Program Standards and Conventions
CS 302 Course Requirement

Frederick Harris, Jr.,
Michael Leverington,
Devyani Tanna

7 October, 2013

Table of Contents

1.0 Document Control

1.1 Present Revision of This Document

The present revision of this document is 1.1

1.2 Revision History

1.2.1 Revision 1.1

Amended 7 October 2013, by D. Tanna. Added Doxygen documentation. Modified all examples to match.

1.2.2 Revision 1.02

Amended 24 August 2005, by F. Harris and M. Leverington. Added specifications for parts A, B, and C of each programming solution design. Modified revision documentation process in examples. Also, an Appendix/ Reference section was added to support the use of a standard default code file called wrkbnch.cpp.

1.2.3 Revision 1.01

Amended 16 August 2005, by M. Leverington. Added document control and revision history component; added description and requirements for document control.

1.2.4 Original Document

Developed and reviewed 08 August 2005, by F. Harris and M. Leverington.

1.3 Document Control For Program Code

1.3.1 All Revision Updates

For all revision updates, a brief description of the update will be added to the appropriate file (e.g., *.cpp or *.h file) in the header area. The version number and revision date must also be changed appropriately.

1.3.2 Major Revision Updates

A major revision update includes: 1) making an interface change that affects the programmer using the code, or 2) making a change to the program code that significantly changes the data structure and/or the performance (e.g., speed, overhead requirements, data storage conditions, etc) of the program. Indication of a major revision change will be to change the whole number part of the revision number (i.e., X of X.YY).

1.3.3 Minor Revision Updates

A minor revision update includes any other changes to the program code. Indication of a minor revision will be to change the fractional part of the revision number (i.e., YY of X.YY).

2.0 Coding Priorities

2.1 Overlying Goals

2.1.1 Readability and Clarity

Unless a program or component specification requires specific speed enhancement or special overhead management, clarity and readability are the most important priorities. While correct operation is critical, if there is any problem with the code, no matter how well it was tested, it will be important that it is easy to understand, trace, and ultimately debug.

2.1.2 Correct Operation

Correct operation of the program or code is obviously critical. Test cases should be generated, implemented, and documented to verify correct operation as well as to provide support for programmers who may need to maintain or extend the code segment.

2.1.3 Alignment with Specification

The program code must demonstrate its ability to safely and appropriately conduct the operations specified by the designer. While generalizability is desired, it may not be used as a rationale to depart from the specifications provided.

2.1.4 Generalizability

The program code or segment should be written so that it can handle variations in size and type of data, and other potential changes driven by maintenance or extension of the program or code.

3.0 Program Layout and Organization

3.1 Goals

The primary goal for organizing the layout of program code is to promote safe and efficient creation, implementation, and management of the code. Secondly, programmers who may not have previously worked with this code should be able to easily find various segments of the code in order to conduct maintenance, debugging, or extension of the code. Finally, it should be easy for the programmer to identify various programming components, such as data acquisition, input or output processes, branching, algorithmic processing, etc.

3.2 Component Blocks

3.2.1 Information Block

The first block should provide information about the program, its author(s), any updates implemented, etc. The information block should contain at least the following:

- file name
- description of program or code
- version number, author, and date formatted as shown below with the most recent version and activity at the top of the list

Example:

```
/**
    @file     Restaurant.cpp

    @brief This program manages hotel and restaurant financial
operations

    @version Revision 1.02 (08/15/2005) - M.E. Designer
        redesigned the data structure for storing restaurant
        financial transactions; data structure now uses linked
        list in place of the array that was used previously

    @version Revision 1.01 (08/10/2005 - E.E. Developer
        added CalcTax function to support taxable sales

    @version Original Code 1.00 (08/05/2005) - C.S. Student

*/
```

3.2.2 Pre-Code Blocks

There should be blocks for the following:

- program description or support, such as:
  - associated support files
  - arguments required by a command line program
  - special use conditions for the program or code
  - description of the program operation, as needed
- pre-compiler directives
- header files and controls
- global constant definitions
- class and/or data structure definitions
- free function prototypes

Example:

```
//  Program Description/Support ////////////////////////////////////
/**
    @mainpage
    This program uses a small file (r_cfg.ini) to support user
    configuration. If the file is not found, defaults will be
    created and used by the program.
*/
//  Precompiler Directives /////////////////////////////////////////
//
    #ifndef REST_CPP
    #define REST_CPP
//
//  Header Files  //////////////////////////////////////////////////
//
    #include <cstdio>

    using namespace std;
//
//  Global Constant Definitions  ///////////////////////////////////
//
    const int SUCCESSFUL_OPERATION = 0;
//
//  Class Definitions  /////////////////////////////////////////////
//
/////  NONE
//
//  Free Function Prototypes  //////////////////////////////////////
//
/////  NONE
//
```

3.2.3 Code Blocks

There should be blocks for the main function if used, free function implementations if used, and class or data structure implementations if used, as shown:

```
//  Main Function Implementation  //////////////////////////////////
//
    int main()
       {
        //
        // program code
        //
        return SUCCESSFUL_OPERATION;
       }
//
//  Free Function Implementation  //////////////////////////////////
//
    int myFunc( int a, int b, int c )
       {
        // free function code
       }
//
```

```
//  Class/Data Structure Member Implementation ////////////////////
//
    MyClass::MyClass()
       {
        // member function code
       }
//
```

3.2.4 File End Precompiler Directives

At the end of the file, `#endif` must be added if the `#ifndef` was used at the beginning. Always use a comment to specify what decision test the `#endif` is terminating, as shown:

```
//  Terminating Precompiler Directives ////////////////////////////
//
    #endif  // REST_CPP
//
```

3.3 Spacing and Layout

3.3.1 Goals

The primary goal of spacing and layout is to provide program code that is easy to view and identify components of the code, and easy to identify algorithmic, branching, or operational parts of the code. In addition, it should support and sustain safe programming practices across the entire life cycle of the code.

3.3.2 Maximum Code Width

Maximum code width will be 80 characters

3.3.3 Spacing Between Tokens and Identifiers

There must be one space between all tokens and identifiers except single open parentheses and brackets, increment and decrement operators, commas and semicolons. Examples:

```
a = b + c;
x = r * MAX_ITEMS / NUM_USERS;
for( index = 0; index < MAX_ITEMS; index++ )
   {
    // code
   }

while( index < MAX_ITEMS )
   {
    cout << "Item No " << index + 1
         << ": "
         << nameArray[ index ] << endl;
    index++;
   }
```

```
// Double open parentheses used below
// First one has no preceding space;
// second one requires a space
if( ( IS_VERBOSE ) && ( index < MAX_ITEMS ) )
   {
    // code
   }
```

3.3.4 Braces

All decision-making operations must use braces even if there is only one line of code implemented. There are two strong reasons for doing this. First, it creates effective white space around the result of the decision-making process, and secondly, it reduces difficulty with bugs generated by programmers who add code to the block at a later time. As a suggestion, the best way to implement the braces is to: 1) type the decision-making statement, then 2) type in both the opening and closing braces properly indented, and finally 3) type in the block of code. This adds a third benefit to the process of not having to trace down missing braces at compile time.

3.3.5 Indenting

There are two options for indenting, depending on personal style. Depending on the editor, automatic indenting may need to be turned off in order to implement this format. Examples:

1) Indent three spaces in from the statement above, then type an open brace, indent four spaces in from the statement above to enter the enclosed statements, and then indent three spaces once again for the closing brace, as shown:

```
if( numStudents < MAX_STUDENTS )
   {
    // code here is indented four spaces
   }
```

or,

2) Align the opening brace with the statement above, indent four spaces in from the statement above to enter the enclosed statements, and then align the closing brace with the statement above, as shown:

```
if( numStudents < MAX_STUDENTS )
{
    // code here is also indented four spaces
}
```

In either form, the opening and closing braces must be vertically aligned with each other.

3.4 Identifier Creation and Usage

3.4.1 Goals

Implementing and maintaining common standards, or a convention, among programmers will support maintainability and sustainability by making the identifiers easy to recognize by any other programmers who may review your code.

3.4.2 Alphabetic Character Case

Classes and structs start with upper case letters, and use upper case letters for new words in identifier, as shown:

```
class StudentList
   {
    // class implementation
   };

struct DataItems
   {
    // struct implementation
   };
```

Variables start with lower case letters, but use upper case letters for new words in the identifier, as shown:

```
int myInt;
double studentGrade, examScore;
```

Free functions start with upper case letters, and use upper case letters for new words, as shown:

```
void RemoveNode();
bool InitializeArray();
double ReturnResult();
```

Member functions start with lower case letters, and use upper case letters for new words, as shown in usage:

```
presentGrade = student.returnGrade();
birthMonth = birthDate.returnMonth();
valueOne.add( 45 );
```

Constants use all capital letters, with underscore dividers between words, as shown:

```
const int MAX_STUDENTS = 100;
const bool IS_VERBOSE = true;
const double CITY_TAX_RATE = 6.75000;
```

3.4.3 Self-documentation

All programmer-created identifiers including constants, variables, functions, and so on must clearly specify the usage or representation of the identifier that is used, with the following qualifications:

- abbreviations of identifier segments are acceptable as long as they are easily understandable by others
    - example: `taxAmt`, `rentalPymnt`
- loop or other counter variables may use `i`, `j`, `k` or `x`, `y`, `z` as long as the counter usage is clear; otherwise, the counter variable must represent its action with names such as `nameCtr`, `rowCtr`, `colCtr`, `arrayIndex`, or just `index`, etc
- no literals may be used other than zero (0) or one (1); all other literal or constant usage must be defined as constants
    - examples: `DAYS_IN_WEEK`, `MONTHS_IN_YEAR`, etc
- characters used in mathematical or processing operations must be used as characters, not as ASCII code; using numbers in code reduces readability, and using numbers to represent characters may not be portable
    - example: `testChar - 'a'`, NOT `testChar - 97`

3.4.4 Global Variables

There are very few cases where the use of global variables outweighs the need for safe, maintainable programming. For that reason, unless it is specified or required to support speed or data management enhancement, global variables are not to be used.

3.4.5 Class and Structured Data

With the exception of classes or objects that are used as simple data structures or nodes, all data used within a class must be protected from external access, and managed with accessor and modifier subroutines.

3.5 Mathematical and Boolean Operations

3.5.1 Goals

Mathematical and Boolean operations must be clearly presented in code so that it is clear to the author that the code is doing what it should be doing, and the author's intent is clear to other programmers who will be reviewing the code.

3.5.2 Mathematical Processes

If there are more than three operands in a mathematical expression, they must be organized and contained using parentheses as needed; it may also be desirable to create interim variables to break down complex mathematical processes into smaller, more easily understandable quantities. This increases programming safety, and it will also make debugging and tracing the mathematical processes easier.

3.5.3 Boolean Processes

All Boolean expressions must be enclosed in parentheses. This will normally be the case for simple expressions, but when multiple component expressions are implemented, clarity and accuracy will be improved when each expression is contained. It is also a good idea to put separate parts of the expression on separate lines. Example:

```
leapYear = ( ( ( year % 4 == 0 )
            && ( year % 100 != 0 ) )
            || ( year % 400 == 0 ) ) );
```

In some cases, it may be clearer and more robust to create Boolean variables for each of the components, and then implement the expression, as shown:

```
bool divisibleByFour, notAnOddCentury,
                            isAnEvenLeapYearCentury, leapYear;

divisibleByFour = ( year % 4 == 0 );
notAnOddCentury = ( year % 100 != 0 );
isAnEvenLeapYearCentury = ( year % 400 == 0 );

leapYear = ( ( divisibleByFour
            && notAnOddCentury )
            || isAnEvenLeapYearCentury );
```

3.6 Subroutine Usage

3.6.1 Goals

The most important part of a given program are the modular components that make up the building blocks of that program. These modules, or subroutines (i.e., functions in C++, methods in Java, functions or procedures in PASCAL) must be implemented in a well-managed format, or they will become difficult to use, debug, extend, and maintain. Carefully constructed subroutines will support a superior program development process, testing and refinement processes, and debugging processes when they are necessary. Poorly constructed subroutines commonly increase the difficulty of these processes. The construction and development of subroutines will define the level of difficulty with managing a segment of code throughout its life cycle.

3.6.2 Subroutine Specification and Documentation

All subroutines (i.e., functions, methods, etc) must have the following information commented in the file preceding the implementation of the subroutine:

1. Precondition: Expected status of all incoming data. Example:

```
/** @pre
   -# double scoreData[] contains student scores 'a'
      such that 0.00 <= a <= 100.00
   -# int numScores contains the number of student scores 'n'
      such that n > 0
   -# doubles mean and median are uninitialized
   -# integer mode is uninitialized
*/
```

2. Postcondition: Expected status of data after processing. Example:

```
/**
   @post
   -# double scoreData[] and int numScores are not changed
      in the calling function
   -# double mean holds the average value
      of all the scoreData[] scores
   -# if numScores is an odd number, double median holds
      the exact middle value of the scoreData[] list
   -# if numScores is an even number, double median holds the average
      of the two middle values in the scoreData[] list
   -# mode holds the highest number of identical values found
      in the scoreData[] list
*/
```

3. Algorithm: Process by which the function will implement its task(s). Example:

```
/** @detail @bAlgorithm
   -# mean is calculated by adding all the values in scoreData[]
      and dividing them by numScores
   -# median is found by sorting the scoreData[] list by value,
      and then finding the middle value of an odd numbered list,
      or the average of the two middle values in an even numbered list
   -# mode is found by sorting scoreData[] list by value,
      and then iterating through the list searching
      for the largest number of identical scores
*/
```

4. Exceptional/Error Conditions: Conditions that might cause failure of correct results, or of program operation, and the actions implemented to resolve them. Examples:

```
/** @exception out_of_range if numScores <= 0, zero (0) is returned
      without further processing
    @exception out_of_range if a value 'a' in scoreData[], is found
      with a value a > 100.00 or a < 0.00,
      a negative one (-1) is immediately returned
      to reflect the error in the data
*/
```

5. Alternative form containing all the above information.

```
/**

@pre
   -# double scoreData[] contains student scores 'a'
      such that 0.00 <= a <= 100.00
   -# int numScores contains the number of student scores 'n'
      such that n > 0
   -# doubles mean and median are uninitialized
   -# integer mode is uninitialized
 @post
   -# scoreData[] and numScores are not changed
      in the calling function
   -# mean holds the average value
      of all the scoreData[] scores
   -# if numScores is an odd number, median holds
      the exact middle value of the scoreData[] list
   -# if numScores is an even number,
      median holds the average of the two middle values
      in the scoreData[] list
   -# mode holds the highest number of identical values found
      in the scoreData[] list
 @detail @bAlgorithm
   -# mean is calculated by adding all the values
      in scoreData[] and dividing them by numScores
   -# median is found by sorting the scoreData[] list by value,
      and then finding the middle value of an odd numbered list,
      or the average of the two middle values
      in an even numbered list
   -# mode is found by sorting scoreData[] list by value,
      and then iterating through the list searching
      for the largest number of identical scores
 @exception out_of_range if numScores <= 0, zero (0) is returned
      without further processing
 @exception out_of_range if a value 'a' in scoreData[], is found
      with a value a > 100.00 or a < 0.00,
      a negative one (-1) is immediately returned
      to reflect the error in the data
*/
```

Note: The above subroutine accomplishes more than one task which contradicts subroutine standards specified below. If this function were implemented as designed and followed good subroutine creation standards, it would likely have been used to call the three processes as subroutines of their own. However, the above is used as an exaggerated example for purposes of demonstrating the specifications of this part.

3.6.3 Parameter Usage and Management

1. All parameters must be specified in comment form. This may be accomplished in different ways, as shown.

Example Form 1, as part of the subroutine specifications:

```
/**
     @param scoreData[]  holds student exam scores
     @param numScores  holds number of exam scores
     @param mean  will provide mean statistic result
     @param median  will provide median statistic result
     @param mode  will provide mode statistic result
*/
void Calculate Statistics( double scoreData[], int numScores,
double &mean, double &median, int &mode )
   {
    // function implementation
   }
```

Example Form 2, implemented within the parameter list:

```
void Calculate Statistics
  (
   double scoreData[], /**< holds student exam scores */
   int numScores,      /**< holds number of exam scores */
   double &mean,       /**< returns the mean of the scores */
   double &median,     /**< returns the median of the scores */
   int &mode           /**< returns the mode of the scores */
  )
   {
    // function implementation
   }
```

2. If there are more than five parameters, there must be a data structure created and used in place of some or all of the parameters, as shown:

```
struct DataItems
   {
    int item1;
    int item2;
    int item3;
    int item4;
    int item5;
   }

double ReturnAverage( int numItems, const DataItems &data )
   {
    // function implementation
   }
```

3. All classes and data structures are to be passed by reference. If the class or data structure is not to be modified by the subroutine, it must be specified as a constant, or otherwise protected from modification. Examples:

```
// data will be entered into each of the DataItems members
int EnterDataItems( DataItems &data )

// no part of PersonList is to be modified by this function
double ReturnBirthDate( const PersonList &list, string personName )

// no part of StudentList is to be modified by this function
int GetStudents( const StudentList &list, int gradeLevel )
```

4. Local variables must be defined at the beginning of the subroutine, including counter or loop variables, as shown:

```
PersonList::processList()
   {
    int rowCtr, colCtr;
    string firstName, lastName, city, state;
    bool found, isProcessed = false;

    // subroutine implementation code
   }
```

3.6.4 Subroutine Design Structure

Subroutines should meet the following specifications:

1. **Subroutines should accomplish one task.** They should only conduct one piece of business. They may be able to do that business repeatedly, or in different parts of a program, or with different data types or quantities, but they should do only one thing.

2. **Subroutines should implement modularity and reusability.** Since subroutines are to accomplish one task, each one is a self-contained and unique module (i.e., its own self contained package) that takes care of its own business. Subroutines should do the one job they are given, but they should be able to do it with a variety of incoming data or conditions; this makes methods both modular and reusable.

3. **Subroutines should meet specifications.** They should be designed, tested, and implemented using the specifications that were given. If a need appears to arise where the subroutine should do something outside the specifications, either review and modify the specifications, or design a different subroutine to meet the new conditions.

4. **Subroutines should be concise.** Use no more than seven (7) significant program statements in a subroutine, as defined below. If it appears that the subroutine needs more significant program statements than seven, break it down into smaller subroutines.

A Significant Statement is:

- an action statement, such as implementing a mathematical operation
- an action statement which includes calling another function
- a branching statement, such as an `if`, `if else`, or `else`
- a repetition or looping process, such as `for`, `while`, or `do...while`
- for branching or repetition, if there are three or fewer lines inside the branch or loop block, the branching or looping statement may be considered one significant statement
- an input or output statement

A Significant Statement is NOT:

- a variable definition and/or initialization at the beginning of a function (e.g., `int myVar;` or `int myVar = 5;`)
- simple assignment operations that do not use math or other operations (e.g., `numberOfBells = 7;` or `taxRate = 0.075;`)
- simple incrementing or decrementing of one or more variables (e.g., `myVar++;` or `taxableItems--;`)
- for a series of similar output statements, if there are fewer than five or six of these actions (e.g., five output statements that are generally the same), then these may be considered one Program Step
- other less significant steps such as returning a value from a method are also not considered significant statements, as long as there is no processing or calculating involved

4.0 Program Code Review

4.1 Goals

The goal of the three stage design and development process is to allow peer or supervisory programmers to evaluate and review your program as it is developed; this process is commonly termed a "gateway" process and/or a "Third Party Review (TPR)" process. In any event, in the industrial environment, programs must be reviewed and evaluated as they are developed in order to support and maintain programming quality control.

4.2 Design Level A Requirements

The first component turned into the reviewer will meet the following specifications:

1. It will contain one or more header files as needed to meet the specifications below

2. If the program only requires one program file, then there will be one header file turned in. The header file will have a segment identified above as the "Program Description/Support" where necessary files and information about the program are found. In addition, the description area should have the program operation and algorithm overview specified as shown in the example below:

```
//  Program Description/Support ////////////////////////////////////
/**

       @brief Program provides a calculation that converts Celsius
temperatures into Fahrenheit.  No other files are necessary for this
program.

       @detail Program will start loop to ask user for a Celsius value.
When the user responds with a number other than -999, the program calls
the function ConverCtoF, which returns the result, and which is     then
displayed
*/
```

3. If there is more than one header file, such as when support classes and/or library functions are used, the overall program description should be located in the driver file, meaning the one that has the `main()` function in it, and a description of the actions and usage of the class(es) or library(s) should be placed in the other header files, as shown

```
//  Program Description/Support ////////////////////////////////////
/**
    @mainpage
    Program requires support from two classes, a menu class and an
    integer management class; it also requires support from a
    header file that holds some common constants and operations;
    the files are:

       constants.h           - file with common constants
                                 and operations

       MenuClass.h           - files that support the menu class
       MenuClass.cpp

       IntegerListClass.h    - files that support the integer
       IntegerListClass.cpp     management class

    Numerical (integer) data is entered, and the program calculates
    statistical values for the mean, median, and mode
    If the data is downloaded to a file, it will be labeled
    "statsdat.txt"; if data is to be uploaded from a file, the file
    name must be the same

    Program Overview:

    The menu operations will be conducted by an object
    of the MenuClass; menu choices will be:

         1. Enter data from keyboard, using the free function
            "KBEnter", specified below
```

```
            2. Load data from file, using the IntegerListClass
               member function "loadValues"

            3. Display results, using the free function
               "DisplayResults", specified below

            4. Clear data set, using the IntegerListClass
               member function "clearList"

            5. Store integer values list, using the IntegerListClass
               member function "storeValues"

            6. Quit, which will stop the menu loop
*/
```

4. After the program overview has been created, define the functions that will be used in the program, and/or the class or library. Each free function, member function, and/or library function must have a brief description immediately above its declaration, as shown:

Free Function Description:

```
/** @brief Free Function:
    Starts a loop and uses the IntegerListClass
    "addValue" function to add items until the user enters
    a negative number; no negative values will be accepted
*/  void KBEnter( IntegerListClass &intList );
```

Member Function Description:

```
/**  @brief Member Function:
    appends a new value to the list with each call.
     If the appending process succeeds, the function
     returns true, otherwise false
*/  bool addValue( int newValue );
```

5. Parameters are not required for the first design pass. If they are known, they should be provided; if not, leave the parameter list blank.

6. The version number for the header file (i.e., <header file>.h) is 1.00 at this point in the design process.

4.3 Design Level B Requirements

The second component turned in to the reviewer will be the stubbed program capable of being both compiled and built, but not necessarily runnable. The program and code segments must contain the following components:

1. All major functions necessary for the program to run as described in the design above.

2. The main or driver program must be written in such a way that it would work if the stubbed functions were operational.

3. Any constants or supporting components required to support compiling and building the program must be included in the appropriate files.

4. Formal function documentation, including at least: 1) Preconditions, 2) Postconditions, 3) Algorithms, and 4) Error/Exceptional Conditions. An example stubbed function with documentation, is shown:

```
/**


@pre
   -# array intArray has been initialized with capacity
     arrayCapacity such that arrayCapacity > 0
   -# array intArray has numItems values,
     such that numItems >= 0
@post
   -# array intArray is resized as needed
   -# newValue is appended to intArray
   -# item count numItems is incremented
 @detail @bAlgorithm
   -# resizeCheck is used to check for and resolve
     array capacity condition
   -# newValue is appended to intArray
   -# numItems count is incremented
 @exception logic_error if resizeCheck fails, no action is taken,
     function returns false, otherwise function returns true
*/
//
    bool IntegerListClass::addValue
    (
     int newValue /**<value to be appended to the integer list array */
    )
       {
            cout << "Stub Function: "
                 << "IntegerListClass addValue called"
                 << endl;

            return false;
       }
```

5. If changes are made to any header file(s) as a result of this part of the process, the revision must be documented and the version number updated.

6. The version number for the implementation file (i.e., the *.cpp file) is 1.00 at this point in the design process.

4.4 Design Level C Requirements

Turn the running program in to the reviewer, capable of being compiled, built, run, and tested under the conditions originally specified, with the following considerations:

1. Significant changes to any of the files created prior to this design point must be documented with brief explanations of the changes, and given new version numbers as specified above. Since this is still the creation/development phase of your program, you do not have to identify every change at this point; however, you must identify, describe, and update your version number for any of the following actions:
   - the addition of any new functions not specified in previous design steps
   - a change to the specifications of any of the functions defined in previous design steps, particularly any changes to the algorithm(s) or process(es) used to implement the function
   - any significant changes to your initial data structure(s)

4.5 Time Log

For purposes of programming operations management, it is necessary to log the time you work on each part of each project. The time log is kept simple so as not to cost working time implementing it. In a text file called "timefile.txt" that is provided to you, enter the following items where prompted. The prompts you will find in the text file are in bold, and example responses are in italics:

Note that the time is in "military" form. You must start a new time sheet for each programming job.

Programmer : *Student, Charles S.*
Project Number : *1*
Date Due : *10/12/2005*

DATE	**START TIME**	**END TIME**	**PART**
10/10/2005	*1400*	*1630*	*A*
10/10/2005	*1800*	*1940*	*A*
10/11/2005	*1300*	*1445*	*B*
10/11/2005	*1600*	*1900*	*B*
10/12/2005	*0800*	*1015*	*C*

5.0 Appendix/Reference

It is a good idea to use the wrkbnch.cpp file as the starting point for developing any code component. The file should be left as a Read-Only file so that it is not accidentally changed. Once the file has been opened, it can be saved as the file name needed, and then modified as necessary.