INTEGER ADDER USING STRINGS

Fordham University

Department of Computer and Information Science

CISC 2200: Data Structures

Instructor: Prof. H. M. Ammari

Students: Destin Piagentini, Kevin Yonamine

Fall 2018

Table of Contents

Introduction

We are building an integer adder that takes in the user input as a sequence of numeric characters that are of string data type instead of integer data type. We are doing this because there is a limit as to how big an integer can be, whereas a string has no limits on size. This allows us to truly add any two numbers instead of there being a maximum. In order to add the numbers accurately, we created a function that reverses the two user inputs and then adds them. This sum must then be reversed again, to create the correct output. Before adding the two numbers we made sure to verify the validity of the inputs. We made sure that there would only be one negative sign and ensured that it would only be in the first position. We also checked if there were any zeros at the beginning of the input that had no effect on the value of the input and removed those from our calculation. These steps ensured that we would be accurately calculating the sum of the user inputs every time.

Design and Solution

Approach and Evolution of the Design

Our approach in tackling this programming challenge was always to mimic the manual form of addition through code. Because the numbers were entered in as a string, and our challenge was to avoid assigning the inputs as integers, it became clear that the algorithm to add the numbers would have to focus on one number place at a time, forming an answer one number place at a time just as a human would as they go through a math problem on pen and paper. At many points during the design and implementation of this project, we found ourselves practicing the manual forms of addition/subtraction with pencils and blank pieces of paper. Add the two digits, then carry the one, etc. Once we had a grasp of the ways in which a human would calculate the problems without a calculator, we began working on a design that would utilize this method.

Our idea of the basic program loop was very simple. Get the input from the user, check to see that it's valid, then determine the operation based on the case, then perform the calculation and print the results. Before we had even started coding, we decided on the four possible cases for numbers entered in by the user. The first case would be where both numbers were positive. The second case would be where both numbers are negative. The third case would be where one number was positive and one number was negative (implying the use of subtraction). The fourth case would be if one number equaled 0, which meant the solution would just be whatever the second number was. With this knowledge, and the knowledge of knowing the foundation of the adding/subtracting algorithms, we began implementing our design.

Our very first steps was to read in two strings from the user, and determine their validity. We used the “cin” function to read in two strings, and then we defined a function that checked the validity of the two strings. We decided not to include this function as part of the class because we needed to test that the two inputs were valid before we instantiated an “Adder” object. Making the function a friend function would be useless as it wouldn’t need to access an private data members (in fact, no objects would be instantiated anyway at the time of its calling). Therefore, we made the decision to keep it separate from the class (declared in StringAdder.cpp, defined in its own file MainFunctions.cpp). This function was tested until we were 100% certain it worked.

At this point we started to create the class interface. Some functions came later (as we didn’t know we needed them at first), while most were known from the start. We knew we needed a default constructor and a constructor that initializes based on two strings (the user inputs). We also knew we needed an accessor for the solution, a public function that would execute the calculation from “main”, a function that would decide which case to execute, a function to find the largest number based on its absolute value, and addition/subtraction functions. We also declared three string variables (S_ENT1, S_ENT2, and S_ENT3) to hold the two user inputs and solution. We also knew we needed to use an enum to serve as a case identifier, which we were able to make the return type of the function that decided the case.

After the interface was created to the best of our ability, we began on implementation. First we created the constructors, which were each as simple as initializing variables to generic values (in the case of the default constructor) or to the user inputs (the constructor that takes two strings). Next, we implemented the DecideCase function to check if both, neither, or only one of the numbers was negative in order to decide which case we needed to apply. Additionally, the function also cycled through each string to see if they were equivalent to 0 (0000 = 0), and, if one or both of them were, to apply that case (OneOrMoreZeros).

The next function to implement was the Perform function. This function used the result of DecideCase (stored in a variable in Perform) to use a switch statement that would execute different scenarios based on the results. The first scenario was, of course, the scenario with one of the integers equaling 0. A function was written to determine which number equaled 0, and to set the answer to the one that was not equal to 0 (if they both were, the answer was simply 0). Though the addition/subtraction functions hadn't been written yet, we were able to pseudo-code the steps needed for those cases (example: removing the negative sign, or finding the largest absolute value digit). These pseudo-codes would translate to real code after the addition/subtraction functions were implemented.

The next function to implement was the addition function. The addition function took two positive integers and added them together. This function would be called in two cases: if both numbers were positive, or both were negative. If both numbers were negative, the two numbers would be stripped of their negative sign, added together, and a negative sign would be inserted to the solution. Initially with this function we decided to have two variables representing the number places. We did not reverse the strings, so we started from the last index and worked our way through the number. This, however, would prove a bit inefficient juggling between two number place variables. Soon after starting the implementation of this function, the decision was made to create a function to reverse the strings, and have the reversed strings be the input of the addition function. From here, we could use the same variable as an index.

The addition algorithm works by examining each character in the same index of the reversed strings, or what would be the same number place in a real number. If the number place was larger than the reach one of the lengths of the numbers, that number place would be treated as a 0 during calculations. Each character was transformed to an int variable using their ASCII numbers. Since the 0 character has an ASCII of 48, and 9 an ASCII of 57, any digit's int value could be determined by taking its ASCII and deducting by 48. After the digits are set up (and the program checks to make sure it isn't time to stop recursion by reaching a number place greater than the reach of both integers), the digits are added together. If they are greater than 9, they are reduced by 10 and the CarryOver boolean is marked to make sure that next pass through around of the digits is increased by 1 (carrying over). The function then calls itself again, advancing one number place. Through this algorithm, human behavior is mimicked. This function makes decisions identical to the ones a human would if they were tackling the same problem on a piece of paper (starting from the back, repeating the process until nothing left, remembering to carry digits when needed).

After addition was working, the last case to handle was if one number was positive while the other was negative. In other words, this would be subtraction. However, in subtraction, the process is a little bit different. The largest number digit-wise must be placed on top. For example, if we had -700 and 56, while 56 is technically the larger number, the "pen and paper" method of subtraction only works when 700 is higher. In this sense, we can subtract 56 from 700 to get 644. However, this isn't the answer to -700 + 56. To get the true answer, we simply add a negative sign, to negate the effects. Thus, we have -644, which is the true answer. So, we need a function to find the largest number based on its absolute value (not accounting for negative signs).

When we began to implement the function checking for the largest number, the first check we needed to do was based on length. However, if the user had an input that inflated the length but not the value by adding 0s to the front of the number (000058), wouldn't work. TO fix this, we quickly realized the need for a function to get rid of the extra zeroes. Once we implemented that function, we added it to the constructor so the strings are made neater from the onset. This also has the added advantage of making the results more readable (we print out the equation to the user). With this in hand, it was relatively easy to implement a function that compared two number strings by using length, or, if they were the same length, cycling through the digits until one was proven to be of greater value.

With the ability to compare in hand, the function that utilized subtraction was implemented. The function operates similarly to the addition function except it subtracts the smaller digit in a number place from the larger digit, and if the difference is a negative number, a number is borrowed from the next spot over. Since the larger number is on top, it is impossible to have the final number place in the sequence result in a negative number, so there was no special case where the function would run one time through at the end. Once the larger integer was put on top, the subtraction algorithm became even easier than the addition algorithm in this sense. Once the subtraction algorithm was implemented, the Perform function was updated to ensure that if the larger number was negative, the answer would be negative and vice versa.

At this point, the program was in a working state. However, the need to improve on the design was still there. To begin with, the case where on integer was 0 was removed. The other cases could calculate a total with the inclusion with 0 accurately, and there was no need for it to be its own special case. Deleting this special case allowed for a lot less space to be taken up in the Perform function, the DecideCase function, and the complete removal of the function that would be performed if that case was met (the function would find which number was equal to 0 by checking each string and set S_ENT3 equal to the one the opposite one). With the removal of this special case, a lot of space in the code was saved, improving readability. Additionally, the LargestDigit function was also updated to report whether the two values were equal. Since this function was only called in the case with one positive and one negative, if the absolute values were equal than the net total would be 0. S_ENT3 could then be set to 0 without having to run the subtraction formula. Additionally, accessor functions were created for S_ENT1 and S_ENT2 so that the "main" function could print the equation out (S_ENT1 + S_ENT2 = S_ENT3) for the user, enhancing user experience. Finally, with user recommendation, we adjusted the instructions and made the program print an equation using the negative sign if the second input was negative by using a temporary variable and erasing the negative sign.

Brief Summary of Each Function

Adder::Adder()

Default constructor that initializes each of the variables to "0", a generic value. Runs at O(1).

Adder::Adder(const string & Input1, const string & Input2)

Constructor that sets the variables to the inputs. The variables will be set to the value of the inputs after the extra zeros at the beginning of the number that have no effect on the value input have been removed. There is no need to check the validity because they are already determined to be valid previously in "main". Runs at O(1).

string Adder::RemoveExtraneousZeros(string Input)

This function is used by the constructor taking in two strings as inputs to remove zeros at the start of any number that have no effect on the value of the input. Used to make user inputs more readable, and to produce a more readable output in subtraction (Previously, for example, 1000 + -1001 would become -0001). Runs at order of O(N) where n is the amount of extraneous 0s in a string input.

void Adder::Perform()

This is the function that is called upon in main. It uses the CaseHandler function to decide what to do, then performs the instructions. For adding two positive numbers, the function reverses them, calls the addition function, and reverses the answer. For adding two negative numbers, the function removes the negative sign, reverses them, calls the addition function, reverses the answer, and inserts the negative sign again. For adding one positive and one negative, the function determines the negative number, removes the negative sign, determines the largest number (based on absolute value), reverses the inputs, calls the subtraction function, reverses the answer, and puts a negative sign on the answer if the largest number was negative. Or, if they have the same absolute value, they will cancel eachother out and the answer will be 0.This is where most of the important calls are. It uses other functions dedicated to doing specific tasks. At the conclusion of the program, the S_ENT3 variable will have been set. This function runs at O(1).

Adder::CaseHandler Adder::DecideCase() const

This function is dedicated to deciding what the sign of the inputs are. It will check and return whether both the values are negative or whether one is positive and one is negative. It does not need to check whether both values are positive because after checking the other two, if they have been determined to be false, then both of the inputs must be positive. Runs at O(1).

void Adder::ReverseString(string & Reverse)

Function is dedicated to reversing the string so that it can be added or subtracted. Doesn't return anything but does set the reverse variable at the end of the function. Runs at an order of O(n) where n is the length of the string passed into the function.

void Adder::AddDigitsFromReverse(const int & DigitPlace, const string &Reverse1, const string &Reverse2, bool CarryOver)

Takes the reversed inputs and adds them. Uses recursion to and adds one number at a time, taking note of whether the sum is greater than 9. If it is then it notes that a carry needs to be applied to the next number that is added. Every time the function runs, it goes to the next number. Will run again if both lengths are expended, but a carry is still needed (a larger number place is added). Runs through digits from back-to-front if we were looking at the unaltered inputs (before reversal). Runs at an order of O(n) where n is the length of the largest number string.

int Adder::LargestDigits(const string & Reverse1, const string & Reverse2)

Based on absolute value, this function will return 1 if the first input is larger, 2 if the second input is larger, or 3 if the inputs are equal. Determines this by first checking length, and if they are equal checking each subsequent digit until one proves to be higher. Runs at a worst case scenario of order of O(n) where n is the length of the two strings (if their lengths are equal), and a best case scenario of O(1) if their lengths are different (in this case, we know from the start which is larger).

void Adder::SubtractDigitsFromReverse(const int &DigitPlace, const string &LargestReverse, const string &SmallestReverse, bool ApplyBorrow)

This function is used when there is one positive input and one negative input. Similarly to the addition function, this function uses recursion and every time it is called on itself, the function is operating on the next place of the inputs. Will record if a borrow is needed to be applied to the next digit. Runs through digits from back-to-front if we were looking at the unaltered inputs (before reversal). Runs at an Order of O(n) where n is the length of the largest string).

string GetS_ENT1() const

Returns the first variable. Runs at O(1).

string GetS_ENT2() const

Returns the second variable. Runs at O(1).

string GetS_ENT3() const

Returns the third variable. Runs at O(1).

int main()

Starts the program and outputs each of the rules. Takes in the inputs for each variable, determines their validity, and then calls on other functions to perform operations. Outputs the answer based on whether the second number is negative or not. If the second number is negative, the equation will become a subtraction equation (a temporary used to erase the negative sign for the output so there may be a space between the negative sign and the second number). If the second number is positive, it outputs a positive equation as normal. At the end of the program asks the user whether or not they would like to do another calculation. Runs at O(n) where n is the amount of times the user plays again, or the amount of times the user enters a invalid input (whichever is larger).

bool DetermineValid(const string & Input1, const string & Input2)

Determines whether or not the inputs from the user are valid. Checks if there is exactly one negative sign and that it is at the start of the input and makes sure there are only digits and no characters besides a possible negative sign. Uses two loops to check each of the inputs and if one is false then returns false. If it makes it to the end of the program then returns true and the inputs are valid. Runs at an order of O(n) where n is the length of the largest string.

A Brief Overview of the General Program Loop

The program starts out by printing a welcome message and instructions for the user. The user is then prompted to input two number strings. If either of the inputs are found to be invalid, the program will tell the user, print the input instructions again, and have the user input a new set of number strings.

Once valid data is accepted, the program then instantiates an object of type “Adder” with the constructor that takes two strings as input. The variables S_ENT1 and S_ENT2 are initialized to the input stings after they go through the process of eliminating extraneous zeros. S_ENT3 is initialized to an empty string.

From here, the Perform function is called from main. The first operation the Perform function does is to call the DecideCase function to determine what case to execute. Stored in a variable of type CaseHandler (the enum), the Result is then passed through a switch statement. In the case of two positive integers, the strings are reversed, passed through the addition algorithm, and the solution is then reversed. If the case is determined to be two negative numbers, the two numbers are stripped of their negative sign, reversed, passed through the addition algorithm, the solution is reversed, and then a negative sign is included in the solution. In the case where one number is positive and one is negative, the negative number is located, recorded, and the two numbers are then passed through the function to determine which is larger. Once the larger number is determined, the subtraction algorithm is executed with the larger number on top (after reversing the two). The solution that results is then reversed, and the solution is also removed of extraneous zeroes, as the subtraction tends to leave extra zeroes if there is a difference in reach for the number places (Ex: 1001 + -1001 = 0001). If the largest number was negative, the solution is made negative (adding a negative sign). Of course this entire process is skipped if the LargestDigits function determined that the two are equal (Ex: -5 and 5), and S_ENT3 is simply made to equal “0”.

After the case is executed and the solution is determined, the solution is printed as an equation to the user (Ex: -5 + 5 = 0). The user is then asked if they would like to run the program again with different numbers. If the user enters ‘Y’ or ‘y’, the program restarts. If the user enters any other key, the program exits.

Implementation

Note: some comments or code may be edited from its original format to be more readable in a word processor (namely, Microsoft Word).

StringAdder.cpp

```
/*

  StringAdder.cpp
  StringAdder

  Created by Destin Piagentini and Kevin Yonamine on 10/7/18.
  Copyright © 2018 Destin Piagentini and Kevin Yonamine. All rights reserved.

 References:

 Knowledge and use of built-in string erase function:
 http://www.cplusplus.com/reference/string/string/erase/

 Knowledge and use of built-in string insert function:
 http://www.cplusplus.com/reference/string/string/insert/

 Refresher on the theory of manual subtraction (specifically subtracting a large number
from a small number):
 https://www.youtube.com/watch?v=hJWMLdDBHqw

 */
#include "pch.h"
#include <iostream>
#include "Adder.h"

bool DetermineValid(const string & Input1, const string & Input2); //determines if the
user input is valid

int main() {
       char Rerun; //For if the user wants to re-run the program
       string Input1, Input2;

       std::cout << "Welcome to the number adder!\n\n"; //A welcome message
       std::cout << "This program seeks to circumvent the maximum size of an integer
                     value by taking the number inputs as strings.\n";
       std::cout << "By taking the number inputs as strings, we use a stepwise-refinement
                     base approach to arrive at an answer.\n";
       std::cout << "In other words, this program can compute numbers of any
                     length.\n\n";


       do { //Rerun loop

              do { //Input invalid
                     std::cout << "A quick reminder:\n\n";//Following lines are
                                                            instructions
```

```
                std::cout << "The integer must comprise entirely of digits (or a
                              negative sign at the front).\n\n";

                std::cout << "Please enter the first integer: ";//Get the input
                std::getline(std::cin, Input1);    //Uses getline function to prevent
                                     spaces being registered as separate input
                std::cout << "Please enter the second integer: ";
                std::getline(std::cin, Input2);
                std::cout << '\n';

                if (!DetermineValid(Input1, Input2))//Check that both inputs are
                                                    valid
                {
                        std::cout << "Invalid inputs! Please try again.\n\n";
                }
        } while (!DetermineValid(Input1, Input2)); //Re-get input if not valid

        Adder Sum(Input1, Input2); //Instantiate object based on the inputs given
                              by the user
        Sum.Perform(); //Calculate

        if (Sum.GetS_ENT2()[0] == '-') { //If we need to print x - y

                string TempS_ENT2 = Sum.GetS_ENT2(); //Temp variable so we can place
                                     negative sign in middle
                TempS_ENT2.erase(0, 1); //Erase negative sign from the temp varible

                std::cout << Sum.GetS_ENT1() << " - " << TempS_ENT2 << " = " <<
                Sum.GetS_ENT3() << "\n\n"; //Print answer with negative sign

        } else
        { //If x + y
                std::cout << Sum.GetS_ENT1() << " + " << Sum.GetS_ENT2() << " = " <<
                Sum.GetS_ENT3() << "\n\n"; //Print the answer
        }


        std::cout << "Would you like to perform another calculation?\n"; //Ask to
                                                                         rerun
        std::cout << "(Type 'Y' or 'y', or any other key to exit): ";

        std::cin >> Rerun;
        std::cin.ignore(); //Avoid getline function errors and clear the input

    } while (Rerun == 'y' || Rerun == 'Y'); //Rerun depending on input

    return 0;
}
```

MainFunctions.cpp

```
//
//  MainFunctions.cpp
//  StringAdder
```

```
//
//  Created by Destin Piagentini and Kevin Yonamine on 10/7/18.
//  Copyright © 2018 Destin Piagentini and Kevin Yonamine. All rights reserved.
//

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

using std::string;

/*
Check to see if inputs are valid. Invalid if character is not a digit or -. There can
only be one - and it must be the first char
*/
bool DetermineValid(const string & Input1, const string & Input2)
{
       //2 loops in case the inputs are not the same length
       for (int i = 0; i < Input1.length(); i++)//first string
       {
              //using ASCII, the digits must be between 48 and 57. 48 is 0 and 57 is 9.
              if not, then the character is invalid
              //if negative sign is not in front or there is invalid character
              if ((Input1[i] == '-' && i != 0)||(Input1[i] != '-'&&(int(Input1[i]) <
                                                48||int(Input1[i]) > 57)))
              {
                     return false;
              }
       }

       for (int j = 0; j < Input2.length(); j++)//second string
       {
                            //using ASCII, the digits must be between 48 and 57. 48 is 0
              and 57 is 9. if not, then the input is invalid
              if ((Input2[j] == '-'&&j != 0)||(Input2[j] != '-' && (int(Input2[j]) <
                                         48||int(Input2[j]) > 57)))
              {
                     return false;
              }
       }
       return true; //If we made it this far, we know both inputs are valid.
}
```

Adder.h

```
//
//  Adder.h
//  StringAdder
//
//  Created by Destin Piagentini and Kevin Yonamine on 10/7/18.
//  Copyright © 2018 Destin Piagentini and Kevin Yonamine. All rights reserved.
//
//  This file contains the class interface for the "Adder" class.

#pragma once

#include <iostream>
#include <string>
```

```
using std::string;

class Adder {
	public:

	Adder(); //Default constructor - Included just for good programming practice
	Adder(const string & Input1, const string & Input2); //Constructor accepting two
	strings to use as the calculations (already confirmed as valid inputs).

	void Perform(); //Function used by the "main" function to initiate the
	              calculation.

	//Accessor function for the variable that stores the first input. Used by "main"
	function to print the solution.
	string GetS_ENT1() const {

		return S_ENT1;

	}

	//Accessor function for the variable that stores the second input.. Used by "main"
	function to print the solution.
	string GetS_ENT2() const {

		return S_ENT2;

	}

	//Accessor function for the variable that stores the solutuion. Used by "main"
	function to print the solution.
	string GetS_ENT3() const {

		return S_ENT3;

	}

	private:

	string S_ENT1, S_ENT2, S_ENT3; //Variables used to store two inputs and answer

	enum CaseHandler {TwoPositive, OnePositiveOneNegative, TwoNegative}; //Enum to
	                                  decide which case the inputted numbers meet.

	//Removes any additional zeroes before a number for better readability by both
	user and program.
	string RemoveExtraneousZeros(string Input);

	CaseHandler DecideCase() const; //Function used to decide which case the "Perform"
	                                  function needs to execute.

	void ReverseString(string & Reverse); //Used to reverse numbers in preparation for
	                                         addition/subtraction.

	//Function to add two number strings once they have been reversed. RECURSIVE
	void AddDigitsFromReverse(const int & DigitPlace, const string & Reverse1, const
	                           string & Reverse2, bool CarryOver);
	//Used to find the largest digit regardless of positive/negative status.
	int LargestDigits(const string & Reverse1, const string & Reverse2);

	//Function used to subtract two number strings via reverse).
	//Largest digit gets put on "top" (goes first).
	void SubtractDigitsFromReverse(const int & DigitPlace, const string &
		LargestReverse, const string & SmallestReverse, bool ApplyBorrow);

};
```

Adder.cpp

```
//
//  Adder.cpp
//  StringAdder
//
//  Created by Destin Piagentini and Kevin Yonamine on 10/7/18.
//  Copyright © 2018 Destin Piagentini and Kevin Yonamine. All rights reserved.
//

#include "pch.h"
#include "Adder.h"

//Default constructor.
//Runs when no inputs are given.
//Initializes each variable to generic values.
//Included for good programming practice.
Adder::Adder() {

	S_ENT1 = "0";
	S_ENT2 = "0";
	S_ENT3 = "0";

}

//Constructor that accepts two number strings (strings consisting of only numbers and/or
a negative sign at the beginning)
//Sets variables to the values of the two inputs.
//The two inputs are stripped of extraneous zeroes (zeroes at the beginning of a number
that do not effect its value)
//for human readability and program readability.
//Note: The two variables were already determined valid in the "main" function.
//@params string Input1, Input2: The user input of numbers in the form of strings.
Adder::Adder(const string & Input1, const string & Input2) {

	S_ENT1 = RemoveExtraneousZeros(Input1);
	S_ENT2 = RemoveExtraneousZeros(Input2);
	S_ENT3 = "";

}

/*Function that removes extra zeros at the start of number that don't change the value
 For readability of program and functionality when using the program
@param string Input: The string of the number to remove zeros from
*/
string Adder::RemoveExtraneousZeros(string Input)
```

```
{
	bool OriginallyNegative = false;//if the number is negative at the start, we need
	                                  to make it negative at the end
	if (Input[0] == '-') { //if number is negative

		Input.erase(0, 1); //erase the element at 0 which is the negative sign, so
		                     that we get the number by itself
		OriginallyNegative = true; //set the boolean to true so we know that the
		                             number was negative at the beginning
	}

	while (Input[0] == '0' && Input.length() >= 2) { //if the first character is 0
	                                                    then remove it. the length will be
	                                                    at minimum, 2 so that last digit
	                                                    always kept

		Input.erase(0, 1); //erase extra 0 from front of string

	}

	if (OriginallyNegative) { //if the number was negative at the start

		Input.insert(0, 1, '-');//insert in the 0 position, length 1, the character
		-; meaning a negative sign before the number

	}

	return Input; //Return the altered string.

}

/*
Function used to perform the calculation.
Called in the "main" function.
First decides which case to execute based on the result of the "CaseHandler" function.
Performs the appropriate case.

TwoPositive: Both numbers positive. Add as normal.

TwoNegative: Two negative numbers. Technically subtraction, but same result
achieved by removing both negative signs, adding as normal, place negative sign on
answer.

OnePositiveOneNegative: Two numbers with different signs.
First we find the absolute largest digit (largest number regardless of negative/positive
status).
Then, we remove the negative signs and subtract (adding two numbers of opposite signs is
the same as subtraction).
The difference between the two numbers will always be the same. The question is whether
the difference is negative or not.
If the larger number was negative, the difference will be negative.
If the larger number was positive, the answer will still be positive.
If equal, the two will cancel eachother out.

By the end of this function, S_ENT3 will be appropriately calculated.
*/
void Adder::Perform() {
```

```
//Make temporary variables based on the real variables so we can alter the
temporaries without changing the real variables.
string Reverse1 = S_ENT1, Reverse2 = S_ENT2;

CaseHandler Result = DecideCase(); //Decide what case to run by running the
                                   DecideCase function.

switch (Result) {

//Case both numbers are positive
case TwoPositive:

       //Reverse
       ReverseString(Reverse1);
       ReverseString(Reverse2);

       //Compute

       AddDigitsFromReverse(0, Reverse1, Reverse2, false);

       //Reverse answer
       ReverseString(S_ENT3);

       break;

       //Case both numbers are negative
       //Adding two negative numbers = Remove negative signs, add as normal, place
       negative sign on answer.
case TwoNegative:
       //RemoveNeg

       Reverse1.erase(0, 1);//Erase first char (which is '-')
       Reverse2.erase(0, 1);

       //Reverse

       ReverseString(Reverse1);
       ReverseString(Reverse2);

       //Add

       AddDigitsFromReverse(0, Reverse1, Reverse2, false);

       //ReverseAnswer

       ReverseString(S_ENT3);


       //PutNeg

       S_ENT3.insert(0, 1, '-');

       break;

case OnePositiveOneNegative:

       //Used to keep track of which number is larger, which is negative.
       int Negative, Largest;
```

```
//Determine which is negative, and erase the negative sign (in preparation
of comparison and subtraction)

if (Reverse1[0] == '-') {

       Reverse1.erase(0, 1);
       Negative = 1;

}
else { //If not 1, then 2

       Reverse2.erase(0, 1);
       Negative = 2;

}

//Compare

//1 = First number is largest, 2 = Seocnd number is largest, 3 = Values are
equal
Largest = LargestDigits(Reverse1, Reverse2);

//Reverse

ReverseString(Reverse1);
ReverseString(Reverse2);

//Cases

if (Largest == 1) {

       //Place larger number on top.Then subtract.
       SubtractDigitsFromReverse(0, Reverse1, Reverse2, false);
       ReverseString(S_ENT3);
       S_ENT3 = RemoveExtraneousZeros(S_ENT3); //Without this, will produce
       extra 0s if sizes are different (Ex: 1000 + -1001 = -0001).

       //If the largest number was negative, we know the answer is
       negative. If not, a smaller number was subtracted from a larger
       number, which results in a positive answer.
       if (Negative == 1) {

              S_ENT3.insert(0, 1, '-'); //If larger number was negative,
                                    insert negative sign into answer.

       }

}
else if (Largest == 2) {

       //Place larger number on top. Then subtract.
       SubtractDigitsFromReverse(0, Reverse2, Reverse1, false);
       ReverseString(S_ENT3);
       S_ENT3 = RemoveExtraneousZeros(S_ENT3);

       //If the largest number was negative, we know the answer is
       negative. If not, a smaller number was subtracted from a larger
```

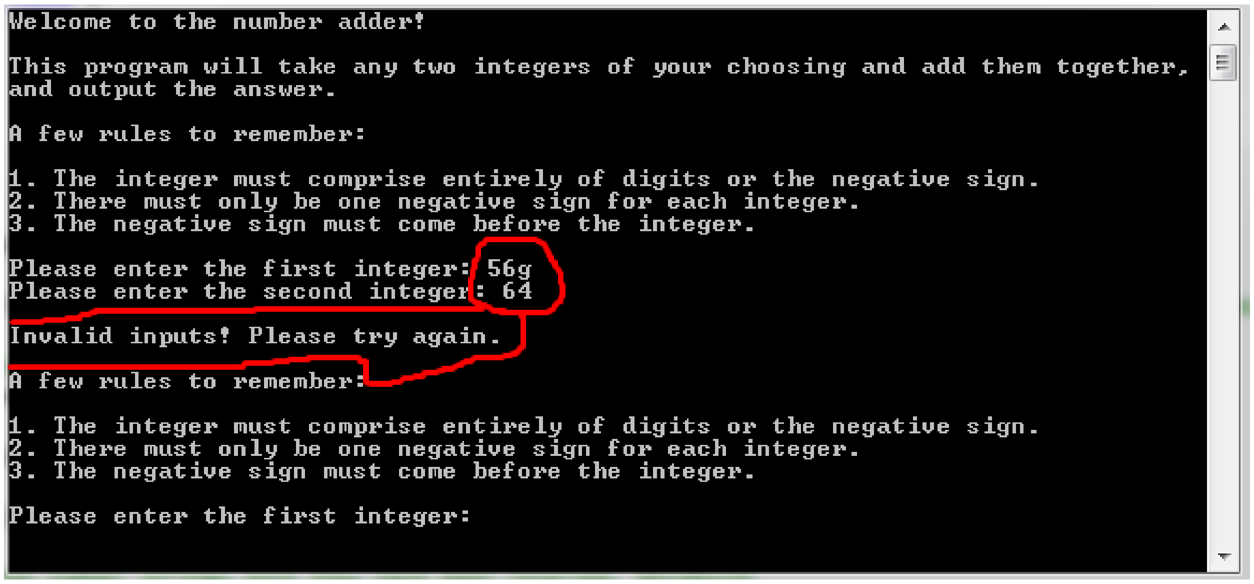
```
                        number, which results in a positive answer.
                        if (Negative == 2) {

                                S_ENT3.insert(0, 1, '-'); //If larger number was negative,
                                                                insert negative sign into answer.

                        }

                }
                else if (Largest == 3) {

                        //If they are equal, they cancel eachother out.
                        S_ENT3 = "0";

                }

                break;

        }

}

//Function utilizes enum in the class to tell the "perform" function which case to
execute.
//Checks for negative signs in front of both numbers, or either number, and if not then
determines neither are negative.
//Note: Numbers already determined valid. We know negative signs must be in proper place,
(in front) if they exist.
Adder::CaseHandler Adder::DecideCase() const {

        //Check for - and - or + and -
        if (S_ENT1[0] == '-' && S_ENT2[0] == '-') {

                return TwoNegative;

        }
        else if (S_ENT1[0] == '-' || S_ENT2[0] == '-') { //If both are not negative, check
                                                                if just one is.

                return OnePositiveOneNegative;

        }

        return TwoPositive; //All other cases have been tested.

}

//Takes an input string, and reverses it in preperation for addition or subtraction,
//or to reverse the solution since it will be calculated in reverse order.
//@param string Reverse: string to be reversed.
void Adder::ReverseString(string & Reverse) {

        string TempString = "";

        for (int i = Reverse.length() - 1; i >= 0; i--) {

                TempString = TempString + Reverse[i];
        }

        Reverse = TempString;

}

/*
Addition function. Takes two number strings (order doesnt matter) and adds them via
recursion and focusing on one number place at a time.
Calls itself, each time moving one number place forward.
If the sum of the two digits in a given place exceeds 9, function will record that the
next number place needs to apply a carry.
Adds to S_ENT3 in reverse order. S_ENT3 reversed in "Perform" function to negate this
effect.

@Params const int & DigitPlace: index of the digits of both functions in focus.
const string & Reverse1, Reverse2: Numbers (which have already been reversed) to be added
together
bool CarryOver: Determines (based on the previous recursive run) if a carry (+1) needs to
be applied to the current digit at the current index

*/
void Adder::AddDigitsFromReverse(const int & DigitPlace, const string &Reverse1, const
string &Reverse2, bool CarryOver) {

        //A negative number will never be entered here, so we dont need to check for it.

        //Temp variables.
        //Digit1 = first number's digit, Digit2 = second number's digit, Digit3 = the sum
        of the two when we calculate it.
        int Digit1, Digit2, Digit3;

        if (DigitPlace < Reverse1.length() && DigitPlace < Reverse2.length()) { //If both
                                                                           are still in play.

                Digit1 = int(Reverse1[DigitPlace]) - 48; //0 ASCII is 48, 9 is 57. - 48 to
                                          the digits int() means we get the digits int value.
                Digit2 = int(Reverse2[DigitPlace]) - 48;

        }
        else if (DigitPlace >= Reverse1.length() && DigitPlace < Reverse2.length()) { //If
                                                      Reverse1 is at its end but not Reverse2.

                Digit1 = 0; //Treat a digit place that is larger than the digit length as a
                                                                                          0.
                Digit2 = int(Reverse2[DigitPlace]) - 48;

        }
        else if (DigitPlace < Reverse1.length() && DigitPlace >= Reverse2.length()) { //If
                                                      Reverse2 is at its end but not Reverse2.

                Digit1 = int(Reverse1[DigitPlace]) - 48;
                Digit2 = 0; //Treat a digit place that is larger than the digit length as a
                                                                                  0.

        }
        else if (DigitPlace >= Reverse1.length() && DigitPlace >= Reverse2.length() &&
CarryOver == true) { //If both are at their end but we still need to carry one over
```

```
			S_ENT3 = S_ENT3 + '1'; //Add the carry to the end (or the front after
                                                          reversal).
			return; //End recursion - WE ARE DONE

		}
		else { //If both are at their end (and no carry to apply).

			return; //End recursion - WE ARE DONE

		}

		//Apply a carry if needed.
		if (CarryOver == true) {

			Digit1 = Digit1 + 1;

		}

		Digit3 = Digit1 + Digit2; //Add the two to calculate this number places value.

		if (Digit3 > 9) { //If we need to apply a carry to the next number place

			Digit3 = Digit3 - 10; //We will add one to the next number place to account
                         for this ten. For now, we want to get the digit back to
                         single digits.
			CarryOver = true; //Record that we need to apply a carry next.

		}
		else { //If we dont need to apply a carry next pass through.

			CarryOver = false;

		}

		//Record the value. + 48 to get proper ASCII of the digit.
		S_ENT3 = S_ENT3 + char(Digit3 + 48); // 0 + 48 = ASCII of 48 = '0'. 9 + 48 = ASCII
                                               of 57 = '9'

		//Call function again for next digit place with correct CarryOver value.
		return AddDigitsFromReverse(DigitPlace + 1, Reverse1, Reverse2, CarryOver);
}

/*
 Function determines which input is larger based on absolute value. Ignores negative
numbers because they will be positive when this function is used.
 Returns 1 if first input is larger, 2 if second input is larger, if they are equal
returns 3
 */
int Adder::LargestDigits(const string & Reverse1, const string & Reverse2)
{
		if (Reverse1.length() > Reverse2.length()) { //if one has a longer length we know
		it is the larger number (extra zeros at the start have already been removed)

			return 1;
		}
		else if (Reverse2.length() > Reverse1.length()) { //same for the second input
```

```
			return 2;

		} else
		{ //if the numbers are the same length it does not imply the same value

			int i = 0; //start at first DigitPlace
			do {//Get int value and compares

				int Digit1 = int(Reverse1[i] - 48);
				int Digit2 = int(Reverse2[i] - 48);

				if (Digit1 > Digit2) { //run until we find a digit that is not
							equal. we can now find which is larger
					return 1;

				} else if (Digit2 > Digit1) {

					return 2;

				}

				i++;

			} while (i < Reverse1.length());

			return 3; //if it exits loop without returning already, we know they are
					equal
		}
}

/*
Subtraction function taking in 2 strings(Larger first) and subtracts them using
recursion. Used if one number is positive and the other is negative.
Calls itself, each repetition moves one digit forward
If the difference of the 2 digits in a given place is less than 0, function records that
the next number place needs to borrow
Adds S_ENT3 in reverse order
*/
void Adder::SubtractDigitsFromReverse(const int &DigitPlace, const string
&LargestReverse, const string &SmallestReverse, bool ApplyBorrow)
{
	//Digit1 = larger number's digit, Digit2 = smallest numbers digit, Digit3 = the
	difference
	int Digit1, Digit2, Digit3;//temporary
	if (DigitPlace < LargestReverse.length() && DigitPlace < SmallestReverse.length())
	{

		Digit1 = int(LargestReverse[DigitPlace]) - 48; //in ASCII 0 is 48, 9 is 57
		Digit2 = int(SmallestReverse[DigitPlace]) - 48;

	}
	else if (DigitPlace < LargestReverse.length() && DigitPlace >=
		SmallestReverse.length()) {

		Digit1 = int(LargestReverse[DigitPlace]) - 48;
		Digit2 = 0;

	}
```

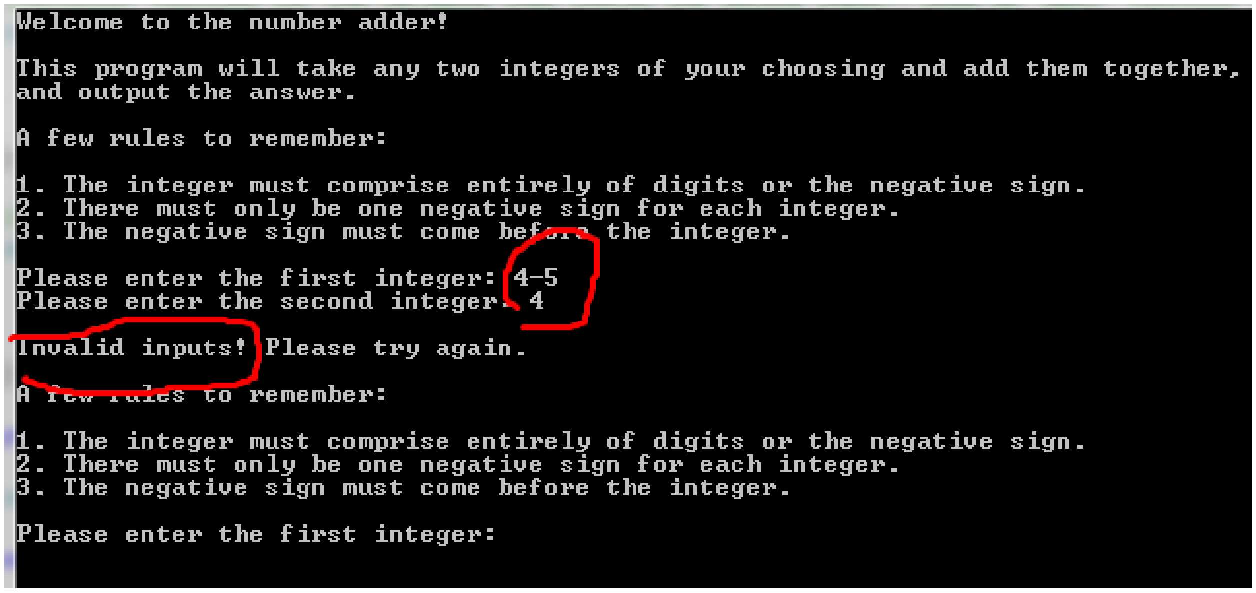
```
        else if (DigitPlace >= LargestReverse.length()) { //no need to check the smaller,
                            if doesn't apply to larger, it wont apply to the smaller
               return;

        }

        if (ApplyBorrow == true) { //need borrow

               Digit1 = Digit1 - 1; //the borrowing subtracts 1 from the top

        }

        Digit3 = Digit1 - Digit2; //subtract

        if (Digit3 < 0) {

               //if difference is less than 0, borrow from the next position
               Digit3 = Digit3 + 10; //borrow makes the digit above 0
               ApplyBorrow = true; //borrow applied

        } else {

               ApplyBorrow = false;

        }

        S_ENT3 = S_ENT3 + char(Digit3 + 48); //set the value to 48 higher to get the
                                       actual value in ASCII
        return SubtractDigitsFromReverse(DigitPlace + 1, LargestReverse, SmallestReverse,
        ApplyBorrow); //Call function again with ApplyBorrow bool

}
```

Tests



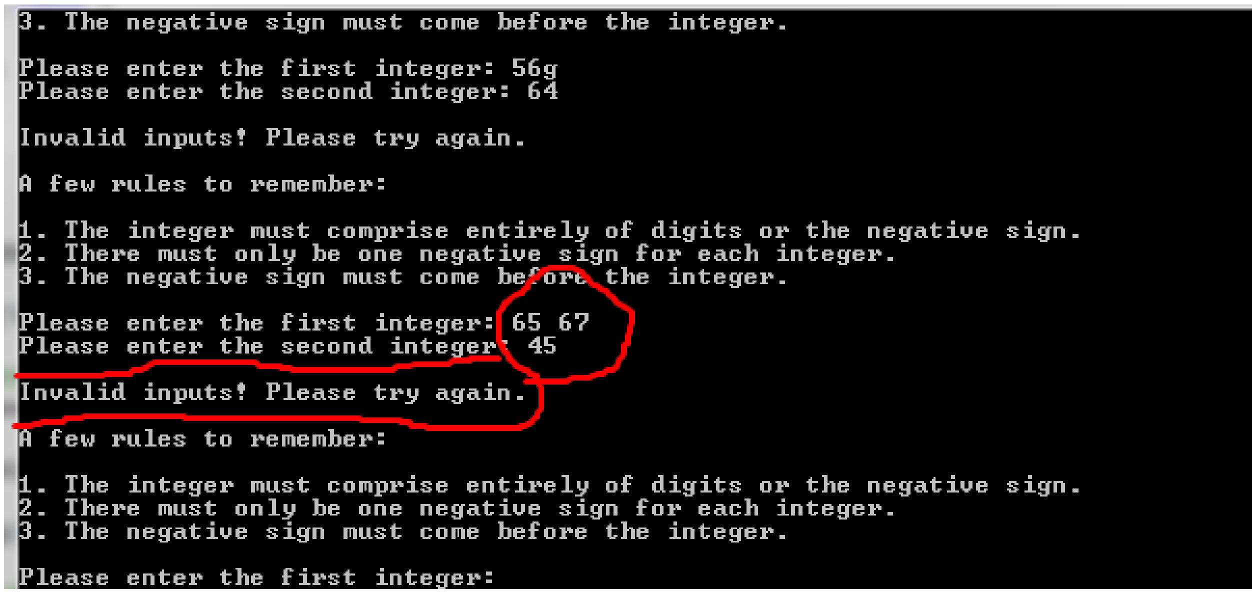
```
Welcome to the number adder!

This program will take any two integers of your choosing and add them together,
and output the answer.

A few rules to remember:

1. The integer must comprise entirely of digits or the negative sign.
2. There must only be one negative sign for each integer.
3. The negative sign must come before the integer.

Please enter the first integer: 56g
Please enter the second integer: 64

Invalid inputs! Please try again.

A few rules to remember:

1. The integer must comprise entirely of digits or the negative sign.
2. There must only be one negative sign for each integer.
3. The negative sign must come before the integer.

Please enter the first integer:
```

This picture shows the DetermineValid function in action. The user entered a string with a letter char, which is obviously not part of an integer string. The program was able to catch this and determine it as invalid.



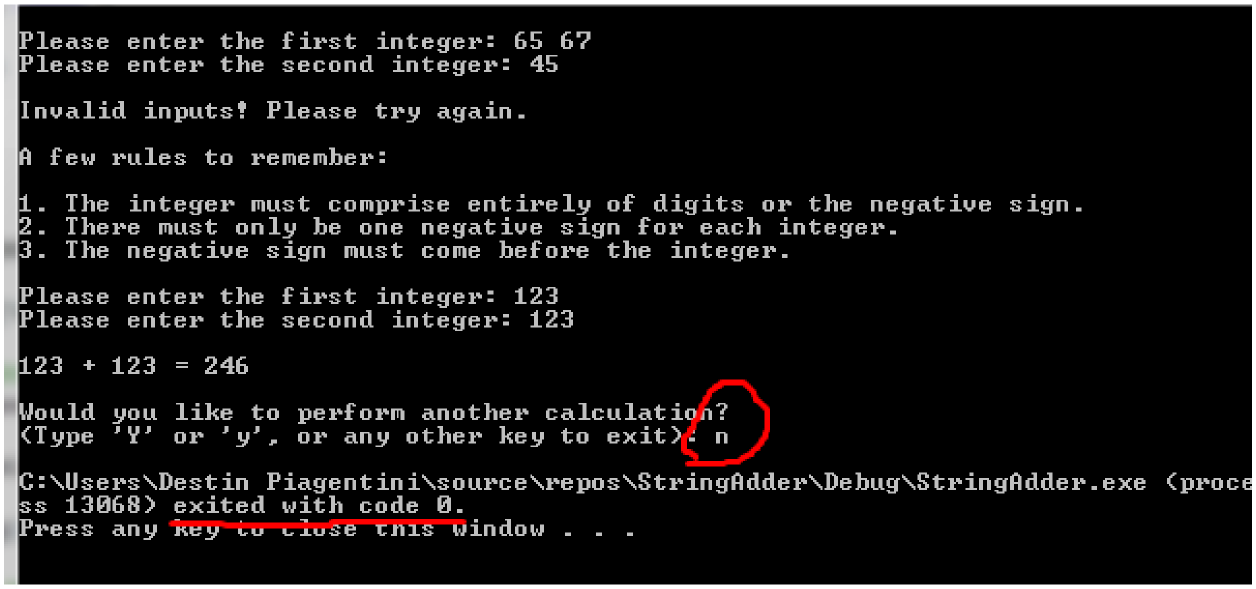
```
Welcome to the number adder!

This program will take any two integers of your choosing and add them together,
and output the answer.

A few rules to remember:

1. The integer must comprise entirely of digits or the negative sign.
2. There must only be one negative sign for each integer.
3. The negative sign must come before the integer.

Please enter the first integer: 4-5
Please enter the second integer: 4

Invalid inputs! Please try again.

A few rules to remember:

1. The integer must comprise entirely of digits or the negative sign.
2. There must only be one negative sign for each integer.
3. The negative sign must come before the integer.

Please enter the first integer:
```

This picture shows the DetermineValid function catching a negative sign in an invalid place, and alerting the user that their input was invalid.



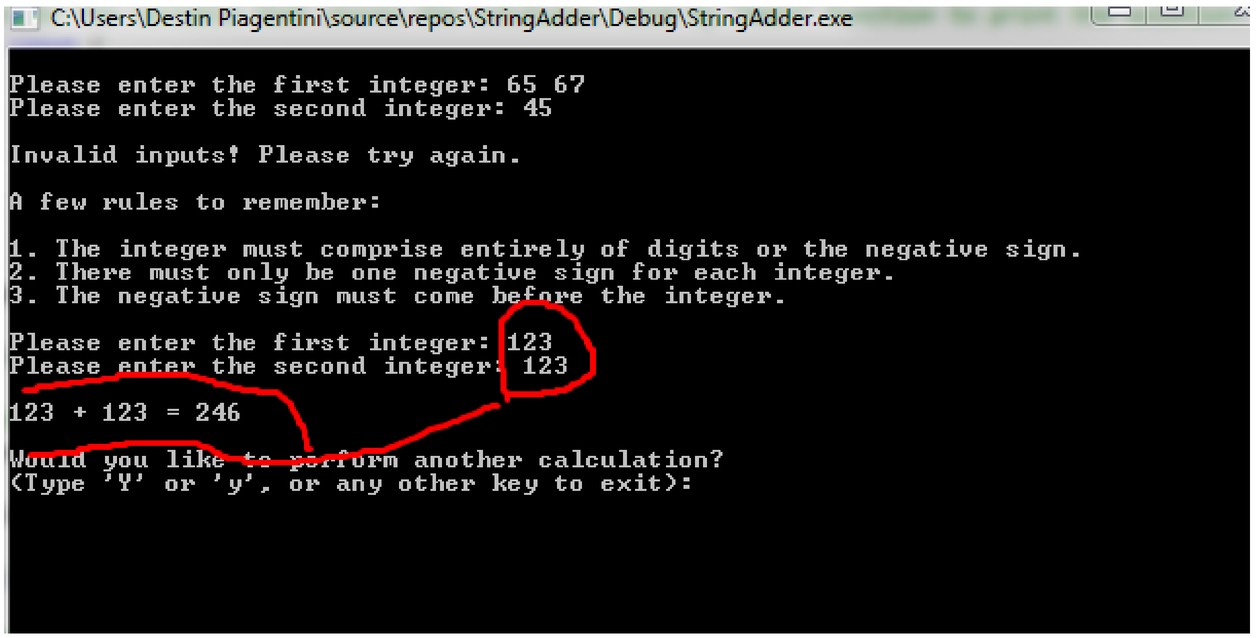
```
3. The negative sign must come before the integer.

Please enter the first integer: 56g
Please enter the second integer: 64

Invalid inputs! Please try again.

A few rules to remember:

1. The integer must comprise entirely of digits or the negative sign.
2. There must only be one negative sign for each integer.
3. The negative sign must come before the integer.

Please enter the first integer: 65 67
Please enter the second integer: 45

Invalid inputs! Please try again.

A few rules to remember:

1. The integer must comprise entirely of digits or the negative sign.
2. There must only be one negative sign for each integer.
3. The negative sign must come before the integer.

Please enter the first integer:
```

This picture shows the importance of using the “getline” function to receive the string inputs. If we were using “cin”, the space char would count as a new line and the program would then initialize with 65 + 67. Instead, since we are using “getline” the space char is considered part of the string, and the DetermineValid function is able to catch that and alert the user.



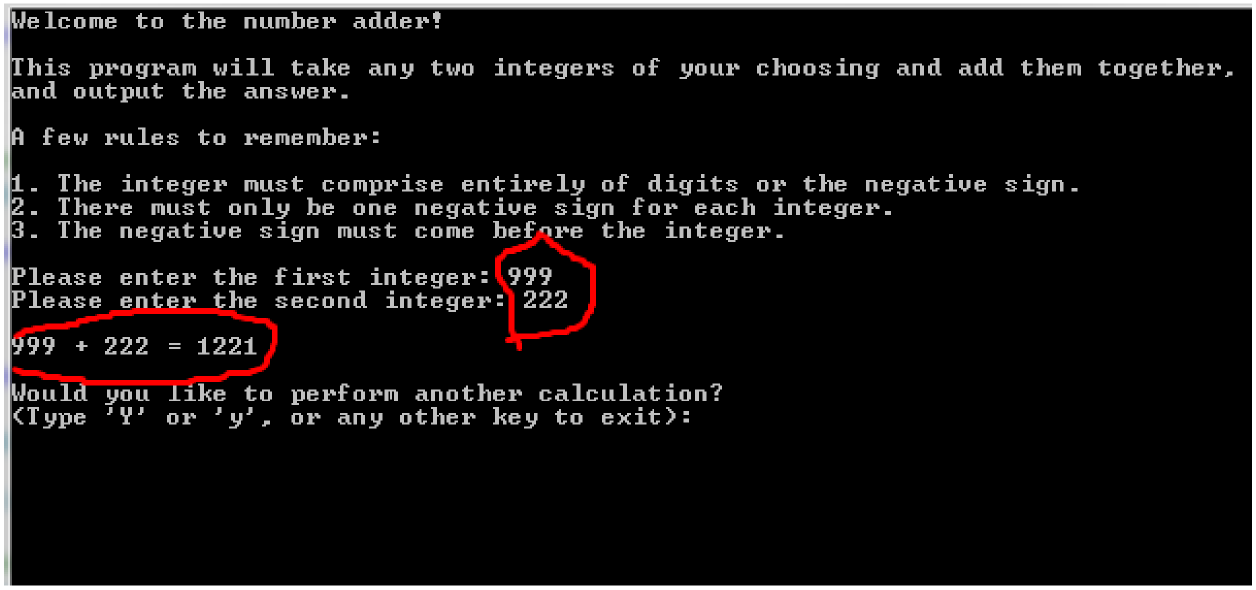
```
Please enter the first integer: 65 67
Please enter the second integer: 45

Invalid inputs! Please try again.

A few rules to remember:

1. The integer must comprise entirely of digits or the negative sign.
2. There must only be one negative sign for each integer.
3. The negative sign must come before the integer.

Please enter the first integer: 123
Please enter the second integer: 123

123 + 123 = 246

Would you like to perform another calculation?
(Type 'Y' or 'y', or any other key to exit): n

C:\Users\Destin Piagentini\source\repos\StringAdder\Debug\StringAdder.exe (proce
ss 13068) exited with code 0.
Press any key to close this window . . .
```

This picture simply shows that the user is able to exit at the end of a calculation.



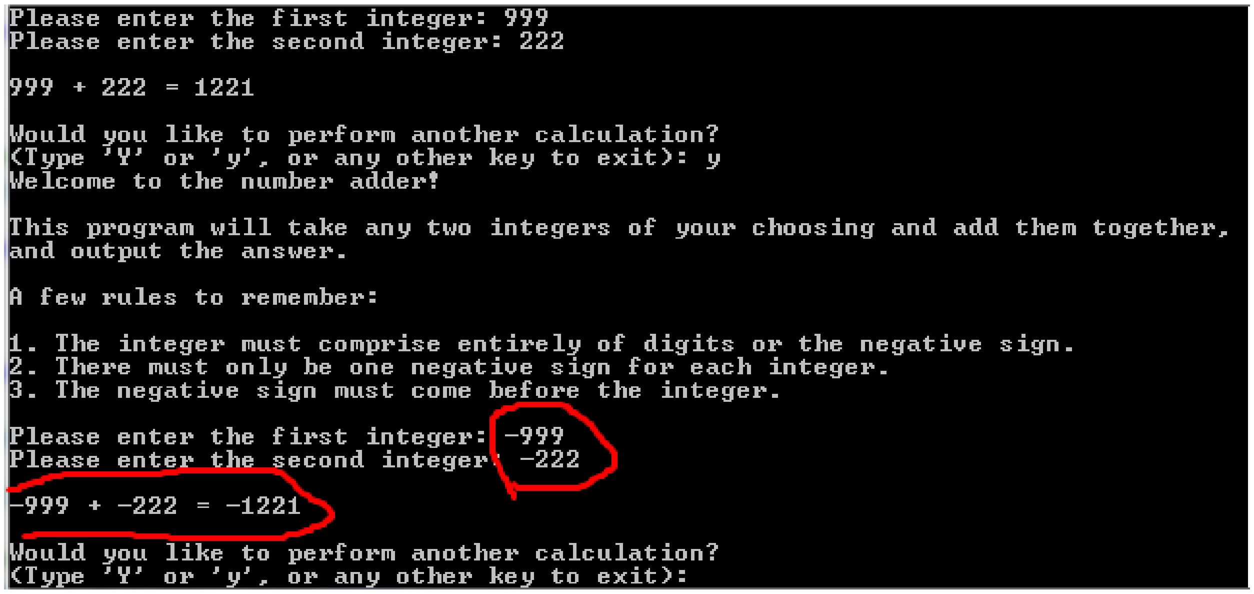
```
C:\Users\Destin Piagentini\source\repos\StringAdder\Debug\StringAdder.exe

Please enter the first integer: 65 67
Please enter the second integer: 45

Invalid inputs! Please try again.

A few rules to remember:

1. The integer must comprise entirely of digits or the negative sign.
2. There must only be one negative sign for each integer.
3. The negative sign must come before the integer.

Please enter the first integer: 123
Please enter the second integer: 123

123 + 123 = 246

Would you like to perform another calculation?
(Type 'Y' or 'y', or any other key to exit):
```

This picture shows the addition algorithm in action. In this example, the there is no carrying involved between the two numbers.



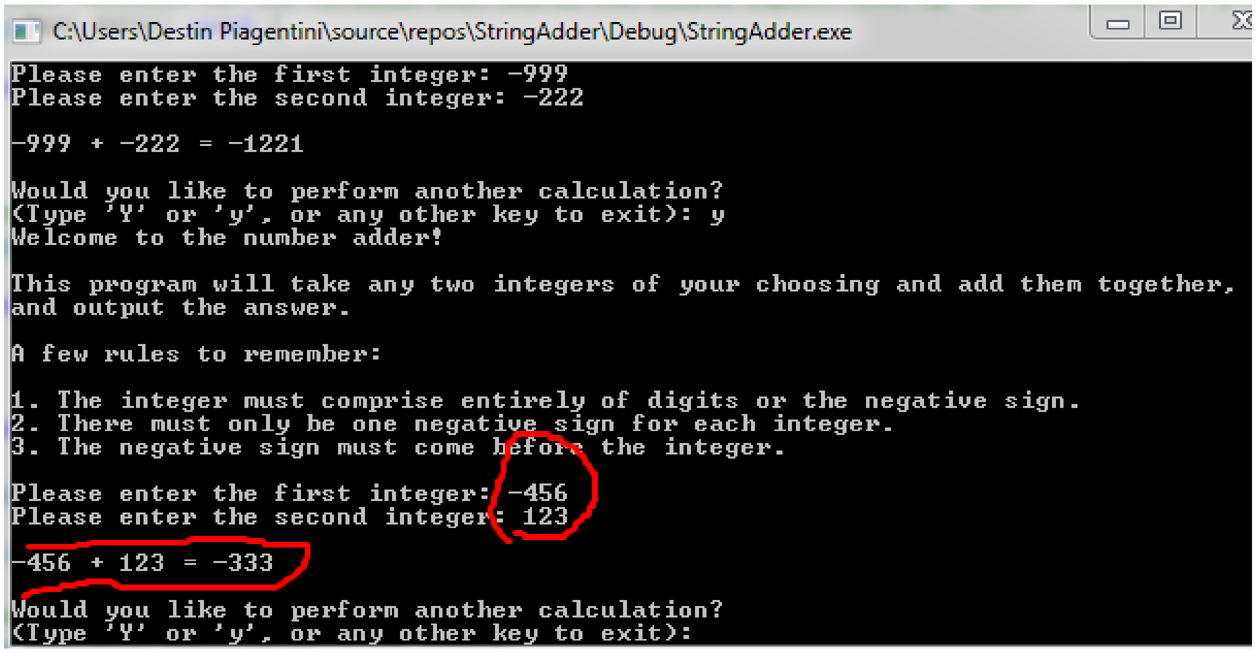
```
Welcome to the number adder!

This program will take any two integers of your choosing and add them together,
and output the answer.

A few rules to remember:

1. The integer must comprise entirely of digits or the negative sign.
2. There must only be one negative sign for each integer.
3. The negative sign must come before the integer.

Please enter the first integer: 999
Please enter the second integer: 222

999 + 222 = 1221

Would you like to perform another calculation?
(Type 'Y' or 'y', or any other key to exit):
```

This picture shows the addition algorithm in action, but each number in the sequence requires a carry.



```
Please enter the first integer: 999
Please enter the second integer: 222

999 + 222 = 1221

Would you like to perform another calculation?
(Type 'Y' or 'y', or any other key to exit): y
Welcome to the number adder!

This program will take any two integers of your choosing and add them together,
and output the answer.

A few rules to remember:

1. The integer must comprise entirely of digits or the negative sign.
2. There must only be one negative sign for each integer.
3. The negative sign must come before the integer.

Please enter the first integer: -999
Please enter the second integer: -222

-999 + -222 = -1221

Would you like to perform another calculation?
(Type 'Y' or 'y', or any other key to exit):
```

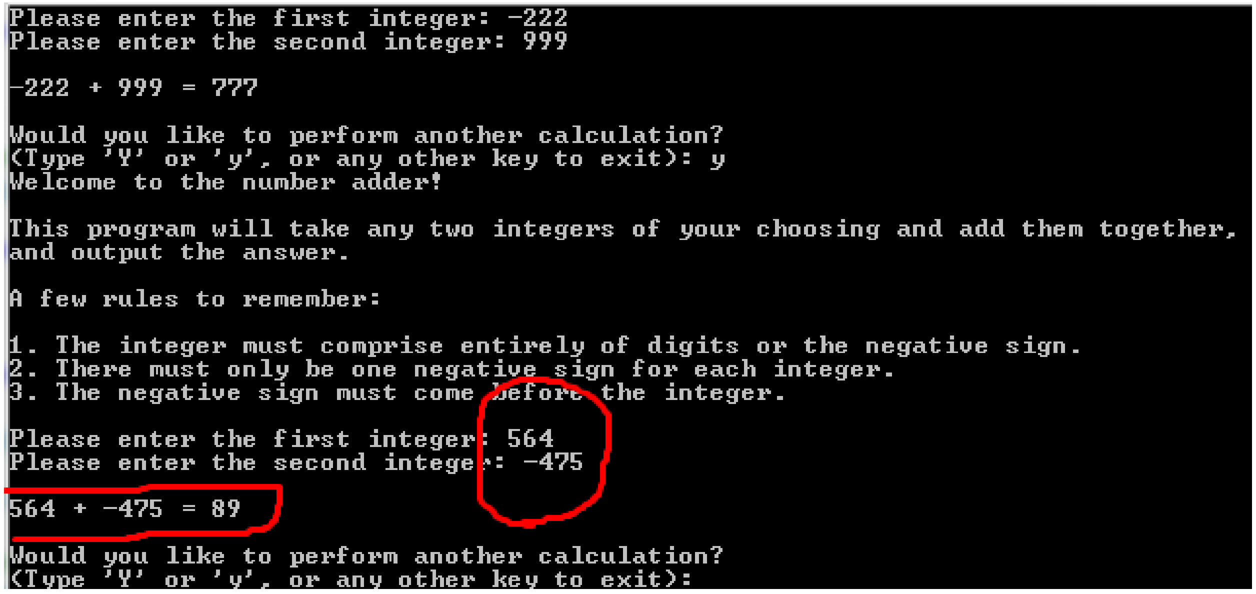
This picture shows the process of two negative numbers. It is the same process of two positive numbers with a negative sign added. This, too, involves carrying.



C:\Users\Destin Piagentini\source\repos\StringAdder\Debug\StringAdder.exe

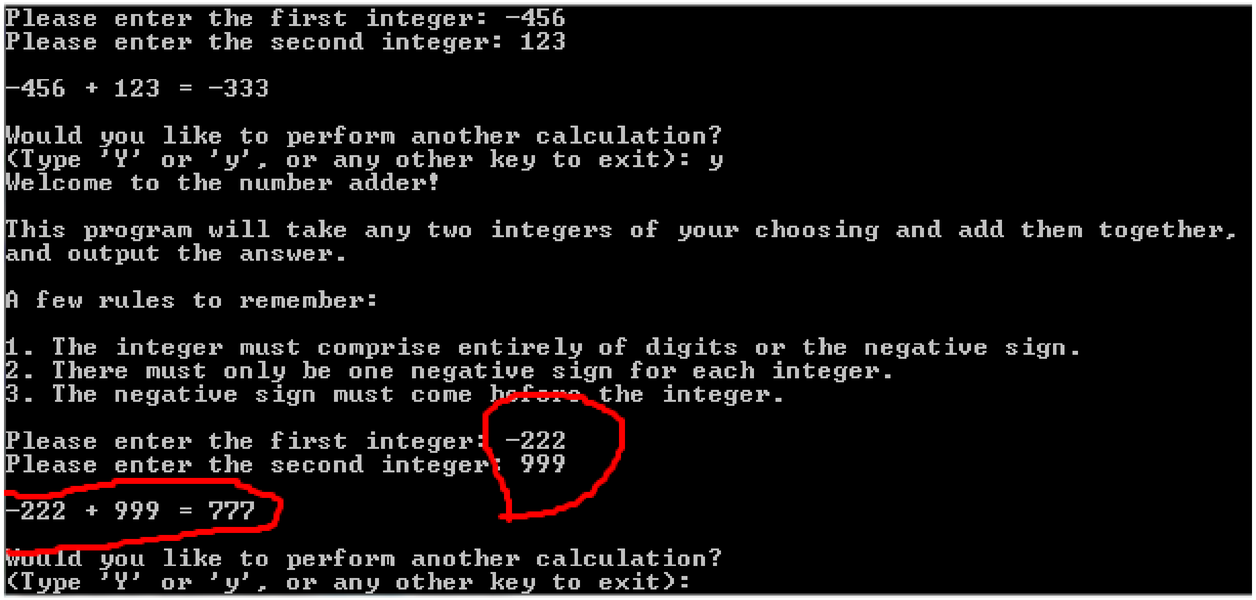
```
Please enter the first integer: -999
Please enter the second integer: -222

-999 + -222 = -1221

Would you like to perform another calculation?
(Type 'Y' or 'y', or any other key to exit): y
Welcome to the number adder!

This program will take any two integers of your choosing and add them together,
and output the answer.

A few rules to remember:

1. The integer must comprise entirely of digits or the negative sign.
2. There must only be one negative sign for each integer.
3. The negative sign must come before the integer.

Please enter the first integer: -456
Please enter the second integer: 123

-456 + 123 = -333

Would you like to perform another calculation?
(Type 'Y' or 'y', or any other key to exit):
```

This is a proof of adding a positive and negative number (subtraction). This example does not require any borrowing, and the first number to appear has the larger absolute value.



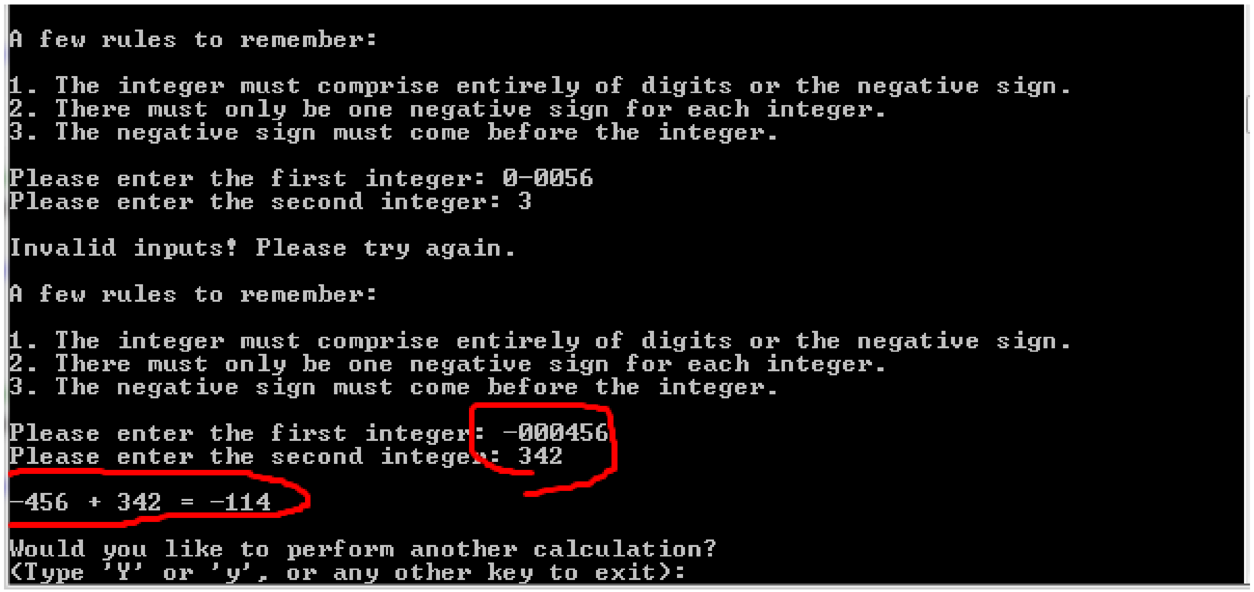
```
Please enter the first integer: -222
Please enter the second integer: 999

-222 + 999 = 777

Would you like to perform another calculation?
(Type 'Y' or 'y', or any other key to exit): y
Welcome to the number adder!

This program will take any two integers of your choosing and add them together,
and output the answer.

A few rules to remember:

1. The integer must comprise entirely of digits or the negative sign.
2. There must only be one negative sign for each integer.
3. The negative sign must come before the integer.

Please enter the first integer: 564
Please enter the second integer: -475

564 + -475 = 89

Would you like to perform another calculation?
(Type 'Y' or 'y', or any other key to exit):
```

This is a proof of adding a positive and negative number (subtraction). In this example, there is some borrowing involved (Ones number place: 4 – 5 = - 1).



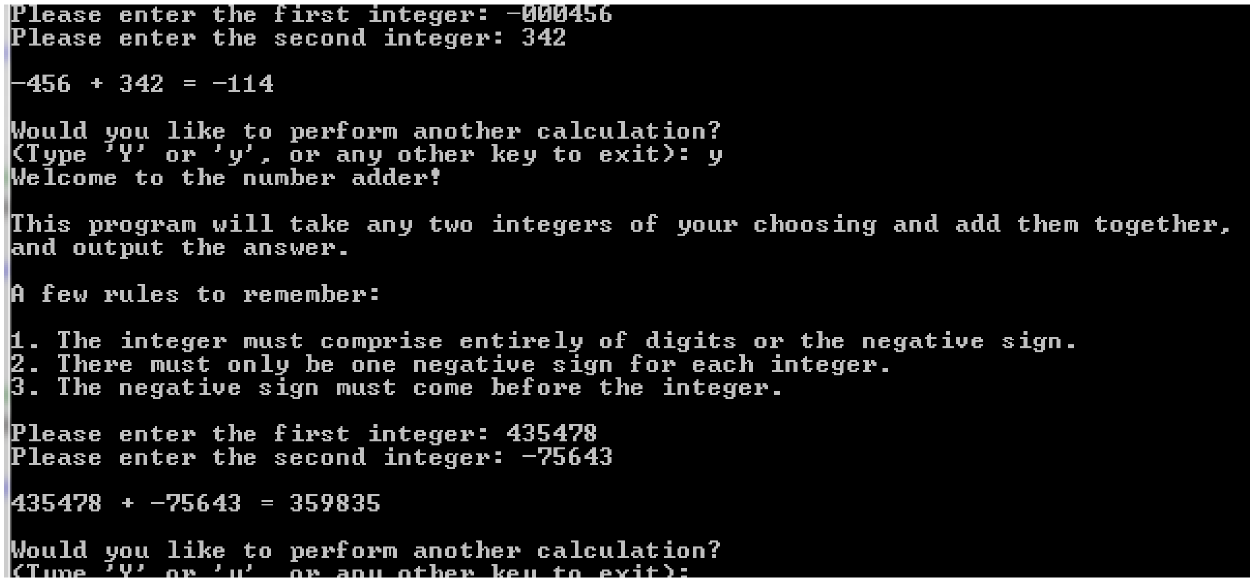
```
Please enter the first integer: -456
Please enter the second integer: 123

-456 + 123 = -333

Would you like to perform another calculation?
(Type 'Y' or 'y', or any other key to exit): y
Welcome to the number adder!

This program will take any two integers of your choosing and add them together,
and output the answer.

A few rules to remember:

1. The integer must comprise entirely of digits or the negative sign.
2. There must only be one negative sign for each integer.
3. The negative sign must come before the integer.

Please enter the first integer: -222
Please enter the second integer: 999

-222 + 999 = 777

Would you like to perform another calculation?
(Type 'Y' or 'y', or any other key to exit):
```

This is a proof of adding a positive and negative number (subtraction). In this example, the number entered first does not have the larger absolute value, so the second number is set to be on the "top" in the algorithm.

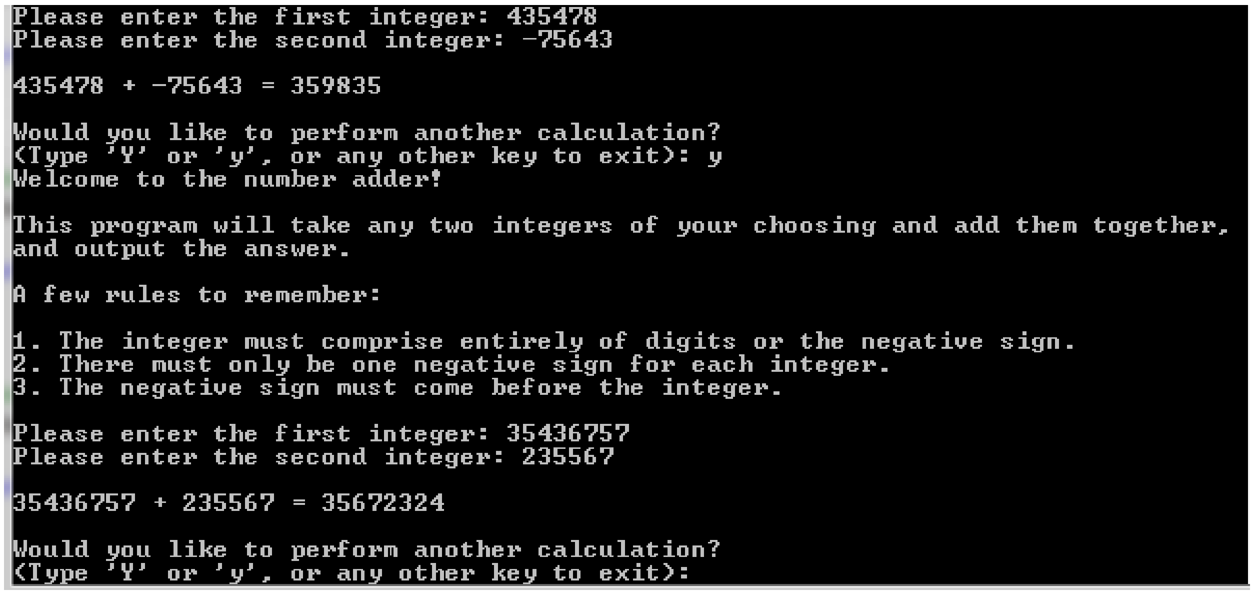


```
A few rules to remember:

1. The integer must comprise entirely of digits or the negative sign.
2. There must only be one negative sign for each integer.
3. The negative sign must come before the integer.

Please enter the first integer: 0-0056
Please enter the second integer: 3

Invalid inputs! Please try again.

A few rules to remember:

1. The integer must comprise entirely of digits or the negative sign.
2. There must only be one negative sign for each integer.
3. The negative sign must come before the integer.

Please enter the first integer: -000456
Please enter the second integer: 342

-456 + 342 = -114

Would you like to perform another calculation?
(Type 'Y' or 'y', or any other key to exit):
```

This is proof of the extraneous zero algorithm removing extra zeroes, making it more readable for the function and for the user experience when it is presented again.



```
Please enter the first integer: -000456
Please enter the second integer: 342

-456 + 342 = -114

Would you like to perform another calculation?
(Type 'Y' or 'y', or any other key to exit): y
Welcome to the number adder!

This program will take any two integers of your choosing and add them together,
and output the answer.

A few rules to remember:

1. The integer must comprise entirely of digits or the negative sign.
2. There must only be one negative sign for each integer.
3. The negative sign must come before the integer.

Please enter the first integer: 435478
Please enter the second integer: -75643

435478 + -75643 = 359835

Would you like to perform another calculation?
(Type 'Y' or 'y', or any other key to exit):
```



```
Please enter the first integer: 435478
Please enter the second integer: -75643

435478 + -75643 = 359835

Would you like to perform another calculation?
(Type 'Y' or 'y', or any other key to exit): y
Welcome to the number adder!

This program will take any two integers of your choosing and add them together,
and output the answer.

A few rules to remember:

1. The integer must comprise entirely of digits or the negative sign.
2. There must only be one negative sign for each integer.
3. The negative sign must come before the integer.

Please enter the first integer: 35436757
Please enter the second integer: 235567

35436757 + 235567 = 35672324

Would you like to perform another calculation?
(Type 'Y' or 'y', or any other key to exit):
```

These pictures show random examples of the program computing larger numbers than the previous examples.

Lessons Learned

One of the difficult parts of the project was that we had to take in the input and store the value as a string. We then had to reverse the input so that we could add them in an easier fashion. It is very important to make the code readable so when we go back to change any part of the function, we know exactly what was happening at all times. This ensures that the program will always run as intended. This program also makes use of the ASCII code to convert each digit, because it was taken in as char (an individual character in a string), into an integer that the computer can perform arithmetic on. Although a brief explanation to it was given last year in previous classes, I did not have a good grasp on how it worked or why it would be needed in any programs but this project showed me how it can be used to solve problems. This is something that will be helpful later on.

Another new feature that I did not know before was the .erase and .insert functions. It had never been introduced in any of the previous computer science courses at Fordham but we needed to use it for this program to remove (or insert) the negative sign at the beginning of a negative number. This allowed us to make the program more clear and concise because there was no need to write a separate function dedicated to doing this task. The same applies for the .insert function which made the program easier to read and to write.

References

1. A challenge we faced during implementation was being able to strip an input of its negative sign without having to write a separate function to create a temp string, cycle through all but the negative sign, and return the temp string. Instead of writing a new function, we decided to research a built-in way to achieve this same result (removing the negative sign from the beginning of the string, and updating its length). We found a way to do this on "cplusplus.com" where there was an informational page on a built in function showing us how to do this. We used the page as a resource for using the string.erase function. The page can be found here: http://www.cplusplus.com/reference/string/string/erase/ .

2. In a similar fashion, we also needed a way to insert the negative sign back into the front of the string when appropriate. To do this, we applied the same method of research for a built-in function to do this. "cplusplus.com" helped us again by providing knowledge on the string.insert function, which allowed us to achieve this goal without having to write our own new function. The page detailing string.insert can be found here: http://www.cplusplus.com/reference/string/string/insert/ .

3. It has undoubtably been a while since we had to perform calculations on paper, and we needed a little bit of help, specifically with dealing with subtraction. We were not sure how to handle a case where a larger number is being subtracted from a smaller number. On YouTube, we found a video that walked us through the theoretical concept of tackling this type of subtraction problem (the solution being to put the larger number on top and, if needed, make the answer negative). The video is made by Brian McLogan. The video can be found here: https://www.youtube.com/watch?v=hJWMLdDBHqw .