

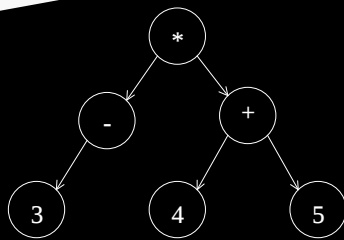
An object-oriented example

Expression trees as objects

The problem

- Find a useful way to represent expression trees
- Do something interesting with them
- Consolidate what we've learned so far
 - data abstraction
 - inheritance
 - dynamic binding

Example of an expression tree



Representation of $(-3) * (4+5)$

Why are trees useful?

- A tree represents the result of parsing an expression
 - Error checking has already been done
 - Recomputation not necessary
- Working with trees is easier
 - We get the notion of “subexpression” for free
 - Common operations are easy

Properties of trees

- Each tree represents an expression (or subexpression)
- Each node represents a token
- Parentheses are represented implicitly in the tree structure
- We will often equate a tree with a pointer to the root of the tree

Kinds of trees

- Integer nodes
- Operator nodes
 - unary
 - binary
- This classification already suggests an inheritance hierarchy

The Expr hierarchy

```
class Expr { /* ... */ };
class IntExpr: public Expr
{ /* ... */ };
class OpExpr: public Expr
{ /* ... */ };
class UnaryExpr: public OpExpr
{ /* ... */ };
class BinaryExpr: public OpExpr
{ /* ... */ };
```

How will we use Expr objects?

- Each object will represent a tree
- We will generally deal with these trees through pointers
- If *p* is a pointer to *Expr*, we would like to be able to call *p->print(stdout)* or obtain the result of *p->eval()*

Memory management

- C++ does not have built-in garbage collection (although commercial implementations are now available)
- We will see later how to manage memory automatically
- For now, we will manage memory manually

Expression tree memory strategy

- We will never copy nodes
- We will use *new* and *delete* to allocate and free nodes as usual
- Whenever we construct a new node from an old one, the new one will free the old one when it goes
 - Necessary but not completely satisfactory
 - Good enough for now

We can already declare class Expr

```
class Expr {
public:
    Expr() { }
    virtual void print(FILE*);
    virtual int eval();
    virtual ~Expr() { }

private:
    Expr(const Expr&);
    Expr& operator=(const Expr&);
};
```

Representing integers

- We want to be able to say

```
Expr* p = new IntExpr(42);
p->print(stdout); // print 42
int n = p->eval(); // n = 42
delete p;
```
- This tells us how to declare class *IntExpr*
 - constructor, destructor
 - virtual members *print* and *eval*

Declaration of IntExpr

```
class IntExpr: public Expr {
public:
    IntExpr(int);
    virtual void print(FILE*);
    virtual void eval();
    // No destructor needed

private:
    int n;
};
```

Implementation

- We will print an expression in fully parenthesized form
 - For an IntExpr, just print the integer
 - For a UnaryExpr, print (@e), where @ is the operator and e is the representation of the operand subexpression
 - For a BinaryExpr, print (e1@e2), where e1 and e2 represent the subexpressions

Implementation of IntExpr

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

Class OpExpr

- An OpExpr object represents an expression with an operator and some number of operands
- We will store the operator (as a char*) in the OpExpr
- Each derived class will worry about its own operands

Class OpExpr declaration

- We will not need print or eval because we will use OpExpr objects only as base classes for other objects

```
class OpExpr: public Expr {
public:
    OpExpr(const char*);

protected:
    const char* op;
};
```

OpExpr implementation

- This one is easy: We need to implement only the constructor, which just saves its argument

```
OpExpr::OpExpr(const char* s):
    op(s) { }
```

Classes UnaryExpr and BinaryExpr

- Each of these classes' constructors takes one or more pointers to objects for which the object will assume responsibility
- The destructor therefore needs to destroy the corresponding object(s)

UnaryExpr declaration

```
class UnaryExpr: public OpExpr {
public:
    UnaryExpr(const char*, Expr*);
    virtual ~UnaryExpr();
    virtual void print(FILE*);
    virtual int eval();

private:
    Expr* operand;
};
```

UnaryExpr implementation

```
UnaryExpr::UnaryExpr
(const char* oper, Expr* opnd):
    OpExpr(oper), operand(opnd) { }
UnaryExpr::~UnaryExpr()
{ delete operand; }
void UnaryExpr::print(FILE* f)
{
    fprintf(f, "(%s", oper);
    operand->print(f);
    fprintf(f, ")");
}
```

BinaryExpr declaration

```
class BinaryExpr: public OpExpr {
public:
    BinaryExpr
        (const char*, Expr*, Expr*);
    virtual ~BinaryExpr();
    virtual void print(FILE*);
    virtual int eval();

private:
    Expr* lhs;
    Expr* rhs;
};
```

BinaryExpr implementation

```
BinaryExpr::BinaryExpr
(const char* oper,
 Expr* l, Expr* r):
    OpExpr(oper), lhs(l), rhs(r) { }
BinaryExpr::~BinaryExpr()
{ delete lhs; delete rhs; }
void BinaryExpr::print(FILE* f) {
    fprintf(f, "("); lhs->print(f);
    fprintf(f, op); rhs->print(f);
    fprintf(f, ")");
}
```

Abstract base classes

- Class OpExpr was unusual
 - We did not implement print or eval
 - We do not expect ever to have a plain OpExpr object
- Such classes are called *abstract base classes*
- Looking back on it, class Expr should also be an abstract base class

Properties of abstract base classes

- Objects of that class do not exist by themselves, but only as bases of other classes
- They contain *pure virtual functions* that we want to implement only in derived classes
- C++ therefore uses pure virtual functions to define abstract bases

Declaring a pure virtual

- If you write, for example

```
class X {
public:
    virtual void foo() = 0;
};
```

then
 - Function X::foo is a pure virtual
 - Class X is an therefore abstract base class
 - Any class derived from X is also an abstract base class unless it defines foo

Rules for abstract base classes

- If a class X is an abstract base class, no objects of class X can exist. In these examples, assume Y is derived from X and is not abstract:

```
X x;           // Error: X is abstract
X* p;          // OK
p = new Y;     // Also OK
p = new X;     // Error: X is abstract
```

Class Expr revisited

```
class Expr {
public:
    virtual void print(FILE*) = 0;
    virtual int eval() = 0;
    virtual ~Expr() { }

private:
    Expr(const Expr&);
    Expr& operator=(const Expr&);
};
```

Class OpExpr revisited

- Class OpExpr inherited print and eval from Expr and did not override them
- Therefore, OpExpr also becomes an abstract base class automatically

Homework (due Monday)

- Make these expression trees work
- Write a main program to exercise them
- Define eval and make it work for binary +, -, *, /, and unary -

C++ puzzle (extra credit)

- Suppose you have an abstract class:

```
class X {  
public:  
    virtual void foo() = 0;  
};
```
- There is no need to define `X::foo`
 - No objects of class `X` can exist
 - Any call to `X::foo` will call `foo` in a derived class, where it will be defined

C++ puzzle, continued

- Although C++ is designed to make it hard to call a pure virtual function, there is a way to do it.
- The puzzle:
 - Find a way to call `X::foo` as defined
 - Try it and see what happens