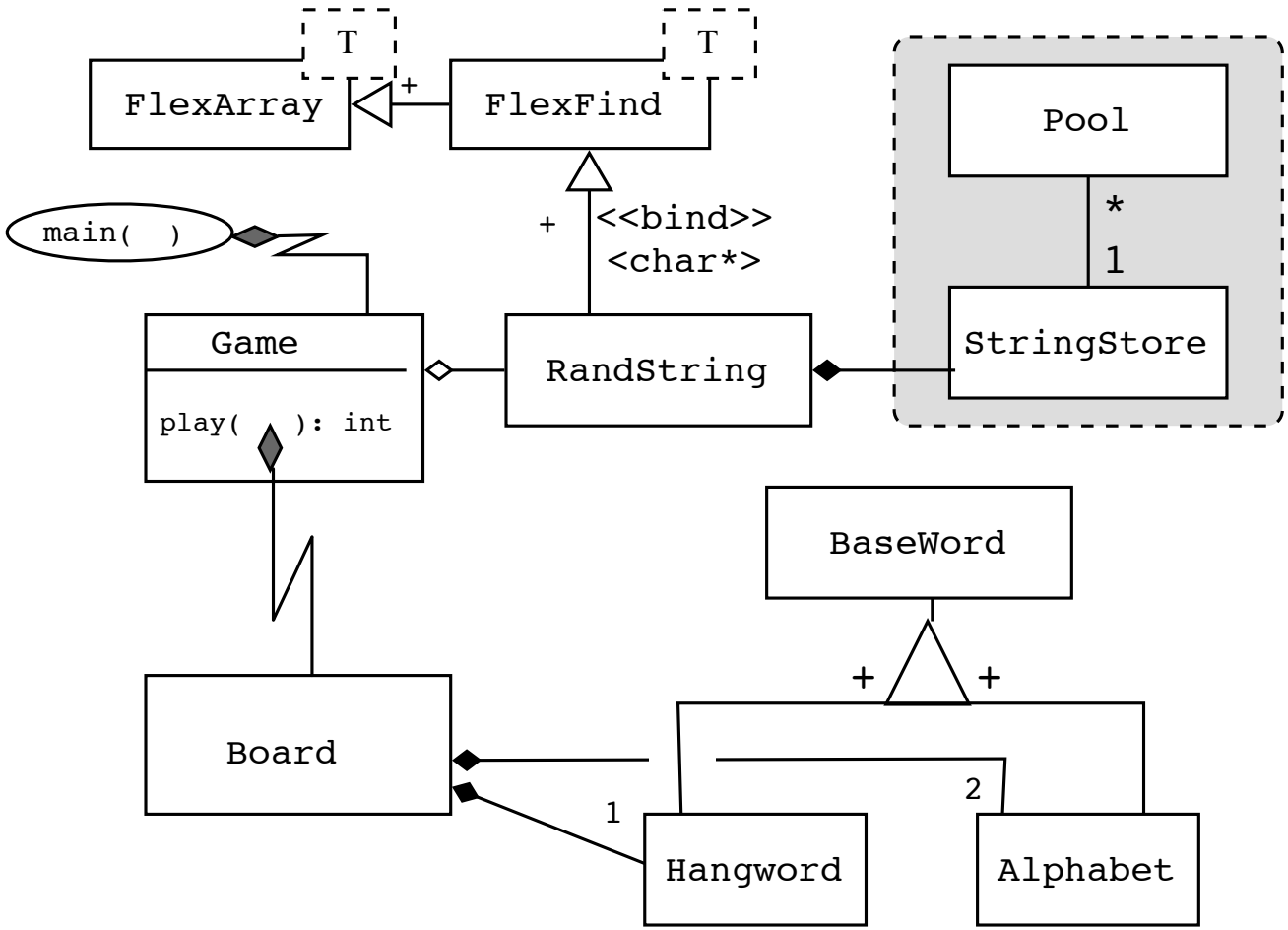
- The puzzle word, initially a series of dashes. As letters are correctly guessed, they replace the dashes.
- A list of wrong guesses, in alphabetical order, with dashes to indicate the number of remaining chances.
- The remaining letters in the alphabet.
- The vocabulary list will be read from a file.
- Words will be selected randomly from it.

Classes and Modules

1. `main`: handle name of vocabulary file; keep the session score.
2. `Game`: Read the vocabulary, choose a random word, construct a `Board`, and play it.
3. `Board`: Keep the state information for one round. Display the board, accept guesses, and keep score.
4. `RandString`: Derived from FlexArray, stores the vocabulary.
5. `Word`: Base class for the Alphabet and Hangword classes.
6. `Alphabet`: Used for the alphabet and errors displays.
7. `Hangword`: The puzzle word display

Alphabet and HangWord are derived from `BaseWord` by simple (non-polymorphic) derivation.

Detailed: UML for the Game



Implementation Notes

Getting started: `main()`
Playing a round
Problem: An unpredictable amount of data.
Classes and UML

The Main Function

- main() accepts a command-line argument: the name of a vocabulary file.
- Here, main() keeps score and does all user communication.
- Often, main() does much less – it only instantiates an application class transfers control to it.
- main might also handle logins and passwords.
- Hangman's main does some cute tricks with English grammar: lines 52–55.

We enter the OO world when main instantiates the Game class and calls play().

Code structure: Playing a Round

`Game::play()` creates an instance of `Board` with a randomly-chosen word.

- `Board::play()` calls `Board::move()` in a loop until the round is either won or lost.
- `Board::move()` prompts for and prints a guess, calls `Board::guess()`, and prints the result. An enum type is used to label the status of the guess.
- `Board::guess()` searches the alphabet for the guess, determines whether it is illegal or redundant. If neither, it searches the puzzle-word for the guess to determine whether it is wrong or right. If right, all occurrences of that letter in the puzzle-word are made visible. The status of the guess is returned.

Documentation Techniques

The Structure Chart
The Call Graph
The Data Diagram
The State Diagram or Flow Diagram

The UML Class Diagram

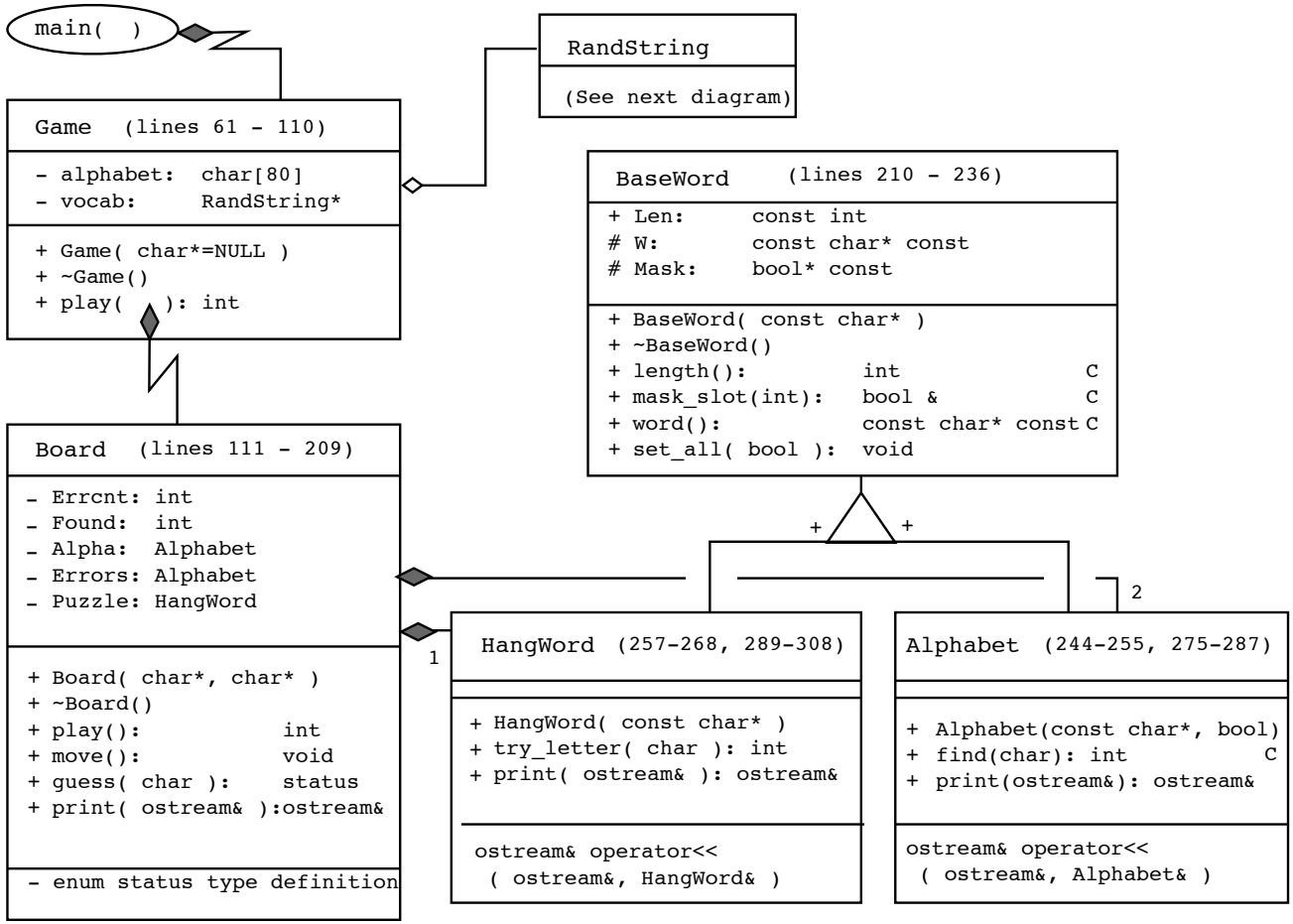
The purpose of a UML class diagram is to answer the question

Which classes will be affected when I change this class?

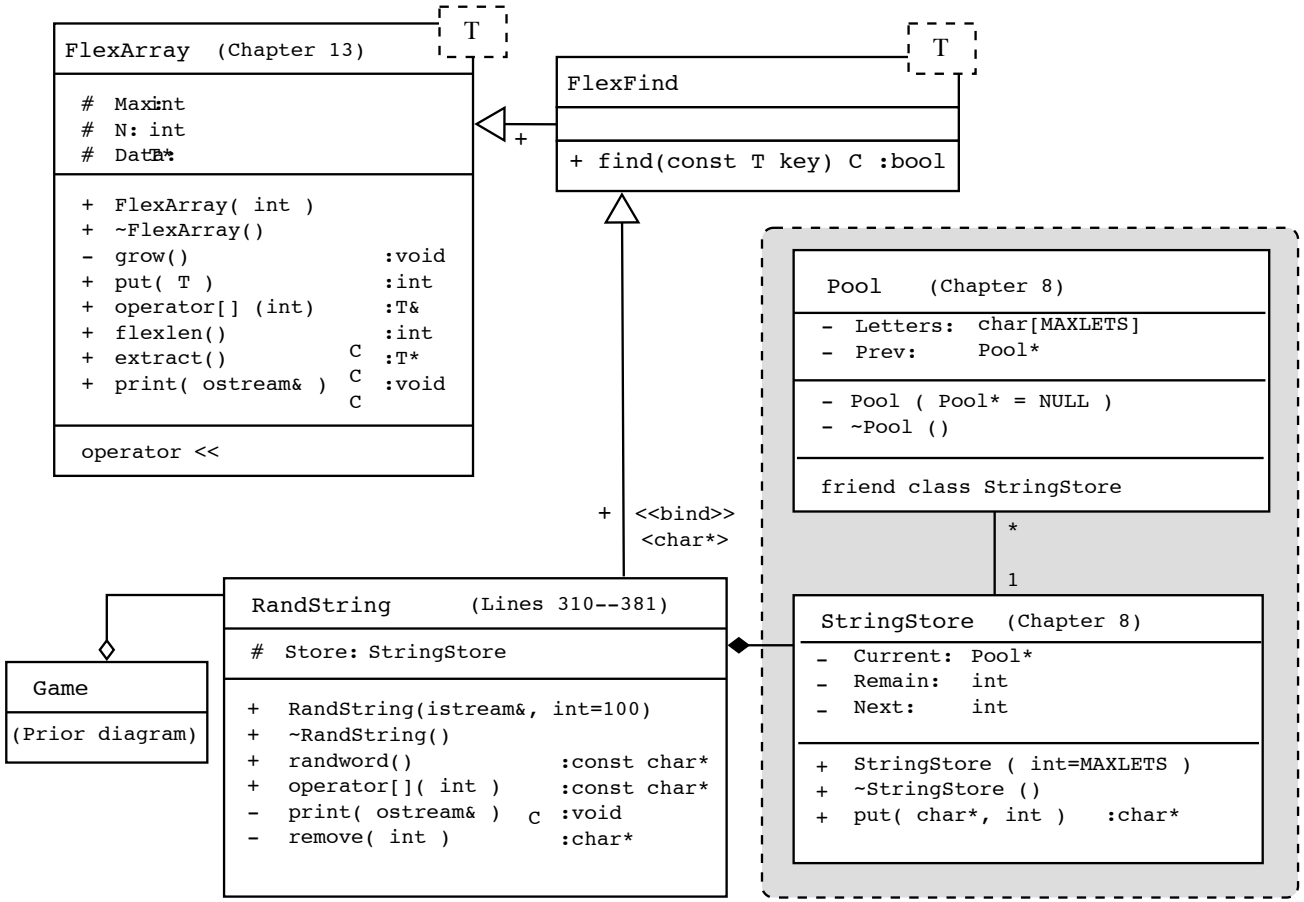
It is a static view of the structure of the code in an application.

- Each class in the application is represented by exactly one box.
- Lines document the relationship between the classes on each end.
- The contents of a box document the services it provides and the information it stores.
- This kind of overview evolves from the design process and is essential when changes must be made to an application.

Detailed UML for the Game



UML for the Vocabulary



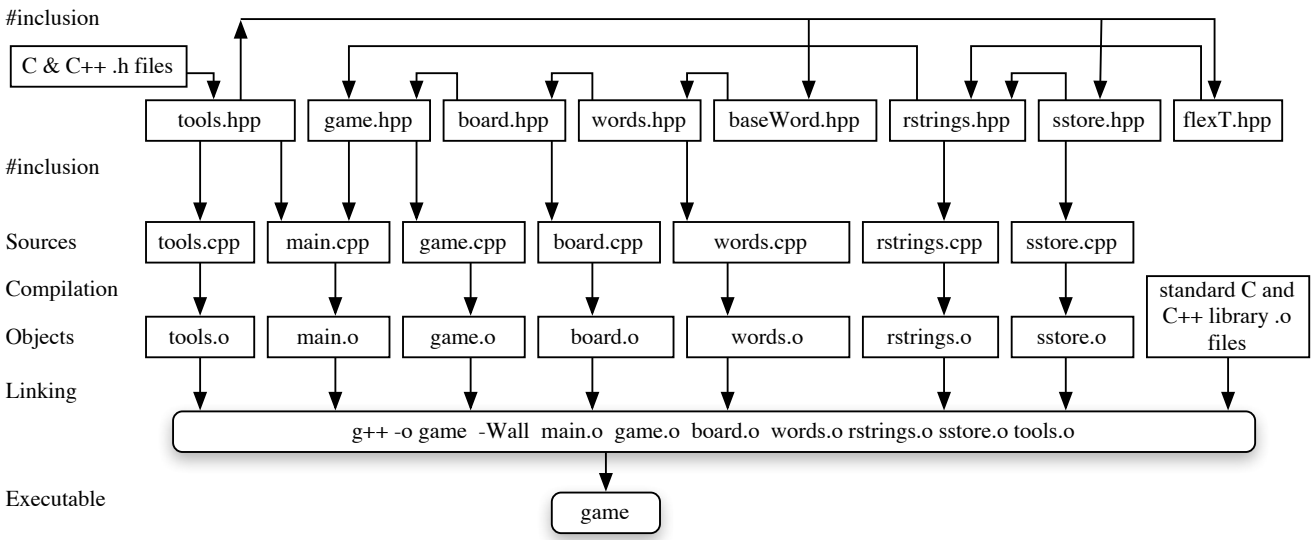
The Structure Chart

A structure chart contains the same information as a makefile or a project file. This information should be part of the written documentation for every project.

- A complete inventory of object files necessary to build the application. (Standard library modules are not listed.)
- For each object file, the related source code file is listed.
- For each code file, the required set of header files is shown.

A makefile provides a plain-text version of this information. It is used by the Unix make facility to recompile a module if any of its code or header files has been edited, and to relink the application if any of the object files has been recompiled. If all files are up to date, make returns without doing any work.

Structure Chart for Hangman



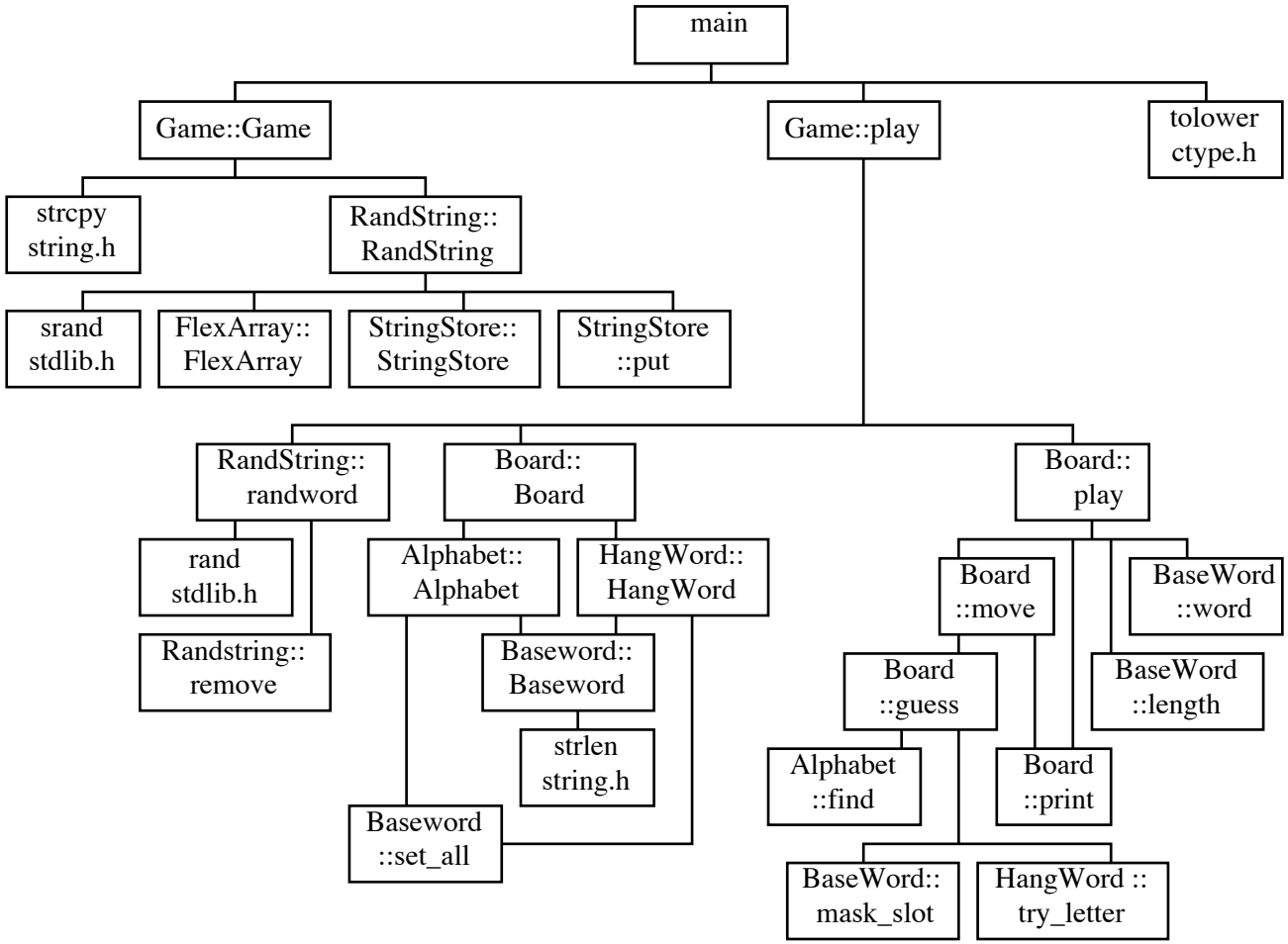
The Call Graph

The purpose of a call graph is to answer the question

How did I get here?

- Each function in the application is represented by exactly one box.
- A line is used to connect a caller (at the upper end of the line) and each function it calls.
- When a program error is discovered, it is not difficult to figure out what function was active.
- The call chart shows all ways that control could reach that function.
- This kind of overview can be useful during debugging and when changes must be made to an application.

Structure Chart for Hangman



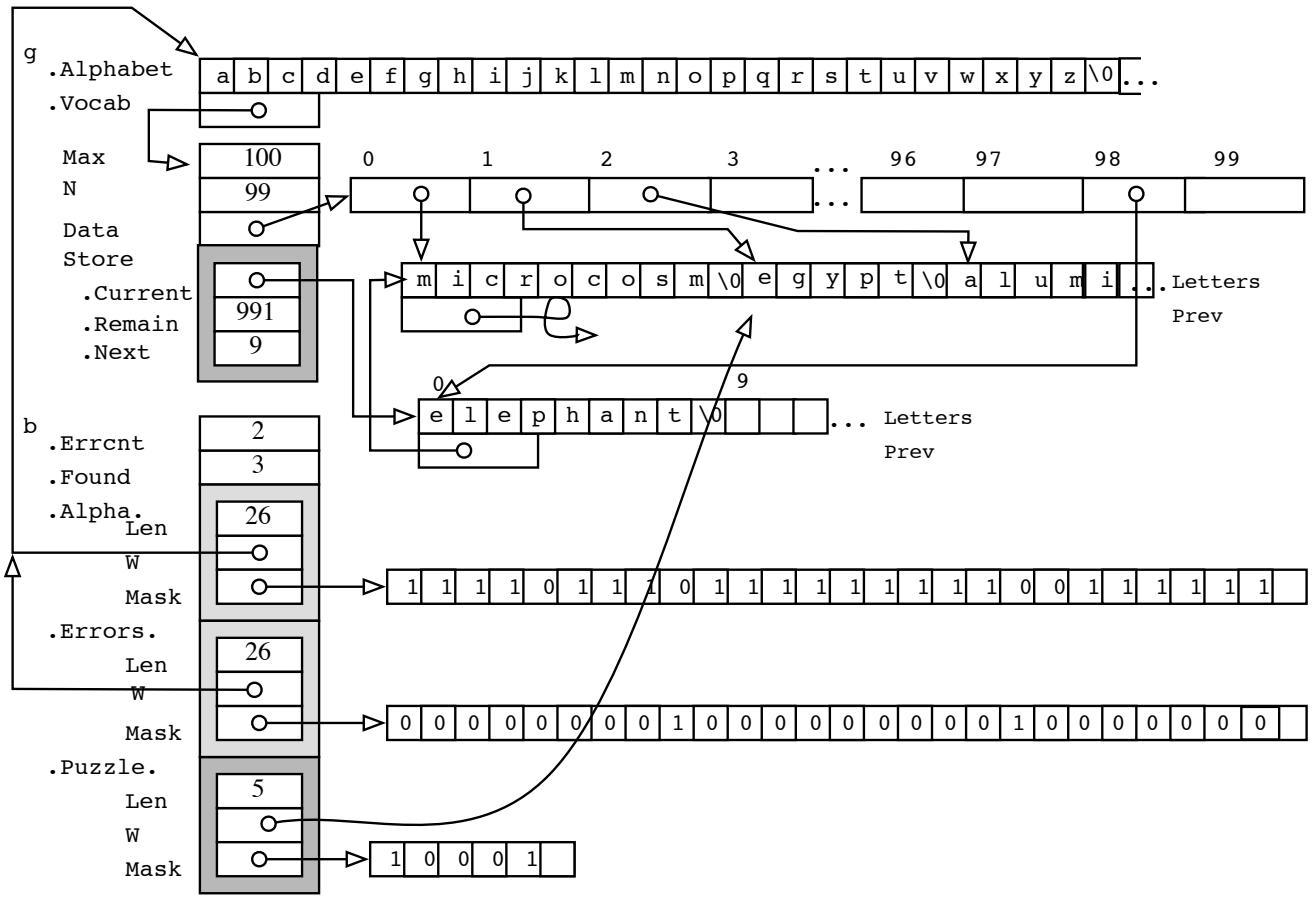
The Data Diagram

The purpose of a data diagram is to answer the question

What data structure is built and used at run time?

- A data diagram is related to a UML class diagram but looks very different.
- Boxes represent variables, structures, and arrays that are created at run time.
- Lines represent pointers.
- The difference between pointing at an object and copying it is clear here – and very important if you want to debug a program.

Structure Chart for Hangman



The State Diagram or Flow Diagram

The purpose of a state or flow diagram is to answer the question

Given a set of inputs to a decision process, what it the outcome?

- If control statements are nested more than 2 deep, a flow diagram can be useful for debugging and for later modification.
- This kind of documentation is good for program fragments, not for entire applications.
- In the Calendar program, you need a state or flow diagram to clarify your code for parsing and validating dates.

Hangman – New Techniques

Storage Management
Masked Arrays

Making the String storage grow.

When we begin to read the vocabulary file, we do not know how many words there will be.

- A `FlexArray<char*>` solves the problem.
- If the goal was to get something working asap, a `vector<string>` might be ideal. However, the goal is to learn about C++ and its capabilities, so we are using a home-built data structure.
- Why not use a `vector<string>`? Performance and complexity. We do not need all of the capabilities of a vector or of a string, nor do we want to deal with their complexities.
- So we will design a data structure that is as efficient as possible, with flexibility where needed.

Making the Character storage grow.

We cannot use a FlexArray to store the characters in the vocabulary words.

- Because the data in a FlexArray is moved when it grows, we cannot have pointers pointing *into* a FlexArray.
- So we need a different strategy for making the character-storage grow: we use a linked list of *pools*.
- Each pool is a long array of chars. The list of pools is managed like a stack, with the currently active pool on top.
- Keeping a list of pools allows us to deallocate the memory properly at the end of the game.

Masked Arrays

The Word classes implement a *masked array*.

- Masking is a useful technique for modifying the VIEW of the data without changing the data itself.
- Each masked object contains an array of data and a parallel array of bools.
- A bool controls whether its matching data is displayed or not.

Masking is used in spreadsheets to allow you to hide columns or rows.

Using the Masks

- For the alphabet, all mask-bits are set initially ON, and the whole alphabet shows. Each time the user guesses a letter, its corresponding bit will be turned off.
- For the list of wrong guesses, all mask-bits are set initially OFF, and no letters show. Each time the user guesses a wrong letter, its corresponding bit will be turned on.
- For the puzzle word, all mask-bits are set initially OFF, and no letters show. Each time the user guesses a correct letter, the bits corresponding to all occurrences of the letter will be turned on.

Non-polymorphic Derivation

Derivation without virtual functions
Template + Derivation

Derivation with Polymorphism

Polymorphic means "having more than one shape or form".

Suppose an application needs two or more similar but not identical data classes:

- We can factor out the common parts of those classes into in a base class, then derive the classes we want.
- If the application wants to put more than one variant into a container, the base class normally needs virtual functions and is, therefore, polymorphic.
- Objects of polymorphic types have run-time type tags that are tested when calling a virtual function on an object . . .
  - using a base-type pointer.
  - after removing the object from a base-type container.

No Polymorphism

Sometimes, however an application has a base class and derived classes but it has NO virtual functions.

- The application simply needs instances of the variations, and those instances are completely independent objects with independent variable names.
- All function calls on these objects can be fully compiled at compile time. No run-time type testing is needed.
- This kind of derivation is used in Hangman to implement the parts of the Board display.

Non-polymorphic Derivation in Hangman

- A polymorphic class has virtual functions that will behave differently depending on the type of an object. A run-time test must be made to determine the actual type of the implied parameter.
- BaseWord is not polymorphic; there are no virtual functions.
- One derived class is used for the two displays that are based on the alphabet.
- We use derivation here to unify common parts of the derived classes and avoid code duplication.
- The derived classes don't even contain methods that override base-class methods. However, if they did, it would still not be polymorphism.

Derivation in the Word classes

We use derivation here to factor out the commonality in the classes. There are no virtual functions. The different kinds of Words are never put into the same container or used through a base-type pointer.

- One derived class is used for the two displays that are based on the alphabet.
- The other is used for the puzzle word.
- The two derived classes have different search and print functions.

The FlexArray template – Ch. 13

- We discussed the basic action of a FlexArray early in the term. This is a growable array, specialized for efficiently adding items at the end of the array.
- The put() function adds items at the end of the array and reallocates the space, if necessary.
- Here, we change the simple version of FlexArray to a template so that we can have arrays of any base type.
- The definition of operator[] permits us to use a FlexArray as if it were an ordinary array. Nice.

The RandString Class

This class lets use choose a random puzzle word from a long vocabulary array.

- The FlexArray template implements a growable array of any base type, T.
- We instantiate it here to make a `FlexArray<char*>`, enabling us to handle as many strings (puzzle words) as the vocabulary file holds.
- We derive from the instantiation of `FlexArray` to produce the class RandString.
- The derivation step adds functionality to the FlexArray and tailors the class to our needs.

Using a StringStore – Ch. 9

- The RandString constructor realizes a long vocabulary file.
- The name of the file can be entered on the command line. If none is given, a default name is used.
- The characters of the vocabulary words are stored in a StringStore, which composes a linked list of large, dynamically allocated character arrays (Pools).
- The pointer to the first letter of each word is stored int the inherited FlexArray. This is our vocabulary list, from which we select puzzle words randomly.
- The resulting data structure is an extremely space and time efficient way to store a large number of variable-length data strings.

String store

A `StringStore` provides 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 the 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? Can it grow?

Derive from FlexArray<char*> to Build the String store

- A `StringStore` is a long array of word-pointers that point into an even longer array of chars.
- Since we do not know how many words will ultimately be in the vocabulary, it makes sense to use a FlexArray to store the list.
- We derive the class RandString from an instantiation of FlexArray with type `char*`. Note the way this is captured in the UML diagram.
  Implwm
- In the derived class, we add functionality.

Making the character storage grow.

We cannot use a FlexArray to store the characters in the vocabulary words.

- Because the data in a FlexArray is moved when it grows, we cannot have pointers pointing *into* a FlexArray.
- So we need a different strategy for making the character-storage grow: we use a linked list of *pools*.
- Each pool is a long array of chars. The list of pools is managed like a stack, with the currently active pool on top.
- Keeping a list of pools allows us to deallocate the memory properly at the end of the game.