Mahdeen Ahmed Khan Sameer

Professor Max Bender

CS333

22 February 2022

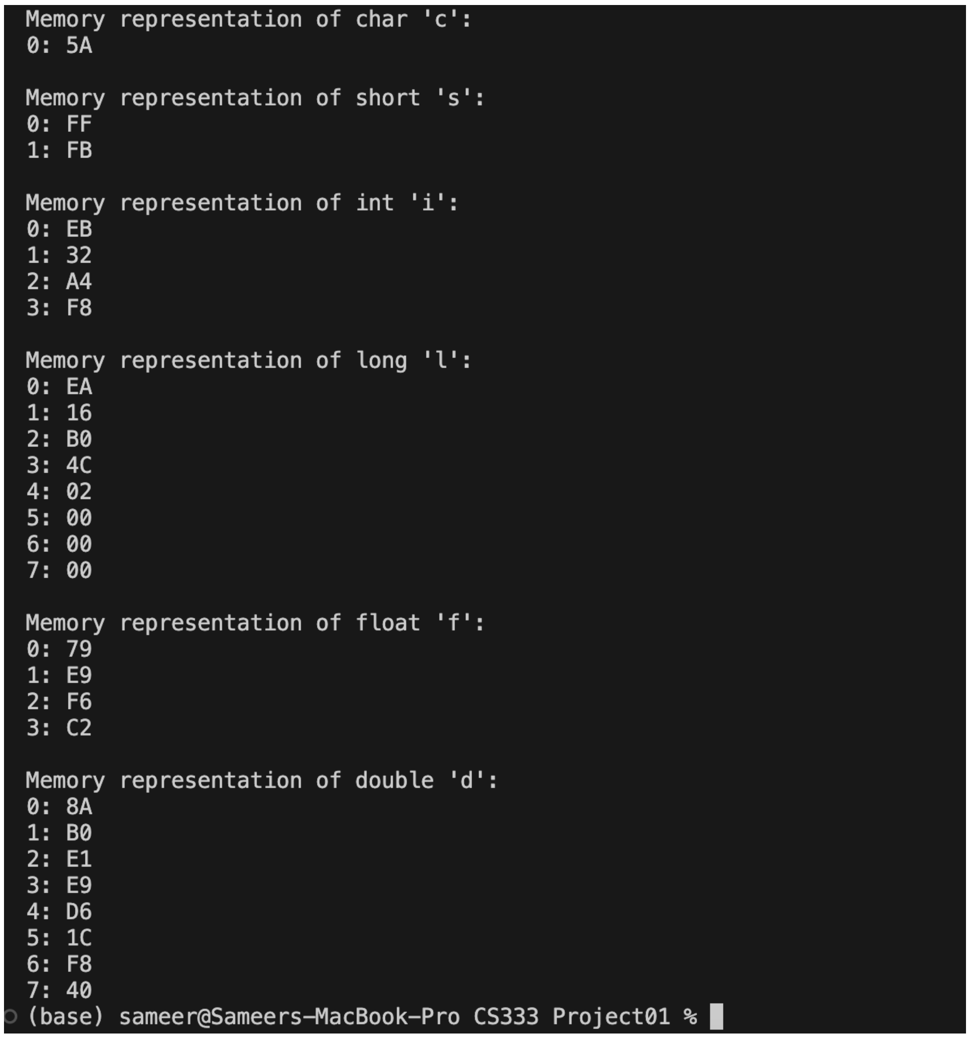
Project 01: Learning C Programming with Javascript, LISP, MySQL!

Google site: https://sites.google.com/colby.edu/sameer-again/home

PART 01:

Task 1

1. Include a screenshot of your output.



```
Memory representation of char 'c':
0: 5A

Memory representation of short 's':
0: FF
1: FB

Memory representation of int 'i':
0: EB
1: 32
2: A4
3: F8

Memory representation of long 'l':
0: EA
1: 16
2: B0
3: 4C
4: 02
5: 00
6: 00
7: 00

Memory representation of float 'f':
0: 79
1: E9
2: F6
3: C2

Memory representation of double 'd':
0: 8A
1: B0
2: E1
3: E9
4: D6
5: 1C
6: F8
7: 40
(base) sameer@Sameers-MacBook-Pro CS333 Project01 %
```

2. Is the machine you are using a big-endian or little-endian machine?

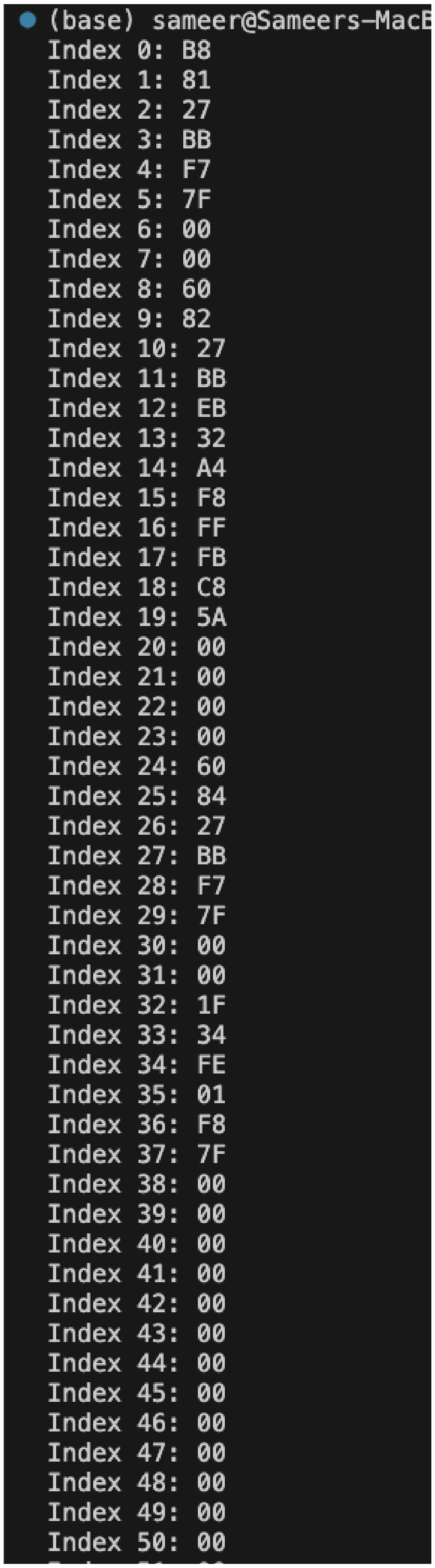
The machine used for this program is a little-endian machine. This is shown by how the bytes of multi-byte data types are stored and displayed in the output. In little-endian systems, the least significant byte (LSB) is stored at the smallest address, and the most significant byte (MSB) is stored at the largest address. This can be observed in the memory representations of the multi-byte variables (short, int, long, float, and double) in the output.

3. How does the program output tell you?

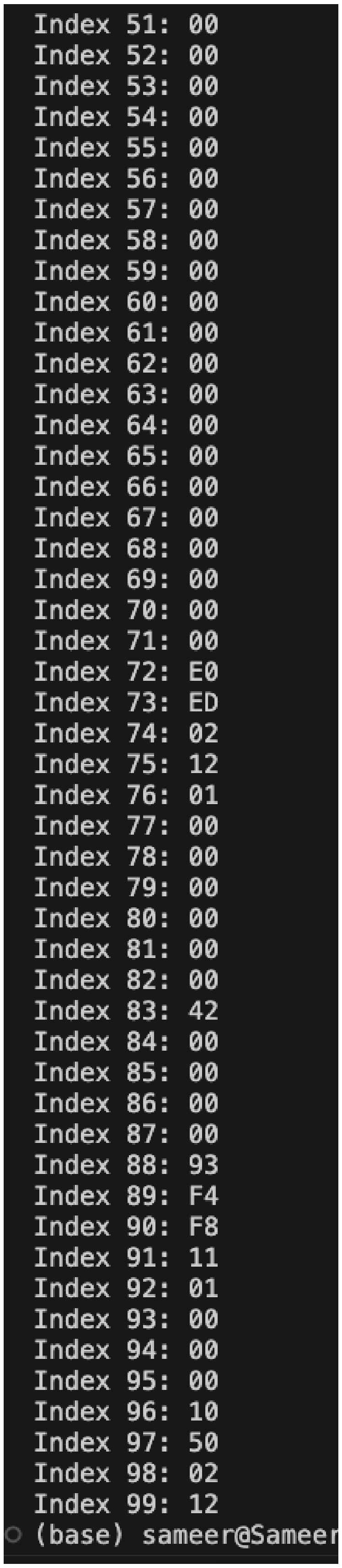
We know that the least significant bytes data types are stored first in memory. This matches the definition of a little-endian architecture, where the least significant bytes come first. So, if it were big-endian, we would see the opposite.

Task 2

1. Include a screenshot of your output.



```
● (base) sameer@Sameers-MacB
  Index 0: B8
  Index 1: 81
  Index 2: 27
  Index 3: BB
  Index 4: F7
  Index 5: 7F
  Index 6: 00
  Index 7: 00
  Index 8: 60
  Index 9: 82
  Index 10: 27
  Index 11: BB
  Index 12: EB
  Index 13: 32
  Index 14: A4
  Index 15: F8
  Index 16: FF
  Index 17: FB
  Index 18: C8
  Index 19: 5A
  Index 20: 00
  Index 21: 00
  Index 22: 00
  Index 23: 00
  Index 24: 60
  Index 25: 84
  Index 26: 27
  Index 27: BB
  Index 28: F7
  Index 29: 7F
  Index 30: 00
  Index 31: 00
  Index 32: 1F
  Index 33: 34
  Index 34: FE
  Index 35: 01
  Index 36: F8
  Index 37: 7F
  Index 38: 00
  Index 39: 00
  Index 40: 00
  Index 41: 00
  Index 42: 00
  Index 43: 00
  Index 44: 00
  Index 45: 00
  Index 46: 00
  Index 47: 00
  Index 48: 00
  Index 49: 00
  Index 50: 00
```



```
  Index 51: 00
  Index 52: 00
  Index 53: 00
  Index 54: 00
  Index 55: 00
  Index 56: 00
  Index 57: 00
  Index 58: 00
  Index 59: 00
  Index 60: 00
  Index 61: 00
  Index 62: 00
  Index 63: 00
  Index 64: 00
  Index 65: 00
  Index 66: 00
  Index 67: 00
  Index 68: 00
  Index 69: 00
  Index 70: 00
  Index 71: 00
  Index 72: E0
  Index 73: ED
  Index 74: 02
  Index 75: 12
  Index 76: 01
  Index 77: 00
  Index 78: 00
  Index 79: 00
  Index 80: 00
  Index 81: 00
  Index 82: 00
  Index 83: 42
  Index 84: 00
  Index 85: 00
  Index 86: 00
  Index 87: 00
  Index 88: 93
  Index 89: F4
  Index 90: F8
  Index 91: 11
  Index 92: 01
  Index 93: 00
  Index 94: 00
  Index 95: 00
  Index 96: 10
  Index 97: 50
  Index 98: 02
  Index 99: 12
○ (base) sameer@Sameer
```

2. What seems to be the overall layout of the stack?

The stack grows downwards (from high memory addresses to low memory addresses). This means that variables declared later in a function are found at lower memory addresses than those declared before them.

3. Are there any non-zero values you can't immediately make sense of?

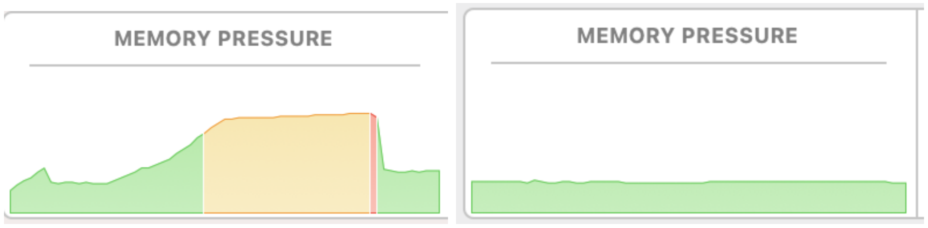
Among the non-zero values presented in the stack memory output, there are several that don't immediately correlate with the variables defined in the C program, making their purpose or origin unclear. These include the initial bytes at indices 0 through 11 and various other bytes beyond the directly identified variables' memory locations. These non-zero values could represent stack frame information, such as return addresses, saved registers, or possibly remnants of previous stack activities, including function calls or other local variables not explicitly.

4. Can you find the variables defined in your C program? Highlight the ones you find and explain how you know you have found them.

In the stack memory output, we can confidently identify the variables defined in the C program based on their initialized values and memory representation, considering the machine's little-endian architecture. The char variable c with the value Z is unmistakably identified at index 19 by its hexadecimal representation 5A, which matches the ASCII code for 'Z'. The short variable s with the value -1025 is found at indices 16 and 17, displayed as FFFB in little-endian order, perfectly aligning with the two's complement representation of -1025 for a short integer. The int variable i with a value of -123456789 is pinpointed at indices 12 through 15, presented as EB32A4F8, which when read in reverse due to the little-endian storage, represents the hexadecimal conversion of -123456789.

Task 3

1. Include a screenshot of your memory usage (e.g. Activity Monitor or Top).

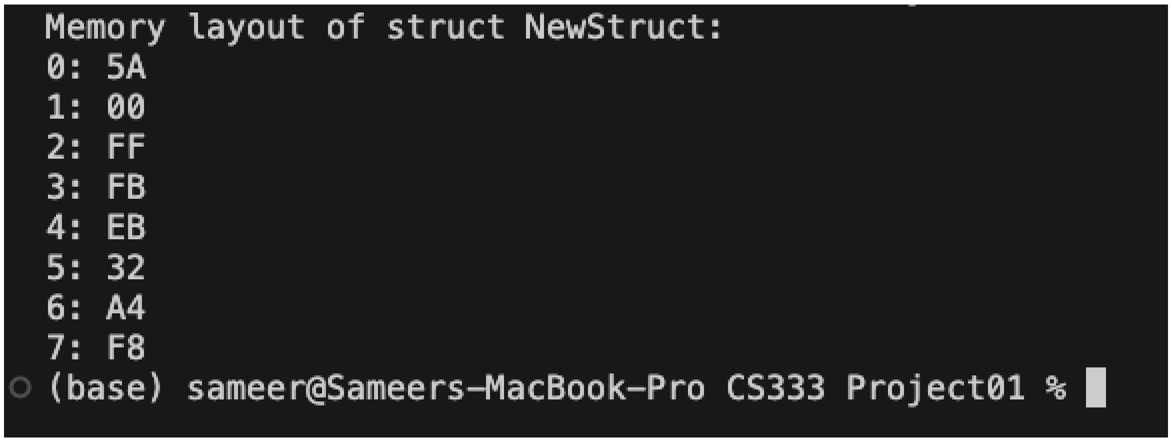


The observed memory consumption increases progressively as memory continues to be allocated without subsequent deallocation. Introducing a `free` statement stabilizes the memory usage, maintaining a constant level because the allocated memory is promptly released after each cycle of the loop.

2. Briefly describe the memory requirements when using and not using the free statement.

Neglecting to release allocated memory results in continuous growth in memory consumption, which can result in memory leaks. Employing the `free` function to liberate the memory once it is no longer needed helps to ensure that the memory is available for reallocation, thereby maintaining minimal memory usage.

Task 4



```
Memory layout of struct NewStruct:
0: 5A
1: 00
2: FF
3: FB
4: EB
5: 32
6: A4
7: F8
(base) sameer@Sameers-MacBook-Pro CS333 Project01 %
```

1. Does the sizeof result match your expectation?

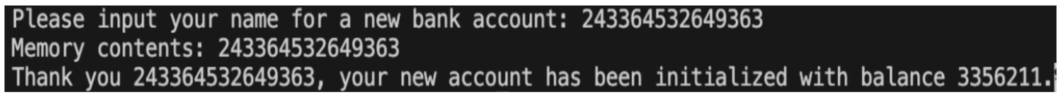
In our examination of the struct NewStruct memory layout, we find that the sizeof` result, being 8 bytes, aligns with our expectations when considering struct padding for alignment in C. Initially, one might expect the struct to occupy 7 bytes, given the sum of its individual member sizes (1 byte for the char, 2 bytes for the short, and 4 bytes for the int). However, the inclusion of a padding byte after the char (to align the short on a 2-byte boundary for efficiency) explains the total size. This padding, observed at index 1 (1: 00), ensures that the short starts at an address that is a multiple of its size, following the char at index 0 (0: 5A). The short and int values then follow in little-endian order, representing their respective initialized values. This automatic padding by the compiler, designed to satisfy the architecture's alignment requirements, results in an 8-byte total size for the struct, illustrating the balance between memory usage and access efficiency.

2. Are there any gaps in the way the fields of the structure are laid out?

Yes, there is a gap (padding) in the structure's layout between the char and short fields to align the short on a 2-byte boundary, ensuring efficient memory access according to the architecture's alignment requirements.

Task 5

1. First, find a string that doesn't work. Run the program on it and show that it starts with a positive balance on your given input.



```
Please input your name for a new bank account: 243364532649363
Memory contents: 243364532649363
Thank you 243364532649363, your new account has been initialized with balance 3356211.
```

2. Use an unsigned char * to identify what the memory contents look like on your bad string: explain in detail what has gone wrong and why.

The first ten characters of the extended input were copied into the name array, but the copying continued beyond the set ten bytes, spilling over into the subsequent four bytes meant for the balance integer. As a result, the balance was overwritten with a significantly large positive number. The issue stems from the static size of the name array, which, when exceeded, leads to the corruption of adjacent memory variables such as the balance. To address this, name should be dynamically allocated using malloc and free to handle strings of any length, or the string length should be checked before copying to avoid overflow.

PART 02:

JavaScript was designed as a scripting language to make web pages interactive. Its main uses involve web development, where it enables dynamic behavior on web pages, such as responding to user actions, validating forms, and creating animations. JavaScript was first introduced on December 4, 1995, by Brendan Eich of Netscape. Although traditionally an interpreted language, modern JavaScript engines use just-in-time compilation for improved performance. This means that while the source code is compiled during execution rather than ahead of time, JavaScript retains its identity as an interpreted language. JavaScript is open source, and it is widely used in various environments beyond browsers, such as server-side development with Node.js.

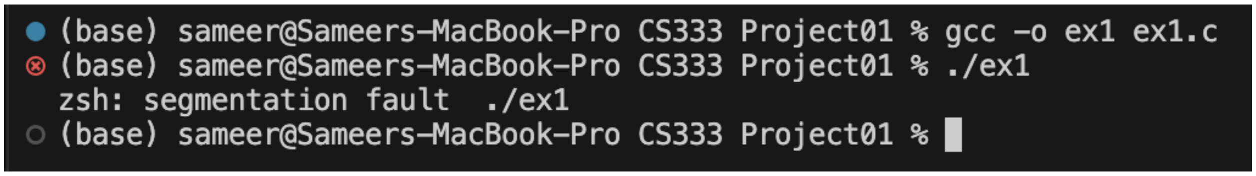
LISP, or List Processing language, was developed in the late 1950s as one of the earliest programming languages and is primarily used for artificial intelligence (AI) research. It is designed for symbolic expression processing, making it suitable for manipulation of data structures, such as lists, which are central to the language. LISP is known for its use in problem-solving, particularly in AI, due to its support for recursive algorithms and symbolic computation. It is an interpreted language, allowing for dynamic writing and testing of code, and it has several dialects, some of which are open source.

MySQL is a relational database management system that was developed to handle large databases efficiently. It is used for a variety of applications, particularly web database applications and online transaction processing. MySQL was first released in 1995 and is known for its reliability and ease of use. It is a key component of the LAMP open-source web application software stack, which is used for building dynamic websites and applications.

MySQL uses a client-server model, and while the core of MySQL is open source, it is owned by Oracle Corporation, which offers paid editions with additional features and support.

Extensions:

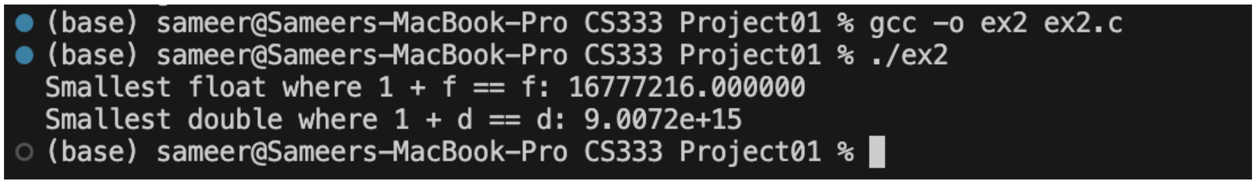
Extension 01:



```
● (base) sameer@Sameers-MacBook-Pro CS333 Project01 % gcc -o ex1 ex1.c
⊗ (base) sameer@Sameers-MacBook-Pro CS333 Project01 % ./ex1
  zsh: segmentation fault  ./ex1
○ (base) sameer@Sameers-MacBook-Pro CS333 Project01 % 
```

When we compile and execute the given C program, our system might encounter a bus error while writing to a misaligned char pointer, as the specified address (0x1) does not meet the memory alignment requirements for larger data types. On the other hand, a segmentation fault may occur while writing to an int pointer targeting an invalid or inaccessible memory address (0xFFFFFFFF). The distinction lies in the fact that bus errors result from violations of the hardware's alignment constraints, whereas segmentation faults arise from unauthorized memory access, which is restricted by the operating system's protective measures. It's important to note that such a program is prone to crash and deliberately triggering these errors is generally discouraged in software development. The actual outcome—whether a bus error, a segmentation fault, or neither—varies depending on the operating system and hardware, as they have different methods of managing such erroneous operations.

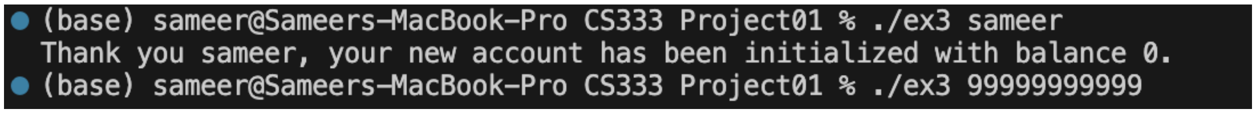
Extension 02:



```
● (base) sameer@Sameers-MacBook-Pro CS333 Project01 % gcc -o ex2 ex2.c
● (base) sameer@Sameers-MacBook-Pro CS333 Project01 % ./ex2
  Smallest float where 1 + f == f: 16777216.000000
  Smallest double where 1 + d == d: 9.0072e+15
○ (base) sameer@Sameers-MacBook-Pro CS333 Project01 % 
```

In our exploration of floating-point numbers in C, we discovered a phenomenon regarding the precision limits of `float` and `double` data types. By incrementally doubling a starting value of 1.0, we searched for the smallest number where adding one no longer alters its value, highlighting the limits of floating-point precision. For the `float` type, this threshold was reached at 16777216.000000, and for `double`, the limit was found to be approximately 9.0072e+15. This experiment underscores the finite precision with which these data types can represent numbers, a consequence of their underlying binary representation and storage in computer memory. It illustrates a practical example of numerical analysis principles, specifically the concept of machine epsilon, which is the smallest difference between two representable numbers, beyond which additional precision is lost.

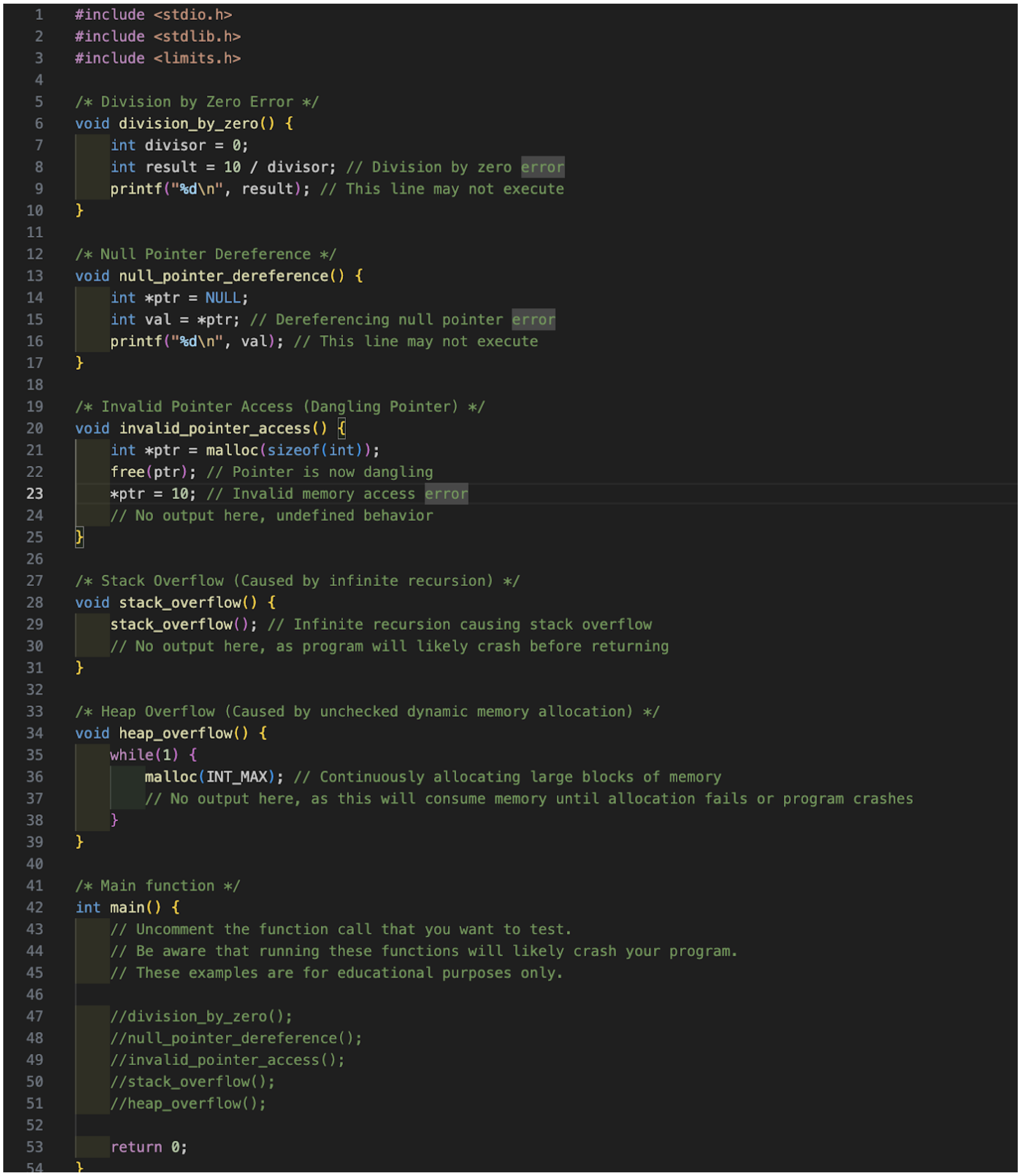
Extension 03:



```
● (base) sameer@Sameers-MacBook-Pro CS333 Project01 % ./ex3 sameer
  Thank you sameer, your new account has been initialized with balance 0.
● (base) sameer@Sameers-MacBook-Pro CS333 Project01 % ./ex3 99999999999
```

In our revised version of the program, we have opted to take the account name directly from the command line arguments, specifically argv[1], to populate the name field of the Account struct. We use strncpy to safely copy the provided name into the struct, ensuring it fits within the designated space and avoiding buffer overflow by explicitly null-terminating the string. This method enhances the program's flexibility and user-friendliness, as it eliminates the need for interactive input and potential errors associated with scanf, streamlining the process for the user to establish a new bank account directly from the command line when running the program.

Extension 04:



```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

/* Division by Zero Error */
void division_by_zero() {
    int divisor = 0;
    int result = 10 / divisor; // Division by zero error
    printf("%d\n", result); // This line may not execute
}

/* Null Pointer Dereference */
void null_pointer_dereference() {
    int *ptr = NULL;
    int val = *ptr; // Dereferencing null pointer error
    printf("%d\n", val); // This line may not execute
}

/* Invalid Pointer Access (Dangling Pointer) */
void invalid_pointer_access() {
    int *ptr = malloc(sizeof(int));
    free(ptr); // Pointer is now dangling
    *ptr = 10; // Invalid memory access error
    // No output here, undefined behavior
}

/* Stack Overflow (Caused by infinite recursion) */
void stack_overflow() {
    stack_overflow(); // Infinite recursion causing stack overflow
    // No output here, as program will likely crash before returning
}

/* Heap Overflow (Caused by unchecked dynamic memory allocation) */
void heap_overflow() {
    while(1) {
        malloc(INT_MAX); // Continuously allocating large blocks of memory
        // No output here, as this will consume memory until allocation fails or program crashes
    }
}

/* Main function */
int main() {
    // Uncomment the function call that you want to test.
    // Be aware that running these functions will likely crash your program.
    // These examples are for educational purposes only.

    //division_by_zero();
    //null_pointer_dereference();
    //invalid_pointer_access();
    //stack_overflow();
    //heap_overflow();

    return 0;
}
```

What we have to do is comment out when we are trying to compile. In C programming, we can generate a variety of run-time errors that each teach us about the importance of proper memory and error management. A division by zero error occurs when we attempt to divide an integer by zero, which is undefined behavior and can cause a program to crash. Null pointer dereference errors are the result of attempting to read or write to a memory location pointed to by a null pointer, indicative of uninitialized or improperly managed memory. Dangling pointer errors, or invalid pointer accesses, happen when we try to use a pointer that has been freed, leading to undefined behavior or crashes due to corruption of memory. Stack overflow errors are typically caused by uncontrolled recursion, where function calls accumulate on the stack without end, eventually exceeding the stack's limit. Heap overflow errors occur when we repeatedly allocate large blocks of memory without freeing them, which can exhaust available memory resources. These run-time errors underscore the critical need for careful checking and handling of inputs, diligent memory allocation and deallocation, and robust error checking in C programs to prevent crashes and undefined behavior.

Extension 05:

(Also found in google site)

R is a programming language and environment specifically designed for statistical computing and graphics. It was created by Ross Ihaka and Robert Gentleman in August 1993 at the University of Auckland and was conceived as an implementation similar to the S language developed at Bell Laboratories. R's primary purpose is to provide an open-source route to statistical methodology and data analysis, with the ability to produce publication-quality graphics. The main uses of R span across various statistical domains, including linear and nonlinear modeling, time-series analysis, classification, clustering, and more. It is highly extensible, allowing users to add new functions and features. R is particularly favored for producing well-designed, publication-quality plots with great control over design choices. As an interpreted language, R facilitates immediate execution of code without the need for compilation, which makes it suitable for interactive work in statistics. It also supports procedural, object-oriented, functional, reflective, and imperative programming paradigms. R is open-source software, licensed under the GNU General Public License, which means it's freely available for anyone to use, modify, and distribute. It runs on various UNIX platforms, Windows, and macOS, ensuring wide accessibility and utility.

Acknowledgements:

None