9.- ft_memcpy.-

Function based on the definition given in the BSD man pages for “memcpy”.
The library associated is <string.h>.

Sinopsis: void * memcpy(void *restrict dst, const void *restrict src, size_t n);

We shall use the next synopsis:

```
void *ft_memcpy(void *dst, const void *src, size_t n);
```

Parameters:

- **dst:** The destination memory location where the contents of **`src`** will be copied to.
- **src:** The source memory location whose contents will be copied to **`dst`**.
- **n:** The number of bytes to be copied from **`src`** to **`dst`**.

Description:

The **`ft_memcpy`** function copies `n` bytes from the memory location pointed to by `src` to the memory location pointed to by **`dst`**. The memory areas **`dst`** and **`src`** must not overlap.

Code:

```
#include "libft.h"

void            *ft_memcpy(void *dst, const void *src, size_t n)
{
        size_t          i;

        i = 0;
        if (dst == NULL && src == NULL)
                return (NULL);
        while (i < n)
        {
                ((unsigned char *)dst)[i] = ((unsigned char *)src)[i];
                i++;
        }
        return (dst);
}
/*
int             main(void)
{
        char    src[18] = "memcpy example";
        char    dst[18];

        ft_memcpy(dst, src, sizeof(src));
        printf("Copied string:  %s\n", dst);
        return (0);
}
*/
```

Explanation:

1. **Include header file:** The **`#include "libft.h"`** statement includes the header file **`libft.h`**, which defines the required libraries for our function.

2. Define function: The **`void *ft_memcpy(void *dst, const void *src, size_t n)`** statement defines the **`ft_memcpy`** function. The function takes three arguments: **`dst`**, **`src`**, and **`n`**.

3. Check for NULL pointers: The **`if (dst == NULL && src == NULL)`** statement checks if both **`dst`** and **`src`** are NULL pointers. If they are, the statement returns **`NULL`**. This is because copying from or to a NULL pointer is undefined behavior.

4. Copy bytes: The **`while (i < n)`** loop copies `n` bytes from **`src`** to **`dst`**. The **`((unsigned char *)dst)[i] = ((unsigned char *)src)[i];`** statement copies one byte from **`src`** to **`dst`** at a time. The **`unsigned char`** cast is necessary to ensure that the data is interpreted as unsigned characters.

5. Return destination pointer: The **`return (dst);`** statement returns the address of the **`dst`** pointer. This is because the **`ft_memcpy`** function is expected to return the destination pointer after the successful copying of data.

6. Main function: The **`int main(void)`** statement defines the main function, which is the entry point of the program. The main function copies the contents of the `src` array to the `dst` array using the **`ft_memcpy`** function. The **`printf("Copied string: %s\n", dst);`** statement prints the contents of the **`dst`** array.

7. Return value: The **`return (0);`** statement exits the program with a status code of 0, indicating that the program executed successfully.

ADDENDUM:

Meaning of `restrict`:

- **Compiler Optimization Hint:** The `restrict` keyword is a qualifier that provides a hint to the compiler about memory aliasing. It indicates that the pointers `dst` and `src` in `memcpy` are the only means to access the memory blocks they point to.
- **Potential for More Efficient Code:** By assuming no aliasing, the compiler can potentially generate more efficient code, such as:
    - Unrolling loops for faster copying.
    - Using vector instructions for optimized memory o

Why `restrict` Isn't Used in `ft_memset`:

- **No Aliasing Concerns:** `ft_memset` only writes to a single memory block, so aliasing isn't an issue. Therefore, `restrict` doesn't offer optimization benefits in this case.
- **Clarity and Compatibility:** Using `restrict` unnecessarily can sometimes make code less readable and potentially introduce compatibility issues with older compilers.

Key Points:

- `restrict` is a hint to the compiler, not a strict constraint.
- It can potentially improve performance, but its impact depends on the compiler and code context.
- Use it judiciously for clarity and compatibility.

Additional Considerations:

- `restrict` is part of the C language standard since C99.
- It's often used in library functions that deal with memory blocks, like `memcpy`, `memmove`, and `memset`.
- It's important to understand its implications for correctness and optimization.

Conclusion:

- Prioritize clarity and compatibility when using `restrict`.
- Consider its potential benefits for optimization, but evaluate its impact in your specific context.
- Adhere to standard C syntax for cross-platform compatibility.

Key Points:

- **Consistency with Standard Library:** Using `restrict` aligns `ft_memcpy` with the standard `memcpy` signature, promoting clarity and consistency.
- **Potential Optimization:** While not strictly necessary for correctness, `restrict` might still offer performance benefits in certain scenarios, especially on modern compilers.
- **Minimal Risks:** The likelihood of compatibility issues with modern compilers and systems is relatively low.

Best Practices:

- **Prioritize Clarity:** Ensure the code remains clear and understandable for other developers.
- **Consider Optimization:** Evaluate the performance impact of `restrict` in your specific context and make informed decisions based on your project's requirements.
- **Adhere to Standards:** Generally, it's good practice to follow standard C syntax and conventions for maintainability and portability.

Additional Tips:

- **Documentation:** Clearly document the use of `restrict` in your code, explaining its purpose and potential implications.
- **Testing:** Thoroughly test your code with and without `restrict` to measure any performance differences and ensure correctness.