**04\_variables**

**What's the difference between:**

1. printf("You are %d years old.", age);

2. printf("You are %i years old.", age);

The %d and %i format specifiers are interchangeable when dealing with integers in printf. However, there is a difference between %d and %i when used with scanf for input. %d expects the input to be in decimal format, while %i allows for input in decimal, octal, or hexadecimal format, based on prefixes like 0x for hexadecimal and 0 for octal.

**30\_nested\_loops**

**If I try to use scanf to read a character after having read a numeric value, it fails.**

The issue you're encountering is due to the behavior of scanf when reading characters after reading numeric values.

When you use scanf to read numeric values (%d for integers in this case), it reads the numeric value but leaves the newline character ('\n') in the input buffer. When you then use scanf to read a character (%c), it reads the next character in the input buffer, which is often the newline character left over from the previous input operation.

To fix this issue, you need to consume the newline character from the input buffer before reading the character. You can do this by adding a space before %c in the format string, like this:

scanf(" %c", &symbol); // Note the space before %c

The space before %c in the format string tells scanf to skip any whitespace characters (including newline characters) before reading the character. This ensures that scanf reads the correct character input from the user.

**33\_print\_an\_array\_with\_a\_loop**

**What’s the size of a pointer?**

The size of a pointer depends on the architecture and the compiler being used.

In 32-bit systems, pointers are typically 4 bytes in size because memory addresses are represented using 32 bits.

In 64-bit systems, pointers are typically 8 bytes in size because memory addresses are represented using 64 bits.

However, the actual size of a pointer can vary depending on the specific architecture and compiler. For example, on some embedded systems or specialized architectures, pointers may have different sizes.

**37\_sort\_an array**

**Why passing the pointer to the first element of the array is enough for the program to access the rest of the elements of the array?**

In C, when you declare an array, it is stored as a contiguous block of memory. The name of the array acts as a pointer to the first element of this memory block. Therefore, when you pass an array to a function, you're actually passing a pointer to its first element.

Let's break it down:

1. **Declaration and Memory Allocation**: When you declare an array, memory is allocated to hold all of its elements. This creates an array named myArray with five elements, and memory is allocated to hold these elements sequentially. For example:

int myArray[] = {1, 2, 3, 4, 5};

1. **Array Name as a Pointer**: The name myArray itself acts as a pointer to the first element of the array. So, myArray is equivalent to &myArray[0].
2. **Passing to a Function**: When you pass myArray to a function, you're passing a pointer to the first element of the array. For example:

void myFunction(int\* array) {

// Here, 'array' is a pointer to the first element of the array passed to the function

}

1. **Accessing Array Elements**: Once you have the pointer to the first element of the array, you can access the rest of the elements by incrementing the pointer. For example:

void d myFunction(int\* array) {

// Access the first element

int firstElement = array[0]; // Equivalent to \*array

// Access the second element

int secondElement = array[1]; // Equivalent to \*(array + 1)

// Access the third element

int thirdElement = array[2]; // Equivalent to \*(array + 2)

// and so on...

}

By manipulating the pointer to the first element (array), you can effectively access all the elements of the array sequentially. This is why passing the pointer to the first element of the array to a function allows the function to access the entire array.