1. How to define a macro function? What are the points that need to pay attention?

   soln:
   Defining a macro function:
   #define MACRO_NAME(parameters) replacement_code

   Key Points to Consider When Defining Macro Functions:

   - Use Parentheses to Avoid Operator Precedence Issues

     example:

     ```
     #define SQUARE(x) x * x  //Incorrect, Missing parentheses

     #define SQUARE(x) ((x) * (x))  // correct
     ```

   - Be Careful with Multiple Evaluations: Macros substitute code, so arguments may be evaluated multiple times.

     example:

     ```
     #define Double(x) (x + x)

     int main() {

        int a = 5;

        int b = Double(++a);

        printf("%d %d\n",a, b);  // Unexpected result due to multiple evaluations

     }
     ```

   - Handling Variadic Arguments (__VA_ARGS__): can create **variadic macros** (macros that accept a variable number of arguments):

     example:

     ```
     #define LOG(fmt, ...) printf("[LOG] " fmt "\n", ##__VA_ARGS__)
     ```

- Use do { ... } while(0) for Multi-Line Macros:

  #define PRINT_SUM(a, b) printf("Sum: %d\n", a + b); printf("Done\n"); //incorrect

  #define PRINT_SUM(a, b) do { printf("Sum: %d\n", (a) + (b)); printf("Done\n"); } while(0) // correct

2. Write a macro MIN that accepts 3 parameters and returns the smallest one

   soln:

```
#define MIN(a, b, c) (((a) < (b)) ? ((a) < (c)) ? (a) : (c)) : (((b) < (c))? (b) : (c)))
```

3. There is a debug function that has no output when the macro DEBUG is not defined. When the macro DEBUG is defined, debug outputs the file name, function name, line number, and content. The call and output are given.
   a. Please write the macro definition

   soln:

```
//macro defination
#ifdef DEBUG
    #define debug(fmt, ...) printf("[DEBUG] File: %s, Function: %s, Line: %d -> " fmt "\n", __FILE__, __func__, __LINE__, ##__VA_ARGS__)
#else
    #define debug(fmt, ...)
#endif
```

b. How to open the macro by command line?

soln:
source code:

```c
#include <stdio.h>

//macro defination
#ifdef DEBUG
    #define debug(fmt, ...) printf("[DEBUG] File: %s, Function: %s, Line: %d -> " fmt "\n", __FILE__, __func__, __LINE__, ##__VA_ARGS__)
#else
    #define debug(fmt, ...)
#endif

void test_function(int x) {
    debug("Value of x: %d square = %d", x, x*x);
}

int main(){

    int a = 42;
    debug("Starting main function");
    test_function(a);

    return 0;
}
```

Command:
gcc -DDEBUG prep_3.c -o prep_3
prep3.exe

Output:

```
C:\Users\Admin\Documents\bdcom_coding_zone>prep_3.exe
[DEBUG] File: prep_3.c, Function: main, Line: 17 -> Starting main function
[DEBUG] File: prep_3.c, Function: test_function, Line: 11 -> Value of x: 42 square = 1764
```

4. What is the difference between macro definition and typedef?

   soln:
   Macro Definition : used to define constants, function-like macros, and type alias.The **#define** directive is handled by the preprocessor before compilation.

   typedef: used to create an alias for an existing data type.typedef processed by compiler, ensuring type safety.
   The difference table given below:

Feature	#define Macro	typedef
Processed by	Preprocessor	Compiler
Type Safety	No	Yes
Can define constants?	Yes	No
Can define function-like macros?	Yes	No
Handles Pointers Well?	No	Yes

5. Both macro definition and typedef can help simplify code and improve its readability. However, sometimes they are different. What's the difference between them for the variables in the following program?
   soln:
   - **U_INT32_1 a, b;** expands to **unsigned int* a, b;**
     - here only a is a pointer, but b is just an unsigned int.Because the macro simply replaces U_INT32_1 with unsigned int*, and b follows normal declaration rule.

   - **U_INT32_2 c, d;** is treated as **unsigned int* c, *d;**
     - here both c and d is pointer.The compiler understands U_INT32_2 as a true type alias for unsigned int*, so both c and d are pointers.

6. When do we need to use typedef? Illustration

   soln: The typedef keyword in C is used to define new names (aliases) for existing types, making code more readable, portable, and maintainable.Below are situations where typedef is beneficial, along with illustrative examples.

   - **Simplifying Complex Data Types**:When a data type is too long or complex, typedef makes it more readable.

     ```
     //example code

     typedef unsigned long int ULI;
     ULI value = 100000;  // Instead of using 'unsigned long int' every time.
     ```

   - **Creating Portable Code**:Different platforms may define data types differently. Using typedef, we can ensure portability.

     ```
     //example code

     #ifdef _WIN32
     typedef unsigned __int64 ULL;  // Windows-specific
     #else
     typedef unsigned long long ULL; // Other platforms
     #endif

     ULL value = 1234567890;
     ```

   - **Improving Code Readability and Maintainability**:Using typedef makes structures more intuitive and easy to understand.

     ```
     //example code

     typedef struct {
        char name[50];
        int age;
        float gpa;
     } Student;

     Student s1;  // No need for 'struct'
     ```

   - **Making Pointers More Readable**:

     ```
     //example code

     int* p1, p2; // many beginnres think both p1 and p2 are pointers, but only p1 is a pointer
     ```

```
typedef int* IntPtr;  // Defines 'IntPtr' as a type alias for 'int*'

IntPtr p1, p2;  // Both 'p1' and 'p2' are now pointers!
```

7. How to use conditional compilation to improve code portability?
   soln:
   Examples of Conditional Compilation for Portability:
   - **Platform-Specific Code:** Some system calls or libraries are **OS-dependent**. Conditional compilation helps handle such cases.

```
//example code:

#include <stdio.h>

#ifdef _WIN32
   #include <windows.h>
   void clearScreen() { system("cls"); }
#elif __linux__
   #include <unistd.h>
   void clearScreen() { system("clear"); }
#else
   void clearScreen() { printf("Screen clearing not supported.\n"); }
#endif

int main() {
   clearScreen();
   printf("Hello, World!\n");
   return 0;
}
```

   - **Compiler-Specific Code (#if defined())** : Different compilers have unique features or built-in macros.

```
//example code:

#include <stdio.h>

#if defined(__GNUC__)

   #define COMPILER "GCC"

#elif defined(_MSC_VER)
```

```
    #define COMPILER "MSVC"
#else
    #define COMPILER "Unknown Compiler"
#endif
int main() {
    printf("Compiled with %s\n", COMPILER);
    return 0;
}
```

- **Architecture-Specific Code**:When targeting different CPU architectures (e.g., x86 vs. ARM), we can conditionally include optimized assembly or processor-specific instructions.

```
//example code:

#if defined(__x86_64__)
    #define ARCH "x86_64"
#elif defined(__arm__)
    #define ARCH "ARM"
#else
    #define ARCH "Unknown Architecture"
#endif

int main() {
    printf("Running on %s architecture\n", ARCH);
    return 0;
}
```

- **Feature Toggle (#define and #ifdef)** : Useful for enabling or disabling features based on user-defined macros.

```
#define DEBUG_MODE  // Comment this line to disable debugging
```

```c
#include <stdio.h>

int main() {
    #ifdef DEBUG_MODE
        printf("Debug mode is ON\n");
    #else
        printf("Debug mode is OFF\n");
    #endif
    return 0;
}
```