Handles and use counts (revisited)

Improving the Expr class Interface

Expr Interface Problems

- The Expr hierarchy we built requires users to explicitly manage memory

```
Expr* e =
    new BinaryExpr("+",
        new UnaryExpr("-", new IntExpr(5)),
        new BinaryExpr("*", new IntExpr(3),
                            new IntExpr(4)));

e->print(cout);
delete e;
```

More problems

- We can't copy `Exprs`, only objects derived from `Expr`
- Memory management is incomplete:

```
Expr* e = new IntExpr(42);
Expr* e2 =
      new BinaryExpr("+", e, e);
delete e2;     // Oops!
```

We can do better...

- Revise our expression classes to behave like values, efficiently
  - Provide efficient & correct copy semantics (do not copy the underlying tree)
  - Memory management should be automatic
- Hide the type hierarchy (because values are not objects, so where would the hierarchy be useful?)

How are we going to use it?

- Only one user-visible expression type
- Overloading can distinguish the constructors

```
Expr(int)
Expr(const char*, const Expr&)
Expr(const char*,
     const Expr&, const Expr&)
```

Example of usage

```
Expr e("*",
       Expr("-", Expr(3)),
       Expr("+", Expr(4), Expr(5)));
e.print(cout);
```

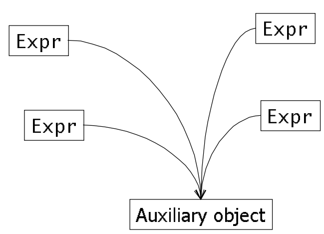
Declaration of `Expr`

```
class Expr {
public:
    Expr(int);
    Expr(const char*, const Expr&);
    Expr(const char*,
         const Expr&, const Expr&);
    Expr(const Expr&);
    Expr& operator=(const Expr&);
    ~Expr();
    void print(ostream&) const;
};
```

Why no virtuals?

- Part of the purpose of this class is to hide the earlier `Expr` hierarchy
- We are not going to inherit from this version of `Expr`; instead, we will define a new hierarchy

The data structure



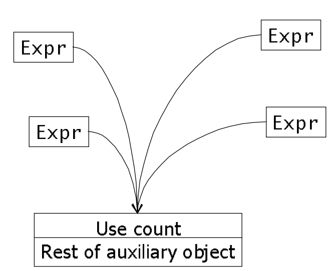
The auxiliary data structure

- Contains the real data associated with an `Expr`
- Does not need to be modified once created (because `print` is nondestructive)
- Needs to be deleted when the last `Expr` pointing to it has gone away

General C++ strategy for such structures

- Class `Expr` contains (only) a pointer to the auxiliary class
- The auxiliary class is an abstract base class for the hierarchy
- The auxiliary class also contains a use count, which `Expr` manipulates

Revised data structure



What classes do we need?

- Class `Expr` is the user interface
- Class `ExprBase` is the root of the auxiliary hierarchy
- Other classes in the hierarchy
  - `IntExpr`
  - `UnaryExpr`
  - `BinaryExpr`

These classes know about each other

- When the user creates an `Expr`, it must know how to create the appropriate auxiliary class
- Class `Expr` must also know how to manipulate the use count in `ExprBase`
- We will have a lot of friendly classes on our hands

The class hierarchy

```
class Expr { /* … */ };
class ExprBase { /* … */ };
class IntExpr: public ExprBase
    { /* … */ };
class UnaryExpr: public ExprBase
    { /* … */ };
class BinaryExpr: public ExprBase
    { /* … */ };
```

Private data in `Expr`

```
class Expr {
public:
    // As before
private:
    ExprBase* p;
};
```

Declaration of `ExprBase`

```
class ExprBase {
    friend class Expr;
protected:
    ExprBase();
    int use;
    virtual void print(ostream&)
        const = 0;
    virtual ~ExprBase() { }
};
```

We can start to define `Expr`

- The `print` function just calls the corresponding `ExprBase` virtuals

```
void Expr::print(ostream& o) const
{
    p->print(o);
}
```

What about the constructors?

- We can start with the one-argument constructor and see how we would like it to work:

```
Expr::Expr(int n):
    p(new IntExpr(n)) { }
```

- What does this desired usage say about class `ExprBase`?

The `ExprBase` constructor

- We want the use count in `ExprBase` to count how many `Expr` objects point to this particular `ExprBase` object
- When we construct an `ExprBase`, that number is about to be 1
- Therefore, the constructor should arrange that:

```
ExprBase::ExprBase(): use(1) { }
```

The `Expr` destructor

- Manipulate the use count of the corresponding `ExprBase` object
- Destroy it if (and only if) the use count becomes zero

```
Expr::~Expr()
{
    if (--p->use == 0)
        delete p;
}
```

`Expr` copy and assignment

- Copying an `Expr` never needs to copy the underlying `ExprBase`, because there are no mutative operations on `ExprBase` objects
- Ditto for assignment
- So we just copy the pointers and manipulate the use counts
- Assignment is slightly tricky

Implementing copy and assignment

```
Expr::Expr(const Expr& e): p(e.p)
{
    ++p->use;
}
Expr& Expr::operator=(const Expr& e)
{
    ++e.p->use;
    if (--p->use == 0)
        delete p;
    p = e.p;
    return *this;
}
```

Look familiar?

- This code looks remarkably like the corresponding code from lecture X in which we implemented use-counted `Strings`
- This technique is sufficiently common that you really want to understand it thoroughly

The other two constructors are simple

```
Expr::Expr(const char* op,
    const Expr& e):
    p(new UnaryExpr(op, e)) { }
Expr::Expr(const char* op,
    const Expr& e1, const Expr& e2):
    p(new BinaryExpr(op, e1, e2)) { }
```

What about those other three classes?

- Class `IntExpr` is pretty simple
- Classes `UnaryExpr` and `BinaryExpr` both have to deal with subexpressions
- Fortunately, we now have a way to deal with such subexpressions, namely class `Expr` itself!
- Instead of storing pointers, we will store `Expr` objects, which are abstractions of pointers

Class `IntExpr` definition

```
class IntExpr: public ExprBase {
    friend class Expr;
    IntExpr(int n);
    void print(ostream&) const;
    int n;
};
```

`IntExpr` implementation

```
IntExpr::IntExpr(int n0): n(n0) { }
void IntExpr::print(ostream& o) const
{
    o << n;
}
```

Class `UnaryExpr` definition

```
class UnaryExpr: public ExprBase {
    friend class Expr;
    const char* op;
    Expr e;
    UnaryExpr
        (const char*, const Expr&);
    void print(ostream&) const;
};
```

Implementation note

- That `Expr` member called `e` in class `UnaryExpr` is magic in several ways
  - Using `Expr::Expr(const Expr&)` to initialize it will take care of memory management
  - Calling `e.print` will result in appropriate virtual calls automatically
  - Destroying the surrounding `UnaryExpr` will destroy `e` appropriately

UnaryExpr implementation

```
UnaryExpr::UnaryExpr
    (const char* x, const Expr& y):
    op(x), e(y) { }
void UnaryExpr::print(ostream& o) const
{
    o << "(";
    e.print(o);
    o << ")";
}
```

BinaryExpr definition

```
class BinaryExpr: public ExprBase {
    friend class Expr;
    const char* op;
    Expr e1;
    Expr e2;
    BinaryExpr(const char*,
        const Expr&, const Expr&);
    void print(ostream&) const;
};
```

BinaryExpr implementation

```
BinaryExpr::BinaryExpr(const char* x,
    const Expr& y, const Expr& z):
    op(x), e1(y), e2(z) { }
void BinaryExpr::print
     (ostream& o) const {
    o <<, "(";
    e1.print(o);
    o << op;
    e2.print(f);
    o << ")";
}
```

Compiling the program

- Getting a program like this to compile can be a bit of a pain
- The hard part is putting the pieces in the right order
- General rules will help
  - Names must be declared before they are used
  - Definitions can usually come fairly late

Typical dependencies

- Class `Expr` must know about class `ExprBase`

```
class Expr {
    // …
    ExprBase* p;
};
```

- The `Expr` constructors must know about the various derived classes

An ordering that works

- Declaration of `ExprBase` (which implicitly declares `Expr` as a class by naming it as a friend)
- Declaration of `Expr` (which uses `ExprBase` as the type of `p`)
- Declarations of the derived classes
- Member function definitions

An alternative ordering

- First, say
  `class ExprBase;`
  to make it known that `ExprBase` is the name of a class
- Next, declare `Expr` (which needs to know about `ExprBase`)
- Then declare `ExprBase`, and continue as before

What about garbage collection?

- It would be nice to have it, but
  - Very little of the code in this example is concerned with memory management
  - Use counts let us free resources accurately and immediately, without waiting for the next garbage collection
  - Garbage collection deals only with memory; there are other resources too.

Discussion

- This whole program has been about defining values that use objects
- So far, however, we really haven't cared much about objects versus values, because we don't modify the objects
- However, we can extend the handle techniques to include "copy on write"

Discussion (continued)

- Even if we don't modify our objects, we have
  - made it possible to copy handles without copying the underlying data structures
  - arranged for the data structures to go away automatically when we're done with them