**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?

**#define** is a [preprocessor directive](http://www.geeksforgeeks.org/interesting-facts-preprocessors-c/). Things defined by #define are replaced by the preprocessor before compilation begins.

**const** variables are actual variables like other normal variable.

The big advantage of const over #define is **type checking**. We can also have pointers to const varaibles, we can pass them around, typecast them and any other thing that can be done with a normal variable. One disadvantage that one could think of is extra space for variable which is immaterial due to optimizations done by compilers.

In general const is a better option if we have a choice. There are situations when #define cannot be replaced by const. For example, #define can take parameters (See [this](http://www.geeksforgeeks.org/interesting-facts-preprocessors-c/)for example). #define can also be used to replace some text in a program with another text.

Give a = 12 and b = 36 write a C function/macro that returns 3612 without using arithmetic, strings and predefined functions.

Below is one solution that uses String [Token-Pasting Operator](http://www.geeksforgeeks.org/interesting-facts-preprocessors-c/) (##) of C macros. For example, the expression “a##b” prints concatenation of ‘a’ and ‘b’.

Below is a working C code.

|  |
| --- |
| #include <stdio.h>  #define merge(a, b) b##a  int main(void)  {      printf("%d ", merge(12, 36));      return 0;  } |

Output:

3612

# Interesting Facts about Macros and Preprocessors in C

In a C program, all lines that start with **#** are processed by preprocessor which is a special program invoked by the compiler. In a very basic term, preprocessor takes a C program and produces another C program without any **#**.

Following are some interesting facts about preprocessors in C.

1. When we use ***include***directive, the contents of included header file (after preprocessing) are copied to the current file.  
   Angular brackets **<** and **>** instruct the preprocessor to look in the standard folder where all header files are held.  Double quotes **“**and” instruct the preprocessor to look into the current folder and if the file is not present in current folder, then in standard folder of all header files.
2. When we use***define***for a constant, the preprocessor produces a C program where the defined constant is searched and matching tokens are replaced with the given expression. For example in the following program *max*is defined as 100.

|  |
| --- |
| #include<stdio.h>  #define max 100  int main()  {      printf("max is %d", max);      return 0;  }  // Output: max is 100  // Note that the max inside "" is not replaced |

1. The macros can take function like arguments, the arguments are not checked for data type. For example, the following macro INCREMENT(x) can be used for x of any data type.

|  |
| --- |
| #include <stdio.h>  #define INCREMENT(x) ++x  int main()  {      char \*ptr = "GeeksQuiz";      int x = 10;      printf("%s  ", INCREMENT(ptr));      printf("%d", INCREMENT(x));      return 0;  }  // Output: eeksQuiz 11 |

1. The macro arguments are not evaluated before macro expansion. For example consider the following program

|  |
| --- |
| #include <stdio.h>  #define MULTIPLY(a, b) a\*b  int main()  {      // The macro is expended as 2 + 3 \* 3 + 5, not as 5\*8      printf("%d", MULTIPLY(2+3, 3+5));      return 0;  }  // Output: 16 |

1. The tokens passed to macros can be concatenated using operator **##** called **Token-Pasting** operator.

|  |
| --- |
| #include <stdio.h>  #define merge(a, b) a##b  int main()  {      printf("%d ", merge(12, 34));  }  // Output: 1234 |

1. A token passed to macro can be converted to a sting literal by using # before it.

|  |
| --- |
| #include <stdio.h>  #define get(a) #a  int main()  {      // GeeksQuiz is changed to "GeeksQuiz"      printf("%s", get(GeeksQuiz));  }  // Output: GeeksQuiz |

1. The macros can be written in multiple lines using ‘\’. The last line doesn’t need to have ‘\’.

|  |
| --- |
| #include <stdio.h>  #define PRINT(i, limit) while (i < limit) \                          { \                              printf("GeeksQuiz "); \                              i++; \                          }  int main()  {      int i = 0;      PRINT(i, 3);      return 0;  }  // Output: GeeksQuiz  GeeksQuiz  GeeksQuiz |

1. The macros with arguments should be avoided as they cause problems sometimes. And Inline functions should be preferred as there is type checking parameter evaluation in inline functions. From [C99](http://en.wikipedia.org/wiki/C99) onward, inline functions are supported by C language also.  
   For example consider the following program. From first look the output seems to be 1, but it produces 36 as output.

|  |  |
| --- | --- |
| #define square(x) x\*x  int main()  {    int x = 36/square(6); // Expended as 36/6\*6    printf("%d", x);    return 0;  }  // Output: 36  If we use inline functions, we get the expected output. Also the program given in point 4 above can be corrected using inline functions.   |  | | --- | | inline int square(int x) { return x\*x; }  int main()  {    int x = 36/square(6);    printf("%d", x);    return 0;  }  // Output: 1 | |

1. Preprocessors also support if-else directives which are typically used for conditional compilation.

|  |
| --- |
| int main()  {  #if VERBOSE >= 2    printf("Trace Message");  #endif  } |

1. A header file may be included more than one time directly or indirectly, this leads to problems of re-declaration of same variables/functions. To avoid this problem, directives like ***defined***, ***ifdef***and ***ifndef*** are used.
2. There are some standard macros which can be used to print program file (\_\_FILE\_\_), Date of compilation (\_\_DATE\_\_), Time of compilation (\_\_TIME\_\_) and Line Number in C code (\_\_LINE\_\_)

|  |
| --- |
| #include <stdio.h>    int main()  {     printf("Current File :%s\n", \_\_FILE\_\_ );     printf("Current Date :%s\n", \_\_DATE\_\_ );     printf("Current Time :%s\n", \_\_TIME\_\_ );     printf("Line Number :%d\n", \_\_LINE\_\_ );     return 0;  }    /\* Output:  Current File :C:\Users\GfG\Downloads\deleteBST.c  Current Date :Feb 15 2014  Current Time :07:04:25  Line Number :8 \*/ |

# What’s difference between header files “stdio.h” and “stdlib.h” ?

These are two important header files used in C programming. While “<stdio.h>” is header file for **St**andar**dI**nput **O**utput, “<stdlib.h>” is header file for **St**andar**d** **Lib**rary. One easy way to differentiate these two header files is that “<stdio.h>” contains declaration of *printf()* and *scanf()* while “<stdlib.h>” contains declaration of *malloc()*and *free()*. In that sense, the main difference in these two header files can considered that, while “<stdio.h>” contains header information for ‘File related Input/Output’ functions, “<stdlib.h>” contains header information for ‘Memory Allocation/Freeing’ functions.

Wait a minute, you said “<stdio.h>” is for file related IO but *printf()* and *scanf()* don’t deal with files… or are they? As  a basic principle, in C (due to its association with UNIX history), keyboard and display are also treated as ‘files’! In fact keyboard input is the default *stdin* file stream while display output is the default*stdout* file stream. Also, please note that, though “<stdlib.h>” contains declaration of other types of functions as well that aren’t related to memory such as *atoi()*, *exit()*, *rand()* etc. yet for our purpose and simplicity, we can remember *malloc()* and *free()* for “<stdlib.h>”.

It should be noted that a header file can contain not only function declaration but definition of constants and variables as well. Even macros and definition of new data types can also be added in a header file.