**Preprocessor in C**

Write a C macro PRINT(x) which prints x

At the first look, it seems that writing a C macro which prints its argument is child’s play. Following program should work i.e. it should print *x*

|  |
| --- |
| #define PRINT(x) (x)  int main()  {    printf("%s",PRINT(x));    return 0;  } |

But it would issue *compile error* because the ***data type of x, which is taken as variable by the compiler, is unknown***. Now it doesn’t look so obvious. Isn’t it?

Guess what, the followings also won’t work

|  |
| --- |
| #define PRINT(x) ('x')  #define PRINT(x) ("x") |

But if we know one of lesser known traits of C language, writing such a macro is really a child’s play.

In C, there’s a **# directive**, also called ‘**Stringizing Operator**’, which does this magic. Basically # directive converts its argument in a string. Voila! it is so simple to do the rest. So the above program can be modified as below.

|  |
| --- |
| #define PRINT(x) (#x)  int main()  {    printf("%s",PRINT(x));    return 0;  } |

Now if the input is *PRINT(x)*, it would print *x*. In fact, if the input is *PRINT(geeks)*, it would print *geeks*.

Variable length arguments for Macros

Like functions, we can also pass variable length arguments to macros. For this we will use the following preprocessor identifiers.

To support variable length arguments in macro, we must include ellipses **(…)** in macro definition. There is also “\_\_VA\_ARGS\_\_” preprocessing identifier which takes care of variable length argument substitutions which are provided to macro. Concatenation operator ## (aka paste operator) is used to concatenate variable arguments.

Let us see with example. Below macro takes variable length argument like “printf()” function. This macro is for error logging. The macro prints filename followed by line number, and finally it prints info/error message. First arguments “prio” determines the priority of message, i.e. whether it is information message or error, “stream” may be “standard output” or “standard error”. It displays INFO messages on stdout and ERROR messages on stderr stream.

#include <stdio.h>

#define INFO    1

#define ERR 2

#define STD\_OUT stdout

#define STD\_ERR stderr

#define LOG\_MESSAGE(prio, stream, msg, ...) do {\

                        char \*str;\

                        if (prio == INFO)\

                            str = "INFO";\

                        else if (prio == ERR)\

                            str = "ERR";\

**fprintf(stream, "[%s] : %s : %d : "msg" \n", \**

**str,** \_\_**FILE**\_\_**,** \_\_**LINE**\_\_**, ##**\_\_**VA\_ARGS**\_\_**);**\

                    } while (0)

int main(void)

{

    char \*s = "Hello";

        /\* display normal message \*/

    LOG\_MESSAGE(ERR, STD\_ERR, "Failed to open file");

    /\* provide string as argument \*/

    LOG\_MESSAGE(INFO, STD\_OUT, "%s Geeks for Geeks", s);

    /\* provide integer as arguments \*/

    LOG\_MESSAGE(INFO, STD\_OUT, "%d + %d = %d", 10, 20, (10 + 20));

    return 0;

}

Compile and run the above program, it produces below result.

[narendra@/media/partition/GFG]$ ./variable\_length

[ERR] : variable\_length.c : 26 : Failed to open file

[INFO] : variable\_length.c : 27 : Hello Geeks for Geeks

[INFO] : variable\_length.c : 28 : 10 + 20 = 30

[narendra@/media/partition/GFG]$

Another example:

**#include <string.h>**

**#define print(x, ... ) \**

**char** a[100];\

strcpy(a,**#\_\_VA\_ARGS\_\_);\**

printf("%s %s",**#x,a);**

**int** main(**void**) {

*// magic...*

print(Arpit, "Jain Enjoy The World","Hello World")

**return** 0;

}

Output:

Arpit "Jain Enjoy The World","Hello World"

# Multiline macros in C

In this article, we will discuss how to write a multi-line macro. We can write multi-line macro same like function, but each statement ends with “\”. Let us see with example. Below is simple macro, which accepts input number from user, and prints whether entered number is even or odd.

#include <stdio.h>

#define MACRO(num, str) {\

            printf("%d", num);\

            printf(" is");\

            printf(" %s number", str);\

            printf("\n");\

           }

int main(void)

{

    int num;

    printf("Enter a number: ");

    scanf("%d", &num);

    if (num & 1)

        MACRO(num, "Odd");

    else

        MACRO(num, "Even");

    return 0;

}

At first look, the code looks OK, but when we try to compile this code, it gives compilation error.

[narendra@/media/partition/GFG]$ make macro

cc macro.c -o macro

macro.c: In function ‘main’:

macro.c:19:2: error: ‘else’ without a previous ‘if’

make: \*\*\* [macro] Error 1

[narendra@/media/partition/GFG]$

Let us see what mistake we did while writing macro. We have enclosed macro in curly braces. According to C-language rule, each C-statement should end with semicolon. That’s why we have ended MACRO with semicolon. Here is a mistake. Let us see how compile expands this macro.

if (num & 1)

{

-------------------------

---- Macro expansion ----

-------------------------

}; /\* Semicolon at the end of MACRO, and here is ERROR \*/

else

{

-------------------------

---- Macro expansion ----

-------------------------

};

We have ended macro with semicolon. When compiler expands macro, it puts semicolon after “if” statement. Because of semicolon between “if and else statement” compiler gives compilation error. Above program will work fine, if we ignore “else” part.

To overcome this limitation, we can enclose our macro in “do-while(0)” statement. Our modified macro will look like this.

#include <stdio.h>

#define MACRO(num, str) do {\

            printf("%d", num);\

            printf(" is");\

            printf(" %s number", str);\

            printf("\n");\

           } while(0)

int main(void)

{

    int num;

    printf("Enter a number: ");

    scanf("%d", &num);

    if (num & 1)

        MACRO(num, "Odd");

    else

        MACRO(num, "Even");

    return 0;

}

Compile and run above code, now this code will work fine.

[narendra@/media/partition/GFG]$ make macro

cc macro.c -o macro

[narendra@/media/partition/GFG]$ ./macro

Enter a number: 9

9 is Odd number

[narendra@/media/partition/GFG]$ ./macro

Enter a number: 10

10 is Even number

[narendra@/media/partition/GFG]$

We have enclosed macro in “do – while(0)” loop and at the end of while, we have put condition as “while(0)”, that’s why this loop will execute only one time.

Similarly, instead of “do – while(0)” loop we can enclose multi-line macro in parenthesis. We can achieve the same result by using this trick. Let us see example.

|  |
| --- |
| #include <stdio.h>    #define MACRO(num, str) ({\              printf("%d", num);\              printf(" is");\              printf(" %s number", str);\              printf("\n");\             })    int main(void)  {      int num;        printf("Enter a number: ");      scanf("%d", &num);        if (num & 1)          MACRO(num, "Odd");      else          MACRO(num, "Even");        return 0;  } |

[narendra@/media/partition/GFG]$ make macro

cc macro.c -o macro

[narendra@/media/partition/GFG]$ ./macro

Enter a number: 10

10 is Even number

[narendra@/media/partition/GFG]$ ./macro

Enter a number: 15

15 is Odd number

[narendra@/media/partition/GFG]$

# The OFFSETOF() macro

We know that the elements in a structure will be stored in sequential order of their declaration.

How to extract the displacement of an element in a structure? We can make use of [offsetof](http://en.wikipedia.org/wiki/Offsetof" \t "_blank) macro.

Usually we call structure and union types (or *classes with trivial constructors*) as *plain old data* (POD) types, which will be used to *aggregate other data types*. The following non-standard macro can be used to get the displacement of an element in bytes from the base address of the structure variable.

**#define OFFSETOF(TYPE, ELEMENT) ((size\_t)&(((TYPE \*)0)->ELEMENT))**

Zero is casted to type of structure and required element’s address is accessed, which is casted to *size\_t*. As per standard *size\_t* is of type *unsigned int*. The overall expression results in the number of bytes after which the ELEMENT being placed in the structure.

For example, the following code returns 16 bytes (padding is considered on 32 bit machine) as displacement of the character variable *c* in the structure Pod.

|  |
| --- |
| #include <stdio.h>    #define OFFSETOF(TYPE, ELEMENT) ((size\_t)&(((TYPE \*)0)->ELEMENT))    typedef struct PodTag  {     int     i;     double  d;     char    c;  } PodType;    int main()  {     printf("%d", OFFSETOF(PodType, c) );       getchar();     return 0;  } |

In the above code, the following expression will return the displacement of element *c* in the structure*PodType*.

**OFFSETOF(PodType, c);**

After preprocessing stage the above macro expands to

|  |
| --- |
| ((size\_t)&(((PodType \*)0)->c)) |

Since we are considering 0 as address of the structure variable, c will be placed after 16 bytes of its base address i.e. 0x00 + 0x10. Applying & on the structure element (in this case it is c) returns the address of the element which is 0x10. Casting the address to *unsigned int* (size\_t) results in number of bytes the element is placed in the structure.

**Note:** We may consider the address operator & is redundant. Without address operator in macro, the code de-references the element of structure placed at NULL address. It causes an access violation exception (segmentation fault) at runtime.

*Note that there are other ways to implement offsetof macro according to compiler behavior. The ultimate goal is to extract displacement of the element.****We will see practical usage of offsetof macro in liked lists to connect similar objects (for example thread pool) in another article.***

# Diffference between #define and const in C?