The standards described in this document are intended to provide a uniform method for writing and presenting C++ programs throughout the School of Computer Science and Information Systems. These standards should be used by instructors and students alike for all classroom presentations, examinations, projects and programming exercises.

1. Compilation Standards

All code must meet the current ANSI standards for the C++ language, to the degree allowable by your compiler. You should not use a compiler that is more outdated than Visual C++ 6.0. Runtime library functions not defined in this standard (such as those in "`dos.h`" and "`bios.h`") may be called as necessary when developing specialized applications.

Compilation should produce no error or warning messages. Programs must be compiled with all standard and frequent error messages activated in order to ensure correctness.

2. Coding Standards

a) ADT (Abstract Data Type) header files

Each source file that contains externally callable functions should have a corresponding *ADT header file*, containing data type definitions, preprocessor constants and macros, and function prototypes necessary for the call to functions in that source file by functions in other source files. The ADT header file should have the same root name as the corresponding source file; for example, the ADT header file for the source file `mystuff.cpp` would be named `mystuff.h`. A different technique will be used for template files.

b) Use of namespace `std`

Programs making use of the standard namespace `std` (where identifiers such as `cin` and `cout` are defined, for example) should include the appropriate header files without the `.h` extension and should use the **`using namespace`** `std` statement. For example, if you are using anything defined in `std`, you might write:

```
#include <iostream>
#include <stdlib>
using namespace std;
```

c) Constants

Constants with a specific meaning and that may be subject to change should not be "hard-coded"; use const declarations or preprocessor **`#define`** commands to specify program-dependent constants. Externally accessible constants should appear in the

appropriate ADT header file; local constants may appear directly in the source file.

Constant names should be written entirely in upper case. Multiple-word names should use the underscore (_) as a separator.

Constants should be used consistently throughout the program.

d) Identifiers

Identifiers should be self-explanatory; cryptic and unnecessary abbreviations are not allowed. Loop control and scratch variables may be written as `i`, `j`, `k`, `x`, `y`, and `temp`, for example.

Identifiers representing variables, objects and member functions of classes or templates (other than constructors and destructors) should be written entirely in lower case, e.g. `total_entries` or `get_value()`.

Identifiers representing data types, classes, templates and constructor/destructor functions should consist of one or more words, each beginning with a capital letter and the remaining in lower case., e.g. `LongInteger`, `Sequence()`, `SimpleList`.

e) Data types

All programs should strive for type consistency. Do not arbitrarily mix data types. When using a data type that represents one of two possible values (off-on, yes-no, true-false) use the **`bool`** data type and the predefined constants **`true`** and **`false`** (not 1 and 0).

All expressions and all assignments (either through an assignment operator or through the passing of parameters) must be type-consistent.

Reliance on the default data type of an object is forbidden; **all objects must be declared before they are used!** This includes functions and parameter names. Note that long is *not* a data type, no matter what you see in books; the correct data type declaration is long int!

f) Declarations and storage classes

No function definition may appear without a prior defining instance of its function prototype. The declaring instance function prototype should appear in the ADT header for the given file if the function is globally accessible; the declaring instance prototype should appear near the top of the file if the function is local to the given source file.

The declaring instance function prototype and the function header of the function definition should be identical.

Functions that are local to a file must be given the storage class static to ensure that they are not accessible outside that file.

Functions that do not return a value must be explicitly declared to be of type void.

All variables must be declared before they are used. Variables should be declared as locally as possible to the scope of their usage. **The use of global variables (either global to a file or global to the program) is to be avoided**. Any such global variable must be fully documented, including an explanation of why the global declaration is *necessary*.

g) Function definitions

The body of a function definition (from the function header to the closing right brace **}**) should be kept short; use the rule of thumb that the entire definition of a function should not exceed 30 lines. On the average, much fewer than 30 lines should be used. An exception to this rule might be a function containing a switch statement with many individual cases.

Each function definition should be written in accordance with the principle of one-entry, one-exit. *Multiple returns are not permitted!*

If you do not understand by what is meant by one-entry, one-exit, you must find out! Severe penalties are applied to any program that violates this cardinal rule of programming.

A call to the standard function `exit`() or any other function that causes program termination may not appear in any function definition except as the last statement in the function main() or in signal-handling functions.

h) Program statements

The use of the statements **`goto`** and **`continue`** are not permitted.

The **`break`** statement may be used only as the terminating statement of a **`case`** within the **`switch`** statement; each case of a **`switch`** statement must be terminated by a **`break`** statement. To properly simulate the standard case construct, you are permitted to have empty cases (without a break) in order to implement multiple case labels.

Each **`switch`** statement should have an explicit default case unless the explicit cases represent all the possible values of the switch expression.

Premature exit from an if-else or any loop construct is not permitted.

i) Braces and indentation

Matching opening and closing braces (the symbols "{" and "}") should always be indented equally. That is, they should appear in the same column. The opening brace should line up with the **beginning** of the line above it; the body of the block should be indented one tab stop from the enclosing braces.

Each level of program nesting should appear one tab stop in from the previous level. You should use the tab character to perform indentation; for purposes of printing, a tab character should correspond to either 4 or 5 characters.

The cases of an if-else-if structure may all be viewed as being at the same level of nesting. Therefore, the else if clauses should line up with the original if clause.

4. Testing Standards

a) All projects must be thoroughly tested before submission. Testing should include all extreme cases of input data and erroneous input data, unless explicitly not required for a given assignment.

b) Any known errors or shortcomings of a given project should be explicitly documented in the accompanying external documentation. Undocumented errors will be graded more harshly.

5. Documentation Standards

a) Internal documentation

Source files should be as self-documenting as possible. This includes the requirements of descriptive identifiers, proper indentation and spacing and well-organized projects.

Each source file and header file should be begin with a comment clearly showing the name of the file, your name, the creation date, and a short description of the purpose of the file. The standard opening comment in a source or header file should look like

```
//
// SYMTAB.CPP - Source file for major symbol table manipulation functions
//
// Author: John C. Park
//
// Date Created: October 1, 1996
// Revisions:
// 10/ 3/96 - JCP - Corrected bug in insert_attribute().
// 10/10/96 - JCP - Resized MAX_SIZE as 200.
//
```

Other comments should be used sparingly, to explain unusual aspects that can not be described by the code itself.

b) General Appearance

When submitting the listings of your source and header file:

1. Present these files in a meaningful order; it is difficult to read a program if the reader has to flip back and forth through the various files. The reader should not encounter the use of definitions before seeing the definitions themselves. It makes most sense to present the listing of an ADT header file and then the listing of the corresponding ADT source file.

2. Be careful of how your printer will display the text of your files. Do not allow long lines to be truncated or to wrap around to the start of the next line. Break up long lines of text yourself so that the corresponding output will be readable. For example instead of:

```
for (int count = 0, cin >> student_score; s
tudent_score != MARKER; count++, cin >> st
udent_score)
```

write

```
for (int count = 0, cin >> student_score;
          student_score != MARKER;
                    count++, cin >> student_score)
```

b) External documentation

External documentation includes sample input and corresponding output files demonstrating the testing of the program, unless the program is interactive in nature.

External documentation includes a description of the contents of the accompanying disk, and instructions for executing and compiling the program.

External documentation includes a description of each globally callable function. This description should include the purpose of the function, the required syntax for calling the function, the location (i.e. the file name) of the function definition and the declaring instance of the function prototype, a description of how the function is to be used, and a description of the return value (if any) of the function.

External documentation includes a description of any known bugs in the project.