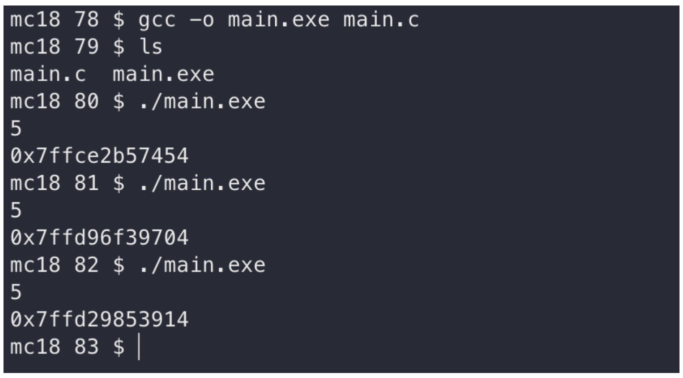
CS 240 Summer 2024
Lab 2: Pointers, passing by value vs. reference, 1-D arrays, and run-time errors

Dong Wang
Student ID: 00370-48599

Due: June 26, 2024

Problem 1

The observed output from running `main.exe` three times is as follows:



```
mc18 78 $ gcc -o main.exe main.c
mc18 79 $ ls
main.c  main.exe
mc18 80 $ ./main.exe
5
0x7ffce2b57454
mc18 81 $ ./main.exe
5
0x7ffd96f39704
mc18 82 $ ./main.exe
5
0x7ffd29853914
mc18 83 $ |
```

Figure 1: problem1

Hexadecimal and Binary Representation

Address 1: 0x7ffce2b57454:

`0111 1111 1111 1100 1110 0010 1011 0101 0111 0100 0101 0100`

Address 2: 0x7ffd96f39704:

`0111 1111 1111 1101 1001 0110 1111 0011 1001 0111 0000 0100`

Address 3: 0x7ffd29853914:

`0111 1111 1111 1101 0010 1001 1000 0101 0011 1001 0001 0100`

Explanation of Output

The program outputs the value of the integer variable `s`, which is set to 5, and the address of `s` in memory. The address of `s` is printed using the `%p` format specifier.

Address Variation The address of `s` is different each time the program is run. This is due to Address Space Layout Randomization (ASLR), a security technique used by modern operating systems. ASLR randomizes the memory addresses used by a program each time it is executed.

Address Width Each address is represented by 48 bits in the binary format. This is because each hexadecimal digit corresponds to 4 binary digits, and the addresses have 12 hexadecimal digits.

```
12 (hex digits) * 4 (bits per hex digit) = 48 bits
```

64-bit CPU Compatibility: The total of 48 bits matches up with our lab machines having 64-bit CPUs. While a 64-bit CPU can address up to 2^{64} different locations, the actual used bits in the addresses can be fewer depending on the specific implementation and the range of addresses used by the operating system. In this case, it appears that only 48 bits are needed to represent the addresses currently in use.

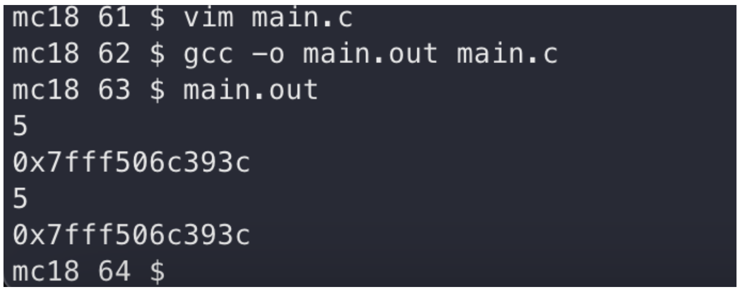
Integer Values of the Addresses:

- `0x7ffce2b57454` in decimal is 140733306597076.
- `0x7ffd96f39704` in decimal is 140733482475268.
- `0x7ffd29853914` in decimal is 140733394179860.

The exampled calculation is as follows:

```
0x7ffce2b57454 = 7 * 16^{11} + f * 16^{10} + f * 16^{9} + c * 16^{8} +
                 e * 16^{7} + 2 * 16^{6} + b * 16^{5} + 5 * 16^{4} +
                 7 * 16^{3} + 4 * 16^{2} + 5 * 16^{1} + 4 * 16^{0}
```

Problem 2



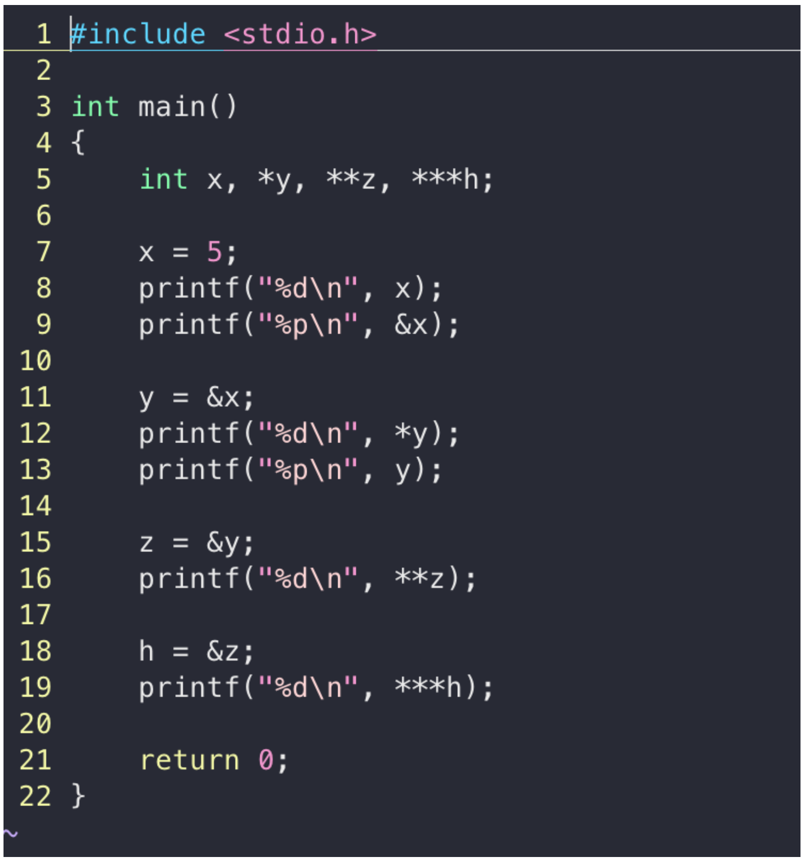
```
mc18 61 $ vim main.c
mc18 62 $ gcc -o main.out main.c
mc18 63 $ main.out
5
0x7fff506c393c
5
0x7fff506c393c
mc18 64 $
```

Figure 2: problem2.c

Explanation of the Output Values

1. `printf("%d", x);` outputs the value of `x`, which is 5.
2. `printf("%p", &x);` outputs the address of `x`. In the given output, this address is `0x7fff506c393c`.
3. `printf("%d", *y);` outputs the value pointed to by `y`. Since `y` is assigned the address of `x` (`y = &x`), `*y` is 5.
4. `printf("%p", y);` outputs the value of `y`, which is the address of `x`, `0x7fff506c393c`.

Modification of `main.c`



```c
#include <stdio.h>

int main()
{
    int x, *y, **z, ***h;

    x = 5;
    printf("%d\n", x);
    printf("%p\n", &x);

    y = &x;
    printf("%d\n", *y);
    printf("%p\n", y);

    z = &y;
    printf("%d\n", **z);

    h = &z;
    printf("%d\n", ***h);

    return 0;
}
~
```

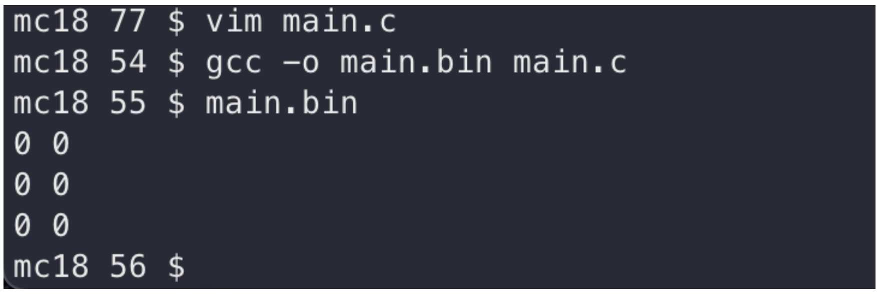
Figure 3: main.c

Explanation of the Modified Code

- Declare `int **z` and assign `z` the address of `y` (`z = &y`).
- `**z` dereferences `z` twice: the first dereference gives `y` (the address of `x`), and the second dereference gives `x`. Therefore, `printf("%d", **z);` outputs the value of `x`, which is 5.
- Declare `int ***h` and assign `h` the address of `z` (`h = &z`).

- `***h` dereferences `h` three times: the first dereference gives `z` (the address of `y`), the second dereference gives `y` (the address of `x`), and the third dereference gives `x`. Therefore, `printf("%d", ***h);` outputs the value of `x`, which is `5`.

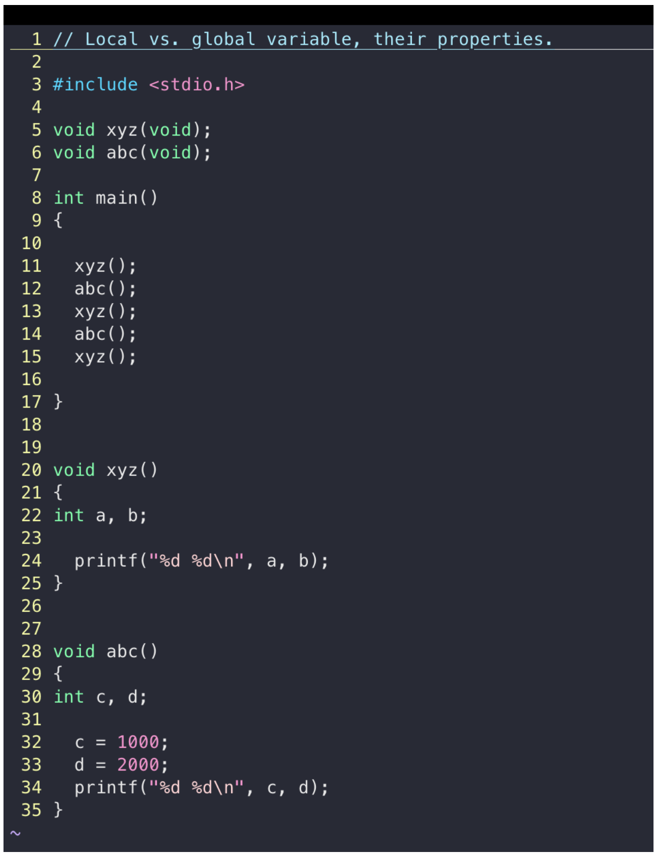
Problem 3



```
mc18 77 $ vim main.c
mc18 54 $ gcc -o main.bin main.c
mc18 55 $ main.bin
0 0
0 0
0 0
mc18 56 $
```

Figure 4: main.bin

This is because the local variables `a` and `b` in the function `xyz()` are uninitialized, and thus they take default values, which happen to be zero in this case. The following picture is the `main1.c`:

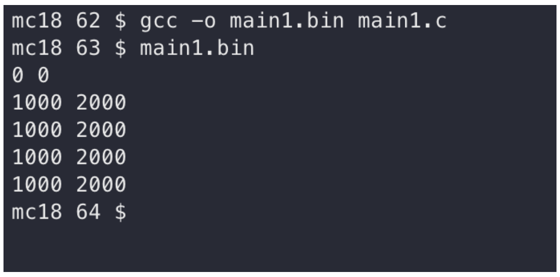


```
// Local vs. global variable, their properties.

#include <stdio.h>

void xyz(void);
void abc(void);

int main()
{

  xyz();
  abc();
  xyz();
  abc();
  xyz();

}


void xyz()
{
int a, b;

  printf("%d %d\n", a, b);
}


void abc()
{
int c, d;

  c = 1000;
  d = 2000;
  printf("%d %d\n", c, d);
}
~
```

Figure 5: main1.c

Observe the output:

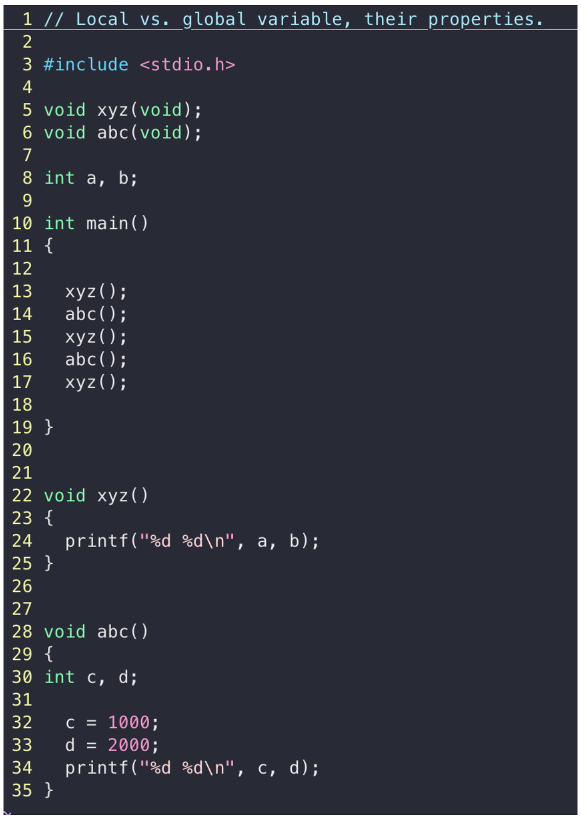
- The variables `c` and `d` in `abc()` are initialized and printed as 1000 and 2000, respectively.
- The local variables `a` and `b` in `xyz()` remain uninitialized and print undefined values. The two local variables of two functions may share the same memory location so function abc() can modify the values of variables in xyz().



```
mc18 62 $ gcc -o main1.bin main1.c
mc18 63 $ main1.bin
0 0
1000 2000
1000 2000
1000 2000
1000 2000
mc18 64 $
```

Figure 6: output of main1.bin

Step 3: Modify and Run main2.c



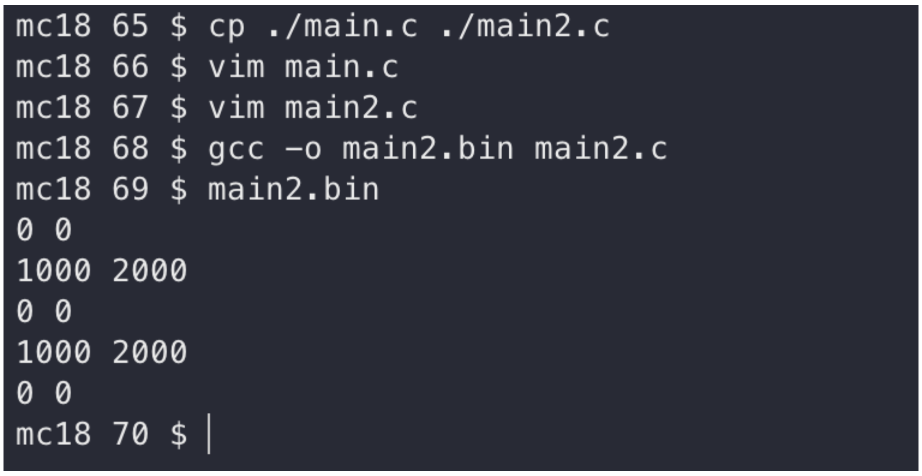
```c
// Local vs. global variable, their properties.

#include <stdio.h>

void xyz(void);
void abc(void);

int a, b;

int main()
{

  xyz();
  abc();
  xyz();
  abc();
  xyz();

}


void xyz()
{
  printf("%d %d\n", a, b);
}


void abc()
{
int c, d;

  c = 1000;
  d = 2000;
  printf("%d %d\n", c, d);
}
```

Figure 7: main2.c

Observe the output

- Since `a` and `b` are global variables, they are automatically initialized to 0 at program startup, so `xyz()` prints 0 0.

- The output of `abc()` remains unchanged, printing 1000 and 2000.



```
mc18 65 $ cp ./main.c ./main2.c
mc18 66 $ vim main.c
mc18 67 $ vim main2.c
mc18 68 $ gcc -o main2.bin main2.c
mc18 69 $ main2.bin
0 0
1000 2000
0 0
1000 2000
0 0
mc18 70 $ |
```

Figure 8: main.bin

Conclusion

- Local variables, when uninitialized, contain indeterminate values and may vary with each function call.
- Global variables are automatically initialized to 0 at program startup if not explicitly initialized.
- Global variables are shared across functions, so changes made in one function affect their values in other functions.

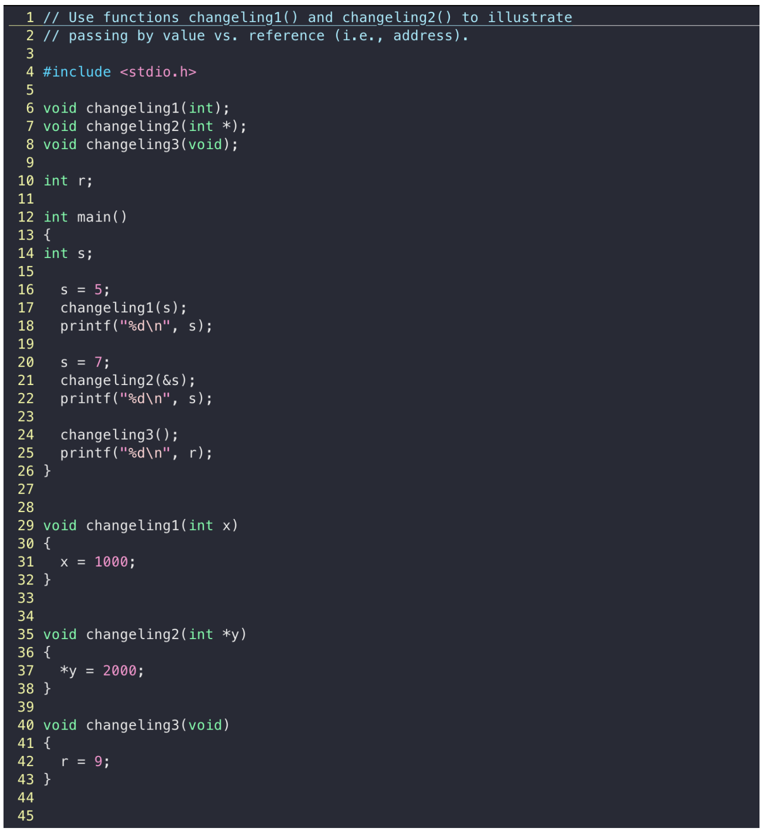
Problem 4

```
5
2000
```

- In the first call to `changeling1(s)`, the value of `s` (which is 5) is passed by value. This means that `changeling1()` receives a copy of `s`, and changes to `x` within `changeling1()` do not affect the original `s`. Therefore, the first `printf()` outputs 5.
- In the second call to `changeling2(s)`, the address of `s` is passed (passing by reference). This means that `changeling2()` can modify the original variable `s` through the pointer `y`. Therefore, the second `printf()` outputs 2000.

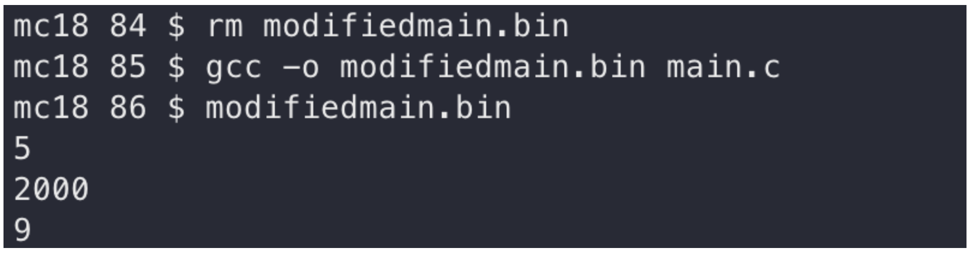
Modified Code and Explanation

- The first and second outputs are the same as previously explained.
- The third `printf()` outputs 9 because `changeling3()` sets the global variable `r` to 9, and `r` is then printed in `main()`.



```c
// Use functions changeling1() and changeling2() to illustrate
// passing by value vs. reference (i.e., address).

#include <stdio.h>

void changeling1(int);
void changeling2(int *);
void changeling3(void);

int r;

int main()
{
int s;

  s = 5;
  changeling1(s);
  printf("%d\n", s);

  s = 7;
  changeling2(&s);
  printf("%d\n", s);

  changeling3();
  printf("%d\n", r);
}


void changeling1(int x)
{
  x = 1000;
}


void changeling2(int *y)
{
  *y = 2000;
}

void changeling3(void)
{
  r = 9;
}

```

Figure 9: main.c



```
mc18 84 $ rm modifiedmain.bin
mc18 85 $ gcc -o modifiedmain.bin main.c
mc18 86 $ modifiedmain.bin
5
2000
9
```

Figure 10: modifiedmain.bin

Pros and Cons of Passing by Reference vs. Using Global Variables

- **Passing by Reference (as in `changeling2()`):**
  - **Pros:**
    - Allows functions to modify the original variable directly.
    - Makes it clear which variables are being modified.
    - Enhances modularity and encapsulation since only the required variables are passed.
  - **Cons:**
    - Can lead to unintended side effects if not used carefully.
    - Requires careful handling of pointers to avoid errors like dereferencing null or invalid pointers.

- **Using Global Variables (as in `changeling3()`)**:
  - **Pros**:
    - Simplifies function signatures since no arguments need to be passed.
    - Provides easy access to variables across multiple functions.
  - **Cons**:
    - Reduces modularity and increases coupling between functions.
    - Makes the code harder to read and maintain, as it becomes unclear where and how global variables are modified.
    - Increases the risk of unintended side effects and bugs due to variable name conflicts and hidden dependencies.

Problem 5

Step 1: Create main4.c and Separate Function Files

`v4/main4.c`:

```c
#include <stdio.h>

void changeling1(int);
void changeling2(int *);
void changeling3(void);

int r; // Declare global variable

int main()
{
  int s;

  s = 5;
  changeling1(s);
  printf("%d\n", s);

  s = 7;
  changeling2(&s);
  printf("%d\n", s);

  changeling3();
  printf("%d\n", r); // Print the value of global variable r
}
```

`v4/changeling1.c`:

```c
void changeling1(int x)
{
  x = 1000;
}
```

v4/changeling2.c:

```
void changeling2(int *y)
{
  *y = 2000;
}
```

v4/changeling3.c:

```
extern int r;
void changeling3(void)
{
  r = 9;
}
```

Step 2: Compile and Execute the Code

```
gcc v4/main4.c v4/changeling1.c v4/changeling2.c v4/changeling3.c -o main4.bin
```

```
./main4.bin
```

Step 3: Verify the Output

```
5
2000
9
```

Explanation of Compilation and Execution Process

When compiling multiple C source files, the compiler needs to be provided with all the files so that it can link them together into a single executable. `gcc` takes `main4.c`, `changeling1.c`, `changeling2.c`, and `changeling3.c`, compiles each of them, and then links the object files together to produce `main4.bin`. In this case, it is beneficial for larger projects where functions are often split into multiple files for better organization and maintainability.

Problem 6

Step 1: Create Object Files

```
gcc -c v4/main4.c -o v4/main4.o
gcc -c v4/changeling1.c -o v4/changeling1.o
gcc -c v4/changeling2.c -o v4/changeling2.o
gcc -c v4/changeling3.c -o v4/changeling3.o
```

Step 2: Statically Link Object Files

Link the object files to generate the executable file `main44.bin`.

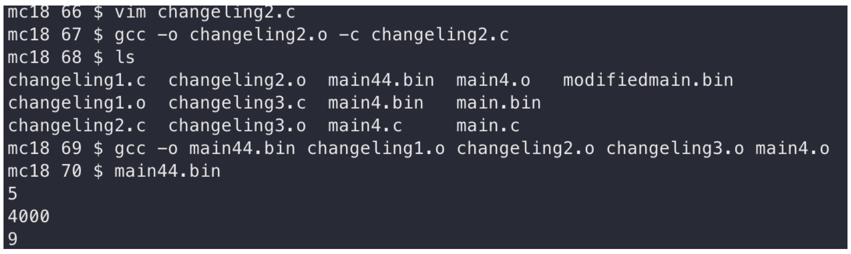
```
gcc -o main44.bin v4/main4.o v4/changeling1.o v4/changeling2.o v4/changeling3.o
```

Step 3: Test and Verify

```
5
2000
9
```

Correctly.

Step 4: Test and Verify the Changed Executable



```
mc18 66 $ vim changeling2.c
mc18 67 $ gcc -o changeling2.o -c changeling2.c
mc18 68 $ ls
changeling1.c  changeling2.o  main44.bin  main4.o   modifiedmain.bin
changeling1.o  changeling3.c  main4.bin   main.bin
changeling2.c  changeling3.o  main4.c     main.c
mc18 69 $ gcc -o main44.bin changeling1.o changeling2.o changeling3.o main4.o
mc18 70 $ main44.bin
5
4000
9
```

Figure 11: output of main44.bin

Correctly.

Problem 7

In the `main()` function, the local variable `s` is declared and initialized to 5. The address of `s` is printed to the standard output, and this address is then passed to the `changevalue()` function as an argument.

When the address of `s` is passed to `changevalue()`, it is passed by value. This means that the address itself is copied to the parameter `x` of the `changevalue()` function. However, `x` itself is a local variable within the scope of `changevalue()`, meaning it has its own memory address, which is different from the address of `s`.

Therefore, the address of the argument `x` (`&x`) printed inside `changevalue()` is different from the address of the local variable `s` in `main()`. The address of `x` refers to where the pointer variable `x` is stored in the stack frame of `changevalue()`, while the value of `x` is the address of `s` in the stack frame of `main()`.

To summarize:

- The address of `x` (`&x`) is the location where the pointer `x` is stored in the `changevalue()` function's stack frame.
- The value of `x` is the address of `s`, which is passed from `main()`.
- This is why the address of `x` (`&x`) is different from the address of `s` (`&s`), while the value of `x` is equal to the address of `s`.

Problem 8

Binary Search Method for Locating Run-time Bugs

1. Divide the code into two approximately equal halves.

2. Insert `printf()` statements at the midpoint.
3. Run the code. If the segmentation fault occurs before the midpoint, the bug is in the first half; otherwise, it is in the second half.
4. Repeat the process on the half where the bug was located.
5. Continue this process until the exact line causing the segmentation fault is identified.

Efficiency of Binary Search

Binary search is efficient for locating run-time bugs because it reduces the search space by half in each step. Instead of checking each statement sequentially, which would take $O(N)$ time, binary search operates in $O(\log N)$ time. This logarithmic complexity significantly speeds up the process of pinpointing the bug.

Worst-case Steps in Binary Search

If a piece of code, `main.c`, has N statements, the number of steps needed in the worst-case to pinpoint the location of a run-time bug using binary search is $\lceil \log_2 N \rceil$. This is because each step halves the number of remaining statements to check.

Example with $N = 4000$

When $N = 4000$, the number of steps required in the worst-case scenario is:

$$\lceil \log_2 4000 \rceil = \lceil 11.97 \rceil = 12$$

Thus, 12 steps are sufficient to locate the run-time bug in the worst-case scenario.

Effect of Removing Newline Characters in `printf()` Calls

If you modify `v6/main.c` so that the newline character '\n' is removed at the end of all debug `printf()` calls, the behavior changes due to how standard output is buffered. Without a newline character, the `printf()` output may be buffered and not immediately flushed to the console. This means that the debug information may not appear in the correct order or may not appear at all before the program crashes, making it difficult to track the execution flow and locate the bug. Including the newline character ensures that the output buffer is flushed, and the debug information is printed to the console immediately, aiding in accurate bug tracing.

Problem 9

Explanation

Layout of 1-D Array in Main Memory

The 1-D integer array `z[3]` is represented in main memory as a contiguous block of memory locations. Each element of the array occupies a specific memory address. For an integer array, each element typically occupies 4 bytes (assuming `sizeof(int) = 4` bytes).

```
z[0] -> | 10 | (address A)
z[1] -> | 20 | (address A + 4)
z[2] -> | 30 | (address A + 8)
```

Why `*(z+2)` is Equivalent to `z[2]`

In C, the expression `z[i]` is equivalent to `*(z + i)`. This is because the array name `z` acts as a pointer to the first element of the array, and adding an integer to a pointer advances the pointer by that many elements. Therefore, `*(z+2)` dereferences the pointer to the element at index 2, which is the same as accessing `z[2]`.

Output of `z` and `&z`

The modified `main()` function prints the addresses of `z` and `&z`:

```
printf("z = %p\n", (void *)z);
printf("&z = %p\n", (void *)&z);
```

The output will show that `z` and `&z` have the same address. This indicates that the name of the array `z` points to the first element of the array, while `&z` points to the array itself.

Deducing the Incorrect Layout

From the output, the layout we discussed is incorrect because `z` and `&z` have the same address, indicating that the array name `z` is simply a pointer to the first element. The array itself does not have a separate address in memory; instead, it is treated as a pointer to its first element.

Optimization by `gcc`

The optimization step carried out by `gcc` for variables declared as 1-D arrays is that the compiler treats the array name as a pointer to the first element. This means the array does not have a distinct address in memory, and the array name is used to reference the first element directly.

Problem 10

Explanation of the Segmentation Fault

A segmentation fault occurs in this program because of the following reasons:

1. **Uninitialized Pointer:**

- The pointer `z` is declared but not initialized.
- This means that `z` points to an undefined memory location.

2. **Dereferencing Uninitialized Pointer:**

- The statement `*z = 10;` attempts to dereference the uninitialized pointer `z`.
- Since `z` does not point to a valid memory location, dereferencing it leads to undefined behavior.
- Specifically, accessing memory that the program does not have permission to access results in a segmentation fault.

3. **Subsequent Dereferencing:**

- The statements `*(z+1) = 20;` and `*(z+2) = 30;` also attempt to write to memory locations relative to the uninitialized pointer `z`.
- These locations are also invalid, and accessing them will cause further segmentation faults.

Where the Segmentation Fault Occurs

The segmentation fault occurs at the first attempt to dereference the uninitialized pointer `z`:

```
*z = 10;
```

At this point, the program tries to write the value `10` to an invalid memory location, resulting in a segmentation fault.

Reason for the Segmentation Fault

The reason for the segmentation fault is that the pointer `z` is not initialized to point to a valid memory location. In C, pointers must be initialized before they are dereferenced. This can be done by:

- Allocating memory dynamically using `malloc()` or `calloc()`.
- Assigning the pointer to the address of an existing variable or an array.

Potential Corrected Code

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

int main()
{
    int *z = (int *)malloc(3 * sizeof(int));
    z[0] = 10;
    z[1] = 20;
    z[2] = 30;

    printf("%d\n", z[0]);
    printf("%d\n", z[1]);
    printf("%d\n", z[2]);

    free(z);

    return 0;
}
```

- `z` is initialized using `malloc()` to allocate memory for three integers.

Problem 11

Silent Run-Time Error

A silent run-time error occurs when a program exhibits incorrect behavior without crashing or generating an error message. In this code, the silent run-time error happens when the third `for` loop accesses and modifies memory beyond the bounds of the array `y`, specifically at index 5:

```
for(i = 0; i < 6; i++)
    y[i] = i;
```

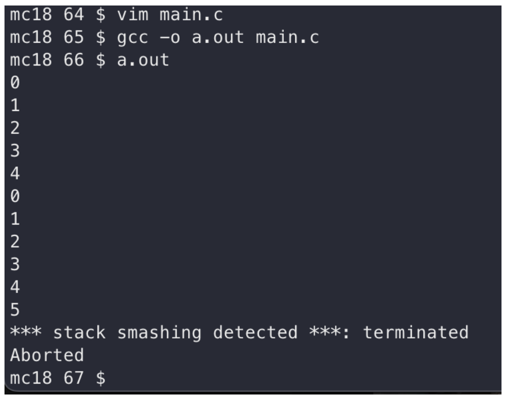
This loop writes to `y[5]`, which is out of bounds since `y` is declared to have only 5 elements (indices 0 to 4).

Comparison with Segmentation Faults

A silent run-time error is worse than an error that causes an application to crash (such as a segmentation fault) because:

- It may go unnoticed and produce incorrect results.
- It can corrupt data and cause unpredictable behavior.
- It is harder to detect and debug since there is no immediate indication that something went wrong.

Changing the Loop Bound



```
mc18 64 $ vim main.c
mc18 65 $ gcc -o a.out main.c
mc18 66 $ a.out
0
1
2
3
4
0
1
2
3
4
5
*** stack smashing detected ***: terminated
Aborted
mc18 67 $
```

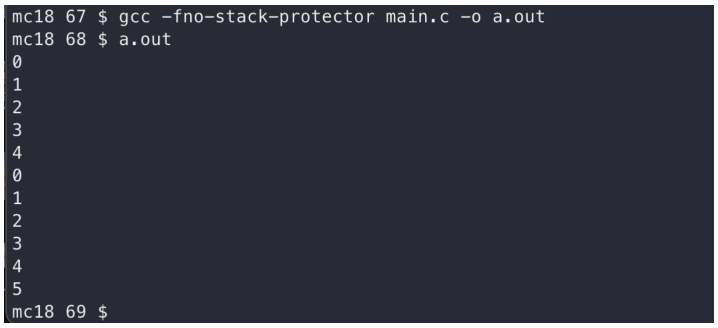
Figure 12: Changing the Loop Bound

This is because the compiler detected that this undefined behavior is trying to overwrite the stack frames which contain the return address or other important data. Therefore, the compiler won't let this dangerous behavior continue. It terminates the program.

Compiling with `-fno-stack-protector`

`gcc -fno-stack-protector main.c -o a.out`

Stack protection helps detect stack overflows by placing a guard value (canary) on the stack. Without this protection, stack-based buffer overflows (such as writing beyond the array bounds) may not be detected, leading to silent data corruption or more severe vulnerabilities such as the execution of arbitrary code by an attacker.



```
mc18 67 $ gcc -fno-stack-protector main.c -o a.out
mc18 68 $ a.out
0
1
2
3
4
0
1
2
3
4
5
mc18 69 $
```

Figure 13: with fno-stack-protector

Explanation of Observed Behavior

When the code is compiled with `-fno-stack-protector` and executed, writing beyond the bounds of the array `y` can corrupt the stack, potentially leading to:

- Overwriting return addresses or other control information.
- Causing the program to behave unpredictably or crash later.
- Making the program more susceptible to security exploits.

By default, enabling stack protection in `gcc` helps detect such issues, but it is not a substitute for careful programming practices. It is the programmer's responsibility to ensure that run-time vulnerabilities, such as buffer overflows, are eliminated from the code.

Problem 12

Modified Code of `main9.c`

```
#include <stdio.h>

int x[5]; // Make the array global

int main()
{
    int i;
    for(i = 0; i < 5; i++)
        x[i] = i;

    for(i = 0; i < 5; i++)
        printf("%d\n", x[i]);

    // doing something sketchy
    for(i = 0; i < 9; i++)
        x[i] = i;

    for(i = 0; i < 9; i++)
        printf("%d\n", x[i]);

    return 0;
}
```

Behavior Comparison and Explanation

In Problem 11, the segmentation fault occurred because the `for` loop attempted to access and modify memory beyond the bounds of the local array `y[5]`. This typically leads to undefined behavior and can cause the program to crash.

In the modified code `main9.c`, the array `x[5]` is declared as a global variable, and the bound of the third `for` loop is changed from 6 to 9. Compiling and executing this code without the `-fno-stack-protector` option results in different behavior:

1. Global Array:

- Since `x` is a global array, it is allocated in the global data segment of memory, not on the stack.
- Writing beyond the bounds of `x[5]` will corrupt the memory in the global data segment rather than the stack.

2. Changing Loop Bound:

- The third `for` loop writes to indices 5 through 8 of `x`. These indices are out of bounds, leading to corruption of adjacent memory.
- The behavior may vary based on the compiler and runtime environment, but typically it leads to corruption of adjacent global variables or other data in the global segment.

3. Stack Protection:

- With the stack protector enabled (not using `-fno-stack-protector`), stack-based buffer overflows are detected and prevented.
- Since `x` is a global variable, stack protection does not directly affect this array, and no stack canary is involved in detecting the overflow.
- Therefore, enabling or disabling the stack protector does not change the outcome of writing beyond the bounds of the global array `x`.

Finding

The key finding is that declaring the array as global changes the location of the overflow from the stack to the global data segment.

- The stack protection mechanisms provided by `gcc` are not triggered, as they primarily protect against stack overflows.
- The overflow in the global array can still cause undefined behavior and potentially corrupt other global variables or data.
- The specific symptoms of this corruption depend on the memory layout and the presence of other global variables or data.

While stack protection is crucial for preventing stack-based overflows, it does not safeguard against overflows in global or heap-allocated arrays.

Problem 13

Solution

To store the string `"hello"` in the array `char x[10]`, the first 6 bytes must be assigned as follows:

```
x[0] = 'h';
x[1] = 'e';
x[2] = 'l';
x[3] = 'l';
x[4] = 'o';
x[5] = '\0'; // Null-terminator
```

The remaining bytes (`x[6]` to `x[9]`) are not used for storing the string `"hello"`.

Bytes used: 6 bytes (5 characters + 1 null-terminator).

Longest string that can be stored in `char x[10]`: 9 characters long because one byte is needed for the null-terminator.

Using `strlen()` to determine the length of `"hello"`:

```
#include <stdio.h>
#include <string.h>

int main() {
    char x[10] = "hello";
    size_t len = strlen(x);
    printf("Length of 'hello' is: %zu\n", len);
    return 0;
}
```

`strlen("hello")` returns 5, which is the length of the string `"hello"` excluding the null-terminator.

Consistency check with man page:

The man page for `strlen()` states that the function returns the length of the string excluding the null-terminator. This is consistent with the value returned by `strlen("hello")`, which is 5.

Return type of `strlen()`:

The man page indicates that the return type of `strlen()` is `sizet`, which is an unsigned integer type. To print a `sizet` value using `printf()`, the format specifier `%zu` should be used.

Bonus Problem

What happens if the argument passed to `strlen()` is missing an end-of-string character?

`strlen()` will continue to read memory beyond the intended end of the string. This can lead to undefined behavior, potentially causing the program to crash or return an incorrect length. The function relies on the presence of the null-terminator to determine where the string ends.

Is `strlen()` fundamentally buggy? Discuss your reasoning.

`strlen()` is not fundamentally buggy. It is designed to work with C-style strings, which are null-terminated. The responsibility of ensuring that the string is properly null-terminated lies with the programmer. When used correctly, `strlen()` functions as intended and efficiently determines the length of a string.

How is the library function `strnlen()` different from `strlen()`?

The library function `strnlen()` is different from `strlen()` in that it takes an additional argument specifying the maximum number of characters to scan. Its prototype is:

```
size_t strnlen(const char *s, size_t maxlen);
```

`strnlen()` scans the string `s` up to `maxlen` characters, returning the length of the string or `maxlen`, whichever is smaller. This ensures that the function does not read beyond a specified number of bytes, providing a safeguard against missing null-terminators.

Does `strnlen()` help address the concern associated with `strlen()`? Explain your reasoning.

Yes, `strnlen()` helps address the concern associated with `strlen()`. By specifying a maximum number of characters to scan, `strnlen()` prevents the function from reading beyond the intended bounds of the string. This can avoid undefined behavior caused by a missing null-terminator. It adds a layer of safety by limiting the number of bytes that the function will examine, reducing the risk of accessing unintended memory areas.