0x0D. C - Preprocessor

C

By: Julien Barbier & Johan Euphrosine, Software Engineer at Google

Weight: 1

Project will start Oct 10, 2022 6:00 AM, must end by Oct 11, 2022 6:00 AM

was released at Oct 10, 2022 12:00 PM

An auto review will be launched at the deadline

Resources

Read or watch:

- Understanding C program Compilation Process (/rltoken/X0ithSsqlz_D0c8V8uA1HQ)
- Object-like Macros (/rltoken/kaqlw352MSJ8xoi1xU09ZA)
- Macro Arguments (/rltoken/wcQZzunlgjepxExZFc2ORQ)
- Pre Processor Directives in C (/rltoken/S4zfCHzg82fUAxdt8_SaZQ)
- The C Preprocessor (/rltoken/G33GiOIZofiIN4Tx9_acbQ)
- Standard Predefined Macros (/rltoken/0OYhpL2cJfsIMBWhTuZsAA)
- include guard (/rltoken/oF2vglZNePdU965jCEZLHA)
- Common Predefined Macros (/rltoken/ROI5xAMKX-JpenEqmf7FnQ)

Learning Objectives

At the end of this project, you are expected to be able to explain to anyone (/rltoken/ipbpW8pLm91jdr3YD-AENg), **without the help of Google**:

General

- What are macros and how to use them
- What are the most common predefined macros
- How to include guard your header files

Copyright - Plagiarism

- You are tasked to come up with solutions for the tasks below yourself to meet with the above learning objectives.

- You will not be able to meet the objectives of this or any following project by copying and pasting someone else's work.
- You are not allowed to publish any content of this project.
- Any form of plagiarism is strictly forbidden and will result in removal from the program.

Requirements

General

- Allowed editors: `vi`, `vim`, `emacs`
- All your files will be compiled on Ubuntu 20.04 LTS using `gcc`, using the options `-Wall -Werror -Wextra -pedantic -std=gnu89`
- All your files should end with a new line
- A `README.md` file, at the root of the folder of the project is mandatory
- Your code should use the `Betty` style. It will be checked using betty-style.pl (https://github.com/holbertonschool/Betty/blob/master/betty-style.pl) and betty-doc.pl (https://github.com/holbertonschool/Betty/blob/master/betty-doc.pl)
- You are not allowed to use global variables
- No more than 5 functions per file
- The only C standard library functions allowed are `malloc`, `free` and `exit`. Any use of functions like `printf`, `puts`, `calloc`, `realloc` etc... is forbidden
- You are allowed to use _putchar (https://github.com/holbertonschool/_putchar.c/blob/master/_putchar.c)
- You don't have to push `_putchar.c`, we will use our file. If you do it won't be taken into account
- In the following examples, the `main.c` files are shown as examples. You can use them to test your functions, but you don't have to push them to your repo (if you do we won't take them into account). We will use our own `main.c` files at compilation. Our `main.c` files might be different from the one shown in the examples
- Don't forget to push your header file
- All your header files should be include guarded

Quiz questions

Great! You've completed the quiz successfully! Keep going! (Hide quiz)

Question #0

What are the steps of compilation?

- 1. compiler 2. preprocessor 3. assembler 4. linker
- 1. preprocessor 2.compiler 3. assembler 4. linker
- 1. preprocessor 2.compiler 3. linker 4. assembler

Question #1

The preprocessor generates assembly code

(/)

- True
- False

Question #2

The preprocessor generates object code

- True
- False

Question #3

The preprocessor links our code with libraries.

- True
- False

Question #4

This portion of code is actually using the library stdlib.

```
#include <stdlib.h>
```

- True
- False

Question #5

The preprocessor removes all comments

- True
- False

Question #6

What is the `gcc` option that runs only the preprocessor?

- -a
- -P
- -p

- -pedantic
- -E
- -cisfun
- -preprocessor

Question #7

`NULL` is a macro

- True
- False

Question #8

What will be the last 5 lines of the output of the command `gcc -E` on this code?

```
#include <stdlib.h>

int main(void)
{
    NULL;
    return (EXIT_SUCCESS);
}
```

-
```
int main(void)
{
 0;
 return (0);
}
```

-
```
int main()
{
 0;
 return (0);
}
```

-
```
int main(void)
{
 ((void *)0);
 return (0);
}
```

(/)

```
int main(void)
{
 '\0';
 return (0);
}
```

Question #9

This code will try to allocate 1024 bytes in the heap:

```
#define BUFFER_SIZE 1024
malloc(BUFFER_SIZE)
```

- True
- False

Question #10

What does the macro `TABLESIZE` expand to?

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
#undef BUFSIZE
#define BUFSIZE 37
```

- 1020
- 37
- nothing

Question #11

This is the correct way to define the macro `SUB` :

```
#define SUB(a, b) a - b
```

- Yes
- No, it should be written this way:

```
#define SUB(a, b) (a - b)
```

- No, it should be written this way:

```
#define SUB(a, b) (a) - (b)
```

- No, it should be written this way:

(/)

```
#define SUB(a, b) ((a) - (b))
```

Question #12

Why should we use include guards in our header files?

- Because we said so, and we should never ask why.
- To avoid the problem of double inclusion when dealing with the include directive.

Question #13

The macro `__FILE__` expands to the name of the current input file, in the form of a C string constant.

- True
- False

Question #14

What will be the output of this program? (on a standard 64 bits, Linux machine)

```
#include <stdio.h>
#include <stdlib.h>

#define int char

int main(void)
{
    int i;

    i = 5;
    printf ("sizeof(i) = %lu", sizeof(i));
    return (EXIT_SUCCESS);
}
```

- Segmentation Fault
- It does not compile
- sizeof(i) = 8
- sizeof(i) = 5
- sizeof(i) = 4
- sizeof(i) = 1

Tasks

0. Object-like Macro

mandatory

Create a header file that defines a macro named `SIZE` as an abbreviation for the token `1024`.

```
julien@ubuntu:~/0x0c. macro, structures$ cat 0-main.c
#include "0-object_like_macro.h"
#include "0-object_like_macro.h"
#include <stdio.h>

/**
 * main - check the code
 *
 * Return: Always 0.
 */
int main(void)
{
    int s;

    s = 98 + SIZE;
    printf("%d\n", s);
    return (0);
}
julien@ubuntu:~/0x0c. macro, structures$ gcc -Wall -pedantic -Werror -Wextra -std=gn
u89 0-main.c -o a
julien@ubuntu:~/0x0c. macro, structures$ ./a
1122
julien@ubuntu:~/0x0c. macro, structures$
```

Repo:

- GitHub repository: `alx-low_level_programming`
- Directory: `0x0D-preprocessor`
- File: `0-object_like_macro.h`

Done! | Help | Check your code

1. Pi

mandatory

Create a header file that defines a macro named `PI` as an abbreviation for the token `3.14159265359`.

```
julien@ubuntu:~/0x0c. macro, structures$ cat 1-main.c
#include "1-pi.h"
#include "1-pi.h"
#include <stdio.h>

/**
 * main - check the code
 *
 * Return: Always 0.
 */
int main(void)
{
    float a;
    float r;

    r = 98;
    a = PI * r * r;
    printf("%.3f\n", a);
    return (0);
}
julien@ubuntu:~/0x0c. macro, structures$ gcc -Wall -pedantic -Werror -Wextra -std=gn
u89 1-main.c -o b
julien@ubuntu:~/0x0c. macro, structures$ ./b
30171.855
julien@ubuntu:~/0x0c. macro, structures$
```

Repo:

- GitHub repository: `alx-low_level_programming`
- Directory: `0x0D-preprocessor`
- File: `1-pi.h`

Done! | Help | Check your code

2. File name

mandatory

Write a program that prints the name of the file it was compiled from, followed by a new line.

- You are allowed to use the standard library

```
julien@ubuntu:~/0x0c. macro, structures$ gcc -Wall -pedantic -Werror -Wextra -std=gn
u89 2-main.c -o c
julien@ubuntu:~/0x0c. macro, structures$ ./c
2-main.c
julien@ubuntu:~/0x0c. macro, structures$ cp 2-main.c 02-main.c
julien@ubuntu:~/0x0c. macro, structures$ gcc -Wall -pedantic -Werror -Wextra -std=gn
u89 02-main.c -o cc
julien@ubuntu:~/0x0c. macro, structures$ ./cc
02-main.c
julien@ubuntu:~/0x0c. macro, structures$
```

Repo:

- GitHub repository: `alx-low_level_programming`
- Directory: `0x0D-preprocessor`
- File: `2-main.c`

Done! | Help | Check your code | >_ Get a sandbox

3. Function-like macro

mandatory

Write a function-like macro `ABS(x)` that computes the absolute value of a number `x`.

```
julien@ubuntu:~/0x0c. macro, structures$ cat 3-main.c
#include <stdio.h>
#include "3-function_like_macro.h"
#include "3-function_like_macro.h"

/**
 * main - check the code
 *
 * Return: Always 0.
 */
int main(void)
{
    int i;
    int j;

    i = ABS(-98) * 10;
    j = ABS(98) * 10;
    printf("%d, %d\n", i, j);
    return (0);
}
julien@ubuntu:~/0x0c. macro, structures$ gcc -Wall -pedantic -Werror -Wextra -std=gn
u89 3-main.c -o d
julien@ubuntu:~/0x0c. macro, structures$ ./d
980, 980
julien@ubuntu:~/0x0c. macro, structures$
```

Repo:

- GitHub repository: `alx-low_level_programming`
- Directory: `0x0D-preprocessor`
- File: `3-function_like_macro.h`

Done! | Help | Check your code

4. SUM

mandatory

Write a function-like macro `SUM(x, y)` that computes the sum of the numbers `x` and `y`.

```
julien@ubuntu:~/0x0c. macro, structures$ cat 4-main.c
#include <stdio.h>
#include "4-sum.h"
#include "4-sum.h"

/**
 * main - check the code
 *
 * Return: Always 0.
 */
int main(void)
{
    int s;

    s = SUM(98, 1024);
    printf("%d\n", s);
    return (0);
}
julien@ubuntu:~/0x0c. macro, structures$ gcc -Wall -pedantic -Werror -Wextra -std=gn
u89 4-main.c -o e
julien@ubuntu:~/0x0c. macro, structures$ ./e
1122
julien@ubuntu:~/0x0c. macro, structures$
```

Repo:

- GitHub repository: `alx-low_level_programming`
- Directory: `0x0D-preprocessor`
- File: `4-sum.h`

Done! | Help | Check your code