Handles and use counts (part 1)

Making objects act like values

Overview

- Suppose we have a class whose objects we do not wish to copy
  - strings
  - arrays and other containers
  - the expression trees we saw last week
- How can we avoid copying such objects except when truly necessary?
- One way: copy something else instead

What is a copy?

- A copy of an object is a distinct object with the same properties as the original
- If the object is part of a large data structure, we might distinguish between
  - copying just that object (shallow copy)
  - copying the whole structure (deep copy)

Why do we want to copy objects?

- How do you tell whether two names refer to the same object?
  - Modify one object
  - Observe the change in the other
- Making a copy of an object
  - usually precedes a change to one of the objects
  - is unnecessary otherwise

Example of necessary copying

- Suppose we have a string class that supports an `append` operation
- Then we can define `concatenate` in terms of `append`:
  - Copy one of the operands
  - Append the other operand to the copy
- We must copy the original string to avoid destroying it

C++ implementation

```
class String {
public:
    String& operator+=(const String&);
    // …
};
String operator+
    (const String& s1, const String& s2)
{
    String result = s1;
    result += s2;
    return result;
}
```

Smalltalk does it differently

- In Smalltalk, copying is always explicit
  - all types are objects
  - all variables are references
  - after `x←y`, `x` and `y` always refer to the same object
- Therefore, there is no way to change the value of objects of primitive types such as `int` and `string`

Java does it differently, too

- Simple types, such as `int`, are values, not objects
  - Each variable of one of these types has its own copy
  - Therefore, operations like ++ present no problem
- Strings are objects, so changing part of a string presents a problem

The root of the problem

- Most programmers think of some types as being values and others as objects
- Languages that come close to treating everything as an object have trouble dealing with values
- C++, which treats almost everything as a value, requires extra awareness to deal with objects

Dealing with objects in C++

- A pointer is a value that is bound to a particular object
- A reference isn't even a value: It's just a name that is bound to an object
- Virtual functions work only through such bindings
- We can define classes that also act like bindings and are useful in other ways

Properties of binding objects

- Attach one to some other object
- Access (modify?) the object to which it is attached
- Copy the binding object (which might copy the other object or not)
- Destroy the binding object (which might destroy the other object or not)

Binding objects have many forms and names

- Pointers
- References
- Smart pointers
- Handles
- Surrogates
- Iterators

A concrete example

- Revise our expression classes to behave like values, efficiently
  - Copying an expression should 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(stdout);
int n = e.eval();
```

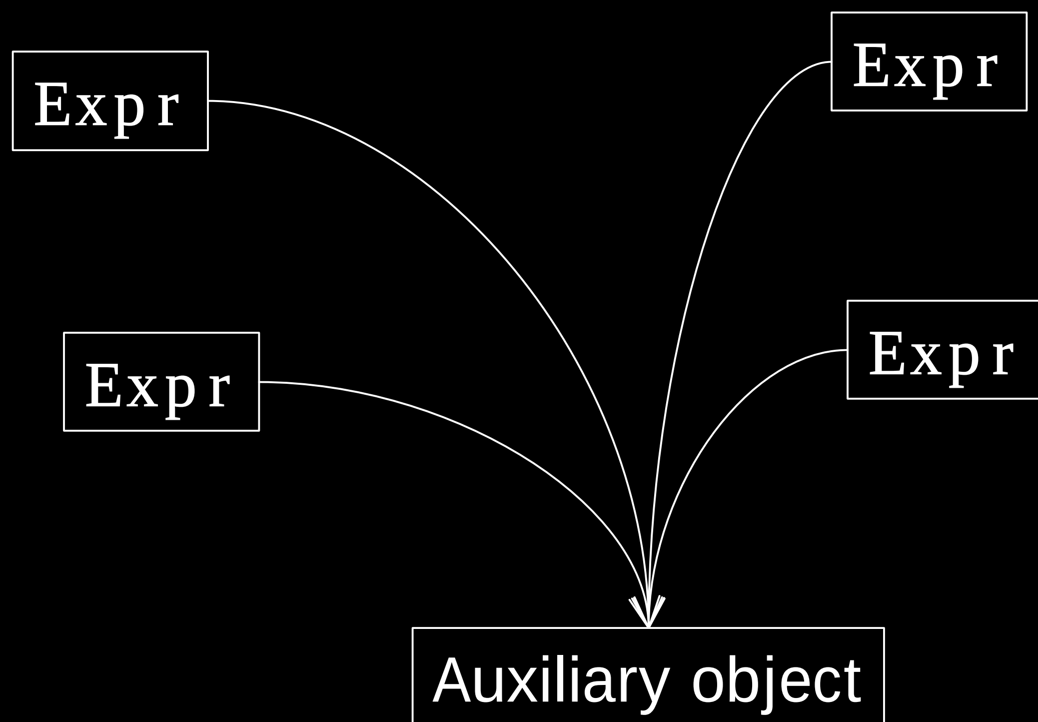
Declaration of Expr

```
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(FILE*) const;
    int eval() 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



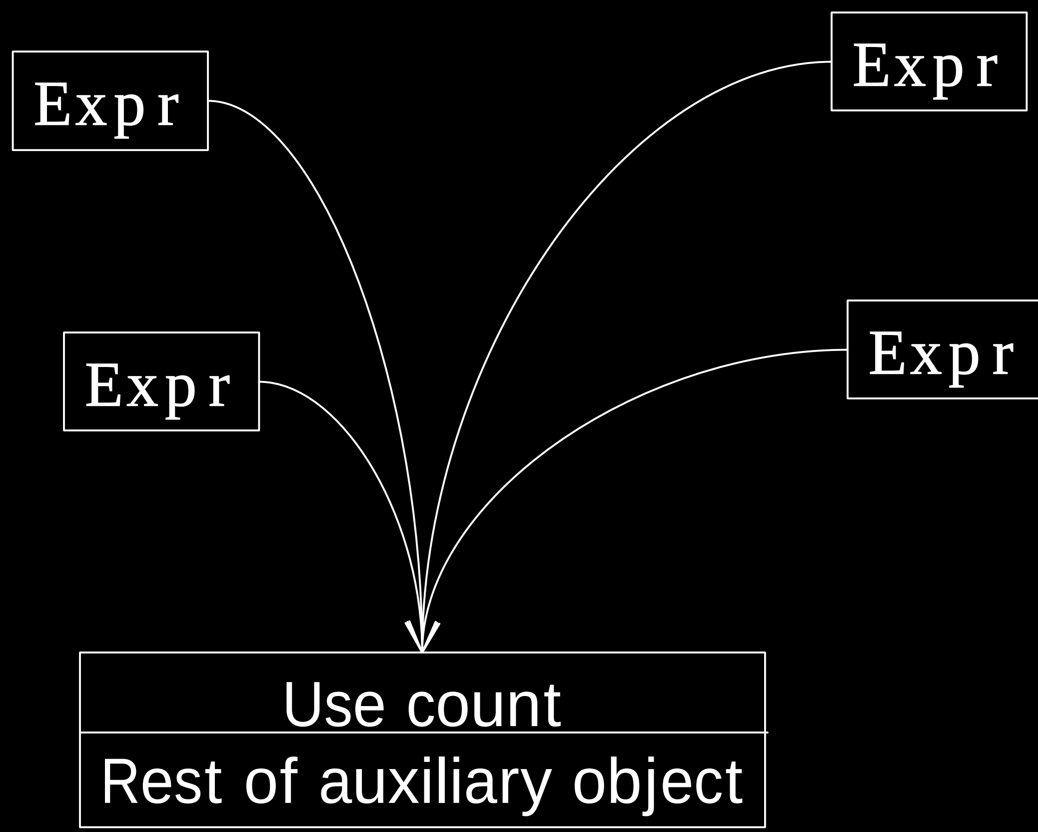
The auxiliary data structure

- Contains the real data associated with an `Expr`
- Does not need to be modified once created (because `print` and `eval` are 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(FILE*)
        const = 0;
    virtual int eval() const = 0;
    virtual ~ExprBase() { }
};
```

We can start to define `Expr`

- The `print` and `eval` functions just call the corresponding `ExprBase` virtuals

```
void Expr::print(FILE* f) const
{
    p->print(f);
}
int Expr::eval() const
{
    return p->eval();
}
```

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) { }
```