Nicholas Colvin

C++

Final Project Pseudo Code

* I recommend you start with just the following methods, get those working and code, test and run additional methods one at a time: constructor, initialize, assignment operator, deep copy, to string and print.
* Next, I’d recommend you get all the other constructors done.
* After that, I’d recommend you get all the “to <data type>” methods done.
* It is **critical** that your main function is written so that it is very clear that **every** function you have written works. This means that you will likely be calling a large number of tests. Every constructor and every other function must have a test that very clearly explains what it does in text that is printed to the terminal when the test is run. Code that is not tested by your main function will **not** receive full points.

ToBool

blnReturnValue

// Get length

If (m\_pstrSuperString == “true”)

{

blnReturnValue = true;

}

// --------------------------------------------------------------------------------

// Name: ToShort

// Abstract: Convert to a short.

// --------------------------------------------------------------------------------

short CSuperString::ToShort() const

{

Int IntLength = strlen(m\_pstrSuperString)

int intValue = 0;

short srtTotal = 0;

intValue = std::atoi(m\_pstrSuperString);

// Not over integer limit?

if (intLengh <= 12)

{

// Yes, Convert to an int

intValue = std::atoi(m\_pstrSuperString);

}

// Less than the minimum or maximum for a short?

if (intValue < SHRT\_MIN)

{

// Yes clip to the smallest number possible

srtTotal = SHRT\_MIN

}

Else if (intValue > SHRT\_MAX)

{

// Yes, clip to the largest number possible

srtTotal = SHRT\_MAX

}

Else

{

srtTotal = intValue;

}

return srtTotal;

}

// --------------------------------------------------------------------------------

// Name: ToInteger

// Abstract: Convert to a Integer.

// --------------------------------------------------------------------------------

short CSuperString::ToShort() const

{

int intLength = strlen(m\_pstrSuperString);

int intIndex = 0;

long long llngValue = 0;

short srtTotal = 0;

// Not over long long limit?

if (intLengh <= 20)

{

// Yes, Convert to a long long

llngValue = std::atoll(m\_pstrSuperString);

}

// Less than the minimum or maximum for a Integer?

if (llngValue < INT\_MIN)

{

// Yes clip to the smallest number possible

intTotal= INT\_MIN

}

Else if (llngValue > INT\_MAX)

{

// Yes, clip to the largest number possible

intTotal= INT\_MAX

}

Else

{

intTotal = llngValue;

}

return intTotal;

}

// --------------------------------------------------------------------------------

// Name: ToLong

// Abstract: Convert to a Long.

// --------------------------------------------------------------------------------

short CSuperString::ToShort() const

{

int intLength = strlen(m\_pstrSuperString);

int intIndex = 0;

long long llngValue = 0;

long lngTotal = 0;

// Not over long long limit?

if (intLength <= 20)

{

// Yes, Convert to a long long

llngValue = std::atoll(m\_pstrSuperString);

}

// Less than the minimum or maximum for a long?

if (llngValue < LONG\_MIN)

{

// Yes clip to the smallest number possible

lngTotal = LONG\_MIN;

}

else if (llngValue > LONG\_MAX)

{

// Yes, clip to the largest number possible

lngTotal = LONG\_MAX;

}

Else

{

lngTotal = llngValue

}

return lngTotal;

}

// --------------------------------------------------------------------------------

// Name: ToFloat

// Abstract: Convert to a float.

// --------------------------------------------------------------------------------

{

int intLength = strlen(m\_pstrSuperString);

int intIndex = 0;

double dblValue = 0;

long lngTotal = 0;

// Not over long long limit?

if (intLength <= 20)

{

// Yes, Convert to a long long

dblValue = std::atof(m\_pstrSuperString);

}

// Less than the minimum or maximum for a long?

if (dblValue < FLT\_MIN)

{

// Yes clip to the smallest number possible

lngTotal = LONG\_MIN;

}

else if (dblValue > FLT\_MAX)

{

// Yes, clip to the largest number possible

lngTotal = FLT\_MAX;

}

else

{

lngTotal = dblValue;

}

return lngTotal;

}

// --------------------------------------------------------------------------------

// Name: operator+=

// Abstract: Appends original instance to the end of the current instance.

// --------------------------------------------------------------------------------

\*pstrStringToCopy

{

intOrginalArraySize = Length();

intPassedArraySize = strlen(pstrStringToCopy);

intNewArraySize = intOrginalArraySize + intPassedArraySize

char\* pastrNewString = 0;

intPassedIndex = 0;

intOriginalIndex = 0;

pastrNewString = new char[intNewArraySize + 1];

// Copy values from original string

for (intOriginalIndex = 0; intOriginalIndex <= intOrginalArraySize; intOriginalIndex += 1)

{

\*(pastrNewString + intIndex) = \*(m\_pstrSuperString + intIndex);

}

// Copy values from passed string

for (intPassedIndex = 0; intPassedIndex <= intPassedArraySize; intPassedIndex += 1)

{

\*(pastrNewString + intOriginalIndex) = \*(m\_pstrSuperString + intIndex);

intOriginalIndex += 1;

}

//Set end of string

\*(pastrNewString + (intOriginalIndex + 1)) = ‘\0’

// Delete old string

delete[] m\_pstrSuperString;

m\_pstrSuperString = 0;

// Set new string

m\_pstrSuperString = pastrNewString;

}

long FindFirstIndexOf(const char chrLetterToFind)

{

intReturnIndex = 0;

intIndex = 0;

// Loop through the string until a

for (intIndex = 0; intIndex < Length(); intIndex += 1)

if \*(m\_pstrSuperString+ intIndex) = chrLetterToFind)

intReturnIndex = intIndex;

}

boolean IsWhiteSpace(char chrLetterToCheck)

{

boolean blnIsWhiteSpace = false;

// Space

if (chrLetterToCheck == ' ') blnIsWhiteSpace = true;

// Tab

if (chrLetterToCheck == '\t') blnIsWhiteSpace = true;

// Carriarge return

if (chrLetterToCheck == '\r') blnIsWhiteSpace = true;

// Line feed

if (chrLetterToCheck == '\n') blnIsWhiteSpace = true;

return blnIsWhiteSpace;

}

int FindFirstIndexOf(const char\* pstrSubStringToFind)

{

char chrCurrentChar;

char chrPreviousChar = ;

char[50] chrCurrentWord;

intReturnIndex = 1;

// Ad litora torquent per conubia nostra inceptos himenaeos.

// Copy string

pastrSuperStringCopy = stringcopy()

// Trim String

Trim(string)

while(\*(pastrSuperStringCopy + intIndex) != 0)

chrCurrentLetter = \*(pastrSuperStringCopy + intIndex)

// Store a word

if (IsWhiteSpace(chrCurrentChar) == true & IsWhiteSpace(chrPreviousChar) == false)

\*(strBuffer + (intIndex + 1) ) = ‘\0’

If ( strcmp(strbuffer, pstrSubStringToFind) == 1)

break;

else

// strBuffer[0] = ‘\0’

// Store index of letter until a whitespace is found

if (IsWhiteSpace(chrCurrentChar) == false & IsWhiteSpace(chrPreviousChar) == true)

intFirstLetterIndex = (\*m\_pastrSuperString + intIndex)

chrPreviousChar = chrCurrentChar;

}

// Substring = Maecenas tempus, tellus eget condimentum

// m\_strSuperString = tellus!

int FindFirstIndexOf(const char\* pstrSubStringToFind, int intStartIndex)

{

intReturnValue = 0;

intBreakIndex = 1;

intIndex = 0;

intSubStringLength = strlen(pstrSubStringToFind);

blnFirstMatch = true;

ABCDEF

for (intIndex = intStartIndex; intIndex < Lenth(); intIndex += 1)

If (pstrSubStringToFind[intIndex] == m\_pstrSuperString[intIndex])

{

If (blnFirstMatch == true)

{

intReturnIndex = intIndex;

blnFirstMatch = false;

}

If (intBreakIndex == intSubStringLength)

{

Break;

}

Else

{

intBreakIndex += 1;

}

}

If (blnFirstMatch = false)

intBreakIndex = 1;

blnFirstMatch = true;

}

// --------------------------------------------------------------------------------

// Name: Reverse

// Abstract: Reverse the string

// --------------------------------------------------------------------------------

int intLength = Length();

int intSourceIndex = 0;

int intCopyIndex = 0;

char\* pstrCopy = new char[intLength + 1];

// Loop through the string backwards

for (intSourceIndex = intLength - 1; intSourceIndex >= 0; intSourceIndex -= 1)

{

\*(pstrCopy + intCopyIndex) = \*(m\_pstrSuperString + intSourceIndex);

intCopyIndex += 1;

}

// Terminate

\*(pstrCopy + intCopyIndex) = '\0';

return pstrCopy;

// --------------------------------------------------------------------------------

// Name: Left

// Abstract: Return characters from the string based on given count

// --------------------------------------------------------------------------------

const char\* CSuperString::Left(int intCharactersToCopy) const

{

int intLength = Length();

int intIndex = 0;

char\* pstrReturnString;

// Less than zero?

if (intCharactersToCopy < 0)

{

// Yes, Set return string to zero

pstrReturnString = new char[1];

\*pstrReturnString = '\0';

}

else

{

// No, Greater than string's length?

if (intCharactersToCopy > intLength)

{

// Yes, Clip to full string

intCharactersToCopy = intLength;

}

// Create copy

char\* pstrDestination = new char[intCharactersToCopy + 1];

// Copy Characters until limit is reached

for (intIndex = 0; intIndex < intCharactersToCopy; intIndex += 1)

{

\*(pstrCopy + intIndex) = \*(m\_pstrSuperString + intIndex);

}

// Terminate

\*(pstrCopy + intIndex) = '\0';

pstrReturnString = pstrCopy;

}

return pstrReturnString;

}

// --------------------------------------------------------------------------------

// Name: Substring

// Abstract: Return a substring based on given parameters

// --------------------------------------------------------------------------------

const char\* CSuperString::Substring(int intStart, int intSubStringLength) const

{

int intLength = Length();

int intSourceIndex = 0;

int intDestinationIndex = 0;

char\* pstrDestination;

char\* pstrReturnString;

// Invalid Input?

if (intStart < 0 || intSubStringLength < 0 || intStart >= intLength)

{

// Yes, return empty string

pstrReturnString = new char[1];

\*pstrReturnString = '\0';

}

else

{

// No, Will substring go over string length?

if (intSubStringLength + intStart > intLength)

{

// Yes, Clip

intSubStringLength = intLength - intStart;

}

// Allocate

pstrDestination = new char[intSubStringLength + 1];

// Set Starting point

intSourceIndex = intStart;

// Copy Characters

for (intDestinationIndex = 0; intDestinationIndex < intSubStringLength; intDestinationIndex += 1)

{

\*(pstrDestination + intDestinationIndex) = \*(m\_pstrSuperString + intSourceIndex);

intSourceIndex += 1;

}

// Terminate

\*(pstrDestination + intDestinationIndex) = '\0';

pstrReturnString = pstrDestination;

}

return pstrReturnString;

}

// --------------------------------------------------------------------------------

// Name: Replace

// Abstract: Replace a character based on given parameters

// --------------------------------------------------------------------------------

const char\* Replace(char chrLetterToFind, char chrReplace)

{

Int intLength = Length();

intLetterIndex = FindFirstIndexOf(chrLetterToFind);

pstrDestination = CloneString();

if (intSourceIndex > intReplaceStart)

{

intReplaceStart = FindFirstIndexOf(pstrFind, intSourceIndex);

}

for(intIndex = 0; intIndex < intLength; intIndex += 1)

if (intIndex > intReplaceStart)

{

intReplaceStart = FindFirstIndexOf(pstrFind, intSourceIndex);

}

If (intIndex == intReplaceStart)

{

\*( pstrDestination + intLetterIndex) = chrLetter;

}

Return pstrDestination;

}

// --------------------------------------------------------------------------------

// Name: Replace

// Abstract: Replace a substring based on given parameters

// --------------------------------------------------------------------------------

const char\* CSuperString::Replace(const char\* pstrFind, const char\* pstrReplace)

{

int intFindCount = 0;

int intLength = Length();

int intReplaceLength = strlen(pstrReplace);

int intFindLength = strlen(pstrFind);

int intDestinationLength = 0;

int intDestinationIndex = 0;

int intSourceIndex = 0;

int intReplaceIndex = 0;

int intReplaceStart = FindFirstIndexOf(pstrFind);

char\* pstrDestination;

bool blnFlag = false;

// Count instances of passed substring

while (intReplaceStart > -1)

{

intFindCount += 1;

intSourceIndex = intReplaceStart + intFindLength;

intReplaceStart = FindFirstIndexOf(pstrFind, intSourceIndex);

if (blnFlag != true)

{

blnFlag = true;

}

}

intSourceIndex = 0;

// Valid Input?

if (blnFlag == true)

{

// Yes, Calculate Length of new substring

intDestinationLength = ((intReplaceLength - intFindLength) \* intFindCount) + intLength;

// Alocate

pstrDestination = new char[intDestinationLength + 1];

// Reset

intReplaceStart = FindFirstIndexOf(pstrFind);

// Loop through the new substring

while (intDestinationIndex < intDestinationLength)

{

// Have we replaced a substring yet?

if (intSourceIndex > intReplaceStart && intReplaceStart != -1)

{

// Yes, Find next instance of substring

intReplaceStart = FindFirstIndexOf(pstrFind, intSourceIndex);

}

// Start replaceing substring?

if (intSourceIndex == intReplaceStart)

{

// Yes, replace substring

for (intReplaceIndex = 0; intReplaceIndex < intReplaceLength; intReplaceIndex += 1)

{

\*(pstrDestination + intDestinationIndex) = \*(pstrReplace + intReplaceIndex);

intDestinationIndex += 1;

}

// Set new location on original string

intSourceIndex += intFindLength;

}

else

{

// No, copy character

\*(pstrDestination + intDestinationIndex) = \*(m\_pstrSuperString + intSourceIndex);

intSourceIndex += 1;

intDestinationIndex += 1;

}

}

}

else

{

// No, return an empty string

pstrDestination = new char[1];

}

// Terminate

\*(pstrDestination + intDestinationIndex) = '\0';

return pstrDestination;

}

// --------------------------------------------------------------------------------

// Name: Insert

// Abstract: Insert a character based on given parameters

// --------------------------------------------------------------------------------

const char\* Insert(const char chrLetterToInsert, int intIndex)

{

int intLength = Length();

intDestinationIndex = 0;

intSourceIndexx = 0;

char\* pstrDestination = [intLength + 2]

for (intDestinationIndex = 0; intDestinationIndex < intLength + 2; intDestinationIndex += 1)

{

If (intDestinationIndex == intIndex)

{

\*( pstrDestination + intDestinationIndex) = chrLetterToInsert;

}

Else

{

\*(pstrDestination + intDestinationIndex) = \*(m\_pstrSuperString + intSourceIndex);

intSourceIndex += 1;

}

\*(pstrDestination + intDestinationIndex) = ‘\0’

}

return pstrDestination;

}

// --------------------------------------------------------------------------------

// Name: Insert

// Abstract: Insert a substring based on given parameters

// --------------------------------------------------------------------------------

const char\* Insert(const char\* pstrSubString, long lngIndex);

{

int intLength = Length();

int intInsertLength = strlen(pstrSubString);

int intDestintionLength = intLength + intInsertLength;

intDestinationIndex = 0;

intSourceIndex = 0;

intInsertIndex = 0;

char\* pstrDestination = [intDestintionLength + 1];

for (intDestinationIndex = 0; intDestinationIndex < intDestinationLength; intDestinationIndex += 1)

{

If (intDestinationIndex == intIndex)

{

for(intInsertIndex = 0; intInsertIndex < intInsertLength: intInsertIndex + 1)

{

\*( pstrDestination + intDestinationIndex) = \*(pstrSubString + intInsertIndex);

intDestinationIndex += 1;

}

}

Else

{

\*(pstrDestination + intDestinationIndex) = \*(m\_pstrSuperString + intSourceIndex);

intSourceIndex += 1;

}

}

\*(pstrDestination + intDestinationIndex) = ‘\0’

return pstrDestination;

}

DOUBLE CALL PROBLEM

m\_intCallCount

// --------------------------------------------------------------------------------

// Name: GetBuffer

// Abstract: Return a buffer to hold a char\*

// --------------------------------------------------------------------------------

const char\* CSuperString::GetBuffer (int intStringSize) const

{

static apstrBuffers[2]

static intCallCount = 0;

int intIndex = 0;

int intBufferIndex = intCallCount % 2;

intCallCount += 1;

If (apstrBuffers[intBufferIndex] != intStringSize + 1)

{

Delete[] apstrBuffers[intBufferIndex];

// Allocate memory to the first buffer;

apstrBuffers[intBufferIndex] = new char[intStringSize + 1];

}

return apstrBuffers[intBufferIndex];

}

String one: “This is a string”

String one: “This is a string”

// Create array of static char\*

// Check call count and delete previous call if needed

// Assign return string to the call count

// Dynamically allocate more size to pstrReturnString

// return