A note on `const`

Class objects and `const`

- What does it mean to change a class object?
  - First-order definition: An object changes if any of its members change
  - Better definition: The class author decides
- How can a class author say what it means to change an object?

Member functions can be `const`

```
class Foo {
public:
    Foo(int);
    void store(int);
    int fetch() const;
private:
    int n;
};
```

Declaration and definition must match

```
int Foo::fetch() const {
    return n;
}
void Foo::store(int k) {
    n = k;
}
```

The type of `this`

- Within a member function of a class `T`, `this` has type `T* const`.
- Within a `const` member function of a class `T`, `this` has type `const T* const`.

Type propagation

- Suppose we had written

```
void Foo::store(int k) const {
    n = k;   // will not compile
}
```

- The point is that `n` means `this->n`, so if `this` is a pointer to `const`, then `this->n` is effectively `const`

Changing members in `const` objects

- Data members can be declared `mutable`
  - Relatively new feature, hence not universally implemented
  - A `mutable` member is not `const`, even if it is a member of a `const` class object
- Such members are rarely necessary

Overloading on `const`

- It is possible to have two member functions with the same name and arguments that differ on whether the function is `const`:

```
class IntArray {
// …
    int& operator[](int);
    int operator[](int) const;
};
```

Handles and use counts (part 2)

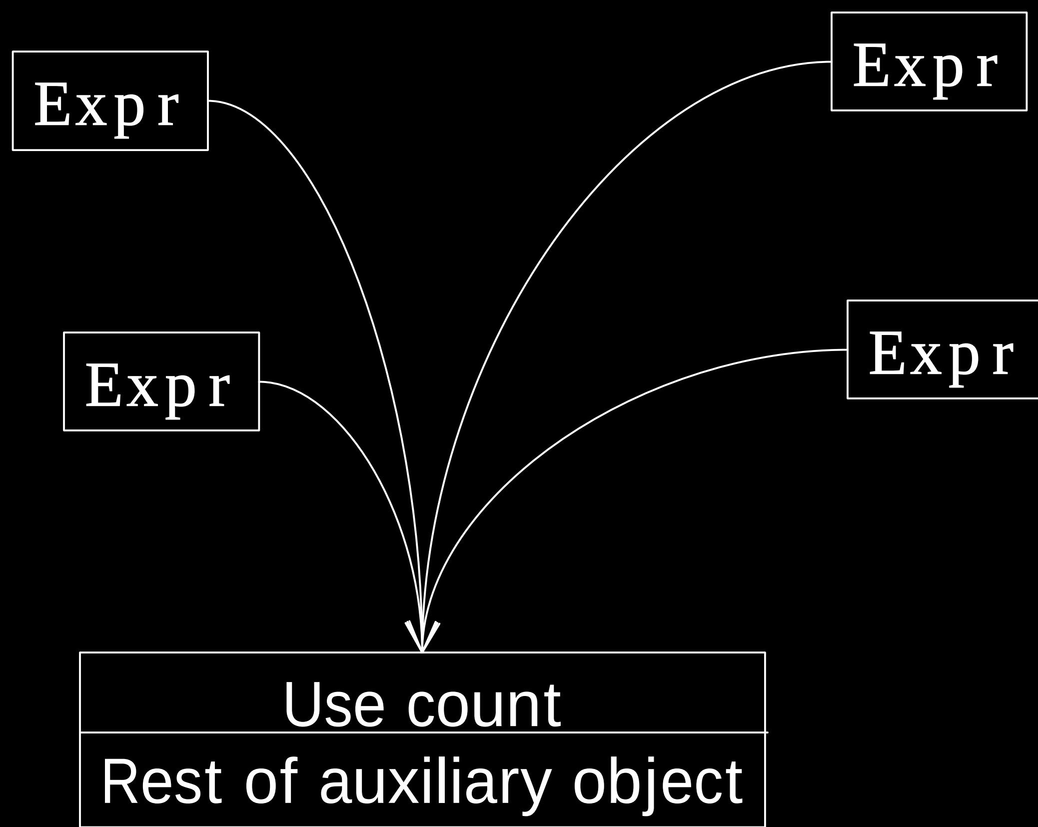
Review— class hierarchy

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

Review— where we are now

- We now have
  - one of the three `Expr` constructors
  - functions `Expr::eval` and `Expr::print`
- We need
  - the destructor
  - copy and assignment
  - the other two constructors
  - the rest of the auxiliary hierarchy

Review— the data structure



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;
}
```

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);
    virtual void print(FILE*);
    virtual int eval();
    int n;
};
```

IntExpr implementation

```
IntExpr::IntExpr(int n0): n(n0) { }
void IntExpr::print(FILE* f)
{
    fprintf(f, “%d”, n);
}
```

Class UnaryExpr definition

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

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(FILE* f)
{
    fprintf(f, “(%s”, op);
    e.print(f);
    fprintf(f, “)”);
}
```

BinaryExpr definition

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

BinaryExpr implementation

```
BinaryExpr::BinaryExpr(const char* x,
    const Expr& y, const Expr& z):
    op(x), e1(y), e2(z) { }
void BinaryExpr::print(FILE* f)
{
    fprintf(f, “(”);
    e1.print(f);
    fprintf(f, “%s”, op);
    e2.print(f);
    fprintf(f, “)”);
}
```

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