CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 6
September 19, 2016

Brackets Example

Brackets Example

Code demo

The 06-BracketsCpp demo contains three interesting classes and illustrates the use of constructors, destructors, and dynamic memory management as well as a number of newer C++ features.

It is based on the example in section 4.5 of the textbook, but there are several significant modifications to the code.

Many of the changes use features of c++14 and would not work under the older standard. Others reflect different design philosophies.

We briefly summarize below some of the features of the demo.

The problem

The problem is to check a file to see if the brackets match and are properly nested.

For example, `([]())` is okay, but `([)]` is not, nor is `((()))` or `[[[`.

A bracket matching algorithm

Rules for bracket matching:

1. Each left bracket is pushed onto the stack.
2. An attempt is made to match each right bracket with the top character on the stack.
3. The attempt fails if
   - The stack is empty, or
   - The top character is a different type of bracket (e.g., round instead of square).
4. If the match fails, an error comment is printed, the mismatched characters are discarded, and processing continues with the next character.
5. At end-of-file, the stack should be empty, for any remaining characters on the stack are unmatched left brackets.

Program design

The program is organized into four modules.

1. Class `Token` wraps a single character. It contains functions for determining which characters are brackets, and for each bracket, its "sense" (left or right), and its "type" (round, square, curly, or angle).
2. Class `Stack` implements a general-purpose growable stack of objects of copyable type `T`. In this case, `T` is typedef'ed to `Token`.
3. Class `Brackets` implements the matching algorithm. It reads the file and carries out the matching algorithm.
4. `main.cpp` contains the main program. It processes the command line, opens the file, and invokes the bracket checker.

Token class

Major points:

1. `enum` is used to encode the bracket type (round, square, etc.) and the sense of the bracket (left, right).
2. The two `enum` types are defined inside of class `Token` and are private.
3. `ch` is the character representing the bracket, used for printing.
4. `classify()` is a private function.
5. The definitions of `print()` and `operator<<` follow our usual paradigms.

Token class (cont.)

6. The `Token` constructor uses a ctor to initialize `ch`, and it calls `classify()` to initialize the other data members.
7. In the ctor `:ch(ch)` , the first `ch` refers to the data member and the second refers to the constructor argument.
8. In the textbook version of `Token`, the static variable `brackets` is *local* to `classify()`. It is now a static *class variable*, initialized in `token.cpp`.

Token design questions

1. The textbook version of `Token` uses getters to return `type` and `sense`. `getType()` was used to test if a newly-read character was a bracket, and it was also used to see if a left bracket and right bracket were the same type.
   Why were they needed?
2. The new version of `Token` replaces `getType()` with boolean functions `isBracket()` and `sameTypeAs()` functions. Similarly, `getSense()` was replaced by boolean function `isLeft()`.

   With these changes, enum `BracketType` and `TokenSense` are no longer needed outside of `Token` and hence are now private.

   What are the pros and cons of this design decision?

Token design questions (cont.)

3. Both the old and new versions of the program work whether or not `brackets` is `static`.
   - Is static a better choice here?
   - Why or why not?
   - Does your answer depend on whether the variable is local (old code) or class (new code)?

Stack class

Major points:

1. `T` is the element type of the stack. This code implements a stack of `Token`. (See `typedef` declaration.)
2. Storage for stack is dynamically allocated in the constructor using `new[]` and deleted in the destructor using `delete[]`.
3. The square brackets are needed for both `new` and `delete` since the stack is an array.
4. `delete[]` calls the destructor of each `Token` on the stack. Okay here because the token destructor is null.
5. `push()` grows stack by creating a new stack of twice the size, copying the old stack into the new, and deleting the old stack. This results in linear time for the stack operations.
6. If `push()` only grew the stack one slot at a time, the time would grow quadratically.

Stack design questions

1. Should `pop()` return a value?
2. Why does stack have a `name` field?
3. `size()` isn't used. Should it be eliminated?
4. `Stack::print()` formerly declared `p` and `pend` at the top. Now they are declared just before the loop that uses them. Is this better, and why?
5. Could they be declared in the loop? What difference would it make?