

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
The simple C++ Console Application Description:	14

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:
 - o **CSC450_CT1_mod1-1.cpp**
 - o **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:

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 overflows 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 overflows 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, Maximum 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;
}
```

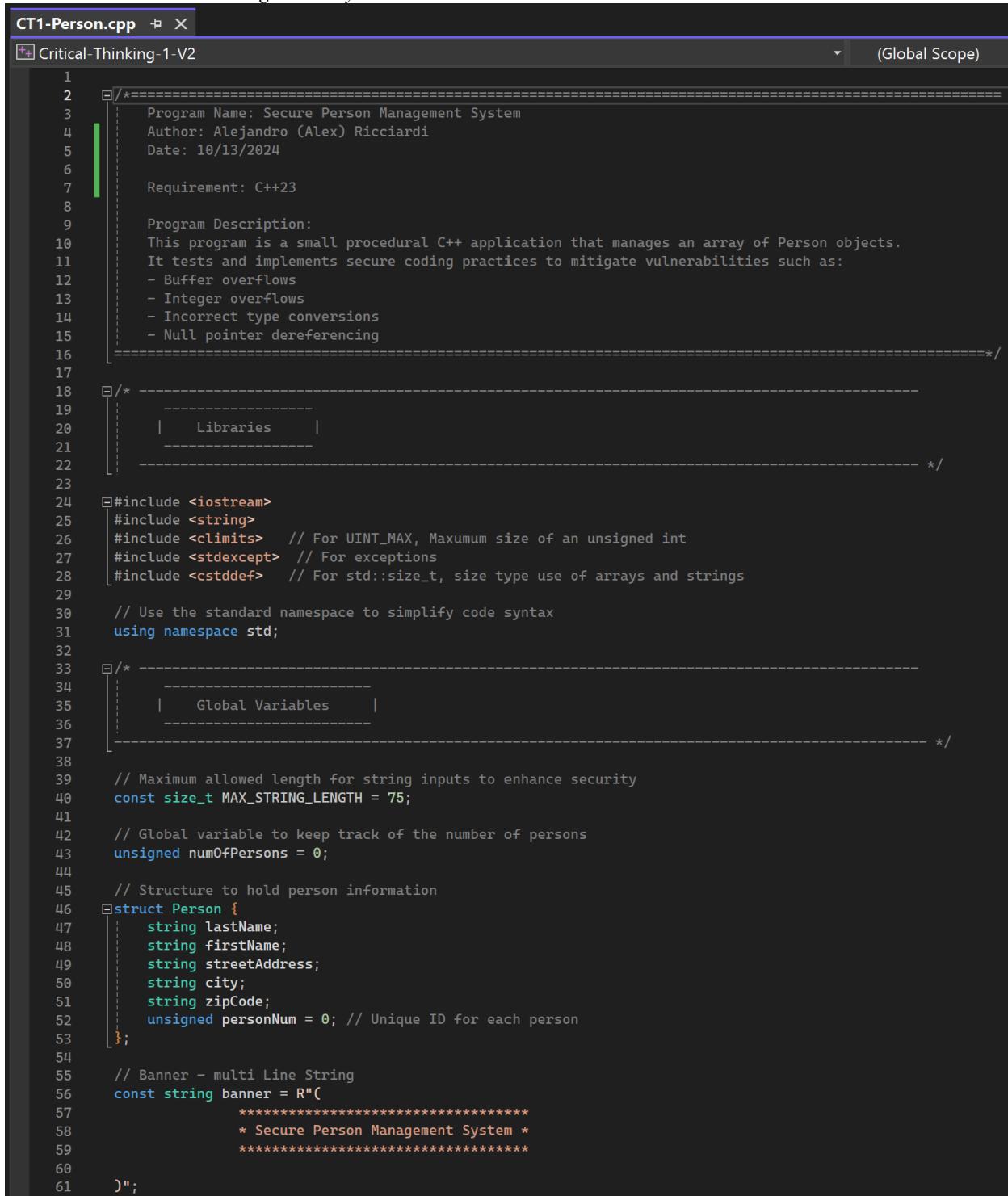
This prevents a null pointer object person value from being cast into a person struct member variable.

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

```
/** -----
 * Struck Person constructor-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 ✘ X
Critical-Thinking-1-V2 (Global Scope)

1  /*
2   *=====
3   * Program Name: Secure Person Management System
4   * Author: Alejandro (Alex) Ricciardi
5   * Date: 10/13/2024
6   *
7   * Requirement: C++23
8   *
9   * Program Description:
10  * This program is a small procedural C++ application that manages an array of Person objects.
11  * It tests and implements secure coding practices to mitigate vulnerabilities such as:
12  * - Buffer overflows
13  * - Integer overflows
14  * - Incorrect type conversions
15  * - Null pointer dereferencing
16  * =====
17  */
18  /*
19  * -----
20  * | Libraries |
21  * -----
22  */
23
24 #include <iostream>
25 #include <string>
26 #include <climits> // For UINT_MAX, Maximum size of an unsigned int
27 #include <stdexcept> // For exceptions
28 #include <cstddef> // For std::size_t, size type use of arrays and strings
29
30 // Use the standard namespace to simplify code syntax
31 using namespace std;
32
33 /*
34  * -----
35  * | Global Variables |
36  * -----
37  */
38
39 // Maximum allowed length for string inputs to enhance security
40 const size_t MAX_STRING_LENGTH = 75;
41
42 // Global variable to keep track of the number of persons
43 unsigned numOfPersons = 0;
44
45 // Structure to hold person information
46 struct Person {
47     string lastName;
48     string firstName;
49     string streetAddress;
50     string city;
51     string zipCode;
52     unsigned personNum = 0; // Unique ID for each person
53 };
54
55 // Banner - multi Line String
56 const string banner = R"(
57 ***** * Secure Person Management System * *****
58 )";

```

Continue next page

```

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

```

Continue next page

```

120 // =====-
121 /* -----
122 |   Main Function   |
123 |-----|
124
125 tests secure coding practices such as:
126 - Buffer overflows
127 - Integer overflows
128 - Incorrect type conversions
129 - Null pointer dereferencing
130
131 ----- */
132 // =====-
133 int main() {
134
135 /* -----
136 |   Variables   |
137 |----- */
138
139 // ---- Variables used to test code vulnerability
140 int negativeValue = -5;           // use for unsigned int vulnerability
141 string longString(100, 'A'); // Create a string with 100 'A's use for string overflow
142 Person* personPtr = nullptr; // Initialize pointer to null use for null pointer vulnerability
143 void* voidPersonPtr; // Generic type pointer use for void pointer vulnerability
144
145 //---- Create an array of persons (size 5)
146 Person persons[5];
147
148 /* -----
149 |   Program   |
150 |----- */
151
152 cout << banner << endl; // std::endl forces flushing of the buffer - good practice
153
154
155 // ----- Test 1: Buffer Overflow with Overly Long Strings
156
157 cout << "-----\n"
158     << "Test 1: Buffer Overflow with Overly Long Strings\n"
159     << "Creates a person with first and last names that are 100 characters long, filled with the letter 'A'.\n"
160     << endl;
161 // not expected fail due to limitStringLength() truncating functionality
162 if (!createPerson(persons[0], longString, longString)) {
163     cout << "\n Error --- Failed to create person ---" << endl;
164 }
165 displayPersons(persons, 1);
166
167 // ----- Test 2: Integer Overflow when Creating Many Persons
168 cout << "\n-----\n"
169     << "Test 2: Integer Overflow when Creating Too Many Persons\n"
170     << "Simulate numPersons = UINT_MAX, Maximum size of an unsigned int\n"
171     << "\nTrying to create a new person"
172     << endl;
173
174 numPersons = UINT_MAX;
175 if (!createPerson(persons[1], "Doe", "John")) {
176     // Expected to fail due to integer overflow security
177     cout << "Failed to create person due to integer overflow." << endl;
178 }
179
180 // Reset numPersons for further tests
181 numPersons = 1;
182

```

Continue next page

```

184 // ----- Test 3: Incorrect Type Conversion
185 cout << "\n" << "Test 3: Incorrect Type Conversion" << endl;
186
187 // Creating a new person
188 if (!createPerson(persons[1], "Conversion", "Alexandria")) {
189     cout << "\n Error --- Failed to create person ---" << endl;
190 }
191
192 displayAPerson(persons, 1);
193
194 cout << "\nTrying to assign persons[1].personNum = -5, which is a negative value"
195     << endl;
196
197 // Incorrectly assign negative value to unsigned personNum
198 persons[2].personNum = static_cast<unsigned>(negativeValue); // negativeValue = -5
199 // Security measure: Check if personNum is valid
200 if (static_cast<int>(persons[2].personNum) < 0) {
201     cerr << "\nIncorrect type conversion --- Negative value assigned to unsigned personNum." << endl;
202 }
203 else {
204     cout << "Person number is: " << persons[2].personNum << endl;
205 }
206
207 // ----- Test 4: Null Pointer Dereferencing with Person Pointer
208 cout << "\n" << "Test 4: Testing Null Pointer before use - null pointer dereferencing\n"
209     << "Checking if personPtr is null, and it is."
210     << endl;
211
212 // Attempt to access member of null pointer
213 if (personPtr == nullptr) {
214     cerr << "\nPerson pointer is null! Cannot use!." << endl;
215 }
216 else {
217     cout << "\nPerson first name: " << personPtr->firstName << endl;
218 }
219
220 // ----- Additional test: Properly adding and displaying persons
221 cout << "\n" << "\nAdditional Test: Adding and Displaying Persons" << endl;
222
223 // Creating a new person
224 if (!createPerson(persons[2], "More", "Bob")) {
225     cout << "\n Error --- Failed to create person ---" << endl;
226 }
227
228 if (!createPersonFull(persons[3], "Marquez", "Anita", "456 Ai Street", "Robot Town", "77442")) {
229     cout << "\n Error --- Failed to create person ---" << endl;
230 }
231
232 if (!createPersonFull(persons[4], "Wan", "Lu", "777 LLM Street", "AI Town", "77772")) {
233     cout << "\n Error --- Failed to create person ---" << endl;
234 }
235
236 displayPersons(persons, 5);
237
238 return 0;
239 }
```

Continue next page

```

241 // -----
242 /* -----
243 |   Function Definition   |
244 |----- */
245 // -----
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, Maximum 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 |   Limits string length to MAX_STRING_LENGTH characters
271 |   Truncates the string if it exceeds the maximum length and issues a warning
272 |   Used by createPerson() and createPersonFull() functions
273 |
274 |   Returns verified string
275 */
276 // Function to limit string length to MAX_STRING_LENGTH characters
277 string static limitStringLength(const string& input) {
278     if (input.length() > MAX_STRING_LENGTH) {
279         // Security measure: Prevent buffer overflows by limiting string length
280         // Truncate the string and issue a warning
281         // cerr << "Warning --- Input string exceeded maximum length of "
282         //       << MAX_STRING_LENGTH << " characters and has been truncated." << endl;
283         return input.substr(0, MAX_STRING_LENGTH);
284     }
285     return input;
286 }
287 // -----
288 // -----
289 /**
290 |   struct Person constructor-1
291 |   Create a new Person with whth only lastName and firstName arguments inputted
292 |   Uses limitStringLength() to verify string lenght
293 |
294 |   Returns true if person created successfully, false otherwise
295 */
296 bool static createPerson(Person& person, const string& lastNameInput, const string& firstNameInput) {
297     // Safely increment the global numOfPersons counter
298     if (!incrementNumOfPersons(numOfPersons)) {
299         return false;
300     }
301     // Limit the length of input strings to prevent buffer overflows
302     person.lastName = limitStringLength(lastNameInput);
303     person.firstName = limitStringLength(firstNameInput);
304     // Initialize address fields with "nan" (Not Available)
305     // "nan" is implemented to so the variables are not null
306     person.streetAddress = "nan";
307     person.city = "nan";
308     person.zipCode = "nan";
309     // Assign a unique person number
310     person.personNum = numOfPersons;
311     cout << "\nA person with number id: " << person.personNum << " was created successfully!" << endl;
312     return true;
313 }
314

```

```

316 // -
317
318 /**
319  * Create a new Person with all arguments inputted
320  * Uses limitStringLength() to verify string length
321  *
322  * Returns true if person created successfully, false otherwise
323  */
324
325 bool static createPersonFull(Person& person, const string& lastNameInput, const string& firstNameInput,
326 const string& streetAddressInput, const string& cityInput,
327 const string& zipCodeInput) {
328     // Safely increment the global numOfPersons counter
329     if (!incrementNumOfPersons(numOfPersons)) {
330         return false;
331     }
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
340     // Assign a unique person number
341     person.personNum = numOfPersons;
342     cout << "\nA person with number id: " << person.personNum << " was created successfully!" << endl;
343     return true;
344 }
345
346 /**
347  * Displays the contents of the persons array
348  * "size_t iterate" is used to iterate the array
349  */
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  * Displays a person data
366  * "size_t index" is the index of the person object in the persons array
367  */
368
369 void static displayAPerson(const Person persons[], size_t index) {
370     cout << "Persons List (Total persons created: " << numOfPersons << ")" :>< endl;
371     cout << "Person " << index + 1 << ": "
372         << persons[index].personNum << " "
373         << persons[index].firstName << " "
374         << persons[index].lastName << ", "
375         << persons[index].streetAddress << ", "
376         << persons[index].city << ", "
377         << persons[index].zipCode << endl;
378
379 }
380
381

```

Figure 2
Output Code Secure Person Management System

```
Microsoft Visual Studio Debug + 

*****
* 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 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA, nan, nan, nan

Test 2: Integer Overflow when Creating Too Many Persons
Simulate numOfPersons = UINT_MAX, Maximum 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 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA, 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, 77722

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 vulgaries. It also displays person data.

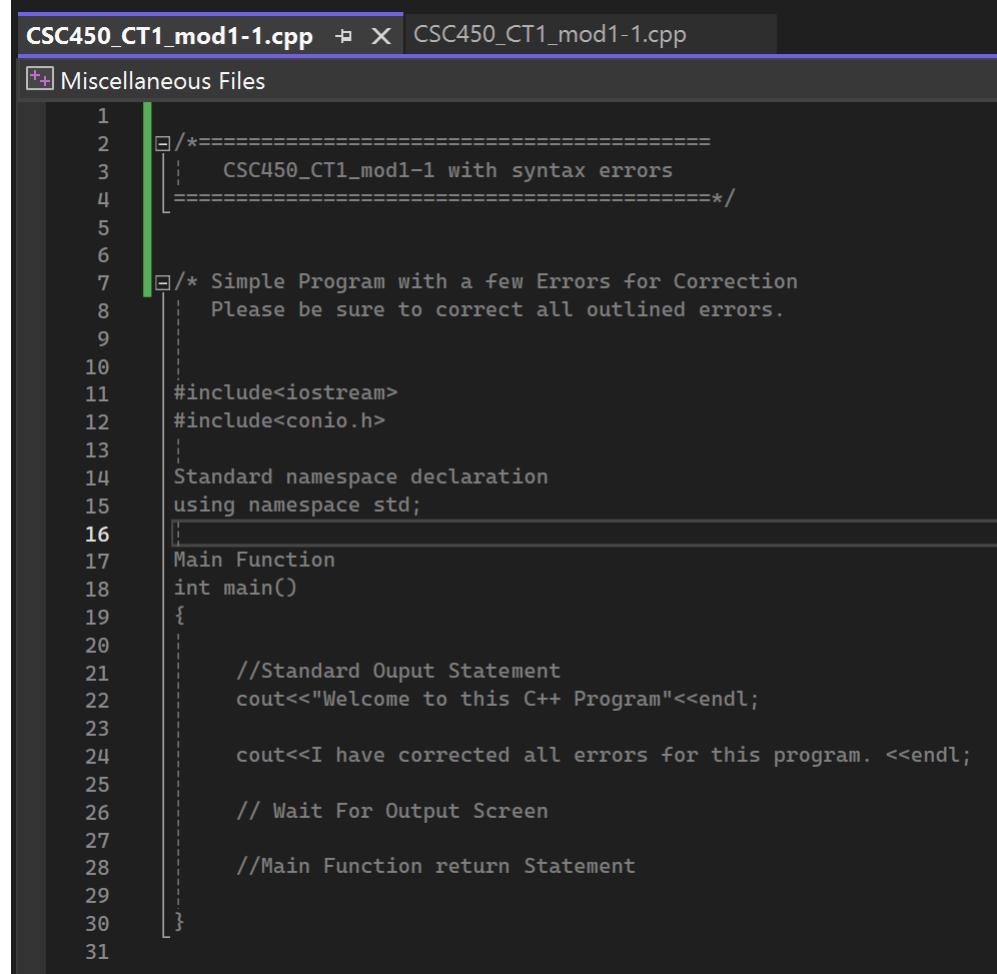
Continue next page

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



```

CSC450_CT1_mod1-1.cpp  X  CSC450_CT1_mod1-1.cpp

[+] Miscellaneous Files

1  /*
2  |=====
3  |   CSC450_CT1_mod1-1 with syntax errors
4  |=====
5
6
7  /* Simple Program with a few Errors for Correction
8  | Please be sure to correct all outlined errors.
9
10
11 #include<iostream>
12 #include<conio.h>
13
14 Standard namespace declaration
15 using namespace std;
16
17 Main Function
18 int main()
19 {
20
21     //Standard Ouput Statement
22     cout<<"Welcome to this C++ Program" << endl;
23
24     cout<<I have corrected all errors for this program. << endl;
25
26     // Wait For Output Screen
27
28     //Main Function return Statement
29
30 }
31

```

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*

```

CSC450_CT1_mod1-1.cpp  ✘ X
CSC450_CT1_mod1-1
=====
1  /*=====
2   |      CSC450_CT1_mod1-1 with Corrected syntax errors
3   |
4   |      Alejandro Ricciardi
5   |      10/13/2024
6   =====*/
7
8
9  /* Simple Program with a few Errors for Correction
10 | Please be sure to correct all outlined errors.
11 */
12
13 #include<iostream>
14
15 // Standard namespace declaration
16 using namespace std;
17
18 // Main Function
19 int main()
20 {
21     // Standard Output Statement
22     cout << "Welcome to this C++ Program" << endl;
23
24     // Corrected Output Statement
25     cout << "I have corrected all errors for this program." << endl;
26
27     // Main Function return Statement
28     return 0;
29 }
30

```

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

Figure 5*Modify CSC450_CT1_mod1-1 Outputs*

```

Microsoft Visual Studio Debug  +  -
Welcome to this C++ Program
I have corrected all errors for this program.

P:\CSU projects\Programming-3-CSC450\Programming-3-cpp\CSC450_CT1_mod1-1\x64\Debug\CSC450_CT1_mod1-1.exe (process 53120) 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 . . .

```

I modified:

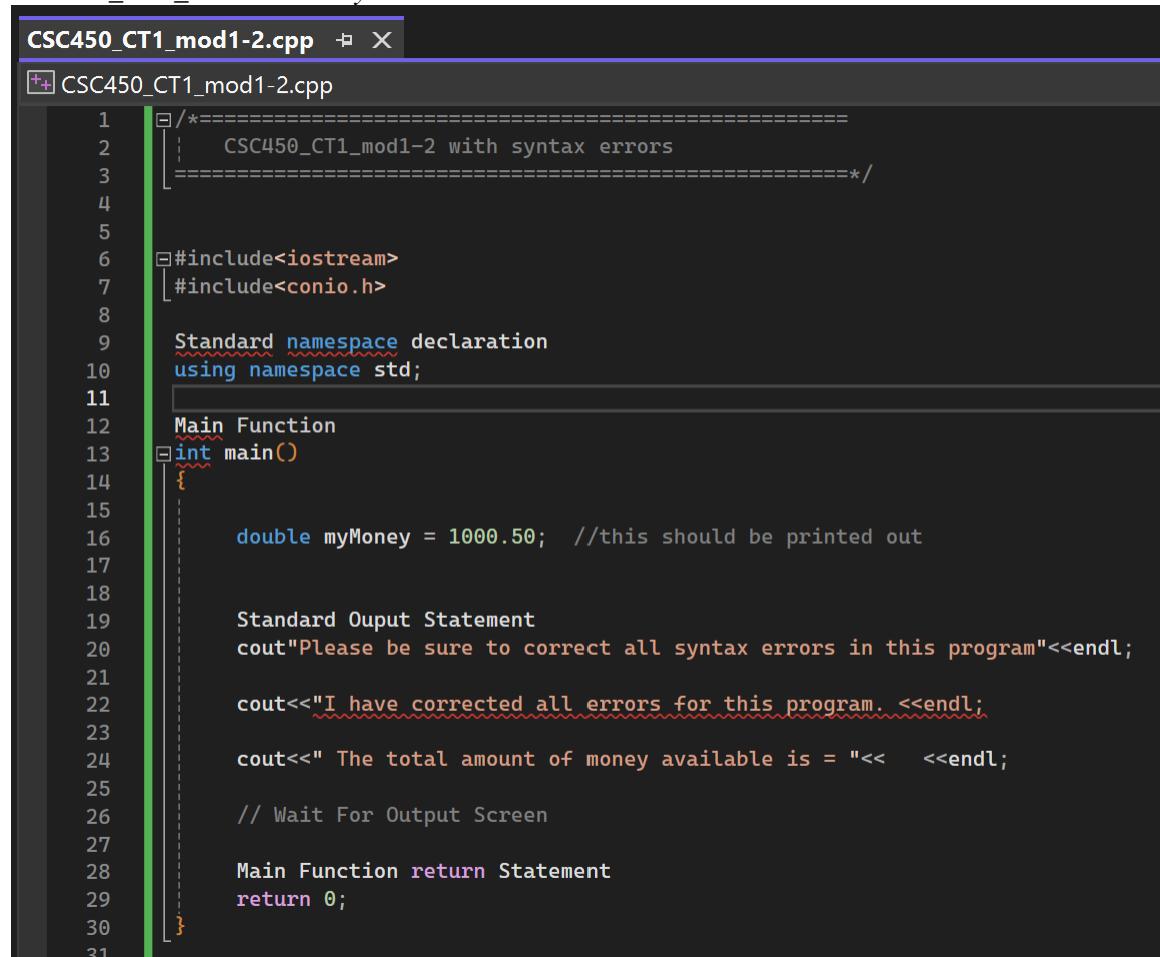
- Lines 1-6, added header comment.
- Line 14, removed the '#include<conio.h>', which is outdated and unnecessary.
- Line 15, added missing '//' in to comment line.
- Line 18, added space after '//' to comment line.
- Line 26, removed comment line '// Wait For Output Screen".
- Line 28. added 'return 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 ✎ X
+ 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 Output 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*  X  (Global Scope)
CSC450_CT1_mod1-2.cpp

1  /*=====
2   CSC450_CT1_mod1-2 with Corrected syntax errors
3
4   Alejandro Ricciardi
5   10/13/2024
6  =====*/
7
8  #include<iostream>
9  #include<iomanip> // setprecision
10
11 // Standard namespace declaration
12 using namespace std;
13
14 // Main Function
15 int main()
16 {
17
18     double myMoney = 1000.50; //this should be printed out
19
20     // Standard Output Statement
21     cout << "Please be sure to correct all syntax errors in this program" << endl;
22
23     cout << "I have corrected all errors for this program." << endl;
24
25     // Set precision to 2 decimal places
26     cout << fixed << setprecision(2);
27
28     cout << "The total amount of money available is = " << myMoney << endl;
29
30     // Main Function return Statement
31     return 0;
32 }
33

```

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

Figure 8*Modify CSC450_CT1_mod1-2 Outputs*

```

Microsoft Visual Studio Debug  X  +  V  -  □  ×
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\CSC450_CT1_mod1-2.cpp.exe (process 36448) 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 . . .

```

I modified:

- Lines 1-6, added header comment.
- Line 9, replaced 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.