**1-usestaticarray.c:**

**Experiment Setup:**

* Initial Memory Address:
  + array1 starts at: 0000000FFE3FF910
  + array2 starts at: 0000000FFE3FF920

**Input Sequence and Observations:**

1. Input: 4
   * array2 (expected to show contents of array1): 4 0 19 0
   * array3 (actual array2): 10 11 12 13
   * Observation: First input is stored correctly within the bounds of array1.
2. Input: 5
   * array2: 4 5 19 0
   * array3: 10 11 12 13
   * Observation: Second input also remains within array1 boundaries.
3. Input: 6
   * array2: 4 5 6 0
   * array3: 10 11 12 13
   * Observation: Third input still within bounds.
4. Input: 7
   * array2: 4 5 6 7
   * array3: 10 11 12 13
   * Observation: Fourth input fills the last slot in array1.
5. Input: 8 (starting out-of-bounds)
   * array2: 4 5 6 7
   * array3: 8 11 12 13
   * Observation: First out-of-bounds input overwrites the beginning of array3, affecting the first element.
6. Input: 10
   * array2: 4 5 6 7
   * array3: 8 10 12 13
   * Observation: Second out-of-bounds input overwrites the second element of array3.
7. Input: 15555555555
   * array2: 4 5 6 7
   * array3: 8 10 12 15555555555
   * Observation: Large numerical input further demonstrates the vulnerability as it overwrites another element in array3.

**Analysis:**

* The experiment clearly demonstrates how writing beyond the allocated array (array1) starts to overwrite adjacent memory space occupied by array3.
* This type of error, known as a buffer overflow, is a common and dangerous vulnerability in C programming. It exposes programs to crashes, data corruption, and security breaches.

**Conclusion:**

* This simple test underscores the importance of implementing and adhering to strict boundary checks in programming to prevent overflow and its potentially severe implications.
* Future coding practices should include using safer functions and thorough testing to manage memory correctly and securely.

**Screen Shots:**

**A screenshot of a computer

AI-generated content may be incorrect.**  
**A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.**

**2-mallocprinter.c:**

**Results:**

**Memory Addresses**:

* + Dynamically Allocated Array (array1): 000001D85F9BEAA0
  + Statically Allocated Array (array2): 0000004A011FF9E0

**Initial Values**:

* + array1 (malloc-allocated): 7602273 (uninitialized, contains garbage data).
  + array2 (statically declared): -46686688 (also uninitialized with garbage data).

**Analysis:**

**Dynamic vs. Static Memory Allocation**:

* + The memory for array1 is allocated on the heap, showing a typical use of malloc for dynamic memory needs. array2 is allocated on the stack, representing static allocation.
  + Both arrays start with garbage values, demonstrating that neither method automatically initializes the memory.

**Security and Stability Concerns**:

* + Using uninitialized memory can lead to security vulnerabilities, such as data leakage or unpredictable behavior.
  + The demonstration underscores the necessity of initializing memory after allocation, particularly when using malloc, and the importance of freeing dynamically allocated memory to avoid memory leaks.

**Screen Shot:**

A screen shot of a computer

AI-generated content may be incorrect.

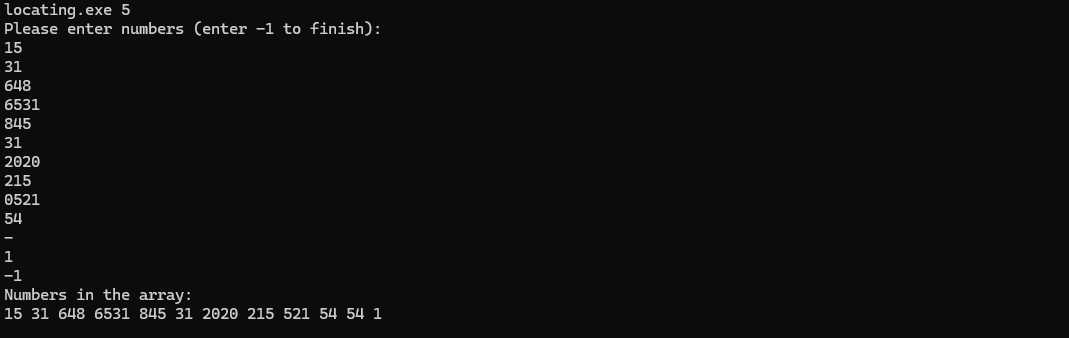
**3-dynamically-allocating.c:**

In some programs, you don't know how many items you'll need to store until the program is actually running. That's why it's useful to be able to change the size of an array while the program is running. The realloc() function is perfect for this because it lets you make the array bigger or smaller depending on what you need at that moment. This way, you don't waste space or run out of room.

When you use realloc(), it can also move the array to a new spot if it needs more space to grow. We make the array bigger each time it gets full by doubling its size. This trick helps avoid having to resize the array too many times, which can slow down the program. It's like moving to a bigger house each time your family grows instead of moving every time a new baby is born.

Also, it’s really important to check if realloc() works properly because if it doesn’t, you need to clean up the old space to avoid losing memory, like making sure you don’t leave anything behind when you move houses. This keeps the program from crashing and helps everything run smoothly.

**Screen Shot:**



**4-simple-dynamic-example.c:**

Calloc is particularly useful in situations where the memory needs to be zero-initialized. Unlike malloc, which merely allocates a block of memory, calloc allocates the memory and sets all bits in the allocated space to zero. This behavior is shown in the program 4-simple-dynamic-example.c, where 12 integers are allocated and initialized to zero before being explicitly set to other values. The output demonstrates how the memory is both allocated and accessed.

**Screen Shot:**

A screen shot of a computer

AI-generated content may be incorrect.

**5-python-lists-are-objects-not-arrays.py:**

**What is an Object in Programming?**

In programming, an object is like a little box that holds information and tools to do things with that information. This box helps keep things organized because all the stuff related to one kind of information is kept together in one place.

**How a Python List Works Like an Object**

A Python list is a great example of one of these boxes. It doesn't just hold a bunch of numbers or words; it also has built-in tools that let you do things like add more items, sort them, or put something in the middle. When you use these tools, you don't need to worry about the tricky parts, like making sure there's enough space to add more items. Python handles all that tricky stuff behind the scenes, so you can focus on what you want to do with your list, not how it actually gets done.

**Screen Shot:**

A number on a black background

AI-generated content may be incorrect.

**6-linkedlist.c:**

Linked lists are like a train where each car is hooked to the next one. This makes it easy to add or remove cars anywhere in the train without having to move all the other cars around. When you use regular arrays, it's like having a row of houses; if you want to add a new house in the middle, it can be a big hassle because you might need to shift all the other houses down. But with linked lists, you just find the spot where you want to add or take away a car (or a data item), hook or unhook the cars, and you're done. This saves a lot of effort and makes things run smoother, especially when you're not sure how many items you'll end up with.

**Screen Shot:**



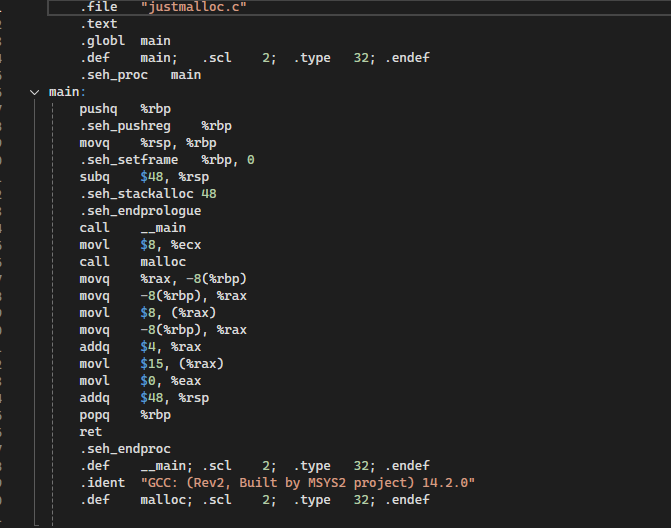
**7-managing-allocation-with-struct:**



**Justarray.c:**

A screen shot of a computer

AI-generated content may be incorrect.

**Justmalloc.c:  
**

**mallocofstructs.c:**

struct book {

    char bookTitle[50];

    int bookNumber;

    int bookPages;

};

**A screenshot of a computer

AI-generated content may be incorrect.**

**8-useofheaders.h:**

**Header File Creation: mathfunctions.h**

The header file mathfunctions.h was designed to declare two fundamental arithmetic functions:

A computer screen shot of a computer code

AI-generated content may be incorrect.

**Header Guards**: The use of #ifndef, #define, and #endif prevents multiple inclusions of this header file, ensuring that the compiler processes the declarations only once.

**Function Implementation: mathfunctions.c**

This source file provides definitions for the functions declared in the header file, ensuring that the actual functionality is abstracted away from their usage:

A screen shot of a computer code

AI-generated content may be incorrect.

**Main Program Functionality: usemath.c**

The main program includes the header file and uses the declared functions to perform arithmetic operations:

A computer screen shot of a program code

AI-generated content may be incorrect.

**Output:**

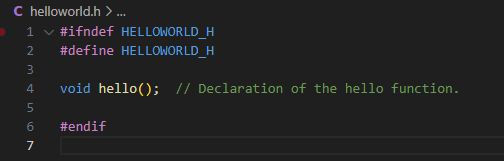
**A black background with white text

AI-generated content may be incorrect.**

**helloworld.c/** **helloworld.h:**

The program consists of the following components:

* **helloworld.h**: The header file that declares the hello() function.



* **helloworld.c**: The implementation file that defines the hello() function.

A screenshot of a computer program

AI-generated content may be incorrect.

* **main.c**: The main program file that includes helloworld.h and calls the hello() function.

A screen shot of a computer

AI-generated content may be incorrect.

**Output:**

