CPSC 427a: Object-Oriented Programming

Michael J. Fischer

Lecture 18
November 3, 2011

Casts and Conversions

Operator Extensions

Virtue Demo

Linear Data Structure Demo

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 `18a-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)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.

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 `18b-Virtue`!

Linear Data Structure Demo

Using polymorphism

Similar data structures:

- Linked list implementation of a stack of items.
- Linked list implementation of a queue of items.

Both support a common interface:

- `void      push(Item*)`
- `Item*     pop()`
- `Item*     peek()`
- `ostream& print(ostream&)`

They differ only in where `push()` places a new item.

The demo `18c-Virtual` (from Chapter 15 of textbook) shows how to exploit this commonality.

Interface file

We define this common interface by the abstract class.

```
class Container {
  public:
    virtual void      put(Item*)        =0;
    virtual Item*     pop()             =0;
    virtual Item*     peek()            =0;
    virtual ostream& print(ostream&) =0;
};
```

Any class derived from it is required to implement these four functions.

We could derive `Stack` and `Queue` directly from `Container`, but we instead exploit even more commonality between these two classes.

Class Linear

```
class Linear: public Container {
  protected:  Cell* head;
  private:    Cell* here; Cell* prior;
  protected:  Linear();
    virtual   ~Linear ();
              void      reset();
              bool      end() const;
              void      operator ++();
    virtual   void      insert( Cell* cp );
    virtual   void      focus() = 0;
              Cell*     remove();
              void      setPrior(Cell* cp);
  public:     void      put(Item * ep);
              Item*     pop();
              Item*     peek();
    virtual   ostream& print( ostream& out );
};
```

Example: Stack

```
class Stack : public Linear {
  public:
    Stack(){}
    ~Stack(){}
    void  insert( Cell* cp ) { reset(); Linear::insert(cp); }
    void  focus(){ reset(); }

    ostream& print( ostream& out ){
        out << "  The stack contains:\n";
        return Linear::print( out );
    }
};
```

Example: Queue

```
class Queue : public Linear {
  private:
    Cell*   tail;

  public:
    Queue() { tail = head; }
    ~Queue(){}

    void  insert( Cell* cp ) {
        setPrior(tail); Linear::insert(cp); tail=cp; }
    void  focus(){ reset(); }
};
```

Class structure

Class structure.

- `Container` specifies the common interface.
- `Linear` contains the bulk of the code. It is derived from `Container`.
- `Stack` and `Queue` are both derived from `Linear`.
- `Cell` is a “helper” class that is aggregated by `Linear`.
- `Item` is the base type for the container elements. It is defined by a `typedef` here but would normally be specified by a template.
- `Exam` is a non-trivial item type used by `main` to illustrate stacks and queues.

C++ features

The demo illustrates several C++ features.

1. [`Container`] Pure abstract class.
2. [`Cell`] Friend functions.
3. [`Cell`] Printing a pointer in hex.
4. [`Cell`] Operator extension `operator Item*()`.
5. [`Linear`] Virtual functions and polymorphism.
6. [`Linear`] Scanner pairs (prior, here) for traversing a linked list.
7. [`Linear`] Operator extension `operator ++()`
8. [`Linear`, `Exam`] Use of `private`, `protected`, and `public` in same class.

#include structure

Getting `#include`'s in the right order.

Problem: Making sure compiler sees symbol definitions before they are used.

Partial solution: Make dependency graph. If not cyclic, each `.hpp` file includes the `.hpp` files just above it.