Programming Languages: An Active Learning Approach

Chapter 3
Object-Oriented Programming with C++

Kent D. Lee

Object-Oriented Programming with C++

- This chapter introduces object-oriented programming using C++ through the development of an interpreter for a calculator language.
- By organizing code and data into objects using *classes*, code reuse is emphasized. Ideas such as *inheritance* enable *polymorphism* among related objects.
- C++ is one of the most widely used object-oriented languages today.
- The main problem with C/C++ programs are memory leaks.
- Modern languages like Java and Ruby provide garbage collection as part of the underlying model of computation.
- Garbage collection does have its own problems.
- Languages like Java and Ruby aren’t as suited to real-time applications where timing is critical.

- The interpreter developed in this chapter is not a real-time application.
- C++ is chosen because there are many interesting aspects to C++ that can be explored by using it.

Design the Calculator

- Let us call the calculator language interpreter *calc*.
- The calculator will have a single memory location in which an integer value can be stored and recalled.
- It will do integer division only which means the fractional part is discarded as a remainder after doing division.

Design the Calculator

Example 3.1

Here is a typical session with the completed calc interpreter.

```
%>calc
Please enter a calculator expression: (4S+5)*(7-R)
The result is 27
%>
```

Design the Calculator

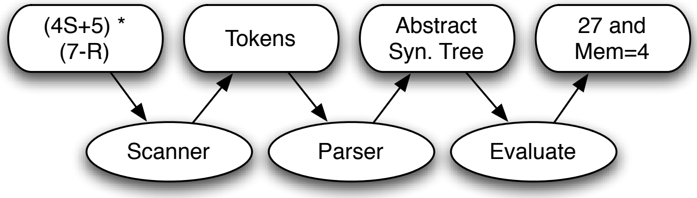
☞ Practice 3.1

Evaluate the following calculator expressions.

1. (4+5)S*R
2. 3S + R
3. R + 3S
4. 2S*4 + R

▸ View Solution

Data flow through the Calc Interpreter



Separate Compilation

- Regardless of which C++ compiler you use, like many modern languages, C++ is organized so that programs may be separately compiled.
- Each piece of a C++ project is stored in a separate file.
- The calc interpreter contains six different modules.
- Each module can be compiled separately.

```
g++ -g -c module.C
```

- The *-g* tells the compiler to include debug information.
- The *-c* option tells the compiler to produce an object file.
- When each module has been compiled, then they may be linked together to produce an executable program.

```
g++ -o executable_program module1.o module2.o module3.o ...
```

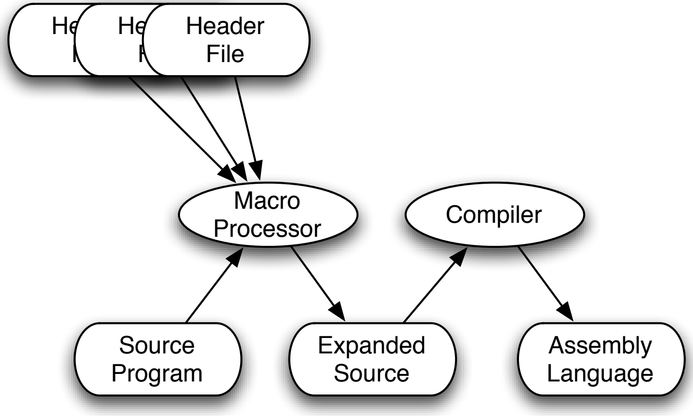
Separate Compilation

Example 3.2

To completely compile the calc project the following compile commands must be issued.

```
g++ -g -c calc.C
g++ -g -c scanner.C
g++ -g -c token.C
g++ -g -c ast.C
g++ -g -c parser.C
g++ -g -c calculator.C
g++ -g -o calc calc.o scanner.o token.o ast.o parser.o \
calculator.o
```

The Macro Processor



Header Files

- A header file in C++ is where declarations go that are to be shared between modules.
- When a program is compiled, the macro processor runs first to expand any macro processor directives in the source code as pictured in figure 1.
- The macro processor looks at all the preprocessor directives and builds the expanded source program that is actually given to the compiler.
- Every macro processor directive starts with a pound sign (i.e. #).

Header Files

Example 3.3

Including a header file in a module is as simple as writing

```
#include "header_file"
```

By convention, header files are named with a .h extension. So there is a header file called scanner.h, token.h, ast.h, parser.h, calculator.h, and a few others in the calculator project containing the declarations of the classes that are a part of the *calc* interpreter.
The parser.C and scanner.C modules include the token.h header file by writing

```
#include "token.h"
```

Header Files

Example 3.4

To solve the problem of circular includes or repeated includes the convention is adopted to start include files with an *#ifndef* and then on the next line define the identifier. So the token.h file begins with

```
#ifndef token_h
#define token_h

// Any declarations go here.

#endif
```

The Make Utility

- Executing all the *g++* commands given in example 3.2 would be very tedious if you had to do it more than once.
- Fortunately, there is a utility called *make* that will take care of our housekeeping for us.
- The make utility takes a file, usually called *Makefile*, as input that specifies the dependencies of a project.

The Make Utility

Example 3.5

Here is the Makefile for the calc project.

```
calc: calc.o scanner.o token.o ast.o parser.o calculator.o
g++ -g -o calc calc.o scanner.o token.o \
        ast.o parser.o calculator.o
calc.o: calc.C scanner.h token.h
g++ -g -c calc.C
calculator.o: calculator.C calculator.h parser.h ast.h
g++ -g -c calculator.C
scanner.o: scanner.C scanner.h token.h
g++ -g -c scanner.C
token.o: token.C token.h
g++ -g -c token.C
ast.o: ast.C ast.h
g++ -g -c ast.C
parser.o: parser.C parser.h
g++ -g -c parser.C
clean:
rm -f *.o
rm -f calc
```

The Make Utility

Example 3.6

To use the make utility you create a file like the one given in example 3.5 and execute the make command

```
make
```

from the directory that contains the Makefile.

The Make Utility

☞ Practice 3.2

Assume that you issued the make command to bring everything up to date. Then you change the ast.h header file. Which compile commands will be executed given the Makefile in example 3.5?

▸ View Solution

The Make Utility

Example 3.7

Good Makefiles are hard to write. They almost always have errors in them and the one in example 3.5 is not completely correct. To deal with this there are tools that will generate make files for you. When all else fails you can add an extra clean rule to start over. If something isn't working you can start over by typing

```
make clean
make
```

The Token Class

- The calculator language has several types of tokens.
- The complete list is *number*,+,-,*,/,(,),*S*,and *R*.
- To begin, the type of token can be desribed using something called an enum in C++.

The Token Class

Example 3.8

Here is the enum for the types of tokens in the intepreter's language.

```
enum TokenType {
   identifier,keyword,number,add,sub,times,divide,
   lparen, rparen,eof,unrecognized
};
```

The Token Class

☞ Practice 3.3

Identify the tokens in these expressions. Refer to the *enum* above to be sure you find them all.

1. 3S + R
2. (4+5)S*R

▸ View Solution

The Token Class

Example 3.9

This is the declaration of the Token class.

```
class Token {
public:
  Token();
  Token(TokenType typ, int line, int col);
  virtual ~Token();
  TokenType getType() const;
  int getLine() const;
  int getCol() const;

private:
  TokenType type;
  int line,col;
};
```

The Token Class

1. The class name on line 1.
2. The keyword public identifies a section that contains all public methods and instance variables. Typically instance variables are not public.
3. Line 3 and 4 are the declarations of the Token constructors. Notice that no code appears in the class declaration.
4. Line 5 is the Token destructor. This is discussed in more detail below.
5. Lines 7-9 declare accessor methods. The getLex() accessor method is declared constant (i.e. *const*) and virtual. We'll discover what those keywords mean soon.
6. The private section is where the instance variables are declared and anything else that should be hidden from users of the class.

The Token Class

Example 3.10

This is the declaration of the LexicalToken class.

```
class LexicalToken: public Token {
 public:
   LexicalToken(TokenType typ, string* lex,
                int line, int col);
   ~LexicalToken();
   virtual string getLex() const;
 private:
   string* lexeme;
};
```

Implementing a Class

Example 3.11

Here is the implementation of the Token and LexicalToken classes contained in the token.C module.

```
#include "token.h"

Token::Token() :
  type(eof),  line(0),  col(0)
{}

Token::Token(TokenType typ, int lineNum, int colNum) :
  type(typ),
  line(lineNum),
  col(colNum)
{}

Token::~Token() {}

TokenType Token::getType() const { return type; }

int Token::getLine() const { return line; }

int Token::getCol() const { return col; }

LexicalToken::LexicalToken(TokenType typ, string* lex,
                           int lineNum, int colNum) :
   Token(typ,lineNum,colNum),
   lexeme(lex)
{}

LexicalToken::~LexicalToken() {
   try {
      delete lexeme;
   } catch (...) {}
}

string LexicalToken::getLex() const {
  return *lexeme;
}
```

Constructors and Initialization Lists

- Lines 3-11 contain the constructor implementations for Token.
- Between the colon and the left brace (i.e. {) appears an initialization list.
- The list initializes instance variables to values.

Constructors and Initialization Lists

Example 3.12

Another way to write the constructor.

```
Token::Token() {
  type = eof;
  line = 0;
  col = 0;
}
```

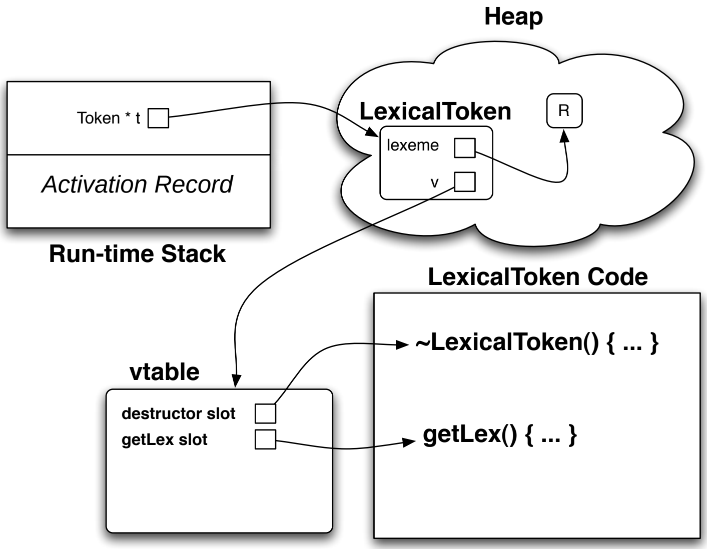
Destructors

- The declaration

  ```
  string* lexeme;
  ```

- declares a string pointer called lexeme.
- When an object like a LexicalToken stores a pointer to another object, in this case a string, it is likely that the referenced object (the string) is stored on the heap.
- When the first object is deleted it must be sure to delete the second from the heap.
- Right before a LexicalToken is deleted the destructor will be called to delete the storage it refers to on the heap.

Implementation of Polymorphism



Inheritance and Polymorphism

1. An object like a LexicalToken is created in some code. In this example it's created in the scanner.
2. When the object is created, it's created as a LexicalToken. The type is known by the compiler at this point. The object is initialized by calling the LexicalToken constructor.
3. The compiler looks at the class description for LexicalToken and sees two virtual functions. It knows that virtual functions require a vtable to be created. The name vtable stands for virtual function table. At a given offset in every object there is a pointer to a vtable. Some objects may not have a vtable if they don't have any virtual functions, but the vtable pointer is still there.

4. The compiler finds the vtable associated with the class of the object being created. There is only one vtable for a class and the compiler knows where each vtable is stored. Code is generated to store the pointer to the vtable in the object.
5. When a virtual method like getLex or the destructor is called the compiler looks at the class and sees that the method is a virtual method. It knows that it must look up the code in the vtable.
6. The compiler generates code to look in the vtable at a specified offset depending on the virtual method being called. The destructor is always in the first slot, the getLex is always in the second slot regardless of whether it is a Token object, LexicalToken object, or some new object class that inherits from either. New entries can be added to the end of a vtable for new virtual functions in derived classes, but existing entries can NEVER be altered. The

compiler must be able to count on the vtable structure remaining the same.

7. The vtable entry chosen contains a pointer to the correct code to be called. At run-time the program jumps to the correct method and polymorphism has just occurred.

Inheritance and Polymorphism

- A constructor can't be polymorphic, nor would we want it to be. When we construct an object we always specify the type of object we are creating. Polymorphism doesn't make sense for constructors.
- Once a method is declared polymorphic (i.e. virtual) it may NEVER become non-polymorphic again. Said another way, once it's in the vtable it stays in the vtable. That's why the destructor and getLex() method in LexicalToken aren't declared virtual, yet they still are. Once virtual, always virtual.

- It is permissable for a function to be declared non-virtual in a base class and then to become virtual in a sub-class since the function can be added to the vtable in a subclass without any problems. However, this is a bad idea in general. In the base class the function would not be polymorphic. You can try this out. Get the cppcalc code and change the definition of token.h so that Token's getLex method is not virtual (delete the keyword virtual) but leave the LexicalToken getLex method as virtual. It should compile but you won't get the right result. This is because when getLex is called on a Token pointer the non-polymorphic `Token::getLex()` method will be called and not the polymorphically correct getLex. Be sure to *make clean* and then *make* between trial runs to get everything to recompile.

- Polymorphic method calls are less efficient than normal function calls. There is some overhead involved in making the function call. It amounts to two extra load instructions in most CPUs to complete the function call. Some hardware may be optimized to reduce the overhead of these loads. However, for most applications this is a small price to pay for code reuse.

A Historical Look at Parameter Passing

Example 3.13

Here is an example of a Pascal function with both value and variable parameters.

```
procedure lookup_term(name:ident_type;var found:boolean;
                                      var place:integer)

var k:integer;

begin
found:=false;
for k:=1 to non_term_index do
  if terminal[k].ident = name then
     begin
     found:=true;
     place:=k;
     end;
end; (*PROCEDURE*)
```

In the code above the `name` parameter is a value parameter. That means a copy of the value is passed to the procedure `lookup_term`. By passing a copy of the value, the caller of the procedure can be assured that any variable passed as the first parameter will not be modified since it is a value parameter. To call the procedure you might write something like:

```
lookup_term(aName, found, aPlace);
```

A Historical Look at Parameter Passing

Example 3.14

Consider the following program:

```c
#include <stdio.h>

struct Point {
  int x;
  int y;
};

void testit(struct Point p) {
  p.x = 0;
  p.y = 0;
}

int main(int argc, char* argv[]) {
  struct Point myPoint;
  myPoint.x = 10;
  myPoint.y = 10;
  testit(myPoint);
  printf("x = %d, y = %d\n",myPoint.x,myPoint.y);
  return 0;
```

```
    }
```

A Historical Look at Parameter Passing

- When compiled and run the program creates a structure called `myPoint` of type `Point`.
- The variable `myPoint` is passed by value and the copy of it is modified in the function called `testit`.
- The program prints `10,10` as the output.
- If a C programmer wishes to pass an argument by reference, he or she must pass a pointer to the original space.

A Historical Look at Parameter Passing

☞ Practice 3.4

Considering what you learned about the run-time stack and calling functions in the first chapter, describe in detail what happens in the following program with regards to the run-time stack and the variables within the program.

```c
#include <stdio.h>

struct Point {
  int x;
  int y;
};

struct Point makePoint() {
  Point aPoint;
  aPoint.x = 0;
  aPoint.y = 0;
  return aPoint;
}

```

```c
void testit(struct Point p) {
  p.x = 0;
  p.y = 0;
}

int main(int argc, char* argv[]) {
  struct Point myPoint = makePoint();
  myPoint.x = 10;
  myPoint.y = 10;
  testit(myPoint);
  printf("x = %d, y = %d\n",myPoint.x,myPoint.y);
  return 0;
}
```

View Solution

A Historical Look at Parameter Passing

Example 3.15

To pass the myPoint by reference in C the following code would need to be written.

```c
#include <stdio.h>

struct Point {
  int x;
  int y;
};

void testit(struct Point* p) {
  p->x = 0;
  p->y = 0;
}

int main(int argc, char* argv[]) {
  struct Point myPoint;
  myPoint.x = 10;
  myPoint.y = 10;
  testit(&myPoint);
  printf("x = %d, y = %d\n",myPoint.x,myPoint.y);
  return 0;
}
```

A Historical Look at Parameter Passing

- In the version of the code in example 3.15 the variable `p` is a pointer to a Point structure.
- First, in the `testit` function the code that was `p.x = 0` is now `p->x = 0`.
- The second and more important difference is that the caller of the `testit` function must now realize that the parameter is a pointer and must pass the address of the Point to the function.
- The caller of the function must also remember to call it in the right way.

A Historical Look at Parameter Passing

☞ Practice 3.5

What could go wrong in the following version of the code? Carefully trace the code by hand to see what the mistake is here.

```
#include <stdio.h>

struct Point {
  int x;
  int y;
};

struct Point* makePoint() {
  Point aPoint;
  aPoint.x = 0;
  aPoint.y = 0;
  return &aPoint;
}

void testit(struct Point* p) {
  p->x = 0;
  p->y = 0;
}

int main(int argc, char* argv[]) {
  struct Point* myPoint = makePoint();
  myPoint->x = 10;
  myPoint->y = 10;
  testit(myPoint);
  printf("x = %d, y = %d\n",myPoint->x,myPoint->y);
  return 0;
}
```

▸ View Solution

A Historical Look at Parameter Passing

Example 3.16

The code in this example passes the Point data by reference.

```
#include <stdio.h>

class Point {
public:
  int x;
  int y;
};

void testit(Point& p) {
  p.x = 0;
  p.y = 0;
}

int main(int argc, char* argv[]) {
  Point myPoint;
  myPoint.x = 10;
  myPoint.y = 10;
  testit(myPoint);
  printf("x = %d, y = %d\n",myPoint.x,myPoint.y);
```

```
    return 0;
}
```

A Historical Look at Parameter Passing

- First, the struct Point is replaced with a class Point since C++ supports classes.
- Then, in this example, the ampersand moves from the caller of the function to the function definition.
- When a formal parameter of a function in C++ has an ampersand after the type it means the parameter is passed by reference.
- In contrast to C++, the Java programming language passes built-in types like `int`, `double`, `float`, and `char` by value.
- All objects in Java are passed by reference. Java is less flexible in its parameter passing, but this simplifies many aspects of the language.

- Because objects can and regularly do have pointers to other objects within them, making a copy of an object when it is passed by value is too complicated for C++ to do by itself.
- A copy constructor makes a copy of the object in the way dictated by the programmer.
- Copy constructors are not discussed here.

Const in C++

- The keyword *const* in C++ can be used in a variety of situations. Perhaps the simplest situation is where you want to declare a constant. You can, for instance, write

```
const int maxVal = 100;
```

- to declare a constant value `maxVal`.
- Perhaps the most important use of *const* is to solve the problem of pass by value copying large objects in C++.
- The problem is that objects, when passed by value, are copied.
- In C++, it is possible to declare that a parameter is not modified by a method by either declaring the parameter is passed by value or by declaring the parameter is passed by constant reference.

Const in C++

Example 3.17

In this code, the Point example has been rewritten to call a method called `printPoint` to print the point data. However, the `printPoint` method will not modify the point so the parameter is declared to be a constant reference to a point.

```
#include <stdio.h>

class Point {
public:
  Point(int x, int y);
  int getX() const;
  int getY() const;
  void setX(int x);
  void setY(int y);

private:
  int x;
  int y;

```

```cpp
};

Point::Point(int x, int y) {
  this->x = x;
  this->y = y;
}

int Point::getX() const {
  return x;
}

int Point::getY() const {
  return y;
}

void Point::setX(int x) {
  this->x = x;
}

void Point::setY(int y) {
  this->y = y;
}

void testit(Point& p) {
  p.setX(0);
  p.setY(0);
```

```
}

void printPoint(const Point & p) {
  printf("Point(%d,%d)\n",p.getX(),p.getY());
}

int main(int argc, char* argv[]) {
  Point myPoint(10,10);
  testit(myPoint);
  printPoint(myPoint);
  return 0;
}
```

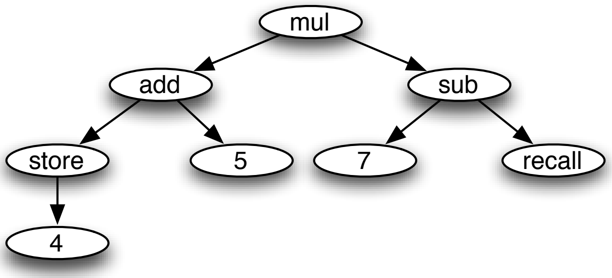
Const in C++

- In the example above you may notice that the methods `getX()` and `getY()` of the Point class are declared as const methods.
- This is an example of how *constness* may creep into your program.
- When the `printPoint` method was declared to take a constant reference to a Point object, it meant that `printPoint` could not call any methods on the Point class that could potentially modify the Point object.
- That meant that the `getX()` and `getY()` methods now need to be declared to be const themselves.

The AST Classes

- According to the diagram in figure 4 the parser must build an abstract syntax tree of the expression to be evaluated.
- The AST classes can be designed as a hierarchy of classes where each type of AST class represents one type of node in an abstract syntax tree.

Abstract Syntax Tree of expression in figure 4



The AST Classes

- Consider the abstract syntax tree in figure 2.
- A postfix traversal of the tree would yield the value.
  1. The traversal begins by recursively descending the left side of the tree down to the 4 node. Visiting that node returns 4.
  2. The store node takes the 4 and stores it in the calculator's memory. It also returns the 4.
  3. The add can't be visited yet since it has a right child (the 5). The 5 node is visited and returns the 5.
  4. The add node can now be visited. It takes the 4 and the 5, adds them together and returns the 9.
  5. The mul node can't be visited until its right child is visited. Postorder traversal of the sub node calls the traversal on the 7 node, which returns 7.
  6. The sub node still can't be visited yet. The recall node is traversed and returns the value in the calculator memory, the 4.
  7. The 7-4 is computed by visiting the sub node and returns 3.
  8. Visiting the mul node computes 9*3 or 27.

The AST Classes

Example 3.18

Here is the AST header file containing three AST class declarations.

```
#ifndef ast_h
#define ast_h

using namespace std;

class AST {
 public:
   AST();
   virtual ~AST() = 0;
   virtual int evaluate() = 0;
};

class BinaryNode : public AST {
 public:
   BinaryNode(AST* left, AST* right);
   ~BinaryNode();
   AST* getLeftSubTree() const;
```

```
    AST* getRightSubTree() const;
  private:
    AST* leftTree;
    AST* rightTree;
};

class UnaryNode : public AST {
  public:
    UnaryNode(AST* sub);
    ~UnaryNode();
    AST* getSubTree() const;
  private:
    AST* subTree;
};

#endif
```

The AST Classes

Example 3.19

The implementation of three AST classes is provided here.

```
#include "ast.h"
#include <iostream>
#include "calculator.h"

//uncomment the next line to see the destructor calls
//#define debug

AST::AST() {}

AST::~AST() {}

BinaryNode::BinaryNode(AST* left, AST* right):
   AST(),   leftTree(left),   rightTree(right)
{}

BinaryNode::~BinaryNode() {
#ifdef debug
   cout << "In BinaryNode destructor" << endl;
#endif
```

```
    try {
       delete leftTree;
    } catch (...) {}
    try {
       delete rightTree;
    } catch(...) {}
}

AST* BinaryNode::getLeftSubTree() const {
  return leftTree;
}

AST* BinaryNode::getRightSubTree() const {
  return rightTree;
}

UnaryNode::UnaryNode(AST* sub):
   AST(),   subTree(sub)
{}

UnaryNode::~UnaryNode() {
#ifdef debug
   cout << "In UnaryNode destructor" << endl;
#endif
    try {
       delete subTree;
```

```
        } catch (...) {}
    }
```

The AST Classes

☞ Practice 3.6

Write the declaration of the AddNode class.

▸ View Solution

The AST Classes

☞ Practice 3.7

Write the implementation of the AddNode class. When writing the evaluate method for the AddNode class don’t worry about how you get the right values. Just assume that those values are available if you call evaluate on the right object or objects. How it’s done is unimportant in this exercise.

▸ View Solution

The Scanner

- Referring back to figure 4 the scanner reads characters from the input and builds Token objects that are used by the parser.
- The parser will get tokens from the scanner by calling a getToken method.
- Sometimes the parser gets a token and needs to put it back.
- In that case a putBackToken method will put back the last token that was returned by getToken.

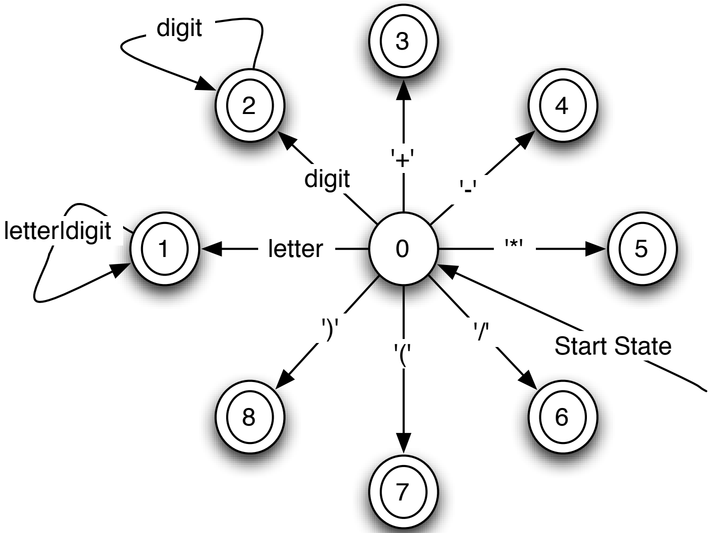
The Scanner

Example 3.20

Here is the declaration of the scanner in the header file scanner.h.

```
#ifndef scanner_h
#define scanner_h

#include <iostream>
#include "token.h"

class Scanner {
public:
  Scanner(istream* in);
  ~Scanner();

  Token* getToken();
  void putBackToken();

private:
  Scanner();

  istream* inStream;
  int lineCount;
  int colCount;

  bool needToken;
  Token* lastToken;
};

#endif
```

A Finite State Machine for the Scanner



The Scanner

- Internally, the scanner object is a finite state machine.
- A finite state machine (fsm) consists of a set of states and a set of transitions from one state to another based on the current character in the input.
- The fsm starts in state zero, reads one character and transitions to one of the eight states depending on the character.
- An fsm is a model of computation for recognizing strings of characters.
- Fsm's are used in many contexts including network protocol implementations, pattern recognition, simulations, and of course language implementation.

The Parser

- Figure 4 shows the parser reading tokens and producing an abstract syntax tree as its output.
- The parser that is discussed in this section is a top-down parser.

The Parser

Example 3.21

This is the Calculator language's grammar.

$$\begin{aligned}
&\mathit{Prog} \rightarrow \mathit{Expr}\ \ \mathit{EOF}\\
&\mathit{Expr} \rightarrow \mathit{Expr} + \mathit{Term} \mid \mathit{Expr} - \mathit{Term} \mid \mathit{Term}\\
&\mathit{Term} \rightarrow \mathit{Term} * \mathit{Storable} \mid \mathit{Term} / \mathit{Storable} \mid \mathit{Storable}\\
&\mathit{Storable} \rightarrow \mathit{Factor}\ \ S \mid \mathit{Factor}\\
&\mathit{Factor} \rightarrow \mathit{number} \mid R \mid (\mathit{Expr})
\end{aligned}$$

The Parser

- The implementation of the parser is given to us by its grammar.
- In the implementation, each nonterminal becomes a function in the parser.
- Each rule in the grammar is part of a function that is named by the nonterminal on the left side of the arrow in the rule.
- In the grammar above each line would correspond to a function in the parser.
- Each appearance of a nonterminal on the right hand side of a production is a function call.
- Each appearance of a token on the right hand side of a production is a call to the scanner to get a token.

A First Attempt at Writing the Parser

Example 3.22

The Prog and Expr functions for the Parser

```
AST* Parser::Prog() {
   AST* result = Expr();
   Token* t = scan->getToken();

   if (t->getType() != eof) {
      cout << "Syntax Error: Expected EOF, found token "
           << " at column " << t->getCol() << endl;
      throw ParseError;
   }

   return result;
}

AST* Parser::Expr() {
   AST* e = Expr();
   Token* t = scan->getToken();
   ...
}
```

A Better Attempt at Writing a Top-Down Parser

1. Eliminate left recursion.
2. Perform left factorization where appropriate.

A Better Attempt at Writing a Top-Down Parser

Example 3.23

An LL(1) Calculator Language Grammar

$Prog \rightarrow Expr\ EOF$

$Expr \rightarrow Term\ RestExpr$

$RestExpr \rightarrow +\ Term\ RestExpr \mid -\ Term\ RestExpr \mid <null>$

$Term \rightarrow Storable\ \ RestTerm$

$RestTerm \rightarrow *\ Storable\ RestTerm \mid /\ Storable\ RestTerm \mid <null>$

$Storable \rightarrow Factor\ S \mid Factor$

$Factor \rightarrow number \mid R \mid (Expr)$

Translating the LL(1) Grammar to C++

1. Construct a function for each nonterminal. Each of these functions should return a node in the abstract syntax tree.
2. Depending on your grammar, some nonterminal functions may require an input parameter of an abstract syntax tree (ast) to be able to complete a partial ast that is recognized by the nonterminal function.
3. Each nonterminal function should call getToken on the scanner to get the next token as needed. If after getting the token, the code determines it didn't need the token after all, the nonterminal function should call the scanner's putBackToken function to put the token back. If the parser is based on an LL(1) grammar, it should never have to put back more than one token at a time.

4. The body of each nonterminal function is a series of if statements that choose which production to expand upon depending on the value of the next token. The body of the function is determined by the productions of the grammar with the given nonterminal on the left hand side of the arrow.

Translating the LL(1) Grammar to C++

Example 3.24

This is the Parser’s header file, “parser.h".

```
#ifndef parser_h
#define parser_h

#include "ast.h"
#include "scanner.h"

class Parser {
 public:
   Parser(istream* in);
   ~Parser();

   AST* parse();

 private:
   AST* Prog();
   AST* Expr();
   AST* RestExpr(AST* e);
   AST* Term();
   AST* RestTerm(AST* t);
```

```
    AST* Storable();
    AST* Factor();

    Scanner* scan;
};
#endif
```

Translating the LL(1) Grammar to C++

Example 3.25

The Parser's Prog and Expr Functions

```
AST* Parser::Prog() {
   AST* result = Expr();
   Token* t = scan->getToken();

   if (t->getType() != eof) {
      cout << "Syntax Error: Expected EOF, found token "
           << "at column " << t->getCol() << endl;
      throw ParseError;
   }

   return result;
}

AST* Parser::Expr() {
   return RestExpr(Term());
}
```

Translating the LL(1) Grammar to C++

Practice 3.8

The RestExpr function is slightly different from the Prog and Expr functions. The RestExpr function has an AST parameter which we'll call `e`. The RestExpr function first gets a token and then decides what to do based on that token. If it is an *add* token it builds a new AST AddNode with the part of the tree given to it (i.e. *e*) as the left subtree and the result of calling Term as the right subtree. The subtract AST nodes are handled similarly. Otherwise, there wasn't a token that the RestExpr knows about so the token is put back and the AST *e* is returned as its AST.

Write the RestExpr function described here. Remember you can refer to the grammar in example 3.23.

View Solution

Putting It All Together

- One more class is required to tie together the pieces that have been developed in this chapter.
- The Calculator class contains a memory location that can hold a stored value.
- The value can also be retrieved on demand.
- The calculator can evaluate an expression that is given to it as a string.

Putting It All Together

Example 3.26

The Calculator’s header file, “calculator.h"

```
#ifndef calculator_h
#define calculator_h

#include <string>

using namespace std;

class Calculator {
 public:
   Calculator();

   int eval(string expr);
   void store(int val);
   int recall();

 private:
   int memory;
};

```

```
extern Calculator* calc;

#endif
```

Putting It All Together

Example 3.27

The Calculator class implementation

```
#include "calculator.h"
#include "parser.h"
#include "ast.h"
#include <string>
#include <iostream>
#include <sstream>

Calculator::Calculator():
   memory(0)
{}

int Calculator::eval(string expr) {

   Parser* parser = new Parser(new istringstream(expr));

   AST* tree = parser->parse();

   int result = tree->evaluate();

```

```
    delete tree;
    delete parser;

    return result;
}

void Calculator::store(int val) {
   memory = val;
}

int Calculator::recall() {
    return memory;
}
```

Putting It All Together

Example 3.28

The main function from “calc.C"

```
#include <iostream>
#include <sstream>
#include <string>
#include "calcex.h"
#include "calculator.h"
using namespace std;

Calculator* calc;

int main(int argc, char* argv[]) {
   string line;

   try {

      cout << "Please enter a calculator expression: ";

      getline(cin, line);

      calc = new Calculator();
```

```

        int result = calc->eval(line);

        cout << "The result is " << result << endl;

        delete calc;

    }
    catch(Exception ex) {
        cout << "Program Aborted due to exception!" << endl;
    }
}
```

Exercises

1. What's the value of (R+7)/4S if the memory contained 4 prior to evaluating this expression?
2. What is the value of the memory location after evaluating the previous expression?
3. What does the abstract syntax tree look like for the expression (R+7)/4S?
4. How could the calculator language be modified to allow more than one memory location like modern calculators? Discuss what changes would be required to implement this enhanced calculator language.

5. Complete the calculator interpreter by downloading the code given in this chapter and finishing the implementation in parser.C, ast.C, and ast.h. The rest of the project is provided.
   When you download the code you will want to unzip the package with some sort of unzip program. On Linux you can issue the command,

```
unzip cppcalc.zip
```

   Then you can make the program and run it. Here is an example of making and running you can use to get started.

```
$ unzip cppcalc.zip
$ cd cppcalc
$ make
make: ‘calc’ is up to date.
$ make clean
rm -f *.o
rm -f calc
$ make
g++ -g -c calc.C
g++ -g -c scanner.C
```

```
g++ -g -c token.C
g++ -g -c ast.C
g++ -g -c parser.C
g++ -g -c calculator.C
g++ -g -o calc calc.o scanner.o token.o ast.o parser.o \
    calculator.o
$ calc
Please enter a calculator expression: 5 + 4
The result is 9
$
```

Commands that you enter are preceded by a dollar sign. The *make clean* above tells make to use the *clean* rule to erase all compiled files and make the project from scratch. See the file *Makefile* for the clean rule or look in the chapter at the section on the make utility.

The program will compile and add two integers together as provided. Your job is to extend the project to the full calculator language. This requires changes to the parser.C and ast.C modules. The parser.C changes are highlighted in section 18. You can complete the parser by completing the functions that are incomplete in the parser.C file. These functions can be patterned after the code presented in the chapter.

The parser code will require that you build AST nodes for storing values and for recalling values from the calculator's memory. You will also need multiply and divide nodes in the abstract syntax tree. These new node types can be added to the ast.C and ast.h files using the existing code as a pattern.

The store and recall nodes in the AST will need to access the memory location of the calculator. The global variable called *calc* can be used to access the calculator's memory. The line

```
calc->store(6)
```

will store 6 in the calculator's memory. Similarly, the expression *calc−>recall()* will retrieve the value stored in the memory of the calculator.

6. Once you have completed the project described above extend the calculator language to allow more than one memory location to hold a value.
7. Modify the project to be a compiler instead of an interpreter. Instead of evaluating the expression, generate EWE code for it instead.

   In addition, to make this interesting, add a new keyword to the language, called I, that when executed waits for user input before proceeding. The value returned by the call to I is the value entered at the keyboard.

   This project can be implemented with a few modifications. First, the evaluate function of the abstract syntax tree will print code to a file called "a.ewe" instead of directly evaluating the expression. To print to a file in C++ you create an ofstream object. The changes can be made in several places but it will work if you create the ofstream in the main function and pass it to the constructor of the Calculator object. The object can be passed as an ostream which ofstream inherits from. Pass the ofstream by reference to try out references in C++. Declare the ostream as a reference in your Calculator object as well. For ofstream to be defined you must add the <fstream> include statement in your code. The Calculator object should have an additional method to return the ostream reference when asked so the evaluate functions in your abstract syntax tree can get the ofstream when printing code. The code that should be printed is EWE code. You will want to employ a correct model of computation when generating EWE code so you can systematically generate the required code for the expression.

To write to an ostream is just like writing to cout. See the cout write statements in calc.C for examples of how this is done.

8. C++ allows parameters to be passed by value, reference, or pointer. Modify the calculator program to pass Tokens by reference instead of by pointer.
9. Modify the calculator program to pass Tokens by value instead of by pointer.
10. Start with a fresh copy of the C++ calculator code. See exercise 5 for directions on downloading and unzipping the files. Compile it with the *make clean* and *make* commands. Run the program to verify that it does correctly add two integers together.

    Then change the *token.h* file and remove *virtual* keyword from the *getLex* method of the *Token* class but leave the *LexicalToken* . Run the program again and it will likely not add two integers together correctly. Explain why this happens. What happened to the C++ code that removing the keyword

Solution to Practice Problem 3.1

1. 81
2. 6
3. Depends on the initial value of memory. It could be an error. Assuming the calculator starts with 0 in memory the answer would be 3. If the calculator is written to evaluate more than one expression in a session then the memory might contain the last value stored.
4. 10

Go Back to Practice Problem

Solution to Practice Problem 3.2

Below are to commands the make file would execute. The Makefile should probably have included a dependency of parser.C on the ast.h as well. But, if you get into trouble, type `make clean` to start over.

```
g++ -g -c ast.C
g++ -g -c calculator.C
g++ -g -o calc calc.o scanner.o token.o ast.o parser.o
          calculator.o
```

▸ Go Back to Practice Problem

Solution to Practice Problem 3.3

1. There are number, keyword, add, keyword tokens in this one.
2. The tokens are: lparen, number, add, number, rparen, keyword, times, keyword.

▸ Go Back to Practice Problem

Solution to Practice Problem 3.4

The program uses pass by value so the Point data is copied between each of the function calls.

1. An activation record is pushed on the stack for the main function containing the variabie called `myPoint`.
2. Immediately the `makePoint` method is called pushing a new activation record on the stack. The new activation record has its own copy of a point called `aPoint`. The `aPoint` variable is initialized to (0,0) and then when `aPoint` is returned the activation record is popped from the run-time stack and the data is copied back into the `myPoint` variable.
3. The `myPoint` variable is changed to (10,10).
4. When `testit` is called another copy of a point is made in the run-time stack's new activation for the call to `testit` and that variabie is initialized to (0,0). This does not change the value of the `myPoint` variable.

5. Finally, the activation record for the call to `testit` is popped returning to main to finish the program by printing the (10,10) to the screen.

▸ Go Back to Practice Problem

Solution to Practice Problem 3.5

In this example, the `makePoint` function returns a pointer to the `aPoint` variable. However that variable lies within the `makePoint` activation record. That is generally a bad idea. When `makePoint` returns, the main function will now have a pointer to memory on the run-time stack that may be reused in the not too distant future. This will almost certainly be true once the program calls the `testit` function. The outcome of this program is not well-defined. It may work and it may not. The outcome of the program depends on the underlying architecture of the target platform and the code generated by the compiler.

This is precisely the reason Java does not contain an `addressof` operator. Problems like this occur all the time in C++ when inexperienced C++ programmers start writing code. It is a bad idea to get the address of a variable, yet this happens all the time in C++, especially when using arrays. In

Java, the only address ever provided is when a programmer uses the `new` keyword to create an object. However, the address returned by new is a reference, which itself is not a pointer and is safer to use than a pointer. A reference must point to an object and may not be a pointer to just anywhere in memory. In addition, the only references in Java are references into the heap. No pointers into the run-time stack are allowed in the language or its underlying implementation.

Go Back to Practice Problem

Solution to Practice Problem 3.6

Here is the AddNode declaration.

```
class AddNode : public BinaryNode {
 public:
   AddNode(AST* left, AST* right);

   int evaluate();
};
```

Go Back to Practice Problem

Solution to Practice Problem 3.7

Here is the AddNode implementation.

```
AddNode::AddNode(AST* left, AST* right):
   BinaryNode(left,right)
{}

int AddNode::evaluate() {
   return getLeftSubTree()->evaluate() +
          getRightSubTree()->evaluate();
}
```

▸ Go Back to Practice Problem

Solution to Practice Problem 3.8

Here is the RestExpr solution.

```
AST* Parser::RestExpr(AST* e) {
   Token* t = scan->getToken();

   if (t->getType() == add) {
      return RestExpr(new AddNode(e,Term()));
   }

   if (t->getType() == sub)
      return RestExpr(new SubNode(e,Term()));

   scan->putBackToken();

   return e;
}
```

▸ Go Back to Practice Problem

Additional Reading

- Namespaces
- Copy constructors
- Type conversion operators
- Streams and stream operators
- Operator overloading