CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 20
April 12, 2016

Templates

Templatizing a class

Demo `20a-BarGraph` results from templatizing `Row` and `Cell` classes in `09-BarGraph`.
Template parameter `T` replaces uses of `Item` within `Row`.

Here is what was necessary to carry this out:

1. Fold the code from `row.cpp` into `row.hpp`.
2. Precede each class and function declaration with `template<class T>`.
3. Follow occurrences of `Row` with template argument `<Item>` in `Graph.hpp` and `Graph.cpp`.
4. Follow each use of `Row` with template argument `<T>` in `row.hpp`.

Using template classes

Demo `20b-Evaluate` uses templates and derivation together by deriving a template class `Stack<T>` from the template class `FlexArray<T>` introduced in `18b-Hangman-full`.

It is a simple expression evaluator based on a precedence parser.

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

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

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. [See demo `20c-Dynamic_cast`.]
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)x`.
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.

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 20d-Virtue!