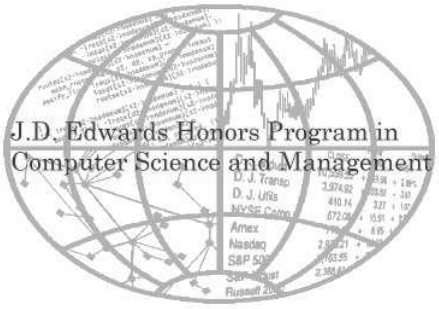

UNIVERSITY OF NEBRASKA–LINCOLN
J.D. EDWARDS HONORS PROGRAM

Coding Standards for C, C++, and Java

Version 3.0
September 3, 2003

1 Introduction

The purpose of these coding standards is to facilitate the maintenance, portability, and reuse of custom C, C++, and Java source code developed by students in the JD Edwards Honor Program, University of Nebraska-Lincoln.

You must receive explicit permission whenever the standards are not followed, and add a comment with the reason for non-conformance.

Language specific information will be denoted by parentheses containing the name of the language. Within tables, "n/a" denotes "not applicable."

1.1 Overview

Section 2 describes file organization, including file content, source file layout, header file layout, and header file guard. Section 3 describes the naming conventions for files, attributes, variables, methods, namespaces, etc. Section 4 describes style guidelines for the source files and header files. Every assignment should observe the standards described from section 2 to 4.

1.2 References and Applicable Documents

GNU standard: **http://www.gnu.org/prep/standards_toc.html**
IBM Standard: http://oss.software.ibm.com/icu/userguide/conventions.html
NASA Standard: **http://ccs.hst.nasa.gov/ccspages/policies/standards/coding_standards.html**
Javasoft Standard: http://Java.sun.com/products/jdk/Javadoc
Possibility Standard: **http://www.possibility.com/Cpp/CppCodingStandard.html**

2 File Organization

This chapter explains what belongs in the source and header files (section 2.1), and how the source and header files should be organized (sections 2.2 and 2.3).

2.1 File Contents

Files should be used to organize related code modules, either at the class (for C++ and Java) or function (for C/C++) level. Always use one file (or header/source file pair) per class or set of logical functions. Of course, nested classes are allowed in Java. Table 2.1 gives the details about what goes in each type of file.

	C	C++	Java
Class declaration	n/a	header file (.h)	n/a
Class implementation	n/a	source file (.cpp)	source file (.java)
Struct declaration	header file (.h)	header file (.h)	n/a
Struct implementation	source file (.c)	source file (.cpp)	n/a
Function prototypes	header file (.h)	header file (.h)	n/a

Function definitions	source file (.c)	source file (.cpp)	n/a

Table 2.1 File Contents

2.2 Source File Layout

Source files should contain the following components **in the order** shown in Table 2.2 (if they are used). Some are sections in themselves (like classes) and may have separate ordering within that scope. Some components may show up in a code block for purposes of scope.

File contents	**C**	**C++**	**Java**
Beginning Comments	X	X	X
Package imports	n/a	n/a	X
Class Document Comments/Prologue	X	X	X
C, C++ Prologue	X	X	n/a
System #includes *****	X	X	n/a
Application #includes *****	X	X	n/a
External functions	X	X	n/a
External variables	X	X	n/a
Constants	X	X	X
Static variable initializations	X	X	X
Class declaration	n/a	n/a	X
Public methods	n/a	X	X
Protected methods	n/a	X	X
Package methods	n/a	n/a	X
Private methods	n/a	X	X
Functions	X	X	n/a

+ ***(Java)*** also called Java Prologue

(C, C++) When it's possible to put a needed #include line in the source file instead of in the header file, do so. This will reduce unnecessary file dependencies and save a little compile time.

Table 2.2. Source File Layout

2.3 Header File Layout (C, C++ only)

Header files should contain the following components **in the order** shown in the Table 2.3 (note that Java does not use header files). Some are sections in themselves (like classes) and may have separate ordering within that scope.

All header files should contain a **file guard** mechanism to prevent multiple inclusion. Below is an example:

```
#ifndef MeaningfulNameH       // first line of the header file
#define MeaningfulNameH       // second line of the header file
.
.                             // body of the header file
.
#endif  // MeaningfulNameH    // last line of the header file
```

File contents	C	C++
File guard	X	X
Prolog	X	X
System #includes	X	X
Application #includes	X	X
#defines	X	X
Macros	X	X
external functions	X	X
external variables	X	X
Constants	X	X
Structs	X	X
Class declaration	n/a	X
Public methods	n/a	X
Protected methods	n/a	X
Private methods	n/a	X
Inline method definitions *	n/a	X
Function declarations	X	X

**(C++)* Small inline methods may be implemented in the class definition.

Table 2.3 Header File Layout

3 Naming Conventions

We use a naming convention to make sure that all names in a project are defined and used in a consistent way, resulting in more readability and understandability. The most important principle for naming everything (files, classes, variables, etc.) is “short but descriptive”. In other words, avoid names that are extremely long, but provide enough detail so it is clear what the name represents. Abbreviations and contractions are discouraged.

For example:

```
public int setNumberReceiver(int aInt); // Recommended

public int setNumberRcv(int aInt);     // Not recommended--uses abbreviations
public int theNumberOfStudentsInTheJDEdwardsHonorsProgramClass;
                                                // Way too long to be useful
```

3.1 General Conventions

The general naming conventions you should follow are summarized on Table 3.1.

3.2 Valid Characters

All names should begin with a letter. Individual words in compound names should be differentiated by capitalizing the first letter of each word. The use of special characters (anything other than letters, digits and underscores) is discouraged.

Identifier	C	C++	Java
package	n/a		ShortName *
class, union, struct	EachWordCapitalized		
interface	n/a		EachWordCapitalized
typedef	EachWordCapitalized		n/a
enum	EachWordCapitalized		n/a
pointers	namePtr		n/a
function, method	internalWordsCapitalized		
class attribute	n/a	internalWordsCapitalized	
convert method	n/a	toX	
accessor method	n/a	getX, setX	
object, variable	internalWordsCapitalized		
#define, macro	ALL_CAPS_AND_UNDERSCORES		n/a
const, static final	ALL_CAPS_AND_UNDERSCORES		
source file	.c	.cpp	.Java
header file	.h		n/a

* If you must combine several words for package name, use format EachWordCapitalized.

Table 3.1 Naming Conventions

3.3 File Names

File names must conform to the following guidelines.

- Names should be descriptive and relate directly to the purpose of the file.
- Do not use spaces between words.
- Do not use more than one period per filename (e.g. my.file.cpp is bad).
- Capitalize the first letter of each word.
- No more than four words in one name.
- Underscores can be used (e.g. Document_Style.doc).

3.4 Function/Method Names

Function names should be an action verb. Functions with Boolean return types should be named with the "is" prefix, as in "isEmpty()" or "isRed()."

3.5 Namespaces/Packages

(C++) There are no requirements for namespaces in C or C++.

(Java) Java packages group classes of related functionality. Package source and class files then reside in a convenient hierarchical directory structure that maps directly to the package name. For example, package edu.unl.jdehp.schmoe.joe corresponds to the directory "…\edu\unl\jdehp\schmoe\joe\".

4 Style Guidelines

The primary purpose of style guidelines is to facilitate long-term maintenance. During maintenance, programmers who are usually not the original authors are responsible for understanding source code from a variety of applications. A common presentation format reduces confusion and speeds comprehension of the code.

4.1 Lines

At most one statement is allowed per line. Keep the code as understandable as possible.

All lines should be displayable without wrapping on an 80-character display. When an expression will not fit on a single line, break it according to these general principles:

- Break after a comma.
- Break before an operator.
- Prefer higher-level breaks to lower-level breaks (see example).
- Align the new line with the start of the expression at the same level on previous line.

Here are some examples.

```
someMethod( firstInteger, secondInteger, thirdInteger,
            FourthInteger);                 //Break at comma, and align

someMethod1( firstInteger,
             SomeMethod( secondInteger, thirdInteger));
                                           //Break at higher-level.

firstInteger = secondInteger * ( thirdInteger + fourthInteger)
               + fifthInteger ;            //Break before operator.
```

4.2 Comments

There are three types of comment delimiters used in C, C++, and/or Java. Table 4.1 shows each, including which can be used in each language.

(C++, Java) Single line comments should use the single line comment delimiter // whenever possible.

When commenting file prologues, classes, methods, and interfaces, you should follow the *Javadoc* format, described in chapter 5.

Type	Begins with	Ends with	C	C++	Java
Single line	//	End of line		**X**	**X**
Multiple line	/*	*/	**X**	**X**	**X**
Javadoc[#]	/**	*/	**X***	**X***	**X**

[#]Javadoc is more fully discussed in chapter 5.

The Javadoc tool does not work with C/C++ code, but since the beginning delimiter has prefix /, this type is still valid in these languages.

Table 4.1 Commenting types

The subsequent sections describe in detail the commenting you must include in your code. Sections 4.2.1 and 4.2.2 describe the comments that every file must begin with, and sections 4.2.3 and 4.2.4 describe the comments used for classes, methods, etc.

4.2.1 Beginning Comments

All source files should begin with a comment that lists the filename, course number, author(s), version, and date. If the code is maintained by a different person/group from the original author(s), consider it a new version and state maintainers name(s). You should **not** use the *Javadoc* format for the beginning comments. Some of the information will be duplicated in the prologue, but do not worry about it. If you are using a software configuration management tool, you do not need to include this. For example:

```
/*
 * SomeClass.java
 *
 * Written by : John Doe, Jane Smith
 * Written for: JDE155
 * Date       : July 30, 2002
 * Version    : 1.0
 *
 * Modified by: Pete Jones, Mary Nelson
 * Written for: JDE155
 * Date       : August 15, 2002
 * Version    : 2.0
 */
```

4.2.2 Prologue

The second type of comment every file should have is the prologue. The purpose of prologue is to provide both users and maintainers of the code with a better understanding of what the code does. In some cases, developers can write a small tool to extract information from the prologues, so observance to the prologue standard is very important. Prologues should always be written with *Javadoc*–type comments, and should **only** contain the following information:

- **Description**. A "big-picture" description (or responsibility) of code, including collaborations with other files.
- **Author**.

- **Version.** The current version number of the code.
- **See**. A list of related class(es), each on a separate line.

Here is an example:

```
/**
 * An class for doing something. A description should be given that
 * will give the user enough information to understand the basics.
 * It will be initialized by some UI class.
 *
 * Version 1.2: Fix bug #3
 * Version 1.3: Fix bug #5
 *
 * @Author     Joe Schmoe jschmoe@cse.unl.edu
 * @Version    1.3, 08/12/00
 * @See        SomethingClass
 * @See        AnotherClass
 */
```

If you are using a software configuration management tool, you only need to include the description, since the tool will take care of the rest for you. The following subsections provide more specific instructions about each component of the prologue.

4.2.2.1 Description

The description must include:

- intent: why the code was developed and how it fits into the process, subsystem; and
- the modification history.

Optionally, you may include

- dependencies; and
- explanation of collaboration with other classes.

Don't include details that are better left as method or function descriptions, or as block comments within the code itself. The description should be as short or as long as is necessary, although one to three paragraphs should cover the majority of files.

For header files and include files, the description should focus on how the class or functions should be used. For source files, the description should focus on how the class or functions are implemented - algorithms, design patterns, etc. For Java files, the description should be a balance of both.

4.2.2.3 Author

List the authors in chronological order, original author first. You may use multiple author tags for multiple authors.

4.2.2.2 Version

The ideal situation is that the configuration management system automatically updates the version and date in the prologue. Otherwise pay attention to the synchronization between the

version number in prologue and actual version number in configuration management system. If you are not using a configuration management system, use version numbers as specified by the instructor. If there are multiple versions, list only the current version using the *Javadoc* tag. You may list the other versions in the description.

4.2.2.3 See

List related books, links, classes, files, etc.

4.2.3 Code Comments

There is a blurry line between cluttering up your code, and putting meaningful comments in. Try to comment in such a way that keeps the code as clean as possible. Not adding comments is definitely not a good answer. This makes it much harder to figure out what you intended. When in doubt, comment the code.

In general, brief comments regarding individual statements may appear at the end of the same line, and should be vertically aligned with other comments in the vicinity for readability.

4.2.4 Classes, Methods/Functions, Interfaces, Class Attributes

Use the *Javadoc* format to comment classes, methods/functions, interfaces, and class attributes in C, C++, and Java.

Block comments should be put in the header files in C/C++ and (obviously) the Java source file. A block comment should be used to describe each method/function, and must be placed before the definition (or implementation) of the method/function. The block comment should include the first three of the below fields, and may contain the last two, particularly if they were written and/or modified by someone other than the original author.

- description: a brief description about the method or function.
- param: the passing parameters and their brief description
- return value
- author
- version

Here is a simple example:

```
/**
 * Compute the sum of two integers
 *
 * @param   param1, The first number
 * @param   param2, The second number
 * @return  the sum of param1 and param2
 * @author  Some guy
 */
```

4.3 Formatting

This sections details the requirements for spacing around operators (section 4.3.1), indentation and braces (section 4.3.2), and blank lines (section 4.3.3).

4.3.1 Spacing Around Operators

One space should be used around all operators with the following exceptions: the `::` and `->` operators in C++; and the `.`, `++`, `--` and `!` operators in C++ and Java. For example,

```
if  (value == 0)                // correct
{
    doSomething();
}
if  ( value==0 )                // wrong, no spaces around ==
{
    doSomething();
}
if  ( value == 0 )              // wrong, spaces around parentheses
{
    doSomething();
}
```

4.3.2 Indentation and Braces

The contents of all code blocks should be indented to improve readability. Four spaces are recommended as the standard indentation. Place the beginning brace on the line below the method, loop, etc., and line the ending brace vertically with the method, loop, class, etc. to which it belongs. For nested if-then-else statements, place the else below the ending brace on the previous if. Use similar format for try-catch-finally blocks. Here is an example.

```
int main()
{
    doSomething();
    switch ( value )
    {
        case 1:
            while ( value == 0 )
            {
                doSomething();
            }
            break;
        case 2:
        case 3:
            doSomething();
            break;
        default:
            break;
    }
}

if  ( value == 0 )
{
    doSomething();
}
else
```

```
{
    doSomething3();
}

try
{
    statement;
}
catch ( ExceptionClass e )
{
    statement;
}
finally
{
    statement;
}
```

The following three examples show the most common alternatives to the style we have chosen. Although there is nothing wrong with them, use the one described above for consistency.

```
if (value == 1) {
    doSomething();
}

if (value == 1)
    {
    doSomething();
    }

if (value == 1) {
    doSomething();
    }
```

4.3.3 Blank Lines

If it makes the code more readable, use a single blank line to separate logical groups of code. Two blank lines to separate each function or method definition may also make it easy to tell where each new function begins. You may also use dashed lines between functions and methods. For example:

```
//-----------------------------------------------------
/**
 * Javadoc style comments
 */
void doNothing()
{
}

//-----------------------------------------------------
/**
 * Javadoc style comments
 */
void returnOne()
{
    return 1;
}
```

```
//-------------------------------------------------
```

4.4 Statements

This section describes some standards you must follow related to control structures (section 4.4.1), conditional statements (section 4.4.2), and include and package import statements (section 4.4.3).

4.4.1 Control Statements

In general, all control statements must be followed by an indented code block enclosed within braces, even if they only contain one statement. This allows the block to be easily expanded in the future. For example:

```
if (value == 0)
{                 // Correct
    doSomething();
}

if (value == 0) doSomething();   // not recommended – no block, not indented

if (value == 0)
    doSomething();               // not recommended - no block
```

4.4.2 Conditional Statements in C/C++

In C and C++, conditional statements do not have to explicitly evaluate to TRUE or FALSE. Any expression that evaluates to zero is considered FALSE, and everything else is TRUE. For clarity, it is recommended that you write your conditional statements so they always evaluate to a Boolean value. Also, do not abuse notation by, for instance, comparing non-pointer values to *null*. Here are a few examples:

```
bool boolValue = getValue();

if ( !boolValue )                 // Correct
{
    doSomethingElse();
}

int intValue = getValue();

if ( intValue == 0 )              // Correct
{
    doSomething();
}

if ( intValue == null )           // Not recommended – null used for pointers
{
    doSomethingElse();
}

if ( !intValue )                  // Not recommended - not explicit test
{
    doSomethingElse();
```

```
}
```

4.4.3 Include Statements and Package Imports

Includes and package imports should be grouped together at the top of each file, after the prolog. They should be logically grouped together, according to system includes, application includes, related packages, etc., with the groups separated by a blank line. Absolute path names should never be explicitly used in `#include` or `import` statements, since this is inherently non-portable.

For C/C++, system includes should use the `<file.h>` notation, and all other includes should use the `"file.h"` notation. For example:

(C/C++)

```
#include <ltstdlib.h>                  // Correct
#include <ltstdio.h>                   //
#include <ltXm/Xm.h>                   //
#include "meaningfulname.h"            //

#include "/proj/util/MeaningfulName.h" // wrong – absolute path given
#include <ltstdlib.h>                  // wrong – out of order
#include </usr/include/stdio.h>        // wrong – path given for system file
```

(Java)

```
import java.awt.*;                     // Correct
import java.awt.event.*;
import java.awt.event.KeyEvent.*;

import javax.swing.*;

import aLibrary.*;
```

4.5 Declarations

In this section the standards relating to declarations are described. Section 4.5.1 discusses variables and attributes in C, C++, and Java. Sections 4.5.2-4.5.6 describe rules that relate to C and/or C++, but not Java.

4.5.1 Variable and Attribute Declarations

Readability and understandability are the goals. Each variable/attribute should be individually declared on a separate line. Variables/attributes can be grouped by permission (public, private, protected and package) or by type (int, float, Applet, etc.), with groups separated by a blank line. Use whichever makes the code clear. The names should be aligned vertically for readability. There is no required ordering of types. A brief comment describing what the variable/attribute is for should be included. Here is one Example:

```
int     area;           // The area of the object, in square inches
int     width;          // The width of the object, in inches
int     height;         // The height of the object, in inches

double  pi;             // An approximation of the constant ‘pi’
```

```
double  e;              // An approximation of the constant ‘e’
```

4.5.2 External Variable Declaration in C/C++

All external variables should be placed in header files. The actual allocation should take place in the implementation file (.c / .cpp). In general, the use of global variables is discouraged (consider creating a singleton class).

4.5.3 Enumerated Type Declaration in C/C++

The `enum` type name and enumerated constants should each reside on a separate line. Constants and comments should be aligned vertically. In general, the `enum` should be within a class. If the user of your class needs direct access to it, then put it in the public section and the user will then simply have to scope it with your class name. This will help reduce the pollution of the global namespace.

Consider using explicit values if these values might be saved to permanent store. If they are not explicit, then they will change if someone inserts a new value. Then, when the values are restored, they may not match the newer enum. In these cases, explicit assignment may help keep the sanity of data on permanent store. Here is an example.

```
enum CompassPoints { // Enums used to specify direction.
    North = 0,       // explicit values not necessary, but recommended.
    South = 1,
    East  = 2,
    West  = 3
};
```

4.5.4 Class Declarations in C++

All class definitions must include a constructor (either default, or at least one parameterized one), (virtual) destructor, copy constructor, and assign (=) operator. If any of these four are not currently needed, create stub versions and place them in the private section so they will not be automatically generated, then accidentally used. (This protects from core dumps and other errors.) It is advisable to put the public section first since the class should represent a concept and the public section holds the services of that concept. The user should not have to know the implementation details. It is suggested that `friend` declarations appear before the public section. All member variables should be either protected or private. It is recommended that definitions of inline functions follow the class declaration (in the .h file), although trivial inline functions (e.g., `{}` or `{ return x; }`) may be defined within the declaration itself.

4.5.6 Function Declaration in C/C++

All functions must be prototyped, with the prototypes residing in header files. If this is not done (especially in C) the parameters may not get checked at compile time, opening the door wide for run-time errors. In general, each class (or struct) will have its own header file (.h) and implementation file (.cpp).

All parameters should either have a meaningful name (even in the prototype) or the use of the parameter must be very clear by the type name. For example, "int" does NOT make the use clear, so you must have something like "int count" that makes the use much more clear.

5 Java Documentation Comments

Java programs can have two kinds of comments: implementation comments and documentation comments. Implementation comments are those delimited by /*…*/, and //. Documentation comments (also known as "doc comments") are Java-only, and delimited by /**…*/. Javadoc comments can be extracted to HTML files using the Javadoc tool, creating a very nice overview, tree structure and description of all the Java files. In fact, you can use HTML in your comments.

As mentioned previously, Javadoc comments are legal in C/C++. In fact, there are several applications available to process them in much the same way as Javadoc does for Java.

Javadoc comments describe Java classes, interfaces, constructors, methods, and fields. Each Javadoc comment is set inside the comment delimiters /**…*/, with one comment per class, interface, or member. The general format of Javadoc comments are list as below:

```
(line 1)   /**
(line 2)    *
(line 3)    * Beginning of description
...
(line x)    * end of description
(line x+1)  *
(line x+2)  * @Tag  tagValue
(line x+3)  * @Tag  tagValue
...
(line x+y)  * @Tag  tagValue
(line x+y+1)*/
```

5.1 Description

Provide a detailed description of the class, interface, method, or field. This field may include the following items.

- intent
- preconditions and/or postconditions
- dependencies
- collaboration with other class, or method
- modification history

You should always give a brief description of the class, interface, method, or field. Unless there are no explicit preconditions to a method, always specify the preconditions. Likewise, postconditions should be included unless they are absolutely obvious. The other items should be used when appropriate, and you can add other items to increase the understandability of the source code.

5.2 Tags

Below are all of the Javadoc tags, listed in the order they should appear.

```
author       (classes and interfaces only, required)
version      (classes and interfaces only, required)
```

```
param        (methods and constructors only)
return       (methods only)
exception    (@throws is a synonym added in Javadoc 1.2)
see
since
serial       (or @serialField or @serialData)
deprecated   (see How and When To Deprecate APIs)
```

Please reference Javadoc homepage http://java.sun.com/j2se/Javadoc/index.html for more detailed descriptions.

See Chapter 6 for an extensive example of Javadoc comments

6 Examples

The next two sections give a complete example of code that follows the standard given in this document. For Java, a single class is given as an example. For C++, there are two examples. The first is a header/source file for a class, and the second a header/source file for a group of related functions.

6.1 Java

The following is an example Java class called **BinaryTree**.

```
//-----------------------------------------------------------------------------
//  BinaryTree.java
//
// Written By  : Chuck Cusack
// Written For : The fun of it
// Date        : June 12, 2001
// Version     : 1.0
//
// Modified on : July 12, 2002
// Version     : 1.3
//-----------------------------------------------------------------------------
import blah.foo.TreeNode;
import blah.foo.Connectives;
//-----------------------------------------------------------------------------
/**
 * BinaryTree is a class that implements the binary tree <i>ADT</i>.
 * The Binary tree is constructed using objects of type TreeNode.
 * Since TreeNodes have char key values, the BinaryTree stores nodes which
 * have char key values.
 * Most of the standard binary tree operations are implemented.
 * <p><font color=red>
 * Keep in mind that this is only serving as an example of the coding standard,
 * and is not meant to be an actual useful class.  You will notice that some
 * methods are not present, and some attributes are not used.  Do not worry
 * about this.  Use this as a guide for the standard, not of proper object
 * oriented design, etc. </font>
 *
 * @author Chuck Cusack
 * @version 1.3
 * @see blah.foo.TreeNode
 * @see blah.foo.Connectives
 */
public class BinaryTree
{
    //-------------------------------------------------------------------------
    // The Class Attribute(s)
    //
```

```
/**
 * A reference to the root of the binary tree.
 */
private TreeNode root;

/**
 * Whether or not the tree is a binary search tree.
 */
private boolean isBinarySearchTree;

/**
 * The number of nodes currently in the tree.
 */
public int numberOfNodes;


//-----------------------------------------------------------------------
/**
 * The default constructor.  It simply creates an empty tree.
 *
 */
public BinaryTree()
{
    root=null;   // Since the tree is empty, the root should point to null.
}

//-----------------------------------------------------------------------
/**
 * A single-node constructor, which creates a new binary tree with one node.
 *
 * @param  rootNode The node which will become the new root.
 *
 */
public BinaryTree(TreeNode rootNode)
{
    root=rootNode;   // Clearly the only thing that needs to be done.
}

//-----------------------------------------------------------------------
/**
 * A three-node constructor, which creates a new binary tree with the first
 * argument as the root, the second argument as the root's left child, and
 * the third argument as the root's right child. If the leftNode or
 * rightNode have children, they will be preserved.  If rootNode has any
 * children, they will be lost.
 *
 * @param  rootNode The node which will become the new root.
 * @param  leftNode The node which will become the root's left child.
 * @param  rightNode The node which will become the root's right child.
 *
 */
public BinaryTree(TreeNode rootNode, TreeNode leftNode, TreeNode rightNode)
{
    root = rootNode;
    root.setLeft(leftNode);
    root.setRight(rightNode);
}

//-----------------------------------------------------------------------
/**
  * @return A reference to the root of the tree.
  */
public TreeNode getRoot()
{
    return root;
}

//-----------------------------------------------------------------------
/**
  * Replaces the root with the given node.  This orphans the old tree, and
  * maintains any children newRoot might have.
```

```
  *
  * @param newRoot The node that will become the new root of the tree.
  */
public void setRoot(TreeNode newRoot)
{
    root = newRoot;
}

//-------------------------------------------------------------------------
/**
  * Performs an in-order traversal of the binary tree.
  * <p>
  * Precondition: startingNode is a node in the BinaryTree, traversalString
  *     references a valid StringBuffer.
  * <p>
  * Postcondition: traversalString has been appended with the in-order
  *     traversal of the tree rooted at startingNode, including parentheses
  *     if addParentheses was set to true.
  * <p>
  *
  * @param startingNode The node to start the traversal at.
  * @param traversalString Stores the string that is created as the tree is
  *        parsed.
  * @param AddParentheses Whether or not to include parentheses in the
  *        appropriate places during a traversal.
  *
  */
private void InOrderTraversal(TreeNode startingNode,
                              StringBuffer traversalString,
                              boolean AddParentheses)
{
    if(startingNode != null)
    {
        // Check for the base case
        // Add a beginning parenthese if appropriate
        if(AddParentheses == true && !startingNode.isLeaf())
        {
            traversalString.append('(');
        }

        // Process the left subtree
        InOrderTraversal(startingNode.getLeft(), traversalString,
                         AddParentheses);

        // Add the current node's key to the string
        traversalString.append(startingNode.getKey());

        // Process the right subtree
        InOrderTraversal(startingNode.getRight(), traversalString,
                         AddParentheses);

        // Add an ending parenthese if appropriate
        if(AddParentheses == true && !startingNode.isLeaf())
        {
            traversalString.append(')');
        }
    }
}

//-------------------------------------------------------------------------
/**
 * Splice the first node into the tree as the parent of
 * the second node, making it the right child, and setting
 * the left child to null.
 *
 * @param currentNode the node in the tree.
 * @param newNode the node add.
 *
 */
public void spliceInRight(TreeNode currentNode, TreeNode newNode)
{
```

```
        // Special case: The root is being replaced
        if(currentNode == root)
        {
            root=newNode;
        }
        else
        {
            // Normal case: An internal node is being spliced in
            TreeNode parentNode = currentNode.getParent();

            // Determine whether newNode should be the left or right child of
            // its new parent.  Then, set the appropriate references.
            if(currentNode == parentNode.getLeft())
            {
                parentNode.setLeft(newNode);
            }
            else
            {
                parentNode.setRight(newNode);
            }
        }

        // Make the currentNode the right child
        // and null the left child of the new node
        newNode.setRight(currentNode);
        newNode.setLeft(null);
    }
    //-------------------------------------------------------------------------
}
```

6.2 C++

The following is an example C++ class called **Queue**.

6.2.1 queue.h

```
#if !defined(QUEUE_000001)
#define QUEUE_000001

/*
 * queue.h
 *
 * Written by : Nick Steinbaugh
 * Written for: JDE310
 * Date       : September 3, 2003
 * Version    : 1.0
 */

/**
 * A class for storing data, specifically integers, in a queue.
 *
 * Version 1.0
 *
 * @Author     Nick Steinbaugh solarflare@gmx.net
 * @Version    1.0, 09/03/2003
 */

#define MAX_SIZE 100

class Queue
{
private:
    //Pointer to the head of the queue
    int head;

    //Pointer to the tail of the queue
    int tail;

    //Array to store the queue data
    int data[MAX_SIZE];
```

```
    //Boolean to keep track of whether or not the queue is empty
    bool empty;
public:

    /**
     * Default constructor
     */
    Queue();

    /**
     * Add an integer to the back of the queue
     *
     * @param   item, The integer to add to the queue
     * @return  Whether or not adding the item was successful
     */
    bool enqueue(int item);

    /**
     * Remove and return the first item in the queue
     *
     * @param   item, The integer to store the value in
     * @return  Whether or not removing and storing the item was successful
     */
    bool dequeue(int &item);

    /**
     * Get the pointer to the first array element of the queue
     *
     * @return  The pointer to the location in the array of the first item
     */
    int *getHead();

    /**
     * Check to see if the queue is empty
     *
     * @return  Whether or not the queue is empty
     */
    bool isEmpty() const;

    /**
     * Check to see if the queue is full
     *
     * @return  Whether or not the queue is full
     */
    bool isFull() const;

    /**
     * Get the size of the queue
     *
     * @return  The current number of items in the queue
     */
    int size() const;
};

#endif
```

6.2.1 queue.cpp

```
/*
 * queue.cpp
 *
 * Written by : Nick Steinbaugh
 * Written for: JDE310
 * Date       : September 3, 2003
 * Version    : 1.0
 */

/**
 * A class for storing data, specifically integers, in a queue.
 *
 * Version 1.0
```

```
 *
 * @Author     Nick Steinbaugh solarflare@gmx.net
 * @Version    1.0, 09/03/2003
 */

#include "queue.h"

Queue::Queue()
{
    //Set the head and tail pointers to zero and empty to true
    head = 0;
    tail = 0;
    empty = true;
}

bool Queue::enqueue(int item)
{
    //Check to see if the queue is full
    if(isFull())
        return false;

    //Adjust tail pointer
    data[tail] = item;
    tail++;
    if(tail >= MAX_SIZE)
        tail -= MAX_SIZE;
    empty = false;
    return true;
}

bool Queue::dequeue(int &item)
{
    //Check to see if the queue is empty
    if(isEmpty())
        return false;

    //Copy queue data to item
    item = data[head];

    //Adjust the head pointer
    head++;
    if(head >= MAX_SIZE)
        head -= MAX_SIZE;
    if(head == tail)
        empty = true;
    return true;
}

int *Queue::getHead()
{
    //Get the pointer to the head of the queue
    return data + head;
}

bool Queue::isEmpty() const
{
    return empty;
}

bool Queue::isFull() const
{
    return !empty && head == tail;
}

int Queue::size() const
{
    if(tail >= head)
        return tail - head;
    else
        return tail + MAX_SIZE - head;
}
```