1. What is a macro in C, and how is it defined?

A **macro** in C is a preprocessor directive used to define a symbol or code snippet that will be expanded by the preprocessor before the actual compilation takes place. Macros are typically used to define constants or simple functions. A macro is defined using the #define keyword followed by the macro name and its value or code.

Eexample:

```c
#define PI 3.14159    // A simple constant macro
#define SQUARE(x) ((x) * (x))    // A macro that acts like a function
```

In this example, PI is a constant, and SQUARE(x) is a macro function that calculates the square of a number.

2. What is the difference between macros and functions?

The main differences between **macros** and **functions** are:

- **Macros** are expanded by the preprocessor before compilation. They have no runtime overhead, and they perform direct substitution in the code, meaning they can be faster than functions. However, macros can have issues like multiple evaluations of their arguments, which can lead to unexpected results or inefficiency.
- **Functions** are executed during runtime. They provide type safety, avoid multiple evaluations of arguments, and can handle complex operations more efficiently than macros. Functions are also subject to the calling overhead at runtime, but they provide better error checking and debugging.

Example:

```c
#define SQUARE(x) ((x) * (x))  // Macro - x is evaluated twice if called as SQUARE(a + 1)
int square_func(int x) { return x * x; }  // Function - x is evaluated once
```

3. What do #ifdef, #ifndef, and #endif do?

These are conditional compilation directives used by the preprocessor in C to include or exclude parts of the code based on whether a certain macro is defined or not.

- **#ifdef**: Checks if a macro is defined. If it is, the code between #ifdef and #endif is included in the compilation.
- **#ifndef**: Checks if a macro is not defined. If it is not defined, the code between #ifndef and #endif is included.
- **#endif**: Ends the conditional block initiated by #ifdef or #ifndef.

Example:

```c
#define DEBUG   // Macro defined

#ifdef DEBUG
    printf("Debugging enabled\n");
#endif
```

Here, since DEBUG is defined, the message "Debugging enabled" will be printed.

4. What does malloc() do, and what type does it return?

The malloc() function allocates a block of memory of a specified size and returns a pointer to the beginning of the allocated memory. The type of the pointer returned by malloc() is void *, meaning it can be cast to any data type.

Example:

```c
int *arr = (int *)malloc(10 * sizeof(int));  // Allocates memory for 10 integers
```

If malloc() fails to allocate memory, it returns NULL, which should always be checked before using the allocated memory.

5. What is the difference between malloc, calloc, and realloc?

- **malloc(size_t size)**: Allocates a block of memory of the given size. The content of the allocated memory is uninitialized, meaning it may contain garbage values.
- **calloc(size_t num, size_t size)**: Similar to malloc, but it allocates memory for an array of num elements of a given size and initializes the entire block to zero.
- **realloc(void *ptr, size_t new_size)**: Resizes a previously allocated memory block to a new size. If the block is enlarged, the new memory is not initialized, and if the block is shrunk, data may be lost. It may return a new pointer if the block cannot be resized in place.

Example:

```c
int *arr = (int *)malloc(10 * sizeof(int));  // Allocation with malloc
arr = (int *)realloc(arr, 20 * sizeof(int)); // Reallocating the array to hold 20 elements
```

6. Why must we always call free() after dynamic allocation?

Dynamic memory allocation (using malloc(), calloc(), or realloc()) reserves memory from the heap. If we do not call free() after we are done using the allocated memory, that memory will remain allocated, leading to a **memory leak**. This can eventually cause the program to run out of memory, especially in long-running applications.

Example:

```c
int *arr = (int *)malloc(10 * sizeof(int));  // Dynamically allocated memory
free(arr);  // Releasing the memory after use
```

7. What is a header guard, and what problem does it solve?

A **header guard** is a mechanism used to prevent multiple inclusions of the same header file in a program. Without a header guard, if a header file is included more than once in the same source file (either directly or through other included files), it can lead to redefinition errors (such as variables, functions, or types being declared multiple times).

A header guard prevents this by ensuring that the contents of a header file are included only once.

8. What is the typical format of a header guard?

The typical format for a header guard involves using #ifndef (if not defined), #define, and #endif to ensure that a header file is included only once.

Example:

```c
#ifndef HEADER_FILE_NAME_H   // Check if the macro is defined
#define HEADER_FILE_NAME_H   // Define the macro to prevent re-inclusion

// Header file content here

#endif  // End the conditional block
```

If the header file is included multiple times, the #ifndef will fail after the first inclusion, skipping the code.

9. How does the preprocessor handle nested includes?

The preprocessor handles **nested includes** by including files recursively. When the preprocessor encounters an #include directive, it checks if the file has already been included, using mechanisms like header guards. If the file has not been included yet, it processes the contents of the file.

Example:

```c
#include "file1.h"   // This includes file1.h
#include "file2.h"   // This includes file2.h
```

If file1.h includes file2.h, then the preprocessor will process file2.h once and skip any subsequent includes, preventing infinite recursion or redefinition errors.