Documentation: Critical Thinking 1

Alejandro Ricciardi

Colorado State University Global

CSC450: Programming III

Professor: Reginald Haseltine

October 13, 2024

Contents

The Assignment Direction: 3

Git Repository 4

The Simple C++ Console Application Description: 5

Corrected Code Syntax CSC450_CT1_mod1-1: 14

Corrected Code Syntax CSC450_CT1_mod1-2: 16

Documentation: Critical Thinking 1

This documentation is part of the Critical Thinking 1 Assignment from CSC450: Programming III at Colorado State University Global. This Project Report is an overview of three programs' functionality and testing scenarios including console output screenshots. The programs re coded in C++ 23.

The Assignment Direction:

Console Application and Syntax Corrections
Demonstrate an understanding of basic C++ programming concepts by completing the following:

1. Create a simple C++ console application using Eclipse IDE that will accomplish the following:
    1. First Name,
    2. Last Name,
    3. Street Address
    4. City
    5. Zip code

    Prints the following information for a fictional person:
2. Given the provided code in file CSC450_CT1_mod1-1.cpp, correct all syntax errors so that the code will compile correctly.
3. Given the provided code in file CSC450_CT1_mod1-2.cpp, correct all syntax errors so that the code will compile correctly.

Compile and submit your pseudocode, source code, and screenshots of the application executing the application, the results and your GIT repository in a single document.

More Instruction:
To be eligible to receive full credit for your submission you must follow these submission requirements.

1) Put each of your 3 C++ source files into a separate .cpp file. Note that I execute all your programs to check them out.

2) In a Word or PDF "Documentation" file, labeled as such, put a copy of your C++ source code and execution output screen snapshots for each of the 3 programs.

3) Include a screenshot showing a listing of your C++ source code in a GitHub repository on github.com. Do not simply include a link to your GitHub repository.

4) Put all files into a single .zip file, and submit ONLY this .zip file for your solution. Do not submit any other separate Word, PDF, or .cpp files.

⚠️ **My notes:**

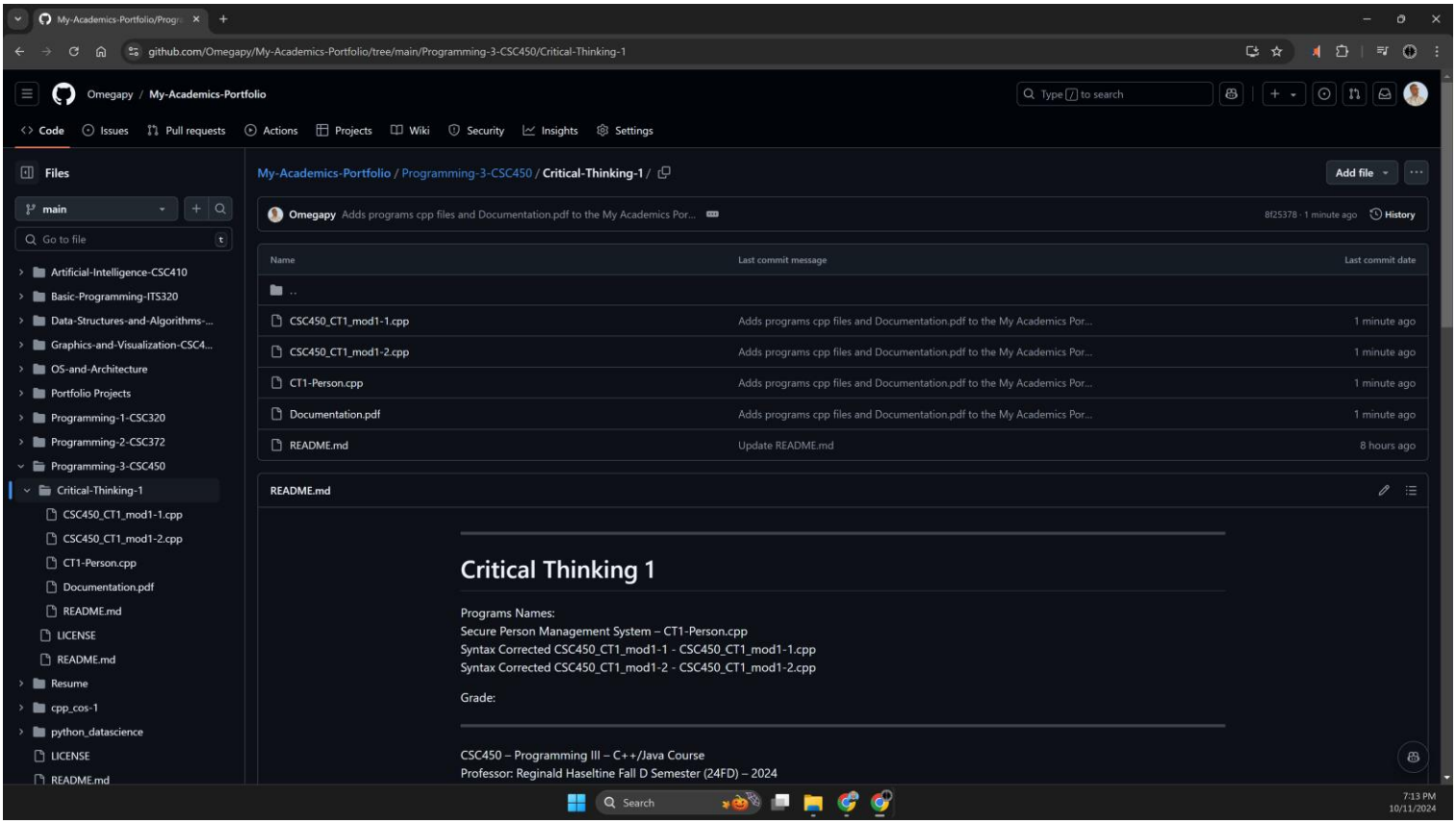
- The simple C++ console application is in file **CTA-1-Person.cpp**
- Corrected code syntax programs are found in the following files:
    - **CSC450_CT1_mod1-1.cpp**
    - **CSC450_CT1_mod1-2.cpp**
- I probably went beyond what was required for the simple C++ console application

Git Repository

I use GitHub as my Distributed Version Control System (DVCS), the following is a link to my GitHub, Omegapy.
My GitHub repository that is used to store this assignment is named My-Academics-Portfolio.

The link to this specific assignment is: https://github.com/Omegapy/My-Academics-Portfolio/tree/main/Programming-3-CSC450/Critical-Thinking-1

Image of the GitHub repository for this assignment:



Continue next page

The Simple C++ Console Application Description:

Secure Person Management System
This program is a small procedural C++ application that manages an array of Person objects.
It tests and implements secure coding practices to mitigate vulnerabilities such as:

- Buffer overflows
- Integer overflows
- Incorrect type conversions
- Null pointer dereferencing

Please see the full program Code in Figure 1

Buffer overflows:
An illustration of the buffer overblows vulnerability management is found in the following function:

```
/** ------------------------------------------------------------------------------------------
  Limits string length to MAX_STRING_LENGTH characters
  Truncates the string if it exceeds the maximum length and issues a warning
  Used by createPerson() and createPersonFull() functions

  Returns verified string
------------------------------------------------------------------------------------------ **/
// Function to limit string length to MAX_STRING_LENGTH characters
string static limitStringLength(const string& input) {
    if (input.length() > MAX_STRING_LENGTH) {
        // Security measure: Prevent buffer overflows by limiting string length
        // Truncate the string and issue a warning
        cerr << "Warning --- Input string exceeded maximum length of "
            << MAX_STRING_LENGTH << " characters and has been truncated." << endl;
        return input.substr(0, MAX_STRING_LENGTH);
    }
    return input;
}
```

This prevents an inputted string from surpassing the size of the string buffer data type.

Integer overflows
An illustration of the integer overblows vulnerability management is found in the following function:

```
/** ------------------------------------------------------------------------------------------
    Increments numOfPersons and check for integer overflow

    Returns true if the incrementation is successfull, false if UINT_MAX is reached
    UINT_MAX, Maxumum size of an unsigned int
------------------------------------------------------------------------------------------ **/
bool static incrementNumOfPersons(unsigned& counter) {
    // Check if counter has reached the maximum value for unsigned int
    if (counter == UINT_MAX) {
        // Security measure: Prevent integer overflow by checking maximum value
        cerr << "\nError --- Maximum number of persons reached!" << endl;
        return false;
    }
    // Increment the counter safely
    ++counter;
    return true;
}
```

This prevents an inputted integer from surpassing the size allowed for an unsigned integer data type. Also, note that an unsigned integer is a non-negative.

Incorrect type conversions
An illustration of an incorrect type conversion management is found in the following code lines:

```
// Incorrectly assign negative value to unsigned personNum
persons[2].personNum = static_cast<unsigned>(negativeValue); // negativeValue = -5
// Security measure: Check if personNum is valid
if (static_cast<int>(persons[2].personNum) < 0) {
    cerr << "\nIncorrect type conversion --- Negative value assigned to unsigned personNum." <<
endl;
}
else {
    cout << "Person number is: " << persons[2].personNum << endl;
}
```

This prevents an inputted negative integer from being converted into an unsigned integer, which can only represent non-negative whole numbers.

Null pointer dereferencing
An illustration of null pointer dereferencing virality management is found in the following code lines:

```
// Attempt to access member of null pointer
if (personPtr == nullptr) {
    cerr << "\nPerson pointer is null! Cannot use!." << endl;
}
else {
    cout << "\nPerson first name: " << personPtr->firstName << endl;
}
```

This prevents a null pointer object person value from being cast into a person struct member variable.

Note that in the following struct construct unused person struct member variables are assigned the value “nan”:

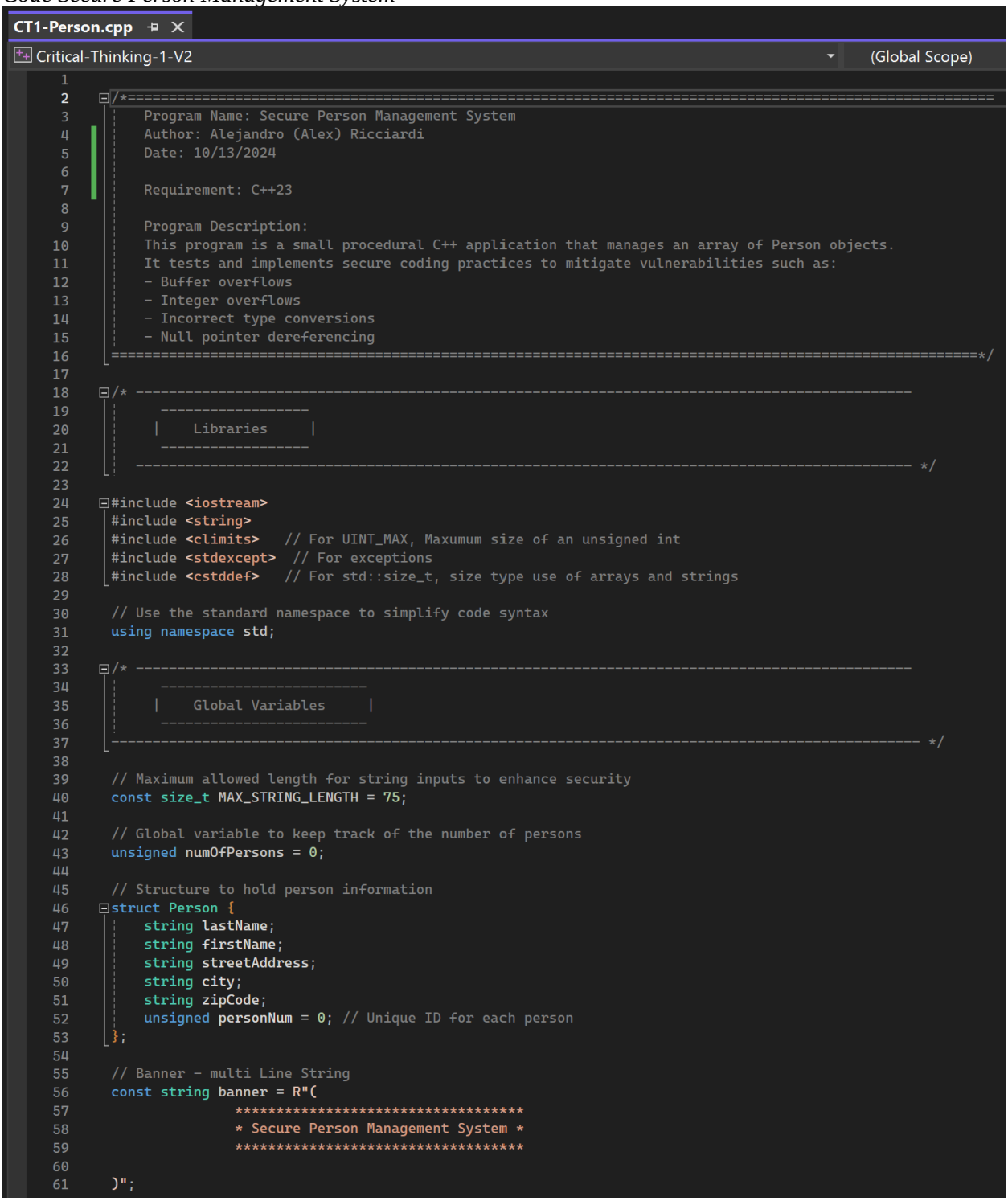
```
/** ----------------------------------------------------------------------------------------
  struck Person construtor-1
  Create a new Person with whth only lastName and firstName arguments inputted
  Uses limitStringLength() to verify string lenght

  Returns true if person created successfully, false otherwise
---------------------------------------------------------------------------------------- **/
bool static createPerson(Person& person, const string& lastNameInput, const string& firstNameInput)
{
    // Safely increment the global numOfPersons counter
    if (!incrementNumOfPersons(numOfPersons)) {
        return false;
    }
    // Limit the length of input strings to prevent buffer overflows
    person.lastName = limitStringLength(lastNameInput);
    person.firstName = limitStringLength(firstNameInput);
    // Initialize address fields with "nan" (Not Available)
    // "nan" is implemented to so the variables are not null
    person.streetAddress = "nan";
    person.city = "nan";
    person.zipCode = "nan";
    // Assign a unique person number
    person.personNum = numOfPersons;
    cout << "\nA person with number id: " << person.personNum << " was created successfully!" <<
endl;
    return true;
}
```

This prevents a null pointer struct person member value to be passed by accident.

Figure 1

Code Secure Person Management System



```
CT1-Person.cpp
Critical-Thinking-1-V2                                    (Global Scope)
/*=============================================================================================================
   Program Name: Secure Person Management System
   Author: Alejandro (Alex) Ricciardi
   Date: 10/13/2024

   Requirement: C++23

   Program Description:
   This program is a small procedural C++ application that manages an array of Person objects.
   It tests and implements secure coding practices to mitigate vulnerabilities such as:
   - Buffer overflows
   - Integer overflows
   - Incorrect type conversions
   - Null pointer dereferencing
=============================================================================================================*/

/* ------------------------------------------------------------------------------------------------
      ------------------
     |    Libraries     |
      ------------------
  ------------------------------------------------------------------------------------------------ */

#include <iostream>
#include <string>
#include <climits>   // For UINT_MAX, Maxumum size of an unsigned int
#include <stdexcept>  // For exceptions
#include <cstddef>   // For std::size_t, size type use of arrays and strings

// Use the standard namespace to simplify code syntax
using namespace std;

/* ------------------------------------------------------------------------------------------------
      -------------------------
     |    Global Variables     |
      -------------------------
------------------------------------------------------------------------------------------------ */

// Maximum allowed length for string inputs to enhance security
const size_t MAX_STRING_LENGTH = 75;

// Global variable to keep track of the number of persons
unsigned numOfPersons = 0;

// Structure to hold person information
struct Person {
    string lastName;
    string firstName;
    string streetAddress;
    string city;
    string zipCode;
    unsigned personNum = 0; // Unique ID for each person
};

// Banner - multi Line String
const string banner = R"(
                **********************************
                * Secure Person Management System *
                **********************************

)";
```

Continue next page

```
62
63    // ================================================================================================================
64    /* ---------------------------------------------------------------------------------------------
65        ----------------------------
66       |    Function Declaration    |
67        ----------------------------
68    --------------------------------------------------------------------------------------------- */
69    // ================================================================================================================
70
71    /** ---------------------------------------------------------------------------------------------
72     * Increments numOfPersons and check for integer overflow
73     *
74     * Returns true if the incrementation is successfull, false if UINT_MAX is reached
75     * UINT_MAX, Maxumum size of an unsigned int
76     * ---------------------------------------------------------------------------------------------- **/
77    static bool incrementNumOfPersons(unsigned& counter);
78
79    /** ---------------------------------------------------------------------------------------------
80     * Limits string length to MAX_STRING_LENGTH characters
81     * Truncates the string if it exceeds the maximum length and issues a warning
82     * Used by createPerson() and createPersonFull() functions
83     *
84     * Returns verified string
85     * ---------------------------------------------------------------------------------------------- **/
86    static string limitStringLength(const string& input);
87
88    /** ---------------------------------------------------------------------------------------------
89     * struck Person construtor-1
90     * Create a new Person with whth only lastName and firstName arguments inputted
91     * Uses limitStringLength() to verify string lenght
92     *
93     * Returns true if person created successfully, false otherwise
94     * ---------------------------------------------------------------------------------------------- **/
95    static bool createPerson(Person& person, const string& lastName, const string& firstName);
96
97    /** ---------------------------------------------------------------------------------------------
98     * struck Person construtor-2
99     * Create a new Person with all arguments inputted
100    * Uses limitStringLength() to verify string lenght
101    *
102    * Returns true if person created successfully, false otherwise
103    * ---------------------------------------------------------------------------------------------- **/
104   static bool createPersonFull(Person& person, const string& lastName, const string& firstName,
105       const string& streetAddress, const string& city,
106       const string& zipCode);
107
108   /** ---------------------------------------------------------------------------------------------
109    * Displays the contents of the persons array
110    * "size_t iterate" is used to iterate the array
111    * ---------------------------------------------------------------------------------------------- **/
112   static void displayPersons(const Person persons[], size_t iterate);
113
114   /** ---------------------------------------------------------------------------------------------
115    * Displays a person data
116    * "size_t index" is the index of the person object in the persons array
117    * ---------------------------------------------------------------------------------------------- **/
118   static void displayAPerson(const Person persons[], size_t index);
119
```

Continue next page

```
// ================================================================================================================
/* ----------------------------------------------------------------------------------------------------------------
     ---------------------
    |    Main Function    |
     ---------------------

    tests secure coding practices such as:
    - Buffer overflows
    - Integer overflows
    - Incorrect type conversions
    - Null pointer dereferencing
---------------------------------------------------------------------------------------------------------------- */
// ================================================================================================================
int main() {

    /* ---------------------------------
     ---------------
    |   Variables   |
     --------------- */

    // ----- Variables used to test code vulnerability
    int negativeValue = -5;      // use for unsigned int vulnerability
    string longString(100, 'A'); // Create a string with 100 'A's use for string overflow
    Person* personPtr = nullptr; // Initialize pointer to null use for null pointer vulnerability
    void* voidPerson1Ptr; // Generic type pointer use for void pointer vulnerability

    //---- Create an array of persons (size 5)
    Person persons[5];

    /* ---------------------------------
     -------------
    |   Program   |
     ------------- */

    cout << banner << endl; // std::endl forces flushing of the buffer - good pratice


    // ---------------------------------- Test 1: Buffer Overflow with Overly Long Strings

    cout << "------------------------------------------------------------\n"
        << "Test 1: Buffer Overflow with Overly Long Strings\n"
        << "Creates a person with first and last names that are 100 characters long, filled with the letter 'A'.\n"
        << endl;
    // not expected fail due tolimitStringLength() truncating functionality
    if (!createPerson(persons[0], longString, longString)) {
        cout << "\n Error --- Failed to create person ---" << endl;
    }
    displayPersons(persons, 1);

    // ---------------------------------- Test 2: Integer Overflow when Creating Many Persons
    cout << "\n------------------------------------------------------------\n"
        << "Test 2: Integer Overflow when Creating Too Many Persons\n"
        << "Simulate numOfPersons = UINT_MAX, Maxumum size of an unsigned int\n"
        << "\nTrying to create a new person"
        << endl;

    numOfPersons = UINT_MAX;
    if (!createPerson(persons[1], "Doe", "John")) {
        // Expected to fail due to integer overflow securety
        cout << "Failed to create person due to integer overflow." << endl;
    }
    // Reset numOfPersons for further tests
    numOfPersons = 1;
```

Continue next page

```cpp
    // ---------------------------------- Test 3: Incorrect Type Conversion
    cout << "\n-----------------------------------------------------------------\n"
        << "Test 3: Incorrect Type Conversion" << endl;

    // Creating a new person
    if (!createPerson(persons[1], "Conversion", "Alexandria")) {
        cout << "\n Error --- Failed to create person ---" << endl;
    }
    displayAPerson(persons, 1);

    cout << "\nTring to assign persons[1].personNum = -5, which is a negative value"
        << endl;

    // Incorrectly assign negative value to unsigned personNum
    persons[2].personNum = static_cast<unsigned>(negativeValue); // negativeValue = -5
    // Security measure: Check if personNum is valid
    if (static_cast<int>(persons[2].personNum) < 0) {
        cerr << "\nIncorrect type conversion --- Negative value assigned to unsigned personNum." << endl;
    }
    else {
        cout << "Person number is: " << persons[2].personNum << endl;
    }

    // ---------------------------------- Test 4: Null Pointer Dereferencing with Person Pointer
    cout << "\n-----------------------------------------------------------------\n"
        << "Test 4: Testing Null Pointer before use - null pointer dereferencing\n"
        << "Checking if personPtr is null, and it is."
        << endl;

    // Attempt to access member of null pointer
    if (personPtr == nullptr) {
        cerr << "\nPerson pointer is null! Cannot use!." << endl;
    }
    else {
        cout << "\nPerson first name: " << personPtr->firstName << endl;
    }

    // ---------------------------------- Additional test: Properly adding and displaying persons
    cout << "\n-----------------------------------------------------------------\n"
        << "\nAdditional Test: Adding and Displaying Persons" << endl;

    // Creating a new person
    if (!createPerson(persons[2], "More", "Bob")) {
        cout << "\n Error --- Failed to create person ---" << endl;
    }
    if (!createPersonFull(persons[3], "Marquez", "Anita", "456 Ai Street", "Robot Town", "77442")) {
        cout << "\n Error --- Failed to create person ---" << endl;
    }
    if (!createPersonFull(persons[4], "Wan", "Lu", "777 LLM Street", "AI Town", "77772")) {
        cout << "\n Error --- Failed to create person ---" << endl;
    }

    displayPersons(persons, 5);

    return 0;
}
```

Continue next page

```
241
242   // ================================================================================================================
243   /* ------------------------------------------------------------------------------------------
244         ---------------------------
245       |    Function Definition     |
246         ---------------------------
247     ------------------------------------------------------------------------------------------ */
248   // ================================================================================================================
249
250   /** ------------------------------------------------------------------------------------------
251       Increments numOfPersons and check for integer overflow
252
253       Returns true if the incrementation is successfull, false if UINT_MAX is reached
254       UINT_MAX, Maxumum size of an unsigned int
255     ------------------------------------------------------------------------------------------ **/
256   bool static incrementNumOfPersons(unsigned& counter) {
257       // Check if counter has reached the maximum value for unsigned int
258       if (counter == UINT_MAX) {
259           // Security measure: Prevent integer overflow by checking maximum value
260           cerr << "\nError --- Maximum number of persons reached!" << endl;
261           return false;
262       }
263       // Increment the counter safely
264       ++counter;
265       return true;
266   }
267
268   // ------------------------------------------------------------------------------------------
269
270   /** ------------------------------------------------------------------------------------------
271     Limits string length to MAX_STRING_LENGTH characters
272     Truncates the string if it exceeds the maximum length and issues a warning
273     Used by createPerson() and createPersonFull() functions
274
275     Returns verified string
276     ------------------------------------------------------------------------------------------ **/
277   // Function to limit string length to MAX_STRING_LENGTH characters
278   string static limitStringLength(const string& input) {
279       if (input.length() > MAX_STRING_LENGTH) {
280           // Security measure: Prevent buffer overflows by limiting string length
281           // Truncate the string and issue a warning
282           cerr << "Warning --- Input string exceeded maximum length of "
283               << MAX_STRING_LENGTH << " characters and has been truncated." << endl;
284           return input.substr(0, MAX_STRING_LENGTH);
285       }
286       return input;
287   }
288
289   // ------------------------------------------------------------------------------------------
290
291   /** ------------------------------------------------------------------------------------------
292     struck Person construtor-1
293     Create a new Person with whth only lastName and firstName arguments inputted
294     Uses limitStringLength() to verify string lenght
295
296     Returns true if person created successfully, false otherwise
297     ------------------------------------------------------------------------------------------ **/
298   bool static createPerson(Person& person, const string& lastNameInput, const string& firstNameInput) {
299       // Safely increment the global numOfPersons counter
300       if (!incrementNumOfPersons(numOfPersons)) {
301           return false;
302       }
303       // Limit the length of input strings to prevent buffer overflows
304       person.lastName = limitStringLength(lastNameInput);
305       person.firstName = limitStringLength(firstNameInput);
306       // Initialize address fields with "nan" (Not Available)
307       // "nan" is implemented to so the variables are not null
308       person.streetAddress = "nan";
309       person.city = "nan";
310       person.zipCode = "nan";
311       // Assign a unique person number
312       person.personNum = numOfPersons;
313       cout << "\nA person with number id: " << person.personNum << " was created successfully!" << endl;
314       return true;
315   }
```

```
316
317     // ------------------------------------------------------------------------------------------------
318
319     /** ------------------------------------------------------------------------------------------
320      * struck Person construtor-2
321      * Create a new Person with all arguments inputted
322      * Uses limitStringLength() to verify string lenght
323      *
324      * Returns true if person created successfully, false otherwise
325      * ------------------------------------------------------------------------------------------ **/
326     bool static createPersonFull(Person& person, const string& lastNameInput, const string& firstNameInput,
327         const string& streetAddressInput, const string& cityInput,
328         const string& zipCodeInput) {
329         // Safely increment the global numOfPersons counter
330         if (!incrementNumOfPersons(numOfPersons)) {
331             return false;
332         }
333         // Limit the length of input strings to prevent buffer overflows
334         person.lastName = limitStringLength(lastNameInput);
335         person.firstName = limitStringLength(firstNameInput);
336         person.streetAddress = limitStringLength(streetAddressInput);
337         person.city = limitStringLength(cityInput);
338         person.zipCode = limitStringLength(zipCodeInput);
339         // Assign a unique person number
340         person.personNum = numOfPersons;
341         cout << "\nA person with number id: " << person.personNum << " was created successfully!" << endl;
342         return true;
343     }
344
345     // ------------------------------------------------------------------------------------------------
346
347     /** ------------------------------------------------------------------------------------------
348      * Displays the contents of the persons array
349      * "size_t iterate" is used to iterate the array
350      * ------------------------------------------------------------------------------------------ **/
351     void static displayPersons(const Person persons[], size_t size) {
352         cout << "Persons List (Total persons created: " << numOfPersons << "):" << endl;
353         for (size_t i = 0; i < size; ++i) {
354             cout << "Person " << i + 1 << ": "
355                 << persons[i].personNum << " "
356                 << persons[i].firstName << " "
357                 << persons[i].lastName << ", "
358                 << persons[i].streetAddress << ", "
359                 << persons[i].city << ", "
360                 << persons[i].zipCode << endl;
361         }
362     }
363
364     // ------------------------------------------------------------------------------------------------
365
366     /** ------------------------------------------------------------------------------------------
367      * Displays a person data
368      * "size_t index" is the index of the person object in the persons array
369      * ------------------------------------------------------------------------------------------ **/
370     void static displayAPerson(const Person persons[], size_t index) {
371         cout << "Persons List (Total persons created: " << numOfPersons << "):" << endl;
372         cout << "Person " << index + 1 << ": "
373             << persons[index].personNum << " "
374             << persons[index].firstName << " "
375             << persons[index].lastName << ", "
376             << persons[index].streetAddress << ", "
377             << persons[index].city << ", "
378             << persons[index].zipCode << endl;
379
380     }
381
```

Figure 2

Output Code Secure Person Management System

```
Microsoft Visual Studio Debu

             **********************************
             * Secure Person Management System *
             **********************************

-------------------------------------------------------------
Test 1: Buffer Overflow with Overly Long Strings
Creates a person with first and last names that are 100 characters long, filled with the letter 'A'.

Warning --- Input string exceeded maximum length of 75 characters and has been truncated.
Warning --- Input string exceeded maximum length of 75 characters and has been truncated.

A person with number id: 1 was created successfully!
Persons List (Total persons created: 1):
Person 1: 1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA, nan, nan, nan

-------------------------------------------------------------
Test 2: Integer Overflow when Creating Too Many Persons
Simulate numOfPersons = UINT_MAX, Maxumum size of an unsigned int

Trying to create a new person

Error --- Maximum number of persons reached!
Failed to create person due to integer overflow.

-------------------------------------------------------------
Test 3: Incorrect Type Conversion

A person with number id: 2 was created successfully!
Persons List (Total persons created: 2):
Person 2: 2 Alexandria Conversion, nan, nan, nan

Tring to assign persons[1].personNum = -5, which is a negative value

Incorrect type conversion --- Negative value assigned to unsigned personNum.

-------------------------------------------------------------
Test 4: Testing Null Pointer before use - null pointer dereferencing
Checking if personPtr is null, and it is.

Person pointer is null! Cannot use!.

-------------------------------------------------------------

Additional Test: Adding and Displaying Persons

A person with number id: 3 was created successfully!

A person with number id: 4 was created successfully!

A person with number id: 5 was created successfully!
Persons List (Total persons created: 5):
Person 1: 1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA, nan, nan, nan
Person 2: 2 Alexandria Conversion, nan, nan, nan
Person 3: 3 Bob More, nan, nan, nan
Person 4: 4 Anita Marquez, 456 Ai Street, Robot Town, 77442
Person 5: 5 Lu Wan, 777 LLM Street, AI Town, 77772

P:\CSU projects\Programming-3-CSC450\Programming-3-cpp\Critical-Thinking-1-V2\x64\Debug\Critical-Thinking-1-V2.exe (process 38936) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

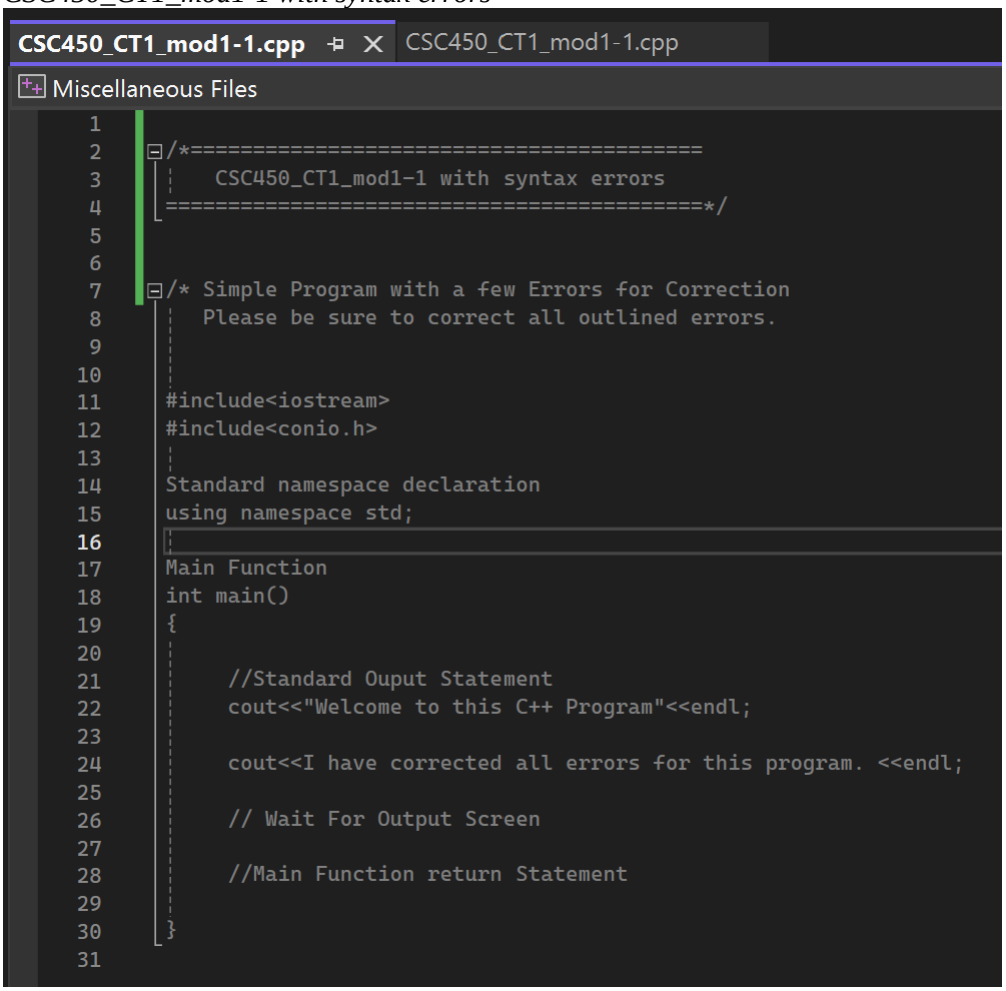
Note: The output of the program showcases a series of tests performed to test the program functionality against the Buffer overflows, Integer overflows, Incorrect type conversions, and Null pointer dereferencing vulgarities. It also displays person data.

Continue next page

Corrected Code Syntax CSC450_CT1_mod1-1:

In this part of the assignment showcases how I corrected code syntaxes from the given C++ program, CSC450_CT1_mod1-1.cpp.

Figure 3
CSC450_CT1_mod1-1 with syntax errors

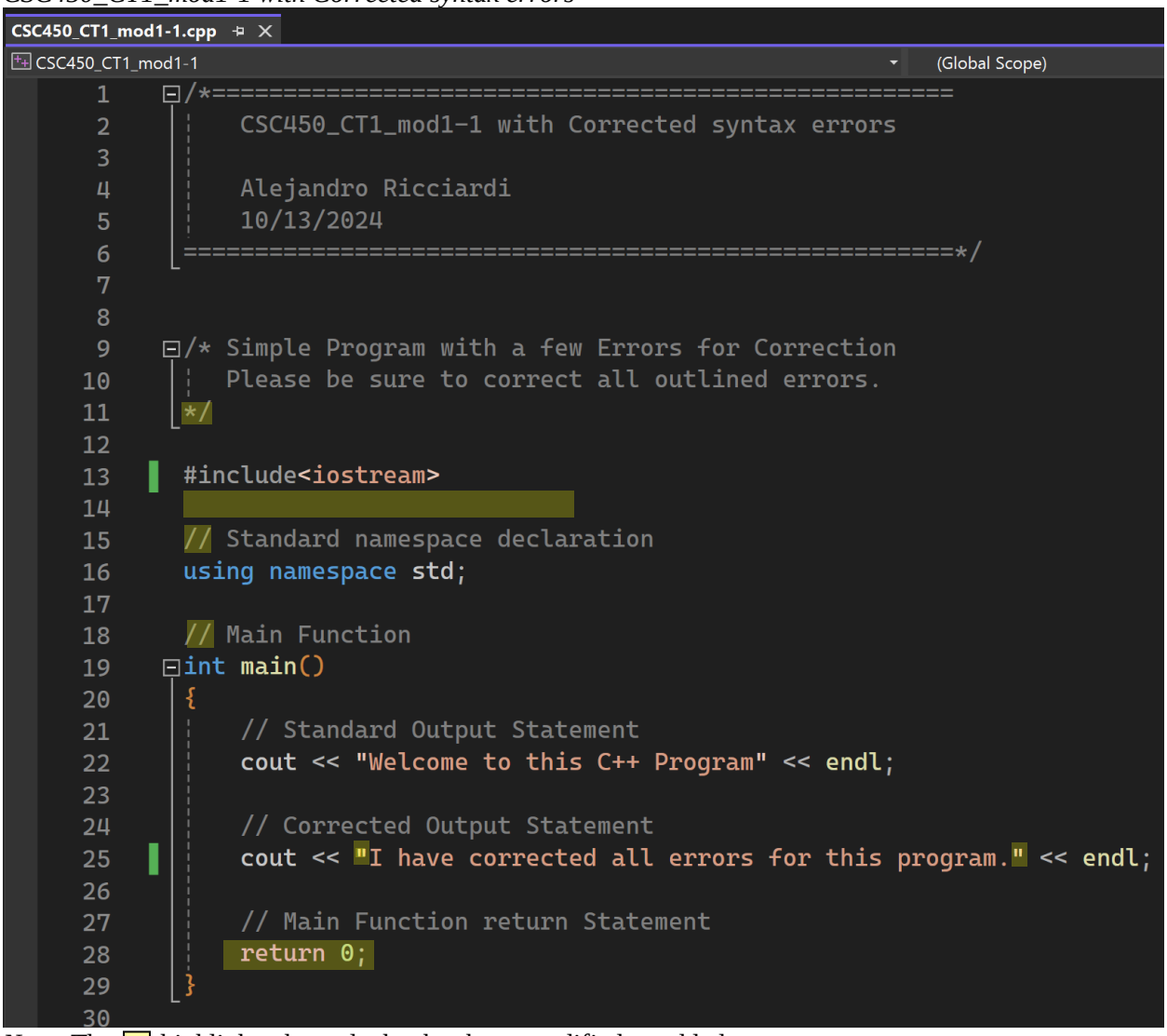


Note: All the code is commented out, this program runs without error. However, it does not output or compute anything as the compiler ignores all the code lines treating it as comments.

Continue next page

Figure 4

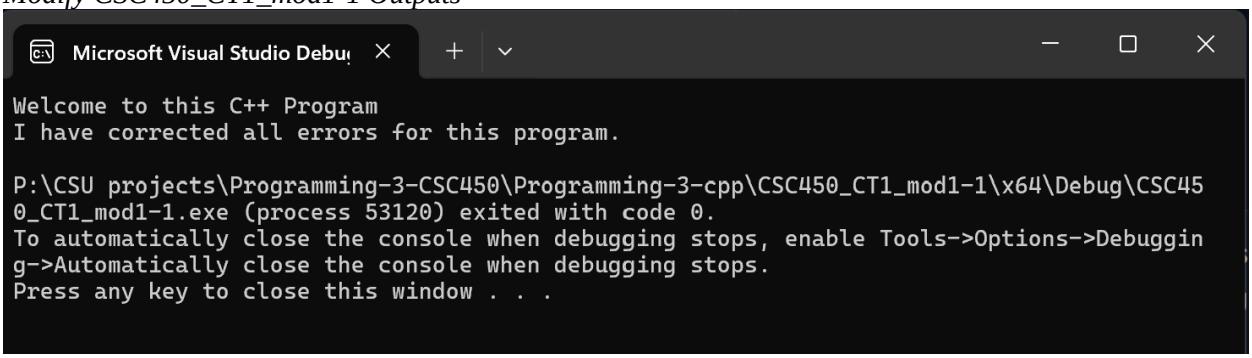
CSC450_CT1_mod1-1 with Corrected syntax errors



Note: The ☐ highlights the code that has been modified or added.

Figure 5

Modify CSC450_CT1_mod1-1 Outputs



I modified:

- Lines 1-6, added header comment.
- Line 14, removed the '#include<conio.h>', which is outdated and unnecessary.
- Line 15, added missing '//' in to comment line.
- Line 18, added space after '//' to comment line.
- Line 26, removed comment line '// Wait For Output Screen".
- Line 28. added 'return 0;'

Corrected Code Syntax CSC450_CT1_mod1-2:

In this part of the assignment showcases how I corrected code syntaxes from the given C++ program, CSC450_CT1_mod1-2.cpp.

Figure 6

CSC450_CT1_mod1-2 with syntax errors

CSC450_CT1_mod1-2.cpp

```
1  /*=====================================================
2     CSC450_CT1_mod1-2 with syntax errors
3  =====================================================*/
4
5
6  #include<iostream>
7  #include<conio.h>
8
9   Standard namespace declaration
10  using namespace std;
11
12  Main Function
13 int main()
14  {
15
16      double myMoney = 1000.50;  //this should be printed out
17
18
19      Standard Ouput Statement
20      cout"Please be sure to correct all syntax errors in this program"<<endl;
21
22      cout<<"I have corrected all errors for this program. <<endl;
23
24      cout<<" The total amount of money available is = "<<   <<endl;
25
26      // Wait For Output Screen
27
28      Main Function return Statement
29      return 0;
30  }
31
```

Note: some of the code lines are flagged by IDE as errors (red squeaky lines). This program generates compiling errors.

Figure 7

CSC450_CT1_mod1-2 with Corrected syntax errors

CSC450_CT1_mod1-2.cpp*

CSC450_CT1_mod1-2.cpp (Global Scope)

```cpp
/*=====================================================
    CSC450_CT1_mod1-2 with Corrected syntax errors

    Alejandro Ricciardi
    10/13/2024
=====================================================*/

#include<iostream>
#include<iomanip>  // setprecision

// Standard namespace declaration
using namespace std;

// Main Function
int main()
{

    double myMoney = 1000.50;  //this should be printed out

    // Standard Output Statement
    cout << "Please be sure to correct all syntax errors in this program" << endl;

    cout << "I have corrected all errors for this program." << endl;

    // Set precision to 2 decimal places
    cout << fixed << setprecision(2);

    cout << "The total amount of money available is = " << myMoney << endl;

    // Main Function return Statement
    return 0;
}
```

Note: The ☐ highlights the code that has been modified, removed, or added.

Figure 8

Modify CSC450_CT1_mod1-2 Outputs

Microsoft Visual Studio Debu

```
Please be sure to correct all syntax errors in this program
I have corrected all errors for this program.
The total amount of money available is = 1000.50

P:\CSU projects\Programming-3-CSC450\Programming-3-cpp\CSC450_CT1_mod1-2\x64\Debug\CSC45
0_CT1_mod1-2.cpp.exe (process 36448) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debuggin
g->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

I modified:

- Lines 1-6, added header comment.
- Line 9, replacd the '#include<conio.h>', which is outdated and unnecessary, with '#include<iomanip> // setprecision'.
- Line 11, added missing '//' in to comment line.
- Line 18, added space after '//' to comment line.
- Line 20, added missing '//' to comment line.
- Line 21, added missing '<<' to the character output line.
- Line 23. added missing ' " ' to the string literal.
- Lines 25-26, added '// Set precision to 2 decimal places' and 'cout << fixed << setprecision(2);'.
- Line 28, added 'myMoney' variable to the character output line.
- Line 29, removed comment line '// Wait For Output Screen".
- Line 30, added missing '//' to comment line.

As shown in Figures 1 through 8 the programs run without any issues displaying the correct outputs as expected.