

ME:

II. Pointers Description 1. Use `void* malloc(size_t size)` to allocate `argv[1]` bytes of memory, i.e., `argv[1]` always is an integer. Do not write codes to check if `argv[1]` is an integer. Notice that you must turn `argv[1]`, a `char*` type data, into the integer type to process. Please use `atoi()` in `stdlib.h` for this conversion. 2. Use `char*` and a for loop, assign to the block of memory, byte-by-byte, 'A' to 'Z' then '1' to '9' repeatedly, to fill up the allocated memory. Use one `printf` to print the string generated above. 3. Then use an `int*` and a while loop, in each iteration, print the memory content and its subtract 1 value in a pair of integers, e.g., 1145258561 : 1145258560 4. In the integer process loop, you need to implement a guard to prevent segmentation faults. Please turn in the source codes only. Do not turn in the executable.

CHATGPT:

Here's a C code implementation based on your problem statement:

"C code is in HW2-2.c"

**Explanation:**

1. **Converting argv[1] to an integer:** We use `atoi()` to convert the input string to an integer size to allocate memory.
2. **Memory allocation:** We allocate memory with `malloc()` and check if the allocation is successful.
3. **Filling memory:** We use a for loop to fill the allocated memory with characters from 'A' to 'Z' and then '1' to '9', repeating as necessary to fill the entire block.
4. **Processing with integers:** We use an `int*` pointer to iterate over the memory, treating the content as integers. In each iteration, we print the integer value and the integer value minus one.
5. **Guard against segmentation fault:** We handle the case where the memory size isn't perfectly divisible by the size of an `int` by checking for any remaining bytes after processing full integers.

6. **Memory cleanup:** We free the allocated memory using `free()`.

Let me know if you have any questions about the code!

**ME:**

For Pointers, write a TESTPLAN file to describe a test plan and verify the program's correctness. Turn in the TESTPLAN file as well.

**CHATGPT:**

## **TESTPLAN**

### **Program: Pointer Memory Allocation and Manipulation**

**Objective:** To verify the correctness and robustness of a program that allocates memory, fills it with characters, and processes the memory contents as integers. The program should be able to handle various input sizes, correctly print the memory content, and avoid segmentation faults.

---

## **Test Plan Outline**

### **1. Test Environment Setup**

- Ensure the C compiler (e.g., GCC) is installed and configured.
- Ensure that the program (pointers.c) compiles without errors.
- Ensure the program is executed from the command line with appropriate arguments.

### **2. Test Cases**

#### **2.1 Test Case 1: Minimum Allocation Size (1 Byte)**

- **Test Objective:** Verify how the program handles a minimal memory allocation.
- **Input:** `./pointers 1`
- **Expected Output:**
  - The program should print a single character (A) followed by a

warning about insufficient bytes to form an integer.

- No segmentation fault should occur.
- **Pass Criteria:** The program runs without errors, and output matches expectations.

## 2.2 Test Case 2: Small Allocation (10 Bytes)

- **Test Objective:** Verify behavior with a small memory allocation that is less than the size of two integers.
- **Input:** ./pointers 10
- **Expected Output:**
  - The program should fill the memory with A to J and print the corresponding characters.
  - Only one integer should be printed, followed by a warning about remaining bytes (2 bytes left).
- **Pass Criteria:** Output is correct, and the program runs without warnings except for the remaining bytes.

## 2.3 Test Case 3: Memory Allocation of Exact Multiple of 4 (8 Bytes)

- **Test Objective:** Ensure that when the memory allocation is exactly divisible by the size of an int, the program processes the integers correctly without leaving any leftover bytes.
- **Input:** ./pointers 8
- **Expected Output:**
  - The program should fill memory with characters from A to H and print them.
  - Two pairs of integers and their subtracted values should be printed without any warnings.
- **Pass Criteria:** No warnings, correct integer pairs output.

## 2.4 Test Case 4: Large Memory Allocation (100 Bytes)

- **Test Objective:** Verify the program's behavior when allocating a relatively large block of memory.
- **Input:** ./pointers 100
- **Expected Output:**
  - The memory should be filled with A-Z, 1-9, repeated as necessary.
  - The string should be printed correctly.
  - Integer pairs should be printed, and if there are leftover bytes, a warning should appear for the remaining bytes.
- **Pass Criteria:** The string prints correctly, integer pairs match the memory content, and the program gracefully handles any remaining bytes.

## 2.5 Test Case 5: Large Input (Non-Multiple of 4)

- **Test Objective:** Test if the program handles large inputs that are not multiples of the integer size (i.e., not divisible by 4).
- **Input:** ./pointers 1023
- **Expected Output:**
  - The memory should be filled as described, and the string should be printed correctly.
  - Integer pairs should be printed for as many full int values as possible.
  - A warning about the remaining 3 bytes should be displayed.
- **Pass Criteria:** The string prints correctly, integer pairs print correctly, and the warning about leftover bytes should match expectations.

## 2.6 Test Case 6: Invalid Input (Non-Numeric Input)

- **Test Objective:** Ensure that non-integer input is handled gracefully by the program (though the program is not designed to handle this explicitly).
- **Input:** ./pointers abc

- **Expected Output:**
  - Since `atoi("abc")` returns 0, the program should attempt to allocate 0 bytes of memory.
  - It should print an empty string with no further output.
- **Pass Criteria:** The program does not crash or produce any erroneous output.

### 3. Boundary and Edge Case Tests

#### 3.1 Test Case 7: Allocation of Zero Bytes

- **Test Objective:** Verify how the program handles an allocation request of 0 bytes.
- **Input:** `./pointers 0`
- **Expected Output:**
  - The program should print an empty string and no integer output.
- **Pass Criteria:** The program runs without errors or segmentation faults.

#### 3.2 Test Case 8: Large Allocation Size (Memory Stress Test)

- **Test Objective:** Test how the program handles a large memory allocation (e.g., 1 GB) to see if it can handle memory allocation limits.
- **Input:** `./pointers 1073741824 (1 GB)`
- **Expected Output:**
  - The program should fill memory as described, though output may be truncated due to the large size.
  - No segmentation fault or crash should occur.
- **Pass Criteria:** The program handles large memory allocation without crashing or slowing down significantly.

---

### Testing Procedure

1. Compile the program:

bash

複製程式碼

gcc -o pointers pointers.c

2. Run each test case and compare the program's output against the expected output.
3. Record the results of each test in a test report.

---

### Pass/Fail Criteria

- **Pass:** The program produces the expected output for the given input, and no segmentation faults or crashes occur.
- **Fail:** The program crashes, produces incorrect output, or causes segmentation faults for any test case.

---

### Additional Notes:

- Monitor memory usage for larger test cases using tools like valgrind to ensure no memory leaks occur.
- For test case 6 (invalid input), since the specification does not mandate error handling for non-numeric inputs, we will treat it as a successful run if no crash occurs.

---

This test plan covers various input sizes, boundary conditions, and edge cases to ensure the correctness and stability of the program.