Integer Pointers Program

Stephan Peters

Colorado State University Global

CSC450-1 23WD: Programming III

Reginald Haseltine

3 March 2024

Integer Pointer Application

For this assignment, the students were asked to create a console application in C/C++ that gets three integers from a user, creates pointers to the memory locations for those pointers, prints the stored integers and memory pointers to the console, then releases the memory the pointers point to for reallocation.

Vetting and Cleaning the Input

As I was thinking of how to force user input to a valid integer, I thought about making sure a human entered a valid integer, and considered what input from an improperly cleaned data source might look like. Examples for machine input for the integer 2112 might be <2112>, [(2112)], "2112,", 2112.0, 2112, etc. So, I wrote a Regular Expression (regex) pattern to mask the input. I get the input as a string using `getline(cin, string)`, then filter it through the following regex mask:

```
regex_replace(input, regex(R"([^\-0-9.]+)"), "")
```

This takes all non-numeric values except for hyphen – (for negative integers) and removes them from the string by replacing them with an empty string. It also will stop at a decimal point, so the value 2112.0 becomes 2112 and then omits the decimal point. It can make invalid data seem correct and then process it, for example 1,2,3, from a csv input (which should throw an exception in a production model) would be interpreted as 123 instead. But it is a good start for a model that will accept improperly sanitized input. It will also strip whitespace which can often occur when a human copies and pastes data from a table. Source Code 1 shows the `get_integer()` function, which can also be ported easily to another program, or even modified and added to a header file, something like int `xstoi(string input)` which could take an

invalid input string and return a valid integer, and would throw an exception if the input were unparsable as an int. The full source code for the entire program is in Appendix A

Source Code 1

Source code for the get_integer function.

```
int get_integer() {
    int integer;
    string input;
    bool valid;
    cout << "Enter an integer value: ";
    do {
        getline(cin, input);
        try {
            integer = stoi(regex_replace
                    (input, regex(R"([^\-0-9.]+)"), ""));
            valid = true;
        } catch (invalid_argument const &ia) {
            valid = false;
            cout << "Input cannot be parsed. "
                    "Enter a valid integer value: ";
            cin.clear();
        } catch (out_of_range const &oor){
            valid = false;
            cout << "Input is out of range for this machine. "
                    "Enter a valid integer value: ";
            cin.clear();
        }
    } while (!valid);
    return integer;
}
```

Creating the Pointers – First Attempt

My first attempt at creating and printing the pointers was a failure, I wasn't really thinking about what a pointer to a memory location for a variable actually is, so I used an array and a foreach loop so the code would be cleaner. But by doing this, I was creating a new variable in the loop which was purged and re-created at each iteration, so the returned pointers did not point to the location of the actual integers, but to the variable in the foreach loop instead. When the program printed the pointers to the console, naturally all the pointer values were identical (as

would be expected). Source code 2 shows the function I wrote that gives pointer information for the variable within the foreach loop rather than pointers to the variables themselves:

Source Code 2

Source code for my failed initial attempt to show the pointer information.

```
void integer_pointers_loop() {
    int counter = 0,
        integer_1 = get_integer(),
        integer_2 = get_integer(),
        integer_3 = get_integer();
    const int INTEGER_ARRAY[3] = {integer_1, integer_2, integer_3};
    for (int integer: INTEGER_ARRAY) {
        counter++;
        printf("Integer %d: %-11d, Pointer %d: %p\n",
               counter, integer, counter, &integer);
    }
}
```

Creating the Pointers – Second Attempt

My second attempt at creating and printing the pointers was successful, but it defeats the purpose of the exercise --to show familiarity with pointer (`int *`, `void *`) data types and removal of the pointers to free memory. Since this function created no pointer variables, and as such they could not be freed, cleared, or deleted, I chose to abandon it and re-write the function.

Source Code 3

Source code for my second attempt function (renamed to alternate_solution()).

```
void alternate_solution() {
    int integer_1 = get_integer(),
        integer_2 = get_integer(),
        integer_3 = get_integer();
    printf("Integer 1: %-11d Integer 1 Pointer: %p\n"
           "Integer 2: %-11d Integer 2 Pointer: %p\n"
           "Integer 3: %-11d Integer 3 Pointer: %p\n",
           integer_1, &integer_1,
           integer_2, &integer_2,
           integer_3, &integer_3);
}
```

Creating the Pointers – Final Function

The final version of the function creates three separate integer variables and creates a pointer for each variable. I chose to use `malloc()` to create the pointers, so I could use `free()` to release them. There are three common ways to create pointers and free the memory associated with them:

1. ```
   int *pointer = new int(value);
   delete pointer;
   ```
2. ```
   int j = value;
   void *pointer = malloc(sizeof(j));
   free(pointer);
   ```
3. ```
   int j = value;
   int *pointer = &j;
   pointer = NULL;
   ```

(LePage 2017).

This will not remove the values inside the volatile memory, they will merely release the memory for re-use. If the values are truly sensitive, more destructive methods such as `memset(pointer, 0, sizeof(*pointer))` or `bzero(pointer)`should be used. I merely used `free()`for this exercise. Using delete is undesirable, as this leads to undefined behavior (Stieber, et. al. 2017). The preferred method is to have the values as local variables in a function or method and allow them to be freed at termination of the function or method (Reid, 2012). I chose to do this as well. Source code 4 shows the function as submitted for compilation.
   ```

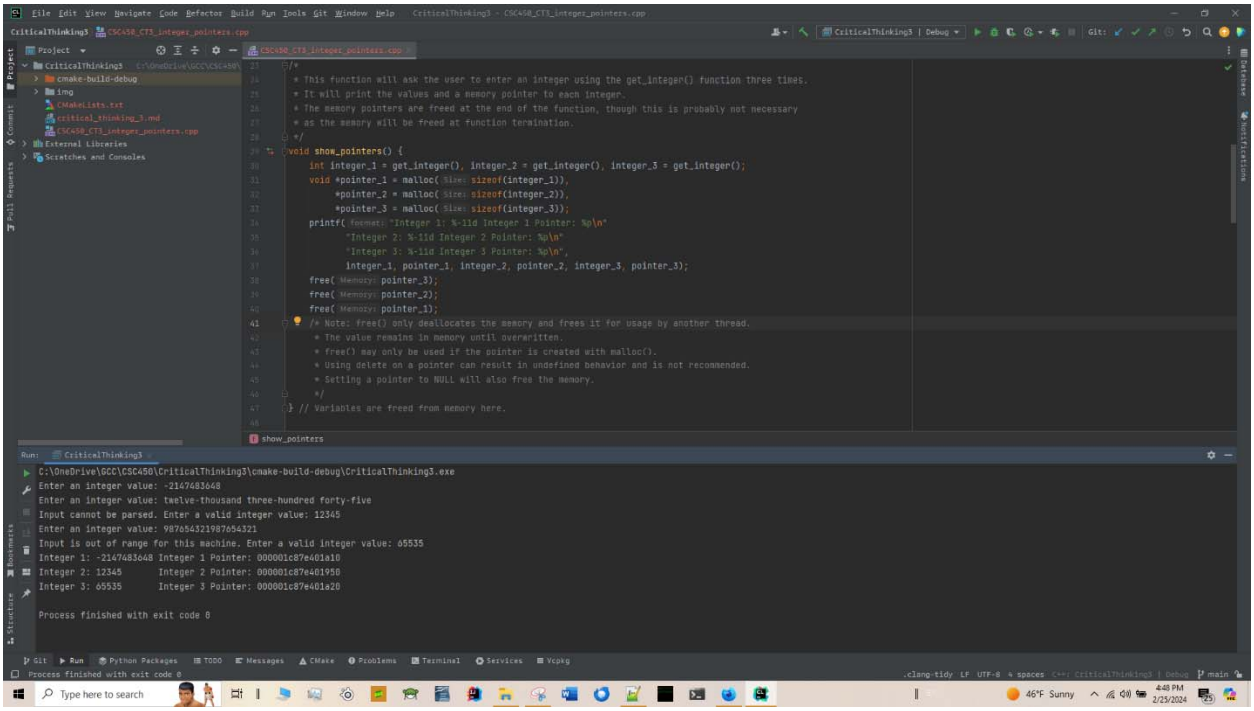
Source Code 4

Final show_pointers() function as submitted for compilation.

```
void show_pointers() {
    int integer_1 = get_integer(),
        integer_2 = get_integer(),
        integer_3 = get_integer();
    void *pointer_1 = malloc(sizeof(integer_1)),
         *pointer_2 = malloc(sizeof(integer_2)),
         *pointer_3 = malloc(sizeof(integer_3));
    printf("Integer 1: %-11d Integer 1 Pointer: %p\n"
           "Integer 2: %-11d Integer 2 Pointer: %p\n"
           "Integer 3: %-11d Integer 3 Pointer: %p\n",
           integer_1, pointer_1,
           integer_2, pointer_2,
           integer_3, pointer_3);
    free(pointer_3);
    free(pointer_2);
    free(pointer_1);
}
```

Figure 1

Running the CSC450_CT3_integer_pointers.cpp program



Vulnerability Assessment

The `free()` command does not wipe the value from volatile memory. If the information is sensitive, `memset(pointer, 0, sizeof(*pointer))` or `bzero(pointer)`might be used to completely wipe the values from memory. These commands will require more processing time to use, negligible in this program but if this is standard practice for non-sensitive information it will unnecessarily task resources in larger enterprise-type applications. This console app initially uses a string for input, so in an unusual case a malicious user *could* enter a string so large it would consume the entire memory allocated to the execution of the program, in essence forcing a Denial of Service (DoS) for concurrent users of the software or machine, an assessment of this would need to take place before deployment. This program also uses printf, which if incorrectly coded may introduce the Format String Vulnerability (CWE 134) (Du, 2014). The printf function as written in this code is compliant with current security standards.

GitHub Repository

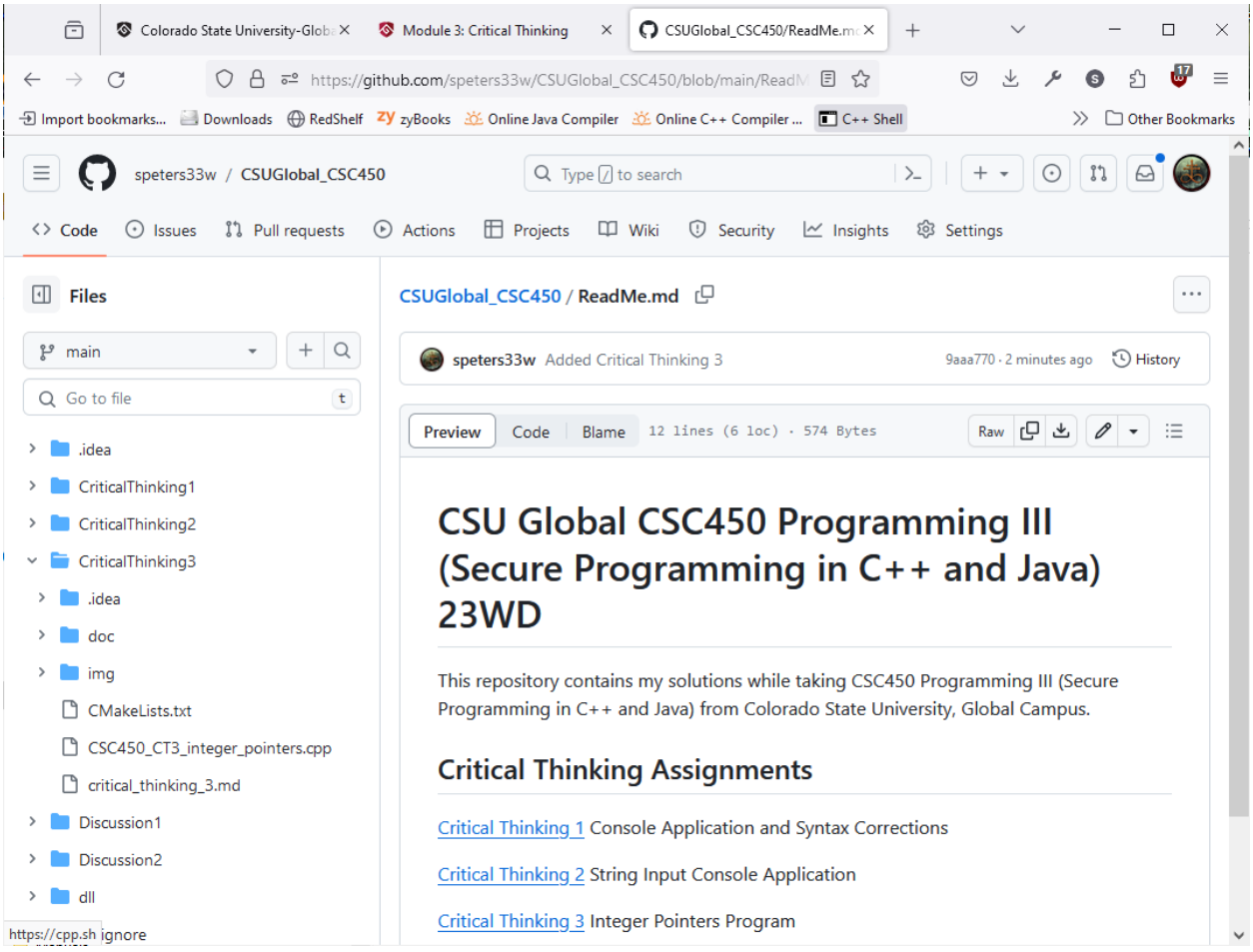
In addition to the solutions above, I was tasked to create a GitHub repository for the project. This repository is located at

https://github.com/speters33w/CSUGlobal_CSC450/tree/main/CriticalThinking3

Figure 2 shows a screenshot of the main page of this repository.

Figure 2

Image of the main page of my CSC 450 GitHub repository



Appendix A – Full Source Code as Submitted

Source Code 5

Final source code as submitted for compilation.

```
/*
 * A simple C/C++ program that takes three integers from a user,
 * creates memory pointers for those integers,
 * then prints the integers and pointer references to the screen.
 * (CSC450_CT3_integer_pointers.cpp)
 */

#include <iostream>
#include <regex>
#include <string>

void show_pointers();

int get_integer();

using namespace std;

int main() {
    show_pointers();
    return 0;
}

/*
 * This function will ask the user to enter an integer
 * using the get_integer() function three times.
 * It will print the values and a memory pointer to each integer.
 * The memory pointers are freed at the end of the function,
 * though this is probably not necessary
 * as the memory will be freed at function termination.
 */
void show_pointers() {
    int integer_1 = get_integer(),
        integer_2 = get_integer(),
        integer_3 = get_integer();
    void *pointer_1 = malloc(sizeof(integer_1)),
         *pointer_2 = malloc(sizeof(integer_2)),
         *pointer_3 = malloc(sizeof(integer_3));
    printf("Integer 1: %-11d Integer 1 Pointer: %p\n"
           "Integer 2: %-11d Integer 2 Pointer: %p\n"
           "Integer 3: %-11d Integer 3 Pointer: %p\n",
           integer_1, pointer_1,
           integer_2, pointer_2,
           integer_3, pointer_3);
    free(pointer_3);
    free(pointer_2);
    free(pointer_1);
```

```
    /* Note: free() only deallocates the memory
     * and frees it for usage by another thread.
     * The value remains in memory until overwritten.
     * free() may only be used if the pointer is created with malloc().
     * Using delete on a pointer can result in undefined behavior
     * and is not recommended.
     * Setting a pointer to NULL will also free the memory.
     */
} // Variables are freed from memory here.

/*
 * This function autocorrects typos or improperly sanitized input
 * from a data source
 * and ensures input may be interpreted as an integer.
 * <2112>, [(2112)], "2112,", 2112.0, <span style = "numeral">2112</span>
 * will all be interpreted as 2112.
 * 12ThreeFour56 will be interpreted as 1256
 * 67.89 will be interpreted as 67
 * 98-76 will be interpreted as 98
 * It will ask the user to re-enter data if there are no digit values.
 * It will ask the user to re-enter if the integer is out of range.
 * Valid integer data, e.g. -1234 or 9876 is returned as entered.
 */
int get_integer() {
    int integer;
    string input;
    bool valid;
    cout << "Enter an integer value: ";
    do {
        getline(cin, input);
        try {integer =
             stoi(regex_replace(input, regex(R"([^\-0-9.]+)"), ""));
            valid = true;
        } catch (invalid_argument const &ia) {
            valid = false;
            cout << "Input cannot be parsed. "
                    "Enter a valid integer value: ";
            cin.clear();
        } catch (out_of_range const &oor){
            valid = false;
            cout << "Input is out of range for this machine. "
                    "Enter a valid integer value: ";
            cin.clear();
        }
    } while (!valid);
    return integer;
} // Variables other than the return value are freed from memory here.
```

```
/*
 * This function will ask the user to enter an integer
 * using the get_integer() function three times.
 * It will print the values and a memory pointer to each integer.
 * The pointers are not stored in a variable
 * and are freed at the function termination.
 * If they were stored and returned, delete or free() could be used
 * to deallocate the storage in memory
 * depending on the way the pointer was created.
 */
[[maybe_unused]] void alternate_solution() {
    int integer_1 = get_integer(),
        integer_2 = get_integer(),
        integer_3 = get_integer();
    printf("Integer 1: %-11d Integer 1 Pointer: %p\n"
           "Integer 2: %-11d Integer 2 Pointer: %p\n"
           "Integer 3: %-11d Integer 3 Pointer: %p\n",
           integer_1, &integer_1,
           integer_2, &integer_2,
           integer_3, &integer_3);
} // Variables are freed from memory here.

/*
 * The loop in this function defeats the purpose of the assignment,
 * as the pointer will always be to the local integer variable
 * within the for each loop.
 * It will print the same pointer address to the screen three times
 * because at each iteration
 * the local integer variable is destroyed and overwritten.
 * It does not provide the pointer address for
 * integer_1, integer_2, or integer_3.
 * It is unused and left here for demonstration.
 */
[[maybe_unused]] void integer_pointers_loop() {
    int counter = 0, integer_1 = get_integer(), integer_2 = get_integer(),
integer_3 = get_integer();
    const int INTEGER_ARRAY[3] = {integer_1, integer_2, integer_3};
    for (int integer: INTEGER_ARRAY) {
        counter++;
        printf("Integer %d: %-11d, Pointer %d: %p\n", counter, integer,
counter, &integer);
    }
} // Variables are freed from memory here.
```

References

Du, W. (Kevin). (2014). Format String Vulnerability printf (user input). In *Department of Electrical Engineering and Computer Science, Syracuse University*. https://web.ecs.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Format_String.pdf

LePage, G. (2017, November 3). *How do I clear an int pointer (int *pointer) from memory on C++?* Quora. https://www.quora.com/How-do-I-clear-an-int-pointer-int-*pointer-from-memory-on-C++

Reid (ModShop), B. (2012, October 18). *How to delete a variable. - C++ Forum*. Cplusplus.com. https://cplusplus.com/forum/beginner/82290/

Stieber et. al., C. (2017, May 23). *What does delete command really do for memory, for pointers in C++?* Stack Overflow. https://stackoverflow.com/questions/11603005/what-does-delete-command-really-do-for-memory-for-pointers-in-c