

**Coding  
and  
Documentation  
Standards**

**Developed by:  
Dr. Jeffrey J. McConnell**



# Preface

It is important for us to remember that even though the programs we write are to be run by a computer, they are to be read by people. Because of this, we must write our programs so that they are as readable as possible. This is true not only of how we design the code, but how it also appears in the file. In other words, it is important that we choose good variable, class, constant, and function names, and it is important that we space and comment our code for improved readability. As you read through these standards, you will see regular references to “the person reading your code.” As you write your programs you should try to help this “person” understand what you’re doing because, in some cases, this person may be you at some time in the future.

This document specifies a standard for naming, indentation and spacing, programming style, and internal and external documentation. These standards are not absolute, but are designed to give students a set of guidelines that they can use to develop a reasonable style of their own. Any style adapted from these standards should not be radically different because significant changes are likely to reduce the resulting readability of the code. It is important that students realize that whatever style they use, they should be consistent in all their work. If you look at different books that introduce computer programming you will notice that the style used in each will vary, but each book will be consistent in the style it uses.

The standard specified here is based on what your instructor has found, by experience, most readable. Many of the examples you will find in this document have no real meaning, but are created to show you the style of indentation, spacing, and commenting that is recommended. Because of this, you should look at these examples for their appearance and form instead of the meaning of the code presented.

## Table of Contents:

<b>Preface</b> .....	<b>i</b>
<b>Table of Contents:</b> .....	<b>i</b>
<b>Introduction</b> .....	<b>1</b>
<b>Naming conventions</b> .....	<b>2</b>
<b>Indentation and spacing</b> .....	<b>3</b>
Statements and blocks.....	3
Declarations.....	4
If statements.....	5
Switch statements.....	5
For loops.....	6
While loops.....	7
Do loops.....	7
Functions.....	7
<b>Programming style</b> .....	<b>8</b>
<b>Internal comments</b> .....	<b>9</b>
File headers.....	9
Function headers.....	9
Declaration comments.....	10
Statement comments.....	10
Block Comments.....	11
Declaration file comments.....	11
<b>External Documentation</b> .....	<b>12</b>
User’s Manual.....	12
System’s Manual.....	13
<b>Appendix - Sample Program Description, Documentation, and Program</b> .....	<b>14</b>



## Introduction

For most programming languages, spaces and comments are unimportant to how a program runs. In fact, many programming language compilers will remove all extra spaces as a first step in producing an executable program. Because of this, the following two functions will run exactly the same:

```
void
Player::Play_Slots()
{
    int winnings = 0;
    int i;
    for (i=0; i < coins; i++)
        winnings += Fruit_Market.Play_Quarter(0);
    coins = winnings;
    cout << playersName << " you have won " << coins <<
        " coins in round 1.\n";
    int wager = 0;
    if (coins > 0)
    {
        int maxSecond = min( coins, 100 );
        do {
            cout << "You can bet up to " << maxSecond <<
                " coins in round two.\n";
            cout << "How many coins do you want to bet? ";
            cin >> wager;
            if ((wager < 0) || (wager > maxSecond))
                cout << "That value is out of the range,"
                    << " please try again.\n";
        } while ((wager < 0) || (wager > 100));
    }
    coins -= wager;
    winnings = 0;
    for (i=0; i < wager; i++)
        winnings += Fruit_Market.Play_Quarter(0);
    coins += winnings;
    cout << "After round 2, you have " << coins <<
        " coins.\n";
}
```

```
void Player::Play_Slots(){int
winnings = 0;int i;for (i=0; i
< coins; i++)winnings +=
Fruit_Market.Play_Quarter(0);co
ins = winnings;cout <<
playersName << " you have won "
<< coins << " coins in round
1.\n";int wager = 0;if (coins >
0){int maxSecond = min( coins,
100 );do{cout << "You can bet
up to " << maxSecond << " coins
in round two.\n";cout << "How
many coins do you want to bet?
";cin >> wager;if ((wager < 0)
|| (wager > maxSecond))cout <<
"That value is out of the
range,"<< " please try
again.\n";}while ((wager < 0)
|| (wager > 100));}coins -=
wager;winnings = 0;for (i=0; i
< wager; i++)winnings +=
Fruit_Market.Play_Quarter(0);co
ins += winnings;cout << "After
round 2, you have " << coins <<
" coins.\n";}
```

It should be obvious that the function with additional space is easier for us to read. Since the computer doesn't care how code is entered, but a person reading your program will, it is best to use as much additional space as necessary.

Comments are an additional aid that help someone reading your program understand what you are trying to do. This is especially important if that person has the responsibility of making a change to your work. If that person does not understand what you have done, any changes that they make may either be wrong or may have additional consequences that are completely unexpected. You should understand that this person may also be you at some point in the future, and even though you clearly understand your code when you have finished writing it, you are not likely to once you have moved on to a new project. Choosing names for the parts of your program is also important. If you choose names that are meaningful to the task at hand, the resulting program will be easier to understand. If, however, you regularly use names like "x" or "y" or "temp," the person reading your program will have to struggle to connect these names to what is being done.

## Naming conventions

For the computer, variables are merely locations in memory where information is held. The computer keeps track of these locations so that it properly loads and stores the values that other statements in the program calculate. Because of this, the computer doesn't care what names you give your variables, since it will translate them into these memory locations. In a similar way, function names are just addresses to transfer control to, and class names are just an elaborate combination of variables and functions.

So, why is it important that you give variables, functions and classes names better than A1, A2, X, Y, and temp? It's because those names make a program very hard to follow when another programmer is trying to read what you have written.

How do you choose good names? Just think about what it is that the function is doing, what the variable is storing, or what the class is representing. Since functions do things, their names are probably going to be action words and verb phrases. Since variables hold information, their names are probably going to be noun phrases. Classes are used to describe groups of objects that have some characteristic, so their names are probably going to be the names of objects.

A couple of warnings are in order about variable names. First, you should try to avoid one letter variable names, except perhaps for loop counters. You should always avoid the letter 'l' as a variable name, because it is very difficult to tell the difference between it and the number '1' when you are looking at code. In fact, there has been a lot of time spent looking at code that has an error only to find that what appeared to be a letter 'l' in an equation or array index was actually the number '1.' Second, you should try to avoid the temptation to use the variable name temp, since it becomes habit forming. There have been a number of pieces of code written that need a temporary variable, so one gets created called temp, and then another is needed, so it becomes temp2, and then temp3, and then temp4. To make matters worse, these temp variables may actually be of different types as well, making the difficult situation of telling them apart even harder, as they are used to store different kinds of information. If you need a temporary variable, give more details in its name by taking on another word like in "tempAnswer."

Some people have a difficult time at first coming up with names. With experience it will become easier. Just remember to name it what it is. You may wind up coming up with very long names in the beginning, but with time and experience you will quickly learn of ways to shorten the names but still keep all of the meaning in them. The following chart gives you examples of some names for functions, variables, and classes:

Function Names		Variable Names		Class Names	
CheckValue	GetInput	myTotal	theAnswer	Building	Record
ShowCount	GreaterThan	count	location	Company	Employee
InRange	FindMax	numberInside	firstZero	Tree	Book
SortValues	Display	tempTotal	tempCard	SlotMachine	CardDeck
DealCard	CountSuccess	numLeft	change	Song	Animal

One thing that you might have noticed is that there is a definite convention or style used for all of these names. It turns out that you can convey information at all sorts of levels including how and where you use capital letters in names. You will notice that function and class names both start with a capital letter and variables start with a lower case letter. This is on purpose, so that when someone looks at a name in a program, if it starts with a capital letter and is a noun, the reader knows it is likely a class name, but if it is a verb phrase, the reader knows it is likely a function. If it starts with a lower case letter, the name is likely a variable. In all cases, however, notice that capital letters are used to identify where second "words" in a name start. This is because it is difficult to read and understand the name "norings" where "noRings" is easier.

The one category of names we have not yet discussed are constants. These are like variables, but their value does not change through out the program. Constants are frequently used for mathematical values (like  $PI = 3.1415$ ), or for things that don't change during the execution of a program (like MAX for the size of a list of values). You will notice that constants are typically given names that are all capital letters. If you have a constant name that is composed of two words, you can use an underscore to separate them as in the example "MINIMUM\_SIZE."

Warning: Naming is the one place where there is wide divergence in style. For example, some programmers dislike capital letters enough that they will instead separate words with underscores, so that “theAnswer” becomes “the\_answer.” These same programmers will not use capital letters for function or class names but expect that the context in which they are used will help the reader differentiate them. You should not use capitalization carelessly by having two item’s names differ only in whether or not a letter is capitalized.

## Indentation and spacing

As was mentioned, indentation and spacing is important to increasing the readability of your code. This section will give guidelines for how to indent and space the various control structures so that it is obvious what the relationship is between various parts of your program.

All of the indentation below is based on tabs, but the important issue is not how much the indentation is, but rather that it makes clear the nesting of the parts. The amount of indentation can be as small as two or three spaces or as large as a tab (typically the same as eight spaces). The trade off is that by typing spaces for indentation, you might have to type a lot of spaces if the indentation gets deep. If instead you use the tab key, each level of indentation requires just one additional tab, however, as the indentation gets deep, the code gets shifted a lot and so the individual statements may have to be broken over two lines. This trade off is seen here:

```
while (!done)
{
    if (x > y)
        cout << "x > y where x = "
            << x << " and y = "
            << y << endl;
    else
    {
        int temp = x;
        x = y;
        y = temp;
    }
}
```

```
while (!done)
{
    if (x > y)
        cout << "x > y where x = " << x
            << " and y = " << y << endl;
    else
    {
        int temp = x;
        x = y;
        y = temp;
    }
}
```

Another issue in spacing is where to put spaces within statements and blank lines within functions. To make code more readable, a space should be put on both sides of all binary operators including assignment, arithmetic, comparison, and stream insertion/extraction operators. A general rule would be “when in doubt, add a space.” Extra lines should be put in a function after declarations of local variables, after significant blocks of code, and anytime the function begins to do something different. So, if a function first totals a set of numbers, and then calculates the average, there should be at least one blank line after the code that totals the numbers and before the code that calculates the average.

### Statements and blocks

A block is a group of statements that are logically grouped. These statements can be those inside a function, a loop or part of a selection statement. A block is started by an open curly brace { and is ended by a close curly brace }. All of the statements inside a block should be indented to the same point. The curly braces for the block should be lined up vertically. If there are nested blocks (i.e. one block inside another), the inner block should be indented more than the containing block.

If a statement is too long, it must be broken over two or more lines. If a statement goes too far to the right of the screen, it may go too far for a printer to output it and so part of the line will not appear on the page. A printer can usually print about 60 characters across a page. You will eventually get a feel for when a statement needs to be broken, but you will only learn this if you look carefully at the files you print. It is surprising how many programs have been turned in with significant parts of the code lost because it was cut off by the printer.

It is also important to pick a good point within a statement to start the new line. All computer languages specify that the new line cannot start in the middle of a variable name or a string of characters. The new line should ideally start with an operator so that at a first glance it is noticeable that this is a continuation of the previous line.

The following function is an example of all of these guidelines put together:

```
void
Sample()
{
    int    theCount, theAmount, theAverage;
    char   usersAnswer;

    cout << "Are you ready to input the data? ";
    cin >> usersAnswer;
    if ((usersAnswer == 'y') || (usersAnswer == 'Y'))
    {
        cout << "What is the count of numbers"
              << " that you have found ? ";
        cin >> theCount;

        cout << "What is the total amount of "
              << "those numbers ? ";
        cin >> theAmount;

        theAverage = theAmount
                    / theCount;

        cout << "The average is "
              << theAverage << endl;
    }
}
```

## Declarations

In C++, declarations of variables can be anywhere within a function or even for that matter within a file. This means that if a variable is only needed inside a loop, it can be declared at the top of the statements inside the loop. This is helpful because this variable is only usable in the loop and so nothing outside of the loop can change its value causing a problem. It is also helpful, because if you need a variable with a similar name in two different places, you can localize their use, making the program easier to read.

This greater flexibility can, however, create a problem if a programmer just declares variables as they are thought of, spreading the declarations out for no apparent reason. To prevent this, it should be standard that all variables should be declared at the top of the block within which they are needed. If a variable is needed inside a function, it should be declared at the top of the function before any executable code appears. If a variable is just needed inside a loop, it should be declared at the top of the loop just after the for, do, or while statement. C++ has a construct that allows counter variables in for loops to be defined only during the for loop's execution. This is done by giving the declaration inside the for statement as follows:

```
for (int i = 0; i < MAX_SIZE; i++ )
{
    declaration_of_variables;

    loop_statements;
}
```

In the above case, the variable “i” is created when the loop is started and it is destroyed when the loop completes. Any variables declared at the top of the loop will be created when the loop is started and will be destroyed when the loop completes.

A similar thing can be done with variables needed inside the then or else parts of an if statement. These variables should be declared at the top of the part’s block and will be created when that group of statements start and will be destroyed when the if statement completes.

### If statements

For an if statement, the word if and any following else should line up. The statements in the “then” and “else” parts should be indented. If there is a block of statements in either the “then” or “else” parts, the opening and closing curly braces should line up with the word if. The following are three examples of how if statements should be indented: (note: some people feel that an if statement is easier to understand if the then part has fewer statements than the else part. It is necessary, however, to be very careful while rewriting the if statement to achieve this, and the condition is then sometimes not as clear.)

```

if (condition)
    then_statement;
else
    else_statement;

if (condition)
{
    then_statements;
}
else
    else_statement;

if (condition)
    then_statement;
else
{
    else_statements;
}

```

In some cases, there are a set of conditions that are checked in nested if statements. These are called “waterfall” if statements. Below shows two ways of indenting these statements, and though both are acceptable, most people prefer the one on the left.

```

if (condition_1)
    then_statement_1;
else if (condition_2)
    then_statement_2;
else if (condition_3)
    then_statement_3;
else if (condition_4)
    then_statement_4;
else if (condition_5)
    then_statement_5;
else
    else_statement;

if (condition_1)
    then_statement_1;
else
    if (condition_2)
        then_statement_2;
    else
        if (condition_3)
            then_statement_3;
        else
            if (condition_4)
                then_statement_4;
            else
                if (condition_5)
                    then_statement_5;
                else
                    else_statement;

```

### Switch statements

Switch statements allow a single check to select from multiple choices. The indenting structure for a switch statement is as follows:

```

char    checkVariable;
      .
      .
      .
switch (checkVariable)
{
    case 'A': case 'a':
        case_statements_1;
        break;
    case 'B':
    case 'b':
        case_statements_2;
        break;
      .
      .
      .
    default:
        case_statements_default;
        break;
}

```

You should note that each case label is indented further than the switch statement, and the statements done as part of each case are indented even further. If there is more than one case that applies to a set of statements they can be listed together on one line or one multiple lines of equal indentation, but in practice, only one of these two options should be used. The set of statements for each case should end with a break statement that is indented to the same point as the set of statements. The last case of a switch does not need a break statement, but one should be included for completeness.

### For loops

For loops are used when we know how many times a set of statements needs to be executed. The indenting structure for a for statement is as follows:

```

for ( int i = 0; i < max_value; i++ )
{
    statement_1;
    statement_2;
    statement_3;
}

```

There are additional style issues that apply to a for loop. You should not change the value of the variables `i` or `max_value` inside the for loop. Doing so will alter the number of times this loop will execute and will make the code difficult to understand. A break statement placed in a for loop will stop the execution of the loop when that break is reached. This is not a good way to program a for loop, and a while loop should be used instead. For example, the while loop on the right is preferred to the for loop on the left.

<pre> int count = 0; for (int i = 0; i &lt; max_value; i++)     if ( list[ i ] &gt;= 0 )         count++;     else         break; </pre>	<pre> int count = 0; int i = 0; while ((i &lt; max_value)       &amp;&amp; (list[ i ] &gt;= 0)) {     i++;     count++; } </pre>
------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------

## While loops

While loops are used when we want to execute a set of statements as long as a condition is true. The indenting structure for a while statement is as follows:

```
while (first < last)
{
    statement_1;
    statement_2;
    statement_3;
}
```

If the condition of the while loop is just checking as a counter changes values, it might be more appropriate to use a for loop. The for loop on the right is preferred over the while loop on the left.

```
int i = 0;
int count = 0;
while (i < max_value)
{
    if (list[ i ] > 0)
        count++;
    i++;
}

int count = 0;
for (int i = 0; i < max_value; i++)
{
    if (list[ i ] > 0)
        count++;
}
```

## Do loops

Do loops function like while loops except that that condition is checked at the end of the loop instead of at the start. This means that do loops execute the statements inside the loop at least once, where a while loop might never execute the statements inside the loop. The indenting structure for a do loop is as follows:

```
do
{
    statement_1;
    statement_2;
    statement_3;
} while (first < last);
```

## Functions

All statements that are part of a function should be indented. The function type should be on the first line, and the function name and parameters should be on the second line. The curly braces that mark the beginning and end of the function should line up. All of these should be in the first column. The indenting structure for a function is as follows:

```
char
GetNextCharacter()
{
    char    theChar;

    cout << "Enter the next character: ";
    cin >> theChar;
    return theChar;
}
```

## Programming style

There are some rules of programming style that should always be followed. These are:

- All input requests from the user should have clear prompts so the user knows what is being requested. If possible or reasonable, these prompts should include either a sample input value or the range of valid values.
- All output should be labeled so the user knows what the values mean or represent.
- Break statements should be used sparingly. The only place where break statements are always acceptable is inside switch statements. In all other cases, you should see if the loop you are working on can be rewritten to eliminate the break. Excessive use of break statements makes the code difficult to understand.
- Return statements should ideally appear only at the end of a function. This way there is only one point where the function stops executing and that is at the bottom of the code. If a routine is small or if there are a number of obvious places for returns, then multiple returns can be used in a function. Multiple returns should always be a rare occurrence.
- There should be no implicit type casting (i.e. the assigning of a variable of one type to another). If the type cast is implicit, a reader of the code will not know if the type case was done on purpose or by accident. If the type cast is explicit, the reader knows the cast was meant. In the following example, the code on the right is preferred.

```
char x;          char x;
int y;          int y;

x = y;          x = char( y );
```

- Conditions should be assigned to variables directly instead of within an if statement. For example, the following pairs of code accomplish the same thing. The code on the right is preferred in each case.

```
if (first < last)
    result = TRUE;
else
    result = FALSE;

if (first < last)
    result = FALSE;
else
    result = TRUE;

if (first == TRUE)
    result = TRUE;
else
    result = FALSE;

if (first == FALSE)
    result = TRUE;
else
    result = FALSE;

if (first < last)
    result = (first < last);

if (first < last)
    result = !(first < last);

if (first == TRUE)
    result = first;
```

- Constants can be declared in two different ways by either a preprocessor declaration (left example) or by a C++ declaration (right example). Either is acceptable, though the preprocessor declaration is slightly preferable. No matter what style is used it should be used consistently throughout a program.

```
#define LIMIT          100                const int LIMIT = 100;
#define MAX_SIZE      25                  const int MAX_SIZE = 25;
#define PI             3.1415             const float PI = 3.1415;
```

## Internal comments

Internal comments are an important part of writing a program and should be done as the program is being written. These are designed to give the reader of your program additional information about what is happening in the program. These can also give information about why code has been written in a particular way. Comments can indicate to the reader what conditions are true at particular points in the code, or what a tricky or confusing piece of code is going to accomplish.

### File headers

Every file should begin with a comment block that gives important information about the rest of the file. This should also give information that helps the reader work with the file. This information includes, but is not limited to, the file name, the programmer's name, the date it was written, the dates it was modified on (if any) and what modifications were done, and a short description of the file contents. This file header should have the following format:

```
//
//   file:           sample.cxx
//   programmer:     Jeffrey J. McConnell, Ph.D.
//   date:           January 15, 1998
//   modified:       January 25, 1998 (function stuff added)
//
//   purpose:        To show what a sample file header looks like.
//
```

### Function headers

Each function should also have a header that is placed immediately next to the function in the file. This header can take the form of a detailed comment and be placed just after the first declaration line, or can be a more formal header and be placed before the entire function. This header comment includes information like the function name, the function's purpose, those parameters that are input, input/output, and output, and the return value (if any). A formal function header could have the following format:

```
//
//   function:       TotalValues
//   purpose:        total the values in a list of integers
//   parameters:
//       input:      list    the collection of integers
//                   count   used for the number of values
//   input/output:   none
//   output:         none
//   return value:   an integer representing the total of the list values
//
int
TotalValues( int list[], int count )
```

**Declaration comments**

There should be a comment at the end of most variable declaration lines. This is required of any significant variables in the function but is optional for loop counters and other limited purpose variables. Declaration comments should have the following format:

```
int    list[100];    // the set of values entered by the user
int    count;       // the number of values in list that are valid
int    total;       // used for the running total of the values in list
float  average;     // stores the average of the values in list
```

**Statement comments**

There should be comments for lines or sets of lines of code that accomplish the tasks necessary to complete the work of a function or program. These are especially necessary if the code is difficult to follow or does a complex process. The comments can be at the end of a single line or can appear before the line or lines.

```
// get the player's name
cout << "Enter player #" << num_players+1
    << "'s name (or \"done\" to end): ";
cin >> name;

// if we are not finished
if (strcmp(name,"done") != 0)
{
    // set up the player and have him/her play
    Player temp(name);
    temp.Play_Slots();

    // save the player's name and winnings
    PlayerList.store( num_players++, temp );
}
```

Comments should also be put before conditional statements and loops to explain what they are about to check. Comments should be put into the “then” and “else” parts of an if statement that explain the conditions that would cause the computer to execute those statements (related to the if condition). Comments should be put into loops, especially while and do loops, that explain what must be true for the loop to keep executing (the loop invariant).

```
// as long as the user enters positive numbers
// continuing reading them. A negative number
// means the user wants to stop.
do {
    cout << "Enter the next number (<0 to stop):";
    cin >> value;

    // check to see if this is still a valid value
    if (value > 0)
        // if we got here the value is valid so
        // add it to the total
        total += value;
} while (value > 0);
```

## Block Comments

If blocks are long or if a number of nested blocks all end at the same place, then comments are needed that help the reader relate the end curly brace to the start of the block. This is done by placing a comment after the close curly brace that gives the item being closed. This can either be simply the control statement at the start of the block (i.e. if, while, switch), or if there are a number of similar control statements this comment can also include the conditional as well. The following example shows what this would look like.

```
void
sample( int x, int y, int z )
{
    if (x == y)
    {
        while (y > z)
        {
            .
            .
            .
        } // end while
    }
    else // if !(x==y)
    {
        if (y == z)
        {
            .
            .
            .
        } // end if (y == z)
    } // end if (x == y)
} // end sample
```

## Declaration file comments

The declaration file for a class should have the standard file header at its start. In addition, there should be a comment before each function prototype that explains what the function does, what parameters it expects (if any), and what value it returns (if any). A example of this follows.

```

//      file:           random.h
//      programmer:    Jeffrey J. McConnell, Ph.D.
//      date:          September 15, 1997
//      purpose:       To generate a series of pseudo-random numbers
//                    using the mixed congruential method.
class MyRandom
{
public:
    // initializes the random number generator the parameter serves as the
    // starting seed value.  to get a random sequence each time this is used,
    // send in a "random" seed for example, the seconds of the system clock.
    MyRandom( int x = 0 );

    // returns a uniformly distributed random number in the range [0,1) with mean 0.5
    float uniform();

    // returns a uniformly distributed random number in the range [low,high) with
    // mean (low+high)/2
    float uniform( int low, int high = 1 );

    // returns a normally distributed random number in the range [0,1) with mean 0.5
    float normal();

    // returns a normally distributed random number in the range [low,high)
    // with mean (low+high)/2
    float normal( int low, int high = 1, int mode = 10);

private:
    // the seed is used as the basis for the next pseudo-random number generated
    int      seed;
};

```

## External Documentation

Program code and internal comments are not enough for someone who is trying to understand what a program does, and how it does it. There are two external documents that must accompany a program - a user's and system's manual. (Your instructor will specify which are necessary.) The user's manual is written for the person who will use the program, and so should give the information necessary to successfully run the program. The system's manual is written for a person who will change the program, and so should give information about what was done and why it was done.

### User's Manual

When you buy a commercial software package, you expect that you will get more than just a disk with an executable program. You expect that you will also get information on how the program works. This documentation can be in paper form or in electronic form. Electronic forms of documentation can either be separate or can be built in to the program as a help function. But the bottom line, in any case, is that the manufacturer is expected to provide information on how the program should be used.

The user's manual should have the following sections:

- 1) Description: This section should give a brief description of the purpose of the program.
- 2) Executing the program: This section should give details about how to start and run the program.
- 3) Input: This section should give details about what kind of information will be needed, what form it should take, the order it will be needed, and any restrictions on its values. It is helpful to have a sample of the computer/user interactions to make the text easier to understand. There should be a subsection giving any error messages that the program

might output based on the input values entered. This subsection should also give instructions on what the user should do to correct those errors.

- 4) Output: This section should give information on the output the program produces. This should include the details of what is output, and where necessary, give information that helps the user interpret and understand the output presented.

This list is not meant to be exhaustive and there may be circumstances where additional sections are necessary or appropriate.

### **System's Manual**

One of the largest tasks in the life cycle of a computer program is maintenance - making improvements and enhancements to computer programs that extend their usefulness. Those programmers who are responsible for this work are dependent on both the internal comments and the system's manual for providing the information necessary to understand how the program was written. A complete understanding of the program is necessary if the changes that are made create only the result wanted and not any other damaging side effects. For this to happen, the documentation must give the maintenance programmer enough information so that he or she feels confident in the changes to be made.

It is understood that when you write some parts of the system's manual it will seem quite strange since you are expected to "explain your choices" when by the nature of academic exercises these choices were actually made for you. This will lessen in later courses, but for now, this places a burden on you to develop a justification for choices you did not make, and this also gives you an opportunity to justify alternatives that you may have noticed. You should be aware that the discussion of alternatives and how the program could be changed to use them is just as important and powerful of a component of these documents as the part that just tells what is in the program.

In the real world, and in an ideal academic setting, you would develop the system's manual before you write the code and then maintained it through the development process. In reality, this is hardly what happens. Students frequently see documentation as the thing you add on after the program works. It is commonly left to the last minute and the resulting product shows this. It would be best if the system's manual was developed in conjunction with the program.

The system's manual should have the following sections:

- 1) Description: This section should give a brief description of the purpose of the program.
- 2) Overall System Design: This section should give a description of the major classes in the program and discuss why those were chosen. There should be a discussion of how these classes interact as well.
- 3) Data Structure Choices: This section should discuss the major data structures used in the implementation of various classes and the main program. This discussion should include information about unusual choices and alternatives considered. If there are any special ways in which data structures are used, this must be discussed here.
- 4) Design Details: This section addresses the implementation choices that were made in writing the code that accomplishes the tasks the program and its objects do. This section can include stepwise refinements or algorithms.
- 5) Compiling Instructions: This section gives details on how the program needs to be compiled including any special compiler "flags."
- 6) Errors and Limitations: This section should give any limitations that the program has placed on the input, processing, or output. There should also be a listing of any errors that still exist in the program. This section must include a details discussion of how these limitations can be overcome, and how these errors can be found and corrected.
- 7) Suggested Improvements: This optional section discusses additional capabilities and improvements that can be included in future versions of the program.
- 8) Testing Information: Since testing is so important to the process of developing a good quality error-free program, it would be helpful to include information about how the program was tested both in pieces at the class level and as those classes were put together. This testing should illustrate both the nature of the tests and the values used in the testing process.

This list is not meant to be exhaustive and there may be circumstances where additional sections are necessary or appropriate.

## Appendix - Sample Program Description, Documentation, and Program

The following represents a sample program and its documentation developed based on a project description.

### CSC 212 - Data Structures Project Description

**Purpose:**

To refresh knowledge of C++ and to give students experience working with tables and linked lists.

**Description:**

Jane Bottom is the owner of Bottom's Up a saloon in Carson City, Nevada. In her establishment she has a number of slot machines that are very popular among her customers. She has one machine, the Fruit Market, that is rarely played, and she has thought about how she could get more people to play it. She has decided that she would like to hold a daily competition to see who can win the most money from this slot machine. She wants to set up the competition so that each person pays \$50 to enter the competition and receives \$25 in quarters (100 quarters). They play the 100 quarters and see how much they win. They then have the option of using up to 100 quarters of their winnings to play a second round. (Note: if they won less than 100 quarters, they can only play up to the amount they won.) They get to keep all of the quarters that they win during the competition, and the person who wins the most quarters gets the \$100 first prize as well. People will be allowed to play more than once a day, but have to pay the \$50 entry fee each time and the results of each of their entries is recorded separately as if it had been two different people.

Jane has contracted with you to write a program that will simulate this competition so that she can see if she will make money on it. She has provided information on her Fruit Market slot machine so that spins can be properly simulated.

**Details:**

To do this project you will need to create a class to keep track of each player's information and control their playing. You will also need to create a table of results and a list of the players ordered by the amount they have won. You will also need to print out some statistics for Jane to decide whether or not this competition will be profitable.

**Deliverables:**

You will, at the end of this project, turn in a hard copy of all of your code, a user's manual that outlines how to run your program, and an electronic copy of your project. Before using "submit" to turn in an electronic copy of your project please be sure to delete all executable programs, and delete all object files for modules that you have created. You can leave object modules for those pieces that I provided to you.

# User's Manual

## Introduction

The purpose of this program is to simulate a slot machine contest, and was prepared for Jane Bottoms, owner of the Bottoms Up Saloon, in Carson City, Nevada. The premise is that Ms. Bottoms is interested in starting a contest on one of her slot machines, and is interested in seeing if this contest will be profitable. The rules she has set up are as follows:

- Each person will pay a \$50 entrance fee to play in the contest.
- People can play more than once, but must pay the entrance fee each time. Their winnings will be kept separate and will not be pooled to determine standings in the contest.
- People will receive 100 quarters, and must play all 100 through the slot machine once. After that first round, the player can choose to bet any portion of their first round winnings up to a limit of 100 quarters.
- The amount each person has won during each contest will be recorded, and at the end of the contest, the person with the largest number of quarters won will receive a bonus of \$100.

Ms. Bottoms has requested that this program not only simulate this contest, but that it also print a summary at the end that will show what the profit or loss of the contest would be. To help in the creation of this program, Ms. Bottoms has provided details of the operation of the Fruit Market slot machine that she intends on using for this contest.

## Setting Up and Running the Program

To run this program, the user should give the command **driver** at the system prompt. The program will then prompt for any and all input it will need. The following section will give the details on the format that must be followed in the specification of the input for the program.

If the executable program **driver** does not exist, it can be recreated by giving the command **make** at the system prompt. This will direct the computer to execute the commands necessary to create a new version of the **driver** executable. After this **make** command completes, the user can follow the procedure given in the previous paragraph to start the program.

## Input to the Program

There are two types of input to this program. The first is the name of the player. Since this is just a simulation, no effort has been made to allow for the full name of the player to be entered. This name is just a sequence of less than 40 characters that can be used to differentiate this “player” from the others. For the sake of simplicity, the user can even enter just a sequence of numbers in place of names.

There is one special name that the user needs to be aware of. If the word “done” is entered as a name, the program will interpret that as a signal that the user does not want to enter any additional players. The program will, therefore, proceed to print out summary information and then terminate when “done” is entered as the player name.

The second type of input is the number of coins the player should play in round two. This entry should be an integer or whole number. The prompt for this value will specify the acceptable range that will be something between zero (0) and the number of coins the player won in round one. If, however, the player won more than 100 coins in round one, the upper limit for the round two bet will be 100.

## Output from the Program

The output from the program includes the prompts for input data as well as information about the progress of the program. The input prompts include those that ask for the player’s “name” and that then ask for the number of coins to play in round two. These prompts will be repeated for each player that is entered into the system.

Other output from the program includes the number of coins won in rounds one and two. At the end of the execution of this program (when the user has entered “done” as the player name), the system will print a list of the players in decreasing order by winnings (figure 1) and summary information (figure 2) about the data that this execution of the simulation has produced.

```
And the winners are:  
1 Player: Fred won: 125 coins  
  
2 Player: Linda won: 115 coins  
  
3 Player: Jeff won: 85 coins  
  
4 Player: Barbara won: 45 coins
```

**Figure 1 - Final Player Winnings List**

```
There was a total of $327.50 paid to winners,  
and a total of $500.00 in entrance fees.  
The contest generated $172.50 in profit.
```

**Figure 2 - Overall Summary Output**

### **Error Messages and Corrections**

There is only one error message that the system will produce. This is when the user enters a bet for a player that is outside of the range. This can either be due to a value entered that is below zero (0) or above the maximum specified bet. The error message reads: “**That value is out of range, please try again.**” The system will then prompt again for the user to enter the number of coins for the player to bet in round two. This prompt gives the maximum bet, so to eliminate this error message, be sure to enter a bet that is not negative and that is less than or equal to the amount specified.

## Sample Execution

```
betelgeuse>driver
Enter player #1's name (or "done" to end): Jeff
Jeff you have won 85 coins in round 1.
You can bet up to 85 coins in round two.
How many coins do you want to bet? 25
After round 2, you have 85 coins.
Enter player #2's name (or "done" to end): Linda
Linda you have won 75 coins in round 1.
You can bet up to 75 coins in round two.
How many coins do you want to bet? 50
After round 2, you have 115 coins.
Enter player #3's name (or "done" to end): Fred
Fred you have won 120 coins in round 1.
You can bet up to 100 coins in round two.
How many coins do you want to bet? 30
After round 2, you have 125 coins.
Enter player #4's name (or "done" to end): Barbara
Barbara you have won 90 coins in round 1.
You can bet up to 90 coins in round two.
How many coins do you want to bet? 90
After round 2, you have 45 coins.
Enter player #5's name (or "done" to end): done

And the winners are:
1 Player: Fred won: 125 coins

2 Player: Linda won: 115 coins

3 Player: Jeff won: 85 coins

4 Player: Barbara won: 45 coins

There was a total of $192.50 paid to winners,
and a total of $200.00 in entrance fees.
The contest generated $7.50 in profit.
```

## Errors and Limitations of the Program

No errors or limitations have been found in this program. The program accomplishes all of the requirements of the project description. (Note: If, however, there were errors in the program, this section would note what they are and also what steps would be necessary to find and correct the errors.)

## Conclusion

This program meets all of the requirements that were specified by the customer, Jane Bottoms. This program can, therefore, be used to test to see whether it would be profitable to implement the slot machine competition that Ms. Bottoms has developed.

# System's Manual

## Introduction

The purpose of this program is to simulate a slot machine contest, and was prepared for Jane Bottoms, owner of the Bottoms Up Saloon, in Carson City, Nevada. The premise is that Ms. Bottoms is interested in starting a contest on one of her slot machines, and is interested in seeing if this contest will be profitable. The rules she has set up are as follows:

- Each person will pay a \$50 entrance fee to play in the contest.
- People can play more than once, but must pay the entrance fee each time. Their winnings will be kept separate and will not be pooled to determine standings in the contest.
- People will receive 100 quarters, and must play all 100 through the slot machine once. After that first round, the player can choose to bet any portion of their first round winnings up to a limit of 100 quarters.
- The amount each person has won during each contest will be recorded, and at the end of the contest, the person with the largest number of quarters won will receive a bonus of \$100.

Ms. Bottoms has requested that this program not only simulate this contest, but that it also print a summary at the end that will show what the profit or loss of the contest would be. To help in the creation of this program, Ms. Bottoms has provided details of the operation of the Fruit Market slot machine that she intends on using for this contest.

## Overall System Design

This program was designed with the goals of reasonable modularity and simplicity of modules. The modules developed cover the major objects that this program must use. Since this program is simulating a competition on slot machines, it is natural to have a class to model a slot machine. Since a competition has players in it, it is natural to have a class to model our players. The inclusion of a random number generator class is to separate this distinct function from that of its primary user the slot machine class. Not only was this separated out because its purpose was distinctly different from the slot machine, it was also separated out so that it would be available to other parts of the program should the design have changed during development or at any time in the future. For instance, one change that might be made is to simulate the choice of how many coins to play in round two. Since we do not have real players making these choices, the number of coins played in round two is arbitrary. So a future version of this program might just ask for a number of people to simulate playing in the contest, and then do all of the calculations after that point on its own. The program could then just assign player numbers, and then use the random number generator to pick the number of coins each of these "players" bets in round two.

So this design lead to the development of the above three classes. The resulting main program is rather lengthy within this design by the standards of object-oriented methodology. One enhancement that can be made in the future is the creation of a competition manager class that will take over the main operations of running the contest, converting the table of players into the winners list, and then printing out the winner's list and final statistics. By developing this additional class, the main program would then be reduced to a few calls to the functions in this manager class.

## Data Structure Choices

There are a number of choices made for data structures throughout this program. This section will look at these choices on a module basis.

### Main Driver Program

This program uses two major data structures in addition to objects of the Player class. The first of these is a one key table. As people play in this tournament, they are assigned a player number in consecutive order. On completion of their play, their information is inserted into this table using their player number as the key. The reason for doing this is to easily compile the information so that it is readily

retrievable. By keying the information by player number, this system could be the basis for a future record keeping program when this contest is implemented. By first storing the information in a table keyed by player number, we have created a design that could easily be enhanced to allow players to ask about their winnings when they last played, and for us to be able to quickly retrieve this information.

Since we need to print a winner's list, ordered by decreasing winnings, at the end of the execution of this program, it was decided to also use a linked list structure. This linked list is ordered by the amount of winnings of each individual player, and is constructed by inserting players into the correct order in this list. The algorithm that constructs this list is, therefore, based strongly on the standard insertion sort algorithm. The general process is to get the next player out of the table, and, starting at the front of the list, step through it comparing this new player's winnings to the winnings of those already in the list. When we find the first location where the winnings are less than those of the new player we stop and insert the new player in the list before the current one. In this way, as the list grows, it grows in order.

### Player

The only significant data structure choice related to the player class was the way in which the slot machine that the players use was declared. This slot machine was declared as a global variable in the definition file for the player class. It was done here so as to create one slot machine that all of the players would share. If this declaration had been in either the declaration of the player class or inside one of the player class functions this would have created a new slot machine for each player. Since the slot machine is dependent on the random number generator, and that random number generator always generates the same sequence of numbers, if we have a new slot machine for each player, each player will produce the exact same results. This would result in a very uninteresting program.

### Slot Machine

The slot machine has two arrays and three integer variables. The first of these arrays, **reelValues**, is an integer array of the values on the three reels in the slot machine. This array is designed with 32 stops or locations. Each location has a value of zero (0) through three (3), representing a blank (0), a cherry (1), an orange (2), or a plum (3). There is one plum, three oranges, 12 cherries, and 16 blanks on the reels. The blanks are located in the even numbered locations, and the fruit are all in odd numbered locations. The oranges are in locations 7, 23, and 31. The plum is in location 15. The cherries are in the remaining 12 odd locations. The slot machine we are simulating pays if either all three reels show some fruit symbol (5 coins), all three are cherries (10 coins), all three are oranges (50 coins), or all three are plums (200 coins).

The reels of the slot machine are simulated with this one array of values and three separate integer indices into this array. So, the three integers of **reel1**, **reel2**, and **reel3** represent what is showing on the three reels of the slot machine by being an index into the array **reelValues**. If the value of **reel1** was 4, the value of **reel2** was 23 and there value of **reel3** was 9, this would mean that the reels of the slot machine would be showing blank, orange, cherry. Spinning the wheels is represented by advancing these values (mod 32) some random number between 0 and 31. The array **reelSymbols** contains character strings that are used if the values of the reels are to be printed.

## Class Design Details and Choices

The previous sections detailed the choices for the data structures used in this program. Since it is impossible to separate a data structure from the algorithm used to manipulate it, part of that discussion also dealt with some of the design choices. The following sections will give additional details on algorithm design details and choices used throughout this program.

### Player

The Player class is designed in a straight forward way. The player's name and number of coins are stored as part of this class. These values are initialized in the constructor function which also sets the number of initial coins to 100. Since the rest of the functions of the class use the value stored in this variable for their processing, if the number of initial coins changes for any future versions of this program, you will only need to change the constructor for this update.

The final amount of coins are determined by the `Play_Slots` function. This function implements the contest rules as specified by Jane Bottoms. This function will run through the coins the player has for the first round with calls to `Play_Quarter` (`SlotMachine`) inside a for loop. Once the winnings for round one are determined, the function determines what the maximum bet is for the second round as the maximum of 100 and the coins won in round one. This is then used to verify that the user input of the number of coins to bet in round two is correct. A second for loop is used to play the coins bet in round two.

### Slot Machine

The locations of the fruit (as described in the data structures section) was very carefully chosen for two purposes. The first was to spread the different fruit out relatively evenly throughout the reels. The second choice of the odd locations was done so as to make the code to check for a reel showing a fruit simpler. Because the fruit are all in odd locations, if we take the value of a reel mod 2 and the result is a 1, a fruit is showing. So we can do a simple check of

```
( (reel1 % 2) & (reel2 % 2) & (reel3 % 2) )
```

and if this is true we have a winner of some sort, and only need to check to see if what is on the three reels are all the same to see how big of a win it is.

### Random

There are a number of ways to generate a sequence of pseudo-random numbers within a computer program. These numbers are not truly random, since they follow a sequence created by a mathematical calculation. Given a good mathematical calculation we can, however, create a sequence of numbers that is relatively close to random for our purposes.

The three most common ways to create a pseudo-random number sequence is by the linear congruential method, the multiplicative congruential method, and a mixed congruential method that combines these two. The mixed congruential method is used in this program and is based on the equation:

```
seedi+1 = (seedi*multiplier + increment) mod limit
```

In the linear congruential method the multiplier would be one (1) and in the multiplicative congruential method the increment would be zero (0). In all of these cases, care must be taken in setting the multiplier, increment, and limit. It is felt that all of these values should be relatively prime (in other words, none should have any factors in common). Also it is best if the limit is very large, because it indicates the cycle of the random number generator. If the values are all relatively prime, and the limit is 10, this means that the random number sequence will repeat after just 10 values. If the limit is increased to 100, the sequence will repeat after 100 values. So, you can see the importance of a large limit to producing a sequence that is not likely to repeat. For this program, the multiplier is 25173, the increment is 13849, and the limit is 65536.

The only remaining concern is the initial seed value. Currently, the random number generator is always seeded with the value zero (0). This was done so that the program generated results that were reproducible each time it was run. This does not produce interesting results since you get the same answer each time. It is common to use some part of the system clock as an initial seed setting, so that each time the program is run the seed will be different and so the results the program produces will be different. This is one enhancement that could be made in future versions of this program.

### Library Classes

There are a set of “library classes” that were supplied with the book *Understanding Program Design and Data Structures with C++* by Kenneth A. Lambert and Thomas L. Naps (West Publishing

Company, 1996). This project uses the classes `association`, `one_key_table`, and `linked_list` from the collection supplied by that text.

The class `linked_list` is a standard linked list structure that includes the operations of insert, retrieve, remove, and next as well as the informational functions of empty, length, and at\_end. In this program, we create a linked list of type `player` that is used to keep the list of players ordered according to decreasing winnings. Details on the use of this linked list are given above.

The class `one_key_table` is a standard keyed table that includes the operations of store, retrieve, and remove as well as the informational functions of empty and length. In this program, we create a one key table of players that is keyed on a sequential player number. Details on the use of this keyed table are given above.

The class `association` is indirectly used when we use the class `one_key_table`. This table is actually a table of associations where each association brings together the key and the element of the `one_key_table`. So in our program, an association links together the integer player number (key) and the player information (element) so that they can be stored together in the table. The class `association` includes an operation to set the value stored in an association as well as informational functions that return the key and value stored in the association. Since this class is hidden in the implementation of a `one_key_table`, and, therefore, is not directly used by this program no further information is provided here.

## Test Program Design

This section will give details on the test programs that were used during the development of this program. This section is sometimes omitted from a System's Manual but is included here to give further information on test program design and use.

### Random

The random number generator was tested in two ways. The first test was to check that the numbers being produced were truly uniformly random. This program initialized a set of 100 counters to zero (0). It then generated a set of 100,000 random numbers between 0 and 100, and counted how many times each of these integer values was generated. These values were then printed. If this random number generator is working, the number of times each of these values is generated should be approximately the same.

The second test was to see if the other functions in the class seemed to be working. A test program was created that made calls to the other functions of the class and their results were printed. This gave the ability to make a visual check of the results of the function. It should be noted that this test program is not a very strong test, and it would have been better to have created a program similar to the first that tested the normal function in a similar way.

### Slot Machine

Statistical analysis of the number of symbols on the three wheels and the pay off schedule indicated that over the long term this slot machine should return about 85% of the money that is bet. To test the slot machine, a program was created that will just repeatedly call the `Play_Quarter` function and total up the winnings. It then prints the number of trials and the amount that would have been won. If this is working correctly, the amount won should be approximately 85% of the number of trials.

### Player

The test for a player was quite simple, since the player class is so dependent on the slot machine class. This test program then just needed to make sure that the player class used the slot machine properly, and that the data provided to it was properly saved. This test program just declares a player and then has that player play the slot machine. This test program is not complete since it does not test the winnings function.

## Compiling Instructions

All compiling instructions, including dependencies among the various modules, are supplied in the **makefile** included with this distribution. This allows the programmer to just give the command **make** and the system will review all date and time stamps and recompile any files that have changed or that are dependent on other files that have changed. As changes are made to this program, it is recommended that this **makefile** be kept up-to-date.

### Special Template Instructions

Since this program uses the template feature there are special compiler options that must be specified. With each template, we must be sure that there are the proper instances of these template classes created. This must be done by specifying the types that will actually be used for each template class. This is done by compiling a file that has a specification of the following form:

```
template class class_name<type {, type}>;
```

The actual use of this specification will replace **class\_name** with the name of the class that we are creating instances of, and **type {, type}** represents the fact that one or more types that this template class requires must be specified. For example, to create the needed instance of the `one_key_table` used in this program, we need to specify the statement:

```
template class one_key_table<int, Player>;
```

within our program.

These template instantiation specifications can be in their own file or at the bottom of the definition file for the class. It is most common to place them at the bottom of the proper definition file.

The last special requirement for compiling template classes that have the instantiation specification in them is to include a special compiler option when those files are compiled. This option is the **no-implicit-templates** flag. A sample compile command that includes this flag is:

```
g++ -c -fno-implicit-templates onetable.cxx
```

The **makefile** provided has the proper compilation instructions for the project as it was submitted at its initial completion.

## Errors and Limitations

No errors or limitations have been found in this program.

## Conclusion

The program produced achieves the results of a reasonable modular design and simplicity of design. This program also achieves all of the requirements of the customer, and, therefore, provides her with the tools that she needs to test her slot machine contest concept. This document also details ways in which this program can be improved and enhanced.

## Program Code

### Driver

```
//
// file:          driver.cxx
// programmer:    Jeffrey J. McConnell, Ph.D.
// date:         October 15, 1997
//
// purpose:      This is the main driver program for the slot
//               machine contest simulator.
//
#include <iostream.h>
#include "player.h"
#include "onetable.h"
#include "linklist.h"

main()
{
    // the table for the players as they play
    one_key_table<int, Player>  PlayerList;

    // the list for the ordered list of winners
    linked_list<Player>        WinnersList;

    char  name[40];
    int   num_players=0;

    do {
        // get the player's name
        cout << "Enter player #" << num_players+1
              << "'s name (or \"done\" to end): ";
        cin >> name;

        // if we are not finished
        if (strcmp(name,"done") != 0)
        {
            // set up the player and have him/her play
            Player  temp(name);
            temp.Play_Slots();

            // save the player's name and winnings
            PlayerList.store( num_players++, temp );
        }
    } while (strcmp(name,"done") != 0);

    Player new_player;

    // put the first player into the winner's list
    PlayerList.retrieve( 0, new_player);
    WinnersList.insert(new_player);

    for (int i=1; i<num_players; i++)
    {
        PlayerList.retrieve( i, new_player );
    }
}
```

```

// put into the winner's list
WinnersList.first();
Player temp;
boolean done = FALSE;
// look through the list of people on the list
for (int j=0; j < WinnersList.length(); j++)
{
    WinnersList.retrieve(temp);
    // does the current player on the list
    // have less money than the new player.
    if (temp.Winnings() < new_player.Winnings())
    {
        // yes so put the new player before
        // and stop
        WinnersList.insert(new_player);
        done = TRUE;
        break;
    }
    // no so try the next player on the list
    WinnersList.next();
}
// has the new player been put in the list yet??
if (!done)
    // no, they must go at the end
    WinnersList.insert(new_player);
}

// Output the winner's list
int totalWinnings = 0;

cout << "\n\nAnd the winners are:\n";
WinnersList.first();
for (int i=0; i<num_players; i++)
{
    Player temp;

    WinnersList.retrieve(temp);
    cout << i+1 << " " << temp << '\n';
    totalWinnings += temp.Winnings();
    WinnersList.next();
}

int moneyIn = num_players*50;
float moneyOut = (totalWinnings/4.0) + 100;
cout << "\n\nThere was a total of $" << moneyOut
    << " paid to winners,\n";
cout << "and a total of $" << moneyIn << " in entrance fees.\n";
cout << "The contest generated $" << moneyIn-moneyOut
    << " in profit.\n";
}

```

**Player**

```

//
// file:      player.h
// programmer: Jeffrey J. McConnell, Ph.D.
// date:      October 15, 1997
//
// purpose:   declaration of the player class
//
#include <iostream.h>

class Player
{
    friend ostream& operator << (ostream& s, Player p);

public:
    // a constructor used when temp objects are declared
    Player( );
    // the main constructor called for each new Player
    // object that will be significant in the program
    Player( char* name );
    // the routine that will control the player's
    // play for both rounds 1 and 2
    void Play_Slots();
    // returns how many coins the player has
    int Winnings();

private:
    char  playersName[40];
    int   coins;
};

//
// file:      player.cxx
// programmer: Jeffrey J. McConnell, Ph.D.
// date:      October 15, 1997
//
// purpose:   Controls the behavior of a slot machine player
//
#include <iostream.h>
#include "SlotMachine.h"
#include "player.h"

// define the function min that returns the smaller of its
// two parameters
#define min(x,y) ((x)>(y)?y:x)

// define a slot machine global to this file so that all of
// the players share this same slot machine.  If this
// was declared in the player class or in any of the
// player functions each player would start in the same
// place and so all the results would be the same.

```

```

SlotMachine  Fruit_Market;

Player::Player( )
// no parameter constructor doesn't need to do anything
// its only used when temporary player objects are declared
{
}

Player::Player( char* name )
// this constructor needs to same the player's name and
// set the starting number of coins to 100.
{
  strcpy(playersName, name);
  coins = 100;
}

void
Player::Play_Slots()
// this function actually plays the slot tournament as
// defined in the project description.
{

  int winnings = 0;

  // play the first set of coins through the machine once
  int i;
  for (i=0; i < coins; i++)
    winnings += Fruit_Market.Play_Quarter(0);

  coins = winnings;

  cout << playersName << " you have won " << coins
    << " coins in round 1.\n";

  int wager = 0;

  if (coins > 0)
  {
    // find out which is smaller the number of
    // coins the player currently has, or 100 the limit
    // set in the project description. This is done so
    // we only need to compare to maxSecond instead
    // of the number of coins and the limit.
    int maxSecond = min( coins, 100 );

    do {
      // get the second round wager
      cout << "You can bet up to " << maxSecond
        << " coins in round two.\n";
      cout << "How many coins do you want to bet? ";
      cin >> wager;

      // is the wager within bounds?? if not, put out and
      // error message and have the user try again.
      if ((wager < 0) || (wager > maxSecond))
        cout << "That value is out of the range,"

```

```
        << " please try again.\n";

    } while ((wager < 0) || (wager > 100));
}

// we've got a good wager so reduce the coins by it,
// and then play them through the machine.
coins -= wager;
winnings = 0;

for (i=0; i < wager; i++)
    winnings += Fruit_Market.Play_Quarter(0);

// update the player's coins and output the results.
coins += winnings;

cout << "After round 2, you have " << coins << " coins.\n";
}

int Player::Winnings()
// returns the current number of coins for this player.
{
    return coins;
}

ostream& operator << (ostream& s, Player p)
// output information for this player
{
    s << "Player: " << p.playersName << " won: " << p.coins << " coins\n";
}
```

**Slot Machine**

```

//
// file:      Slot_machine.h
// programmer: Jeffrey J. McConnell, Ph.D.
// date:      October 15, 1997
//
// purpose:   declaration of the SlotMachine class
//
#include "random.h"

#define stops 32

class SlotMachine
{
public:

    // set up the slot machine member data and
    // initialize the random number generator it uses.
    SlotMachine();

    int Play_Quarter(int display);    /* if display = 1 the reels and      */
        /* winnings are printed. if */
        /* display=0, nothing is printed*/
        /* in both cases the winnings */
        /* are returned.              */

private:
    int    reel1, reel2, reel3;
    MyRandom RNG;
    char*  reelSymbols[4];
    int    reelValues[stops];
};

// SlotMachine.cxx
// Jeffrey J. McConnell, Ph.D.
//
// This routines handle objects of the class
// SlotMachine.

#include <iostream.h>
#include "SlotMachine.h"

// specify the locations of the fruit symbols
// the cherries go into all other odd locations
// the odd locations were used so that a mod
// by 2 would give a 1 or true if there was
// a symbol displayed. This makes determining
// a winner easier.
#define orange1 7
#define orange2 23
#define orange3 31
#define plum1 15

```

```

SlotMachine::SlotMachine()
// this sets up the data needed to control
// the play of the slot machine.
{
    int i;

    // set up the position of the wheels
    // as numbers between 0 and 31
    reel1 = (int) RNG.uniform(0,32);
    reel2 = (int) RNG.uniform(0,32);
    reel3 = (int) RNG.uniform(0,32);

    // now put in numbers that represent
    // what is on the wheels.
    // there are blanks in the 16 even spots,
    // cherries in 12 of the odd spots, oranges
    // in 3 of the odd spots, and a plum in one
    // odd spot.
    for (i=0; i<stops; i=i+2)
        /* blanks into the even spots */
        reelValues[i] = 0;
    for (i=1; i<stops; i=i+2)
        /* cherries into the odd spots */
        reelValues[i] = 1;
    /* put in the oranges */
    reelValues[orange1] = 2;
    reelValues[orange2] = 2;
    reelValues[orange3] = 2;
    /* put in the plum */
    reelValues[plum1] = 3;

    // the following arrays are used when output
    // of the wheels is requested
    reelSymbols[0] = " ---- ";
    reelSymbols[1] = " CHERRY ";
    reelSymbols[2] = " ORANGE ";
    reelSymbols[3] = " PLUM  ";
};

int SlotMachine::Play_Quarter( int display )
// this routine will spin the wheels once and
// return the amount of money won.
//
// if display = 0 nothing will be output to the screen.
// if display = 1 the wheels and amount won will be output.
{
    /* spin the reels */
    reel1 = (int) (reel1 + RNG.uniform(1, 33)) % 32;
    reel2 = (int) (reel2 + RNG.uniform(1, 33)) % 32;
    reel3 = (int) (reel3 + RNG.uniform(1, 33)) % 32;

    /* calculate the winnings */
    int winnings = 0;
    /* if all reels are odd, winnings are at least 5 */
    if ((reel1 % 2) & (reel2 % 2) & (reel3 % 2))
        winnings = 5;
}

```

```
/* check if the symbols are all the same */
if ((reelValues[reel1] == reelValues[reel2]) &
    (reelValues[reel1] == reelValues[reel3]))
{
    /* The following checks only need to look at one */
    /* reel since if we are here we know they are all */
    /* the same value. */

    /* if they are all cherries make the winnings 10 */
    if (reelValues[reel1] == 1)
        winnings = 10;

    /* if they are all oranges make the winnings 50 */
    if (reelValues[reel1] == 2)
        winnings = 50;

    /* if they are all plums make the winnings 200 */
    if (reelValues[reel1] == 3)
        winnings = 200;
}

/* if display is set, output the results */
if (display)
{
    cout << reelSymbols[ reelValues[ reel1 ] ];
    cout << reelSymbols[ reelValues[ reel2 ] ];
    cout << reelSymbols[ reelValues[ reel3 ] ];
    cout << "\tYou win: " << winnings << '\n';
}

return winnings;
}
```

**Random**

```

//
// file:      random.h
// programmer: Jeffrey J. McConnell, Ph.D.
// date:      October 15, 1997
//
// purpose:   declaration of the random class
//
class MyRandom
{
private:

    int      seed;

public:
    // initializes the random number generator`
    // the parameter serves as the starting
    // seed value.  to get a random sequence each
    // time this is used, send in a "random" seed
    // for example, the seconds of the system clock.
    MyRandom( int x = 0 );

    // returns a uniformly distributed random number
    // in the range [0,1) with mean 0.5
    float uniform();

    // returns a uniformly distributed random number
    // in the range [low,high) with mean (low+high)/2
    float uniform( int low, int high = 1 );

    // returns a normally distributed random number
    // in the range [0,1) with mean 0.5
    float normal();

    // returns a normally distributed random number
    // in the range [low,high) with mean (low+high)/2
    float normal( int low, int high = 1, int mode = 10);

};

// random.cxx
// Jeffrey J. McConnell, Ph.D.
//
// This file has the routines to generate a sequence of
// random numbers either uniformly or normally distributed.

#include <stdlib.h>
#include "random.h"

#define modulus 65536
#define multiplier 25173
#define increment 13849

```

```

MyRandom::MyRandom(int x)
// initialize the seed for the random number generator
{
    seed = x;
};

float MyRandom::uniform()
// generate the next uniform random number
// that is >= 0 and < 1
// this has the side effect of updating seed
// for the next call of this function.
{
    seed = ((multiplier * seed) + increment) % modulus;
    return ((1.0*seed)/(1.0*modulus));
};

float MyRandom::uniform( int low, int high )
// generate the next uniform random number
// that is >= low and < high
// this has the side effect of updating seed
// for the next call of this function.
{
    seed = ((multiplier * seed) + increment) % modulus;
    float temp = ((1.0*seed)/(1.0*modulus));
    return ((high-low)*temp + low);
};

float MyRandom::normal()
// generate a random number that is normally
// distributed about 0.5 and is in the range
// >= 0 and < 1. The mode value determines
// the variance about 0.5 (the higher the mode
// the lower the variance).
{
    float total=0;
    int i;
    int mode = 10;
    for ( i = 0; i < mode; i++)
        total = total + uniform();
    float result = total / mode;
    return result;
};

float MyRandom::normal( int low, int high, int mode)
// generate a random number that is normally
// distributed about the mean of (low+high)/2 and
// is in the range >= low and < high. The mode value
// determines the variance about mean (the higher the
// mode the lower the variance).
{
    float total=0;
    int i;
    for ( i = 0; i < mode; i++)
        total = total + uniform();
    float temp = total / mode;
    float result = (high-low)*temp + low;
};

```

```
    return result;  
};
```

## Test Programs

### Test of Random 1

```
//
// file:          test_random1.cxx
// programmer:    Jeffrey J. McConnell, Ph.D.
// date:         October 15, 1997
//
// purpose:      This tests the distribution of random numbers produced by MyRandom

#include <iostream.h>
#include "random.h"

main()
{
    MyRandom generator( 0 );
    int          counters[100];

    // initialize the counters
    for (int i=0; i<100; i++)
        counters[i] = 0;

    // generate a lot of random numbers and record their distribution.
    for (int i = 1; i <= 100000; i++)
        counters[ int(generator.uniform()*100) ]++;

    // output the results
    for (int i =0; i<25; i++)
        cout << "count[" << i << "] = " << counters[i]
            << "\tcount[" << i+25 << "] = " << counters[i+25]
            << "\tcount[" << i+50 << "] = " << counters[i+50]
            << "\tcount[" << i+75 << "] = " << counters[i+75] << '\n';
}

```

### Test of Random 2

```
//
// file:          test_random2.cxx
// programmer:    Jeffrey J. McConnell, Ph.D.
// date:         October 15, 1997
//
// purpose:      This test the random number generator by simply generating
//              some numbers. This is not a great test, but does just make
//              sure the overall ranges of values produced are reasonable.

#include <iostream.h>
#include "random.h"

main()
{
    MyRandom generator( 0 );

    cout << generator.uniform();
    cout << '\n';
    cout << generator.uniform(1, 10);
    cout << '\n';
}

```

```

cout << generator.normal();
cout << '\n';
cout << generator.normal(1, 10);
cout << '\n';
cout << generator.normal(10, 20, 20);
cout << '\n';
cout << generator.normal(10, 20, 50);
cout << '\n';
}

```

### Test of Slot Machine

```

//
// file:          test_slot.cxx
// programmer:    Jeffrey J. McConnell, Ph.D.
// date:         October 15, 1997
//
// purpose:      According to a statistical analysis of the slot machine
//              simulated, it should return about 85% of the money it
//              takes in.  this program tests that.

#include <iostream.h>
#include "SlotMachine.h"

main()
{
    SlotMachine FruitMarket;
    int winnings = 0;
    int trials;

    cout << "How many trials? ";
    cin >> trials;

    for (int i=0; i < trials; i++)
        winnings += FruitMarket.Play_Quarter(0);

    cout << "after " << trials << " the winnings were " << winnings << '\n';
}

```

### Test of Player

```

//
// file:          test_player.cxx
// programmer:    Jeffrey J. McConnell, Ph.D.
// date:         October 15, 1997
//
// purpose:      This program tests the player class.

#include "player.h"

main()
{
    Player pl("Jeff");
    pl.Play_Slots();
}

```