Eric Roberts
CS 106B

Handout #44
May 13, 2009

Expression Trees

Expression Trees

Eric Roberts
CS 106B
May 13, 2009

Outline for Today

1. Review the structure of expressions
2. Review the idea of inheritance in general
3. Discuss how inheritance is represented in C++
4. Design the expression hierarchy
5. Implement `eval` for each of the subclasses
6. Explore the relationship between parsing and grammars
7. Look at the final version of `parser.cpp`
8. Play with the BASIC interpreter

Recursive Structure of Expressions

- An arithmetic expression in a programming language is a recursive structure that can be thought of as a tree.
- In the simple model I use in Chapter 14, every expression falls into one of the following forms:
  - An integer constant
  - A variable name that holds an integer value
  - Two expressions joined by an operator
  - An expression enclosed in parentheses
- This structure can be represented in the form of a ***grammar***.

E → *constant*
E → *identifier*
E → E *op* E
E → (E)

Expression Trees

- When the C++ compiler looks at an expression, it needs to understand what the expression means by translating it into an internal form. This process generally consists of two steps:
  - ***Lexical analysis***, in which the line is broken up into units
  - ***Parsing***, in which the tokens are assembled into a tree that embodies the expression structure

```
y = 3 * ( x + 1 )
```

Classes Hierarchies

- In any object-oriented language, classes serve as templates for individual objects. Each object is an ***instance*** of a particular class, which can serve as a pattern for many different objects.
- One of the defining characteristics of the object-oriented paradigm is that classes form ***hierarchies***. Any class can be designated as a ***subclass*** of some other class, which is called its ***superclass***. As noted on this week's section handout, most class hierarchies are tree-structured even though C++ permits more complicated structures.
- A class represents a specialization of its superclass. If you create an object that is an instance of a class, that object is also an instance of all other classes in the hierarchy above it in the superclass chain.
- When you define a new class in C++, that class automatically **inherits** the behavior of its superclass.

Biological Class Hierarchy

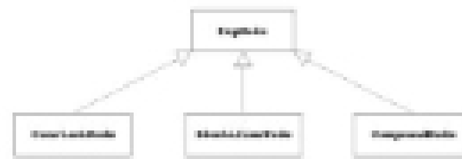
Living Things
Kingdom: Plants, Animals, Fungi

Classification of the red ant *Iridomyrmex purpureus*

Note that there can be many individual red ants, each of which is an **instance** of the same basic class.

Inheritance Hierarchies

- Because expressions have more than one form, a C++ class that represents expressions can be represented most easily by a class hierarchy in which each of the expression types is a separate subclass, as shown in the following diagram:



- Even though the class hierarchy is organized in terms of the different types of nodes, clients of the expression package will almost always work with pointers to nodes instead. As I did last time, I will therefore give the pointer type the name `expressionT`.

Representing Inheritance in C++

- The first step in creating a C++ subclass is to indicate the superclass on the header line, using the following syntax:

```
class subclass : public superclass {
    body of class definition
}
```

- In contrast to Java, a subclass cannot automatically override the definition of a method in its superclass. To permit such overriding, both classes must mark the prototype for that method with the keyword `virtual`.
- An ***abstract class*** is a class that doesn't actually represent any objects but instead serves only as a common superclass for ***concrete classes*** that do generate objects. In C++, methods for an abstract class that are always implemented by the concrete subclasses are indicated by including `= 0` before the semicolon on the prototype line.

The `exp.h` Interface

```
/*
 * File: exp.h
 * -----------
 * This interface defines a class hierarchy for expressions,
 * which allows the client to represent and manipulate simple
 * binary expression trees.
 */

#ifndef _exp_h
#define _exp_h

#include "genlib.h"
#include "map.h"

/*
 * Type: expressionT
 * -----------------
 * For clients, the most important type exported by this interface
 * is the expressionT type, which is defined as a pointer to an
 * expNode object.  This is the type used by all other functions
 * and methods in the expression package.
 */

class expNode;
typedef expNode *expressionT;
```

The `exp.h` Interface

```
/*
 * Class: evalState
 * ----------------
 * This class is passed by reference through the recursive levels
 * of the evaluator and contains information from the evaluation
 * environment that the evaluator may need to know.  The only
 * such information implemented here is a symbol table that maps
 * variable names into their values.
 */

class evalState {

public:
    evalState();
    ~evalState();
    void setValue(string var, int value);
    int getValue(string var);
    bool isDefined(string var);

private:
    Map<int> symbolTable;

};
```

The `exp.h` Interface

```
/*
 * Class: expNode
 * --------------
 * This class is used to represent a node in an expression tree.
 * expNode is an example of an abstract class, which defines the
 * structure and behavior of a set of classes but has no objects
 * of its own.  Any object must be one of the three concrete
 * subclasses of expNode:
 *
 *  1. ConstantNode   -- an integer constant
 *  2. IdentifierNode -- a string representing an identifier
 *  3. CompoundNode   -- two expressions combined by an operator
 *
 * The expNode class defines the interface common to all expNode
 * objects; each subclass provides its own specific implementation
 * of the common interface.
 *
 * Note on syntax: Each of the virtual methods in the expNode class
 * is marked with the designation = 0 on the prototype line.  This
 * notation is used in C++ to indicate that this method is purely
 * virtual and will always be supplied by the subclass.
 */
```

The `exp.h` Interface

```
/* Enumeration type for the three expression types */

enum expTypeT { ConstantType, IdentifierType, CompoundType };

/* [illegible] for expNode class */

class expNode {

public:
    expNode();
    virtual ~expNode();
    virtual expTypeT type() = 0;
    virtual int eval(evalState & state) = 0;
    virtual string toString() = 0;

};
```

The `exp.h` Interface

```
[illegible]
```

The `exp.h` Interface

```
[illegible]
```

The `exp.h` Interface

```
[illegible]
```

Code for the `eval` Method

```
[illegible]
```

A Two-Level Grammar

- The problem of parsing an expression can be simplified by changing the grammar to one that has two levels:
  - An ***expression*** is either a ***term*** or two expressions joined by an operator.
  - A ***term*** is either a constant, an identifier, or an expression enclosed in parentheses.
- This design is reflected in the following revised grammar.

$$E \rightarrow T$$
$$E \rightarrow E \ \textit{op} \ E$$
$$T \rightarrow \textit{constant}$$
$$T \rightarrow \textit{identifier}$$
$$T \rightarrow (\ E\ )$$

The `parser.cpp` Implementation

```
[illegible]
```