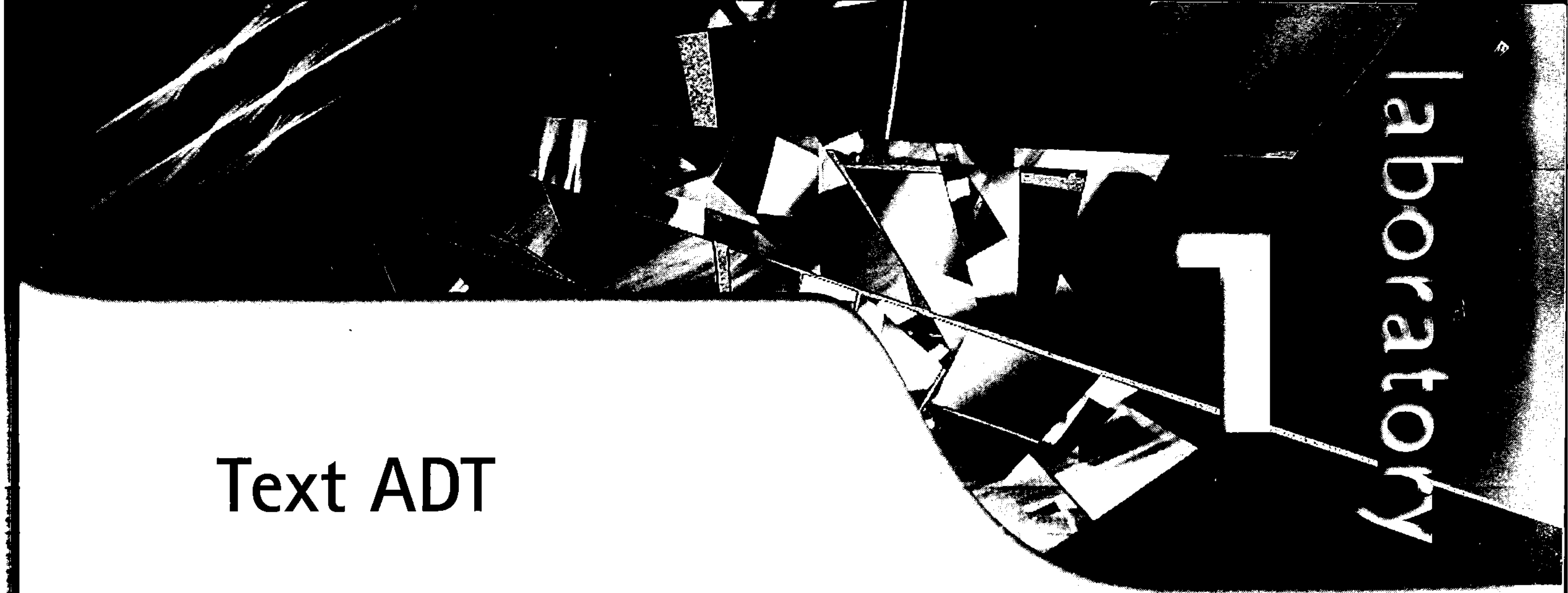

Text ADT

Objectives

In this laboratory, you

- are introduced to the concept of an Abstract Data Type (ADT).
- implement an ADT using a C++ class.
- use the C++ operators `new` and `delete` to dynamically allocate and deallocate memory.
- learn the C++ mechanisms for implementing classes correctly and efficiently.
- create a program that performs lexical analysis using your new Text data type.
- examine the problems with the standard C string representation.

ADT Overview

The purpose of this laboratory is for you to explore how you can use C++ classes to implement an abstract data type (ADT). In this laboratory, you will implement your own string type, the Text class.

When computers were first introduced, they were popularly characterized as giant calculating machines. This characterization ignores the fact that computers are equally adept at manipulating other forms of information, including sequences of symbols called strings. Text is a type of string that is composed of human-readable characters. In this lab we will be working with text.

C++ provides a set of predefined, or "built-in," data types (e.g., `int`, `char`, and `float`). Each of these predefined types has a set of operations associated with it. You use these operations to manipulate variables of a given type. For example, type `int` supports the basic arithmetic and relational operators, as well as a number of numerical functions (`abs` and `div`, etc.). These predefined data types provide a foundation on which you construct more sophisticated data types, types that are collections of related data items, rather than individual data items. Each of the user defined data types in this book will be discussed in two steps. First, we describe them as ADTs (abstract data types). An ADT is the description of a set of data items and of the operations that can be performed on that data. The "abstract" part means that the data type is described independent of any implementation details. Second, we discuss implementation details for turning the ADT description into a usable language-specific data structure. You will implement each ADT by writing one or more C++ classes that make it possible to instantiate actual C++ objects—variables of the user-defined data type—that meet the ADT specifications.

When specifying an ADT, you begin by describing the data items that are in the ADT. You describe how the ADT data items are organized to form the ADT's structure. In the case of the Text ADT, the data items are the letters associated with a text string and the structure is linear: the entries are arranged in the same order as they would appear in written form.

Having specified the data items and the structure of the ADT, you then define how the ADT can be used by specifying the operations that are associated with the ADT. For each operation, you specify what conditions must be true before the operation can be applied (its preconditions, or requirements) as well as what conditions will be true after the operation has completed (its postconditions, or results). The following Text ADT specification includes operations to create and initialize a Text object, assign text to it, compare it to other Text objects, and retrieve the letters at specific positions within the Text object.

C++ Concepts Overview

Many important C++ concepts are introduced in this lab. In this section, we briefly describe each.

Constructors: This is a member function that is automatically run to initialize new objects when they are created. If each C++ class has an appropriate constructor, it will guarantee that all objects of that class are initialized before the programmer can do anything else with the object, thus ensuring that the programmer will always be working with properly initialized data. Like overloaded functions, there can be multiple constructors provided that each has a unique combination of parameter types (the function's signature). To be safe, all C++ classes should have one or more constructors.

Destructor: This is a member function that automatically runs when an object is destroyed. The destructor is responsible for doing any object "clean up" that needs to happen before the object is destroyed. The destructor's most important responsibility in C++ is to ensure that any memory the object borrowed from the memory manager gets returned correctly. Object destruction happens implicitly when the object goes out of scope, or as a result of the programmer explicitly deleting an object.

Dynamic memory allocation: It is quite common to need a block of memory of a specific size—typically an array—at runtime. In C++, the code requests this memory from the memory management routines by using the `new` keyword. When the memory block is no longer needed, it is returned to the dynamic memory pool by using the `delete` keyword. Forgetting to return dynamically allocated memory leads to a serious problem called a memory leak. Any class that performs dynamic memory allocation needs a destructor to help return the memory when the object gets destroyed.

Class protection and access mechanisms: Data and member functions in C++ classes can be protected from access outside the class by declaring them in one of three sections of the class: public, private, and protected. Anything declared in the `public` section is freely accessible from anywhere outside the class. The object-oriented programming (OOP) philosophy is that data and member functions should be public only on a "need to be public" basis, so they should only be placed in the public section if they really need to be there. Data is almost never declared as public. Items in the `private` section are accessible only to member functions within that class. Member functions and data in the `protected` section are treated as private, except that classes that are derived from the given class (through inheritance) are given access to anything in the protected section. C++ also allows the use of `friend` to let a class give a function or another class access to its contents. The use of `friend` will be discussed later when it is needed.

Operator overloading: C++ knows what to do when it sees operators used with predefined data types, but it generally does not know what to do with the user-defined data types and doesn't even try. And when it does try, the results are often not what the programmer intended. To solve the behavior problem, C++ allows a class to redefine the meaning of an operator used in the context of an object of that class. Examples include the '<', '=', '==', '`[]`', and '+' operators.

Const protection: In C++, there are sometimes objects that may not be changed in a specific context. In those cases, the compiler needs to know whether calling a specific member function would illegally change the object. A member function can be declared to be a `const` function, meaning that running it does not cause changes to any of the object's data values. Additionally, a function parameter can be declared `const`, meaning that the function may not modify the parameter.

Text ADT Specification

Data items

A set of characters, excluding the null character.

Structure

The characters in a Text object are in sequential (or linear) order—that is, the characters follow one after the other from the beginning of a string to its end.

Operations

```
Text ( const char* charSeq = "" )
```

Requirements:
None

Results:
Conversion constructor and default constructor. Creates a Text object containing the character sequence in the array pointed to by `charSeq`. This constructor assumes that `charSeq` is a valid C-string terminated by the null character ('`\0`') and allocates enough memory for the characters in the array plus any delimiter that may be required by the implementation of the Text ADT.

```
Text ( const Text& other )
```

Requirements:
None

Results:
Copy constructor. Initializes the object to be an equivalent copy of `other`. This constructor is invoked automatically whenever a Text object is passed to a function using call by value, a function returns a Text object, or a Text object is initialized using another Text object.

```
void operator= ( const Text& other )
```

Requirements:
None

Results:
Assigns the value of `other` to this Text object.

```
~Text ()
```

Requirements:
None

Results:
Destructor. Deallocates (frees) the memory used for the implementation of the Text ADT.

```
int getLength () const
```

Requirements:
None

Results:
Returns the number of characters in the Text object (excluding the delimiter).

```
char operator [] ( int n ) const
```

Requirements:
None

Results:
Returns the n^th^ character in the Text object—where the characters are numbered beginning with zero. If the object does not have an n^th^ character, then it returns the null character.

```
void clear ()
```

Requirements:
None

Results:
Clears the Text object, thereby making it an empty Text object.

```
void showStructure () const
```

Requirements:
None

Results:
Outputs the characters in the Text object, as well as the delimiter. Note that this operation is intended for testing/debugging purposes only.

Implementation Notes

C-strings: C++ supports the manipulation of character data through the predefined data type `char` and the associated operations for the input, output, assignment, and comparison of characters. Most applications of character data require character sequences—or strings—rather than individual characters. A string can be represented in C++ using a one-dimensional array of characters. By convention, a string begins in array data item zero and is terminated by the null character, `'\0'`. (That is how C and early C++ represented strings. Although C++ now has a standard string class, many current programming APIs—Application Programming Interfaces—require a knowledge of the C-string representation.)

Representing a string as an array of characters terminated by a null suffers from several defects, including the following:

- The subscript operator (`[]`) does not check that the subscript lies within the boundaries of the string—or even within the boundaries of the array holding the string, for that matter.
- Strings are compared using functions that have far different calling conventions than the familiar relational operators (==, <, >, and so forth).
- The assignment operator (=) simply copies a pointer, not the character data it points to. The code fragment shown here, for example,

```
char *str1 = "data",
  *str2;
str2 = str1;
```

makes `str2` point to the array already pointed to by `str1`. It does not create a new array containing the string `"data"`. This results in changes to `str2` affecting `str1` since they share the array, and vice versa.

Either the length of a string must be declared at compile-time or a program must explicitly allocate and deallocate the memory used to store a string. Declaring the length of a string at compile-time is often impossible, or at least inefficient. Allocating and deallocating the memory used by a string dynamically (that is, at run-time) allows the string length to be set (or changed) as a program executes. Unfortunately, it is very easy for a programmer to forget to include code to deallocate memory once a string is no longer needed. Memory lost in this way—called a memory leak—accumulates over time, gradually crippling or even crashing a program. This will eventually require the program or computer system to be restarted.

In this laboratory, you develop a Text ADT that addresses these problems. The Text ADT specification that follows includes a diverse set of operations for manipulating strings.

The first decision you must make when implementing the Text ADT is how to store the characters in a string. Earlier, you saw that original C++ represented a string as a null-terminated sequence of characters in a one-dimensional buffer. Adopting this representation scheme allows you to reuse existing C++ functions in your implementation of the Text ADT. This code reuse, in turn, greatly simplifies the implementation process.

Your Text ADT will be more flexible if you dynamically allocate the memory used by the string buffer. The initial memory allocation for a buffer is done by a constructor. One of the constructors is invoked whenever a text declaration is encountered during the execution of a program. Which one is invoked depends on whether the declaration has as its argument an integer or a string literal. Once called, the constructor allocates a string buffer using C++'s `new[]` operator. The constructor that follows, for example, allocates a string buffer of `bufferSize` characters and assigns the address of the string buffer to the pointer `buffer`, where `buffer` is of type `char*`.

```
Text:: Text ( char* chaseSeq = "" )
{
    ...
    buffer = new char [bufferSize];
}
```

Whenever you allocate memory, you must ensure that it is deallocated when it is no longer needed. The class destructor is used to deallocate a string buffer. This function is invoked whenever a string variable goes out of scope—that is, whenever the function containing the corresponding variable declaration terminates. The fact that the call to the destructor is made automatically eliminates the possibility of you forgetting to deallocate the buffer. The following destructor frees the memory used by the string buffer allocated previously.

```
Text:: ~Text ()
{
    ...
    delete [] buffer;
}
```

Constructors and destructors are not the only operations that allocate and deallocate memory. The assignment operation may also need to perform memory allocation/deallocation in order to extend the length of a string buffer to accommodate additional characters.

Strings can be of various lengths and the length of a given string can change as a result of an assignment. Your string representation should account for these variations in length by storing the length of the string (`bufferSize`) along with a pointer to the buffer containing the characters in the string (`buffer`). The resulting string representation is described by the following declarations.

```
int bufferSize;     // Size of the text buffer
char *buffer;       // Text buffer containing a null-terminated
                    // sequence of characters
```

Defaulting parameters: C++ allows the programmer to set up default values for parameters to a function. A default parameter is used in the Text constructor function. Given the function prototype

```
Text ( const char* charSeq = "" );
```

the constructor function should be called with some sort of C-string as a parameter. E.g., a new Text object could be declared by writing

```
Text myText("This is my text");
```

in which case the constructor parameter `charSeq` points to the C-string `"This is my text"`. However, if the programmer instead creates a new Text object by writing

```
Text myText2();
```

then the compiler will make `charSeq` point to the empty C-string `""`.

Self-assignment protection: C++ allows the programmer to overload the assignment operator. For example, the following function will set the value of the Text object to the contents of `other`. We have to deal with the situation when `other` refers to the object to be changed (self-assignment). The "`this != &other`" code performs the self-assignment check. See the Laboratory 5 implementation notes pointer section for a more detailed explanation of what this code does.

```
void Text:: operator= ( const Text& other ) {
    if (this != &other) {
        ...
    }
}
```

Double inclusion protection: C++ allows source code files to include other source code files by using the C++ preprocessor `#include "filename"` syntax. This is commonly done to include class and library declaration header files. The problem is that as files include one or more other files, which may in turn include others, it is quite easy to `#include` the same file more than once. Then the compiler issues errors about double declarations. The standard way to avoid this problem is to use the C++ preprocessor label definition mechanism to ensure that code only gets processed once. To avoid double-including the *Text.h* file, the file is written so as to define a preprocessor variable based on the name of the file. We take the file name, change the '.' to a '_' (because periods are not legal in the preprocessor names), and structure the file as follows:

```
#ifndef TEXT_H
#define TEXT_H
...             // Rest of code in file
#endif
```

When the compiler preprocessor is reading the file, it encounters the `#ifndef TEXT_H` statement. "ifndef" means "IF Not DEFined". The string `#ifndef TEXT_H` asks whether the preprocessor has created (defined) an identifier called `TEXT_H`. If a TEXT_H identifier has not been created, the statement is true; it processes the `#define TEXT_H` statement, defines a `TEXT_H` identifier, and processes everything else in the file up through the `#endif` statement. Next time the file gets included, TEXT_H is already defined, so the #ifndef TEXT_H is false and all lines get skipped through the #endif.

Step 1: Implement the operations in the Text ADT using this string representation scheme. Base your implementation on the following class declaration from the file *Text.h.*

(Note: all header files for this book are included in the student file set. We are including a copy of the header file for this lab for the student's convenience while being introduced to implementing ADTs with C++.)

```
class Text
{
  public:
    // Constructors and operator=
    Text ( const char* charSeq = "" );              // Initialize using char*
    Text ( const Text& other );                     // Copy constructor
    void operator = ( const Text& other );          // Assignment
    // Destructor
    ~Text ();
    // Text operations
    int getLength () const;                              // # characters
    char operator [] ( int n ) const;                    // Subscript
    void clear ();                                       // Clear string
    // Output the string structure -- used in testing/debugging
    void showStructure () const;
    // toUpper/toLower operations (Exercise 2)
    Text toUpper( ) const;                         // Create lower-case copy
    Text toLower( ) const;                         // Create upper-case copy
    // Relational operations (Exercise 3)
    bool operator == ( const Text& other ) const;
    bool operator <  ( const Text& other ) const;
    bool operator >  ( const Text& other ) const;
  private:
    // Data members
    int bufferSize;     // Size of the string buffer
    char* buffer;       // Text buffer containing a null-terminated char
                        //  sequence
};
```

Step 2: Save your implementation of the Text ADT in the file *Text.cpp*. Be sure to document your code.

Compilation Directions

Compile your implementation of the Text ADT in the file *Text.cpp* and the test program in the file *test1.cpp*. Compilation directions will depend on your compiler and operating system. This will typically be through a project file or through the command line.

Testing

Test your implementation of the Text ADT using the program in the file *test1.cpp*.

Step 1: Download the online test plans for Lab 1.

Lab 1 Online Test Plans

Test	*Action*
1-1	Tests the constructors.
1-2	Tests the length operation.
1-3	Tests the subscript operation.
1-4	Tests the assignment and clear operations.

Step 2: Complete the test plan for Test 1-1 by filling in the expected result for each string.

Step 3: Execute Test Plan 1-1. If you discover mistakes in your implementation of the Text ADT, correct them and execute the test plan again.

Step 4: Complete the test plan for Test 1-2 by filling in the length of each Text object.

Step 5: Execute the test plan. If you discover mistakes in your implementation of the Text ADT, correct them and execute the test plan again.

Step 6: Complete the test plan for Test 1-3 by filling in the character returned by the subscript operation for each value of `n` and the string `"alpha"`. Remember, the description of the subscript operator ('`[]`') states that given `[n]`, it returns the nth character, counting from zero. Otherwise, it returns `char('\0')`.

Step 7: Execute the test plan. If you discover mistakes in your implementation of the Text ADT, correct them and execute the test plan again.

Step 8: Complete the test plan for Test 1-4 by filling in the expected result for each assignment statement.

Step 9: Execute the test plan. If you discover mistakes in your implementation of the Text ADT, correct them and execute the test plan again.

Programming Exercise 1

A compiler begins the compilation process by dividing a program into a set of delimited strings called tokens. This task is referred to as lexical analysis. For instance, given the C++ statement,

```
if ( j <= 10 ) cout << endl ;
```

lexical analysis by a C++ compiler produces the following ten tokens.

```
"if"  "("  "j"  "<="  "10"  ")"  "cout"  "<<"  "endl"  ";"
```

Before you can perform lexical analysis, you need operations that support the input and output of delimited strings. A pair of Text ADT input/output operations is described here.

```
friend istream& operator >> ( istream& input,
                              Text& inputText )
```

Requirements:
The specified input stream must not be in an error state.

Returns:
Extracts (inputs) a string from the specified input stream, returns it in `inputText`, and returns the resulting state of the input stream. It begins the input process by reading whitespace (blanks, newlines, and tabs) until a non-whitespace character is encountered. This non-whitespace character is returned as the first character in the string. It continues reading the Text string character-by-character until another whitespace character is encountered.

```
friend ostream& operator << ( ostream& output,
                              const Text& outputText )
```

Requirements:
The specified output stream must not be in an error state.

Returns:
Inserts (outputs) `outputText` in the specified output stream and returns the resulting state of the output stream.

Note that these operations are *not* part of the Text class. However, they do need to have access to the data members of this class. Thus, they are named as friends of the Text class.

Step 1: The file *textio.cpp* contains implementations of these Text string input/output operations. Copy these operations into your implementation of the Text ADT in the file *Text.cpp*. Prototypes for these operations are included in the declaration of Text class in the file *Text.h*.

Step 2: Create a program that uses the operations in the Text ADT to perform lexical analysis on a text file containing a C++ program. Save your program in the file *lexical.cpp*. Your program should read the tokens in this file and output each token to the screen using the following format.

```
1 : [1stToken]
2 : [2ndToken]
...
```

This format requires that your program maintain a running count of the number of tokens that have been read from the text file. Assume that the tokens in the text file are delimited by whitespace—an assumption that is not true for C++ programs in general.

Step 3: Test your lexical analysis program using the C++ program in the file *progsamp.dat* as input. The contents of this file are shown here.

```
void main ( )
{
   int j ,
       total = 0 ;
   for ( j = 1 ; j <= 20 ; j ++ )
       total += j ;
}
```

Test your program using Test Plan 1-5.

Programming Exercise 2

It is useful to have a way of getting an entirely uppercase or entirely lowercase copy of a string. For this exercise, you are to implement the Text class `toUpper` and `toLower` member functions that return uppercase and lowercase copies of the Text object.

```
Text toUpper( ) const
```

Requirements:
None

Results:
Returns a new Text object containing an entirely uppercase copy of the object's internal text string.

```
Text toLower( ) const
```

Requirements:
None

Results:
Returns a new Text object containing an entirely lowercase copy of the object's internal text string.

Implementing each function requires creating a new object that is initialized with a C-string composed of the characters in the Text object, where all alphabetic characters are of the correct upper or lower case. The C++ `toupper` and `tolower` library functions are helpful in doing the case conversion. For instance, given a character `ch`, `toupper(ch)` returns the uppercase value of `ch` if `ch` is alphabetic. Otherwise it returns the unaltered value of `ch`. When using these functions, use "`#include <cctype>`".

Step 1: Implement the operations described above and add them to the file *Text.cpp*. Prototypes for these operations are included in the declaration of the Text class in the file *Text.h*.

Step 2: Activate Test 1 in the test program *test1.cpp* by changing the definition of LAB1_TEST1 from 0 to 1 in *config.h* and recompiling.

Step 3: Complete the test plan for Test 1-6 by filling in the expected result for each test.

Step 4: Execute Test Plan 1-6. If you discover mistakes in your implementation of the copy constructor, correct them and execute the test plan again.

Programming Exercise 3

Most applications that use strings will at some point sort the string data into alphabetical order, either to make their output easier to read or to improve program performance. In order to sort strings, you first must develop relational operations that compare the Text strings with one another.

`bool operator == ( const Text& other )`

Requirements:
None

Results:
Returns `true` if the object is lexically equivalent to `other`. Otherwise, returns `false`.

`bool operator < ( const Text& other )`

Requirements:
None

Results:
Returns `true` if the object occurs lexically before `other`. Otherwise, returns `false`.

`bool operator > ( const Text& other )`

Requirements:
None

Results:
Returns `true` if the object occurs lexically after `other`. Otherwise, returns `false`.

All of these operations require moving through a pair of Text object strings in parallel from beginning to end, comparing characters until a difference (if any) is found between the strings.

The standard C++ string library includes a function `strcmp` that can be used to compare C-strings character-by-character. You will need to "`#include <cstring>`" if you choose to use `strcmp`. Alternatively, you can develop your own code (or private member function) to perform this task.

Step 1: Implement the relational operations described above using the C++ `strcmp` function (or your own private member function) as a foundation. Add your implementation of these operations to the file *Text.cpp*. Prototypes for these operations are included in the declaration of the Text class in the file *Text.h*.

Step 2: Activate Test 2 in the test program *test1.cpp* by changing the definition of LAB1_TEST2 from 0 to 1 in *config.h* and recompiling.

Step 3: Complete the test plan for Test 1-7 by filling in the expected result for each pair of Text objects.

Step 4: Execute the test plan. If you discover mistakes in your implementation of the relational operations, correct them and execute the test plan again.

Analysis Exercise 1

A full-page version of this exercise with space for writing in answers is available in the online supplements for Lab 1.

Part A

What are the implications of having no destructor in a class like Text that does dynamic memory allocation? What are the practical consequences of not having a destructor for these classes in a long-running program?

Part B

What other operators might it make sense to overload in the Text class? Name four and briefly describe how they would work.

Part C

Are there any operators that it does not make sense to overload in the Text class? Why not?

Analysis Exercise 2

A full-page version of this exercise with space for writing in answers is available in e online supplements for Lab 1.

Part A

Design another method for the Text ADT and give its specification below. You need not im ment the method, simply describe it.

Function prototype:

Requirements:

Results:

Part B

Describe an application in which you might use your new method.