CPSC 427a: Object-Oriented Programming

Michael J. Fischer

Lecture 14
October 26, 2010

Demo: Hangman Game (cont.)

Refactored Game

Refactored hangman game

Demo `11-Hangman-full` extends `10-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.

Flex 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.

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

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?

Coding Practices Reminders

Get and set functions

What's wrong with the following code?

```
class C {
private:
    int cost;
public:
    int getCost() const { return cost; }
    void setCost(int n) { cost = n; }
    ...
}
```

Rule: Don't use set functions!

(As with all rules, there are sometimes exceptions, but they are rare.)

Coping with privacy problems

Privacy violations usually mean that an action is being taken in the wrong class.

If a function in class `B` wants to manipulate variables in class `A`, it should delegate the operation to an appropriate function in `A`.

Type safety

What's wrong with the following code?

```
// void* return to make function generic
void* pop();
```

Reliable programming comes through checks and balances.
Types provide important protections in C++.
`void*` and certain kinds of casts circumvent the type system.

Rule: Don't bypass the type system!

Casts and Conversions

Casts in C

A C cast changes an expression of one type into another.

Examples:

```
int x;
unsigned u;
double d;
int* p;

(double)x;    // type double; preserves semantics
(int)u;       // type unsigned; possible loss of information
(unsigned)d;  // type unsigned; big loss of information
(long int)p;  // type long int; violates semantics
(double*)p;   // preserves pointerness but violates semantics
```

Different kinds of casts

C uses the same syntax for different kinds of casts.

Value casts convert from one representation to another, partially preserving semantics. Often called *conversions*.

- `(double)x` converts integer `x` to equivalent `double` floating point representation.
- `(short int)x` converts integer `x` to equivalent `short int`, *if the integer falls within the range of a* `short int`.

Pointer casts leave representation alone but change interpretation of pointer.

- `(double*)p` treats bits at destination of `p` as the representation of a double.

C++ casts

C++ has four kinds of casts.

1. *Static cast* includes value casts of C. Tries to preserve semantics, but not always safe. Applied at compile time.
2. *Dynamic cast* Applies only to pointers and references to objects. Preserves semantics. Applied at run time.
3. *Reinterpret cast* is like the C pointer cast. Ignores semantics. Applied at compile time.
4. *Const cast* Allows `const` restriction to be overridden. Applied at compile time.

Explicit cast syntax

C++ supports three syntax patterns for explicit casts.

1. C-style: `(double)p`.
2. Functional notation: `double(x); myObject(10);`.
   (Note the similarity to a constructor call.)
3. Cast notation:
   `int x; myBase* b; const int c;`
   - `static_cast<double>(x);`
   - `dynamic_cast<myDerived*>(b);`
   - `reinterpret_cast<int>(p);`
   - `const_cast<int>(c);`

Implicit casts

General rule for implicit casts: If a type `A` expression appears in a context where a type `B` expression is needed, use a semantically safe cast to convert from `A` to `B`.

Examples:

- Assignment: `int x; double d; x=d; d=x;`
- Pointer assignment:
  ```
  class A { ... };
  class B : public A { ... };
  A* ap; B* bp; ap = bp;
  ```
- Initialization:
  `A a=x;` converts `x` to an `A`, then copies.
- Construction:
  `A a(x);` calls `A` constructor, possibly casting `x`.

Ambiguity

Can be more than one way to cast from `B` to `A`.

```
class B;
class A { public:
  A(){}
  A(B& b) { cout<< "constructed A from B\n"; }
};
class B { public:
  A a;
  operator A() { cout<<"casting B to A\n"; return a; }
};
int main() {
  A a; B b;
  a=b;
}
```

error: conversion from 'B' to 'const A' is ambiguous

explicit keyword

Not always desirable for constructor to be called implicitly.

Use `explicit` keyword to inhibit implicit calls.

Previous example compiles fine with use of `explicit`:

```
class B;
class A {
public
  A(){}
  explicit A(B& b) { cout<< "constructed A from B\n"; }
};
...
```

Question: Why was an explicit definition of the default constructor not needed?

Operator Extensions

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.

Other special cases

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.