# The TokenScanner Class

**Qi SONG**
**221019037@link.cuhk.edu.cn**

# Motivation

- Many applications need to divide a string into words, or more generally, into **tokens** (i.e. logical units that may be larger than a single character).
- Given that the problem of dividing a string into individual tokens comes up so frequently in applications, it is useful to build a library package that takes care of that task. The primary goal is to build a package that is **simple** to use but also **flexible** enough to meet the needs of a variety of clients.

# Design

- ✓Task 1: **Associate** the token scanner with a source of tokens, which might be a string, an input stream, etc.
- ✓Task 2: **Retrieve** individual tokens from the source of tokens and deliver them one at a time.
- ✓Task 3: **Test** whether the token scanner has any tokens left to process.

pseudocode - reading tokens from a scanner:

```
Set the input for the token scanner to be some string or input stream.
while (more tokens are available) {
    Read the next token.
}
```

# Design

- ✓ TokenScanner should define tokens. What should be considered as a token?
  - ✓ a word in a string?
  - ✓ a single character?
  - ✓ a punctuation mark?
  - ✓ a space?
- ✓ Different applications define tokens in different ways
  - ✓ TokenScanner class must give the client some control over what types of tokens are recognized, e.g. whether a space should be recognized as a token.

# Design

## Methods in the **TokenScanner** Class

| |
|---|
| scanner.setInput(str) *or* scanner.setInput(infile)<br>Sets the input for this scanner to the specified string or input stream. |
| scanner.hasMoreTokens()<br>Returns true if more tokens exist, and false at the end of the token stream. |
| scanner.nextToken()<br>Returns the next token from the token stream, and "" at the end. |
| scanner.saveToken(token)<br>Saves token so that it will be read again on the next call to nextToken. |
| scanner.ignoreWhitespace()<br>Tells the scanner to ignore whitespace characters. |
| scanner.scanNumbers()<br>Tells the scanner to treat numbers as single tokens. |
| scanner.scanStrings()<br>Tells the scanner to treat quoted strings as single tokens. |

# Demo

Ignoring white space:

```
Test program for the TokenScanner class
Input line: hello!world!
 "hello"
 "!"
 "world"
 "!"

Input line: 116010000@link.cuhk.edu.cn
 "116010000"
 "@"
 "link"
 "."
 "cuhk"
 "."
 "edu"
 "."
 "cn"

Input line: today is Monday
 "today"
 "is"
 "Monday"
```

Not ignoring white space:

```
Test program for the TokenScanner class
Input line: hello!world!
 "hello"
 "!"
 "world"
 "!"

Input line: 116010000@link.cuhk.edu.cn
 "116010000"
 "@"
 "link"
 "."
 "cuhk"
 "."
 "edu"
 "."
 "cn"

Input line: today is Monday
 "today"
 " "
 "is"
 " "
 "Monday"
```

# Demo

Scan single digits:

```
Input line: pi = 3.1415
 "pi"
 "="
 "3"
 "."
 "1"
 "4"
 "1"
 "5"
```

Scan the numbers as a whole:

```
Test program for the TokenScanner class
Input line: 2020/3/8 Women's Day
 "2020"
 "/"
 "3"
 "/"
 "8"
 "Women"
 "'"
 "s"
 "Day"
```

# Implementation

In tokenscanner.h:

```
/*
 * Constructor: TokenScanner
 * Usage: TokenScanner scanner;
 *        TokenScanner scanner(str);
 * ----------------------------------
 * Initializes a scanner object.  The initial token stream comes from
 * the string str, if it is specified.  The default constructor creates
 * a scanner with an empty token stream.
 */

   TokenScanner();
   TokenScanner(std::string str);
```

In tokenscanner.cpp:

```
TokenScanner::TokenScanner() {
    ignoreWhitespaceFlag = false;
    singleDigit = false;
}

TokenScanner::TokenScanner(string str) {
    ignoreWhitespaceFlag = false;
    singleDigit = false;
    setInput(str);
}
```

# Implementation

In tokenscanner.h:

```
private:

/* Instance variables */

   std::string buffer;          /* The input string containing the tokens */
   int cp;                      /* The current position in the buffer     */
   bool ignoreWhitespaceFlag;   /* Flag set by a call to ignoreWhitespace */
   bool singleDigit;
```

# Implementation

In tokenscanner.h:

```
/*
 * Method: setInput
 * Usage: scanner.setInput(str);
 * ----------------------------
 * Sets the input for this scanner to the specified string.
 * Any previous input string is discarded.
 */

   void setInput(std::string str);
```

In tokenscanner.cpp:

```
void TokenScanner::setInput(string str) {
   buffer = str;
   cp = 0;
}
```

# Implementation

In tokenscanner.h:

```
/*
 * Method: ignoreWhitespace()
 * Usage: scanner.ignoreWhitespace();
 * ----------------------------------
 * Tells the scanner to ignore whitespace characters.
 * By default, the nextToken method treats whitespace
 * characters (typically spaces and tabs) just like any
 * other punctuation mark and returns them as single-
 * character tokens. Calling
 *          scanner.ignoreWhitespace();
 * changes this behavior so that the scanner ignores
 * whitespace characters.
 */

   void ignoreWhitespace();
```

```
/* Private methods */

   void skipWhitespace();
```

In tokenscanner.cpp:

```
void TokenScanner::ignoreWhitespace() {
   ignoreWhitespaceFlag = true;
}

void TokenScanner::skipWhitespace() {
   while (cp < buffer.length() && isspace(buffer[cp])) {
      cp++;
   }
}
```

# Implementation

In tokenscanner.h:

```
/*
 * Method: hasMoreTokens
 * Usage: if (scanner.hasMoreTokens()) . . .
 * ----------------------------------------
 * Returns true if there are additional tokens
 * for this scanner to read.
 */

   bool hasMoreTokens();
```

In tokenscanner.cpp:

```
bool TokenScanner::hasMoreTokens() {
   if (ignoreWhitespaceFlag) skipWhitespace();
   return cp < buffer.length();
}
```

# Implementation

In tokenscanner.h:

```
/*
 * Method: nextToken
 * Usage: token = scanner.nextToken();
 * ----------------------------------
 * Returns the next token from this scanner. If
 * called when no tokens are available, nextToken
 * returns the empty string.
 */

   std::string nextToken();
```

In tokenscanner.cpp:

```
string TokenScanner::nextToken() {
   if (ignoreWhitespaceFlag) skipWhitespace();
   if (cp >= buffer.length()) {
      return "";
   } else if (isalnum(buffer[cp])) {
      if ((not isalpha(buffer[cp]) && singleDigit)) {
          cp += 1;
          return buffer.substr(cp-1, 1);
      }
      int start = cp;
      while (cp < buffer.length() && isalnum(buffer[cp])) {
         cp++;
      }
      return buffer.substr(start, cp - start);
   } else {
      return string(1, buffer[cp++]);
   }
}
```

# Thank You