Demo: Hangman Game (continued)
Refactored Game

Templates
Demo: Hangman Game (continued)
Refactored Game
Refactored hangman game

Demo 15-Hangman-full extends 15-Hangman in three respects:

1. It removes the fixed limitation on the vocabulary size.
2. It removes the fixed limitation on the string store size.
3. It more clearly separates the model of Board from the viewer/controller.

We’ll examine each of these in detail.
Flexible arrays

A FlexArray is a growable array of elements of type T.

Whenever the array is full, private method `grow()` is called to increase the storage allocation.

`grow()` allocates a new array of double the size of the original and copies the data from the original into it (using `memcpy()`).

Note: After `grow()`, array is 1/2 full.

By doubling the size, the amortized time is $O(n)$ for $n$ items.
Flex array implementation issues

**Element type:** A general-purpose **FlexArray** should allow arrays of arbitrary element type $T$.

If only one type is needed, we can instantiate $T$ using `typedef`. Example: `typedef int T;` defines $T$ as synonym for `int`.  

C++ **templates** allow for multiple instantiations.

**Class types:** If $T$ is a class type, then its default constructor and destructor are called whenever the array grows.

They must both be designed so that this does not violate the intended semantics.

This problem does not occur with numeric or pointer flexarrays.
String store limitation

Can’t use **FlexArray** to implement **StringStore** since pointers to strings would change after `grow()`.

Instead, when one **StringStore** fills up, start another.

Only really want another **storage pool**, not another **StringStore** object.

Each new **Pool** is linked to the previous one, enabling all pools to be deleted by `~StringStore()`.
Refactored Game

Refactoring Board class

Old design for Board contained the board model, the board display functions, and the user-interaction code.

New design puts all user interaction into a derived class Player.

This makes a clean separation between the model (Board) and the controller (Player).

The viewer functionality is still distributed between the two.

What are the pros and cons of this distribution?
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:

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

Later, can instantiate

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

and use

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

Definition:

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

Use:

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

Definition:

```cpp
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.

```cpp
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:

```cpp
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:

```cpp
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 16-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 15-Hangman-full.

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

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