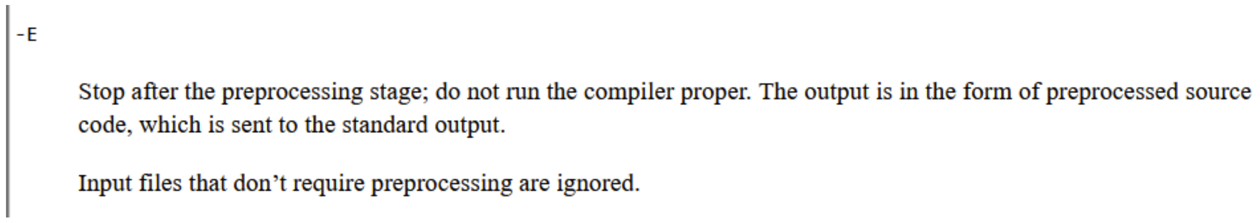
CPE 2600- Systems Programming Dr. Turney
Lab Week #2 Leigh Goetsch
09/16/2024 09/10/2024

Lab 2: Building Programs

Learning Outcomes

- Create C source files
- Link C source files together through header files and include statements
- Use preprocessor directives
- Display and analyze the output at each step in the compilation of a C program

Preprocessing


```
-E
```

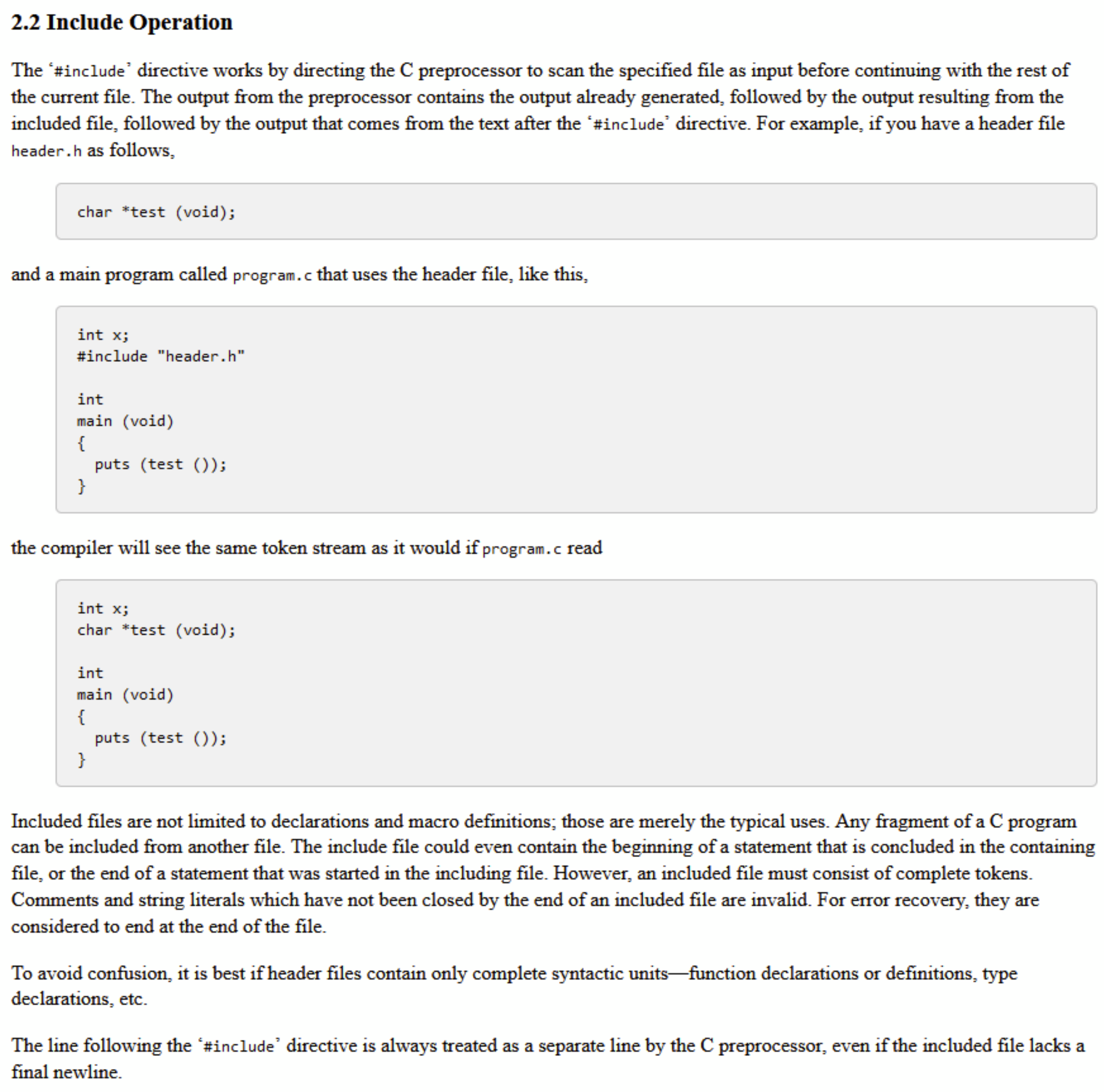
Stop after the preprocessing stage; do not run the compiler proper. The output is in the form of preprocessed source code, which is sent to the standard output.

Input files that don't require preprocessing are ignored.

Figure 1: gcc manual entry for -E flag

The -E flag forces gcc to stop after the preprocessor phase [1]

[1] Determine what command 'flag' parameter that needs to be specified to force gcc to stop after the preprocessor phase.



2.2 Include Operation

The '#include' directive works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the '#include' directive. For example, if you have a header file header.h as follows,

```
char *test (void);
```

and a main program called program.c that uses the header file, like this,

```
int x;
#include "header.h"

int
main (void)
{
  puts (test ());
}
```

the compiler will see the same token stream as it would if program.c read

```
int x;
char *test (void);

int
main (void)
{
  puts (test ());
}
```

Included files are not limited to declarations and macro definitions; those are merely the typical uses. Any fragment of a C program can be included from another file. The include file could even contain the beginning of a statement that is concluded in the containing file, or the end of a statement that was started in the including file. However, an included file must consist of complete tokens. Comments and string literals which have not been closed by the end of an included file are invalid. For error recovery, they are considered to end at the end of the file.

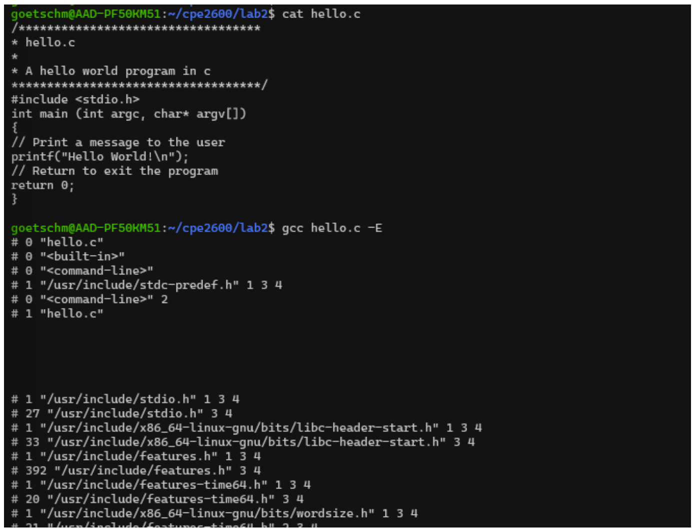
To avoid confusion, it is best if header files contain only complete syntactic units—function declarations or definitions, type declarations, etc.

The line following the '#include' directive is always treated as a separate line by the C preprocessor, even if the included file lacks a final newline.

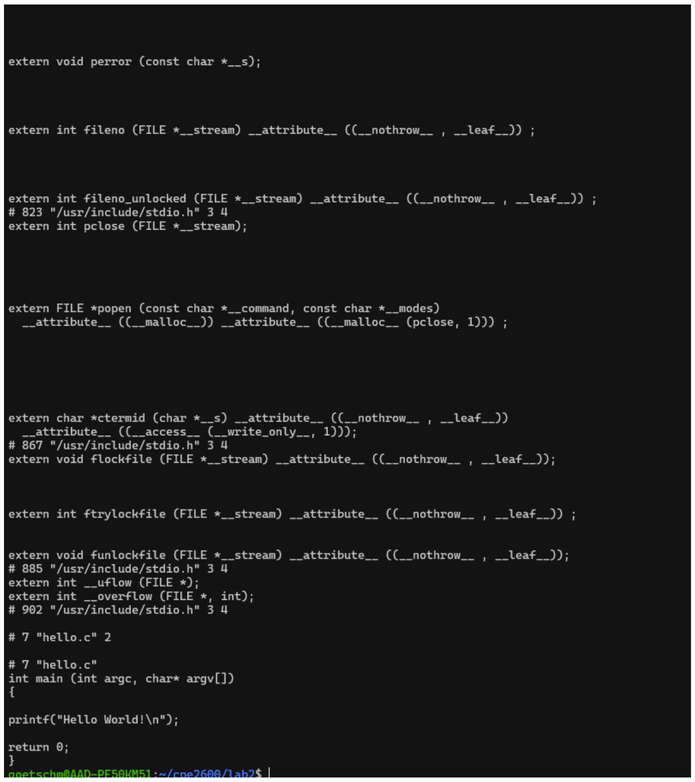
Figure 2: gcc cpp manual entry for the include operation

The #include directive is a way of including other files in your program. [2]

[2] What is the purpose of the #include preprocessor directive?



```
goetschm@AAD-PF50KM51:~/cpe2600/lab2$ cat hello.c
/**********************************
* hello.c
*
* A hello world program in c
***********************************/
#include <stdio.h>
int main (int argc, char* argv[])
{
// Print a message to the user
printf("Hello World!\n");
// Return to exit the program
return 0;
}

goetschm@AAD-PF50KM51:~/cpe2600/lab2$ gcc hello.c -E
# 0 "hello.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "hello.c"





# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
# 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 392 "/usr/include/features.h" 3 4
# 1 "/usr/include/features-time64.h" 1 3 4
# 20 "/usr/include/features-time64.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 21 "/usr/include/features-time64.h" 2 3 4
```

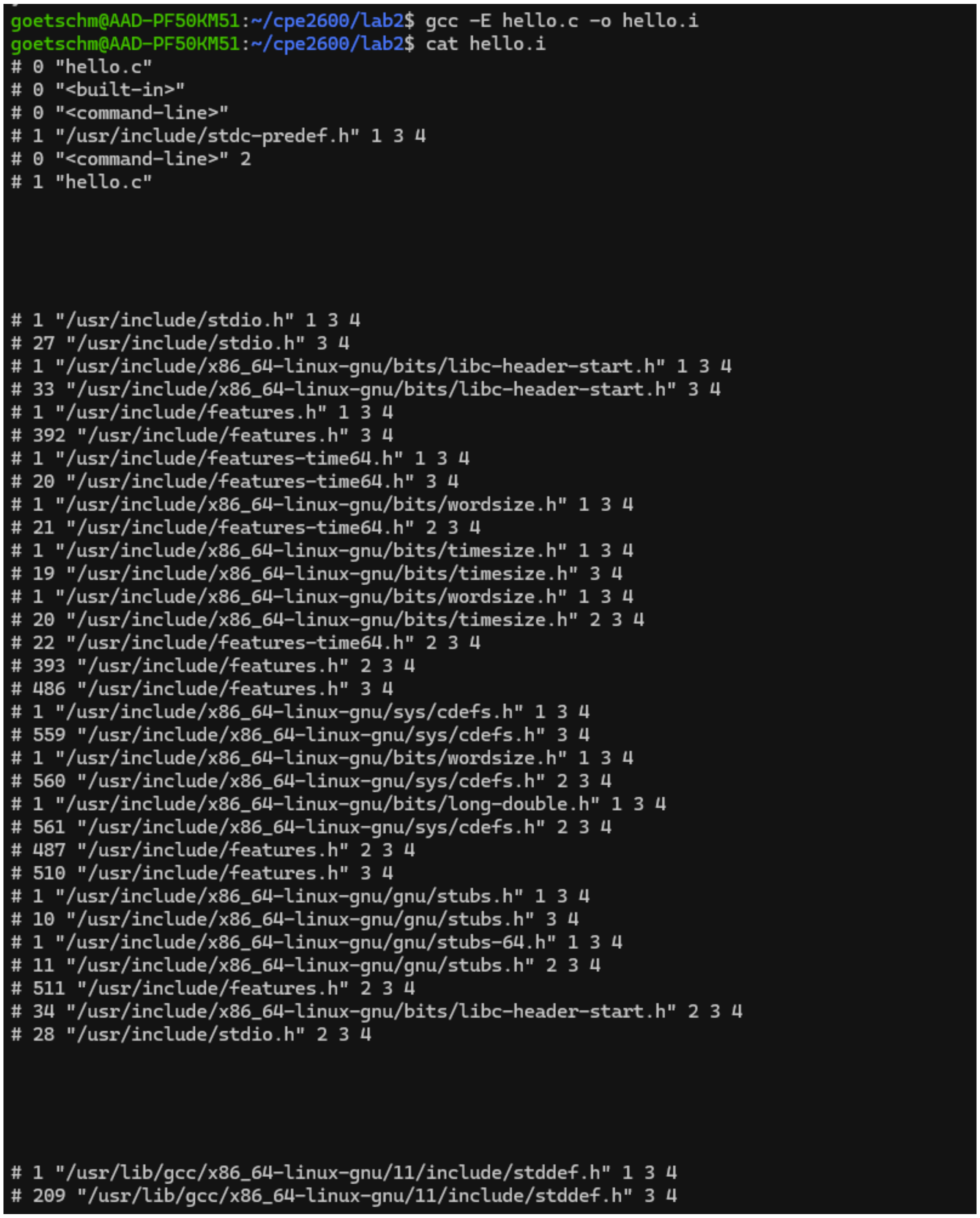


```


extern void perror (const char *__s);




extern int fileno (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;




extern int fileno_unlocked (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;
# 823 "/usr/include/stdio.h" 3 4
extern int pclose (FILE *__stream);





extern FILE *popen (const char *__command, const char *__modes)
  __attribute__ ((__malloc__)) __attribute__ ((__malloc__ (pclose, 1))) ;






extern char *ctermid (char *__s) __attribute__ ((__nothrow__ , __leaf__))
  __attribute__ ((__access__ (__write_only__, 1)));
# 867 "/usr/include/stdio.h" 3 4
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));



extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;


extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
# 885 "/usr/include/stdio.h" 3 4
extern int __uflow (FILE *);
extern int __overflow (FILE *, int);
# 902 "/usr/include/stdio.h" 3 4

# 7 "hello.c" 2

# 7 "hello.c"
int main (int argc, char* argv[])
{

printf("Hello World!\n");

return 0;
}
goetschm@AAD-PF50KM51:~/cpe2600/lab2$
```

Figure 3: preprocessor output for gcc hello.c -E

The output of the preprocessor includes the code from the #include directive. The preprocessor scans the file included as input before continuing with the current file. [3]

[3] How does output of the preprocessor differ from your input source?



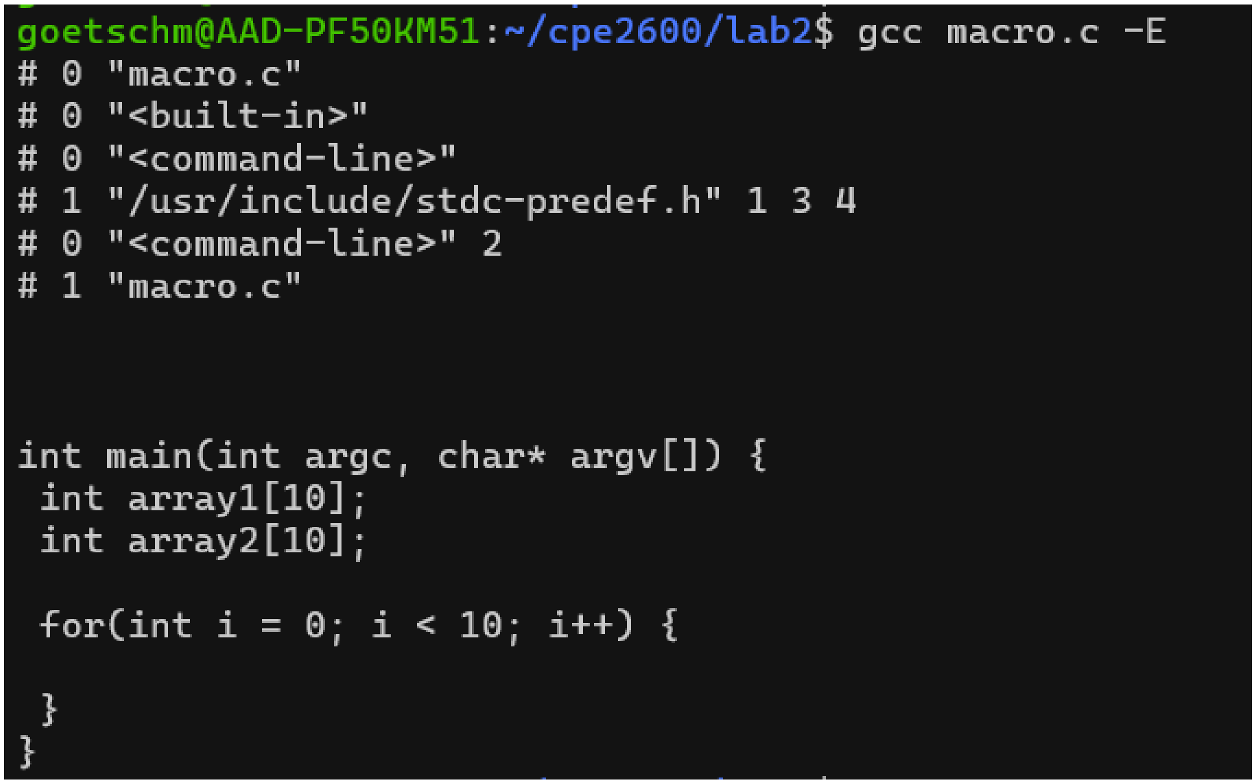
```
goetschm@AAD-PF50KM51:~/cpe2600/lab2$ gcc -E hello.c -o hello.i
goetschm@AAD-PF50KM51:~/cpe2600/lab2$ cat hello.i
# 0 "hello.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "hello.c"





# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
# 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 392 "/usr/include/features.h" 3 4
# 1 "/usr/include/features-time64.h" 1 3 4
# 20 "/usr/include/features-time64.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 21 "/usr/include/features-time64.h" 2 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/timesize.h" 1 3 4
# 19 "/usr/include/x86_64-linux-gnu/bits/timesize.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 20 "/usr/include/x86_64-linux-gnu/bits/timesize.h" 2 3 4
# 22 "/usr/include/features-time64.h" 2 3 4
# 393 "/usr/include/features.h" 2 3 4
# 486 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
# 559 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 560 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/long-double.h" 1 3 4
# 561 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 487 "/usr/include/features.h" 2 3 4
# 510 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 1 3 4
# 10 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs-64.h" 1 3 4
# 11 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 2 3 4
# 511 "/usr/include/features.h" 2 3 4
# 34 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 2 3 4
# 28 "/usr/include/stdio.h" 2 3 4





# 1 "/usr/lib/gcc/x86_64-linux-gnu/11/include/stddef.h" 1 3 4
# 209 "/usr/lib/gcc/x86_64-linux-gnu/11/include/stddef.h" 3 4
```

Figure 4: result of saving the preprocessor result to hello.i

[4]

[4]Write the exact command necessary run the preprocessor on hello.c and output the result to hello.i. Include the screenshot of a portion of this file in your submission.

Macros



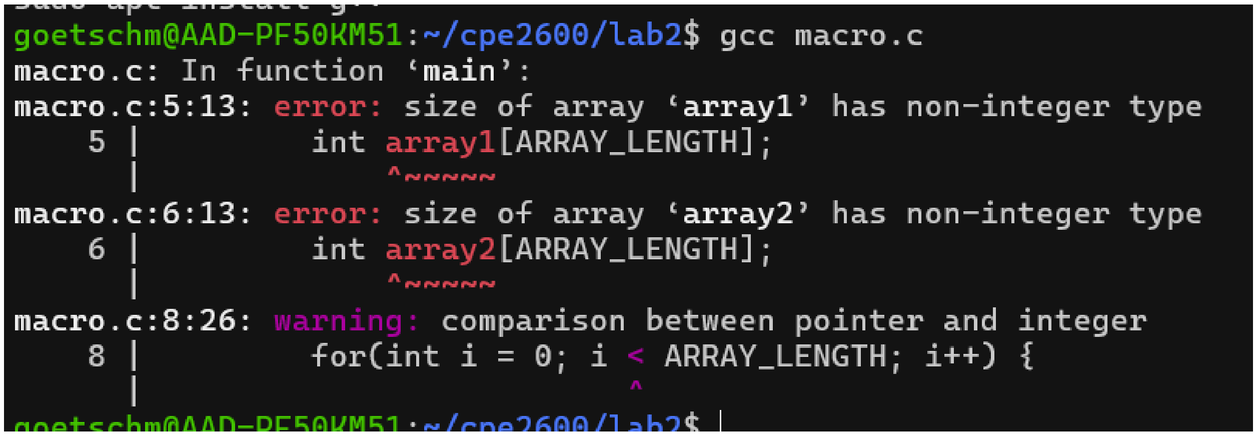
```
goetschm@AAD-PF50KM51:~/cpe2600/lab2$ gcc macro.c -E
# 0 "macro.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "macro.c"



int main(int argc, char* argv[]) {
 int array1[10];
 int array2[10];

 for(int i = 0; i < 10; i++) {

 }
}
```

Figure 5: Included result of preprocessing stage of macro source code

[5] [6] In the output, the constant ARRAY_LENGTH is replaced with the value given. [7]

One issue with this is that you can set ARRAY_LENGTH to anything, including values that are not legal array lengths, such as characters. [8]

[5]Run gcc on this source and force it to stop at the preprocessor output phase.
[6]Save your output and include it with your submission.
[7]What do you observe about the output?
[8]Can you foresee any issues with this mechanism?

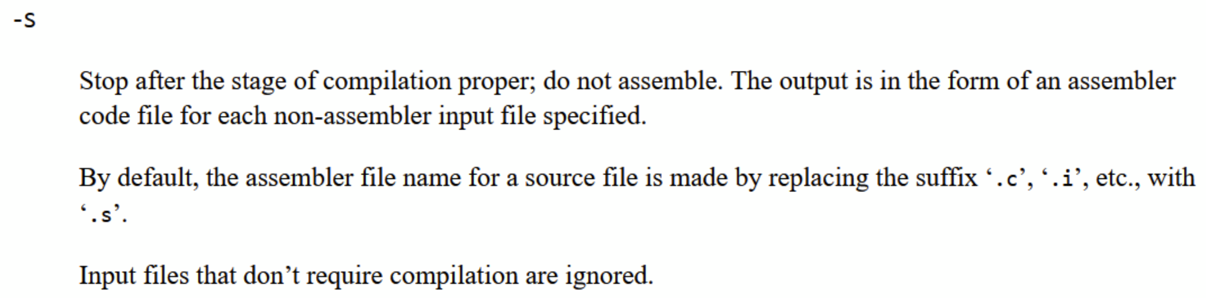


```
goetschm@AAD-PF50KM51:~/cpe2600/lab2$ gcc macro.c
macro.c: In function ‘main’:
macro.c:5:13: error: size of array ‘array1’ has non-integer type
    5 |         int array1[ARRAY_LENGTH];
      |             ^~~~~~
macro.c:6:13: error: size of array ‘array2’ has non-integer type
    6 |         int array2[ARRAY_LENGTH];
      |             ^~~~~~
macro.c:8:26: warning: comparison between pointer and integer
    8 |         for(int i = 0; i < ARRAY_LENGTH; i++) {
      |                          ^
goetschm@AAD-PF50KM51:~/cpe2600/lab2$
```

Figure 6: The errors say that the size is of the wrong type. The error messages are clear.

[9]

Compiling



`-S`

Stop after the stage of compilation proper; do not assemble. The output is in the form of an assembler code file for each non-assembler input file specified.

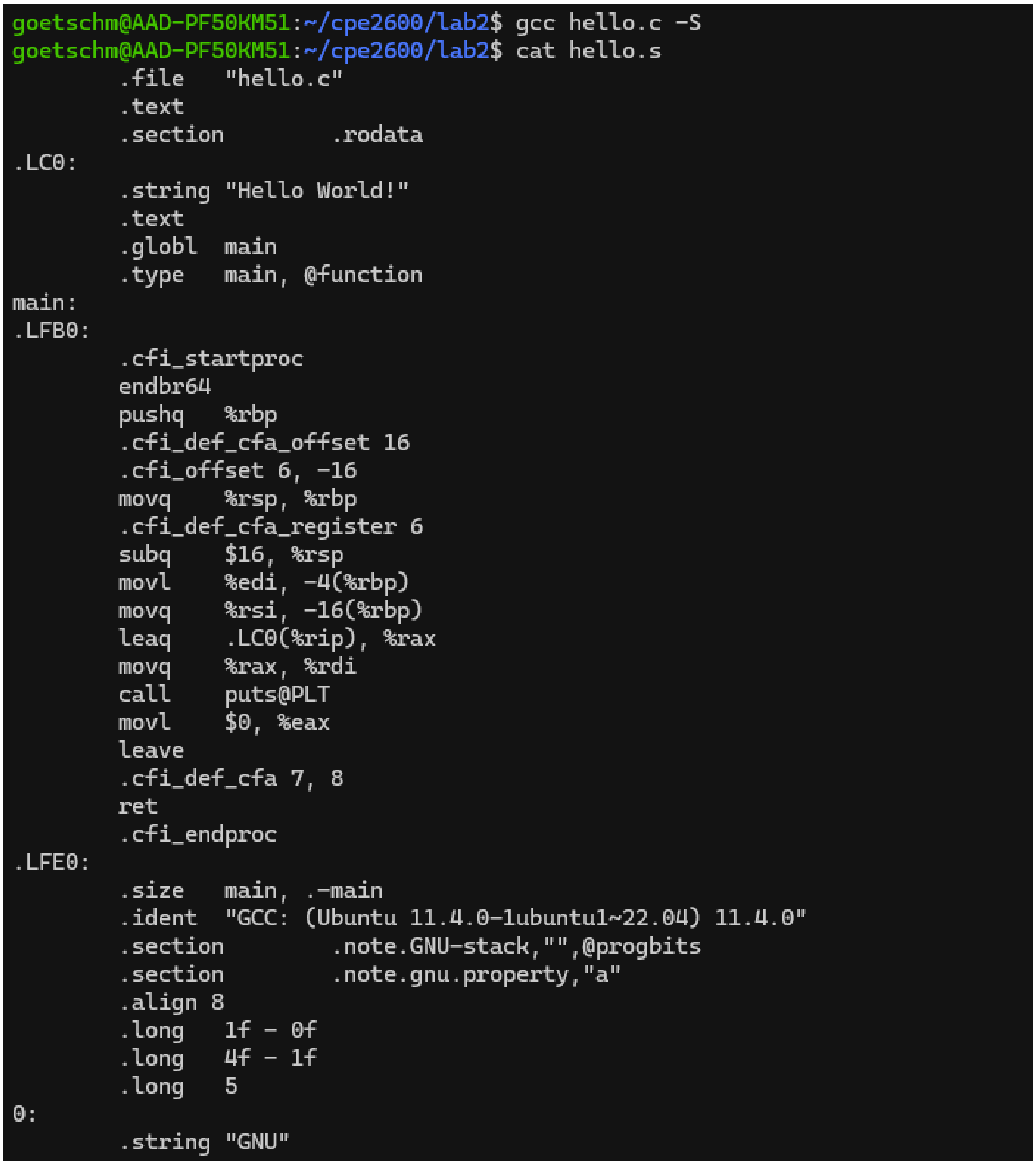
By default, the assembler file name for a source file is made by replacing the suffix ‘`.c`’, ‘`.i`’, etc., with ‘`.s`’.

Input files that don’t require compilation are ignored.

Figure 7: the -S flag stops the compiler after compilation and saves the assembly source files as ".s"

[10]

[9]As an experiment, change ARRAY_LENGTH from 10 to “xyz” (include quotes). Compile and look at the error message(s). Are the error messages clear?
[10]From your research, determine what command ‘flag’ parameter that needs to be specified to force gcc to stop after the compiler phase.



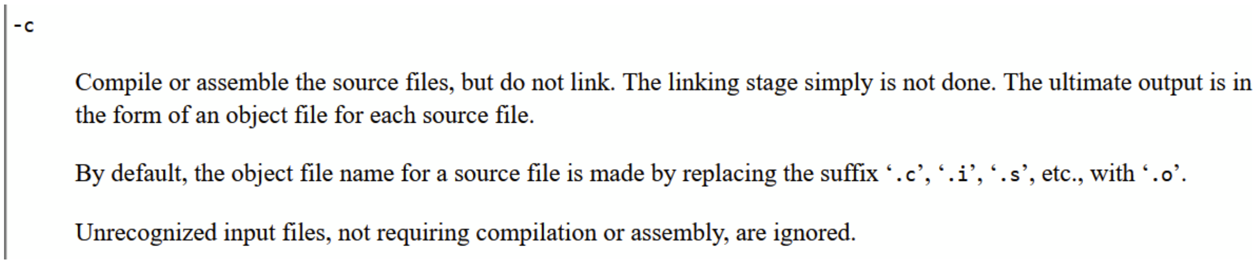
```
goetschm@AAD-PF50KM51:~/cpe2600/lab2$ gcc hello.c -S
goetschm@AAD-PF50KM51:~/cpe2600/lab2$ cat hello.s
        .file   "hello.c"
        .text
        .section        .rodata
.LC0:
        .string "Hello World!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    %edi, -4(%rbp)
        movq    %rsi, -16(%rbp)
        leaq    .LC0(%rip), %rax
        movq    %rax, %rdi
        call    puts@PLT
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0"
        .section        .note.GNU-stack,"",@progbits
        .section        .note.gnu.property,"a"
        .align 8
        .long   1f - 0f
        .long   4f - 1f
        .long   5
0:
        .string "GNU"
```

Figure 8: assembly file for hello.c which is the result of gcc hello.c -S

[11]

[11] Write the exact command necessary run the compiler on hello.c and output the assembly language to hello.s. Include a snip this file in your submission.

Assembling



`-c`

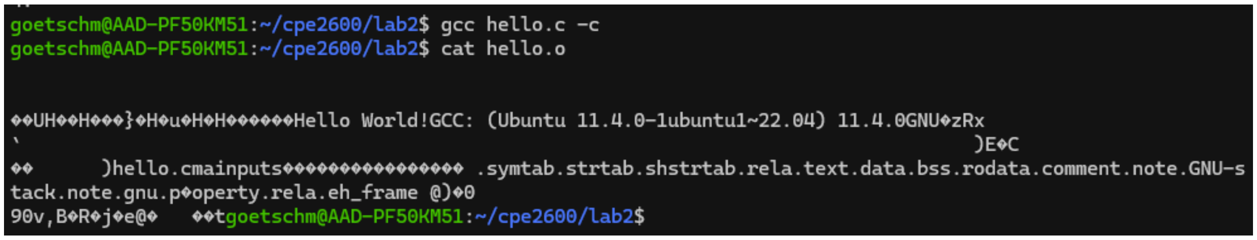
Compile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file.

By default, the object file name for a source file is made by replacing the suffix '`.c`', '`.i`', '`.s`', etc., with '`.o`'.

Unrecognized input files, not requiring compilation or assembly, are ignored.

Figure 9: the -c flag stops the compiler after compilation and assembly and saves the object files as ".o"

[12]

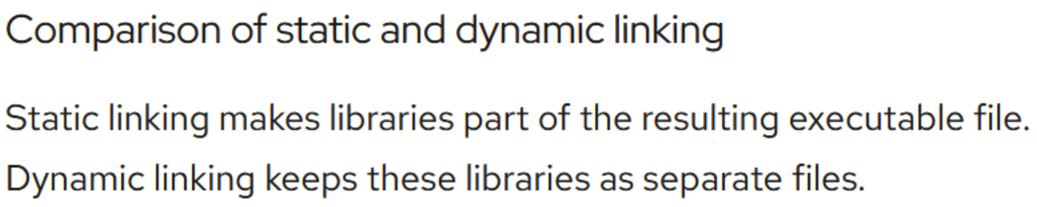


```
goetschm@AAD-PF50KM51:~/cpe2600/lab2$ gcc hello.c -c
goetschm@AAD-PF50KM51:~/cpe2600/lab2$ cat hello.o


��UH��H���}�H�u�H�H������Hello World!GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0GNU�zRx
`                                                                                   )E�C
��      )hello.cmainputs���������������� .symtab.strtab.shstrtab.rela.text.data.bss.rodata.comment.note.GNU-s
tack.note.gnu.p�operty.rela.eh_frame @)�0
90v,B�R�j�e@�   ��tgoetschm@AAD-PF50KM51:~/cpe2600/lab2$
```

Figure 10: object file for hello.c which is the result of gcc hello.c -c

[13]

Linking



Comparison of static and dynamic linking

Static linking makes libraries part of the resulting executable file.

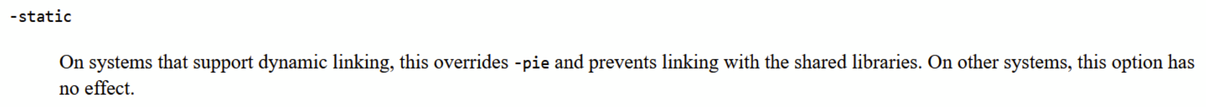
Dynamic linking keeps these libraries as separate files.

Figure 11: Explanation of static and dynamic linking on the red hat docx

[12]Determine what command 'flag' parameter that needs to be specified to force gcc to stop after the assembler phase.

[13]Write the exact command necessary run the gcc on hello.c and output the object file to hello.o. Include a snip this file in your submission.

Static linking makes the library part of the executable, while dynamic linking loads the library into memory separate from the executable. Dynamic linking takes up less space and is useful when multiple programs share the same libraries. [14]

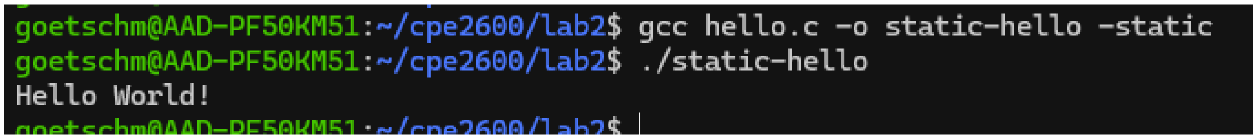


-static

On systems that support dynamic linking, this overrides -pie and prevents linking with the shared libraries. On other systems, this option has no effect.

Figure 12: gcc manual entry for -static

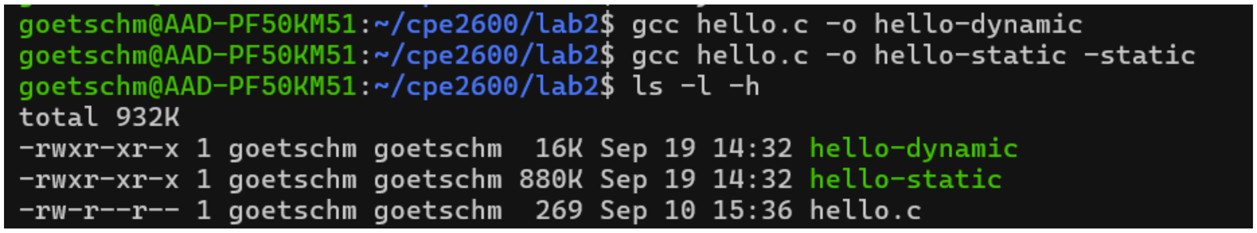
The flag `-static` prevents dynamic linking. [15]



```
goetschm@AAD-PF50KM51:~/cpe2600/lab2$ gcc hello.c -o static-hello -static
goetschm@AAD-PF50KM51:~/cpe2600/lab2$ ./static-hello
Hello World!
goetschm@AAD-PF50KM51:~/cpe2600/lab2$
```

Figure 13: result of compiling hello.c with static cmd

The exact command is `$ gcc hello.c -static`. [16]



```
goetschm@AAD-PF50KM51:~/cpe2600/lab2$ gcc hello.c -o hello-dynamic
goetschm@AAD-PF50KM51:~/cpe2600/lab2$ gcc hello.c -o hello-static -static
goetschm@AAD-PF50KM51:~/cpe2600/lab2$ ls -l -h
total 932K
-rwxr-xr-x 1 goetschm goetschm  16K Sep 19 14:32 hello-dynamic
-rwxr-xr-x 1 goetschm goetschm 880K Sep 19 14:32 hello-static
-rw-r--r-- 1 goetschm goetschm  269 Sep 10 15:36 hello.c
```

Figure 14: The relative sizes of both executables

The relative size of the static executable is around 50 times larger than the size of the dynamically linked executable. [17] The sizes are different because the statically linked executable holds the libraries used by the program. [18]

[14]What is static linking vs dynamic linking?

[15]Determine what 'flag' parameter needs to be specified to force gcc to perform static linking.

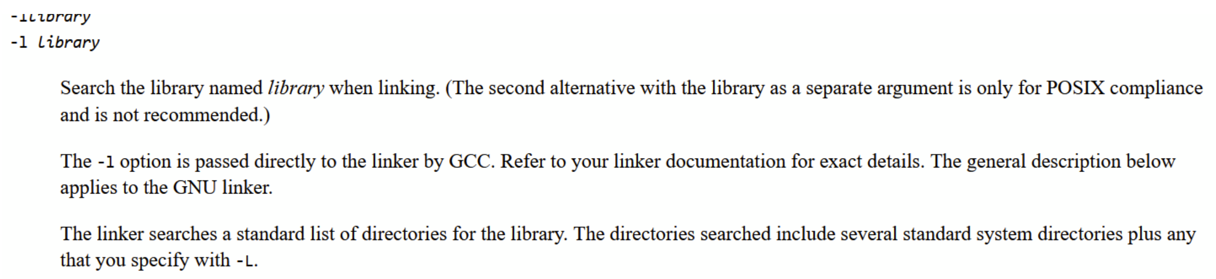
[16]Write the exact command necessary and run the compiler on hello.c (you may wish to explicitly set the output of gcc to something other than a.out).

[17]Compare the file sizes of the statically linked executable to a dynamically linked executable. Include a screenshot showing the relative sizes of both executables.

[18]Why are they different?

Building Programs from Multiple Files

To build a program with multiple files, you need to list all of the files included in the build. [19] To avoid warnings and errors, you should have a header file for any project files with functions. [20] The command `-l` is need for external libraries, like the math library (which you can use `-lm`) need to be included in the build command to properly build the project. [21] This is because it tells the compiler to include the library when linking. [22]



```
-llibrary
-l library
```

Search the library named *library* when linking. (The second alternative with the library as a separate argument is only for POSIX compliance and is not recommended.)

The `-l` option is passed directly to the linker by GCC. Refer to your linker documentation for exact details. The general description below applies to the GNU linker.

The linker searches a standard list of directories for the library. The directories searched include several standard system directories plus any that you specify with `-L`.

Figure 15: gcc entry for -l cmd

This can be found in the gcc compiler docs. [23]

[19]From what you have learned in class and through your research how do you build a program from multiple files?

[20]What additional file is needed in order to avoid warnings or errors?

[21]What additional argument is needed for gcc to properly build the project?

[22]Why?

[23]Where did you find this documented?

Resulting Project Files

Exact command used to build: `$ gcc main.c mymath.c -o mymath -lm -Wall`

6.1.1 main.c

```
/**
* @file main.c
* @brief contians the main function.
*
* Course: CPE2600
* Section: 121
* Assignment: Lab 2
* Name: Leigh Goetsch
*
* to compile: $ gcc main.c mymath.c -lm
*/

#include "mymath.h"
#include <stdio.h>

#define TEST_ARRAY_SIZE 4

int main (){
    // test arrays
    int bitMax[] = {4, 8, 16, 32};
    int bitsNum[] = {7, 8, 15, 16};

    int i;

    // iterate through maxvalue test values
    for (i = 0; i < TEST_ARRAY_SIZE; i++){
        int bits = bitMax[i];
        printf("The␣max␣value␣that␣can␣be␣represented␣by␣%d␣bits␣is␣", bits);
        printf("%ld\n", maxvalue(bits));
    }

    // iterate through numbits test values
    for (i = 0; i < TEST_ARRAY_SIZE; i++){
        int value = bitsNum[i];
        printf("The␣min␣number␣of␣bits␣that␣can␣represent␣%d␣is␣", value);
        printf("%d\n", numbits(value));
    }

    return 0;
}
```

6.1.2 mymath.h

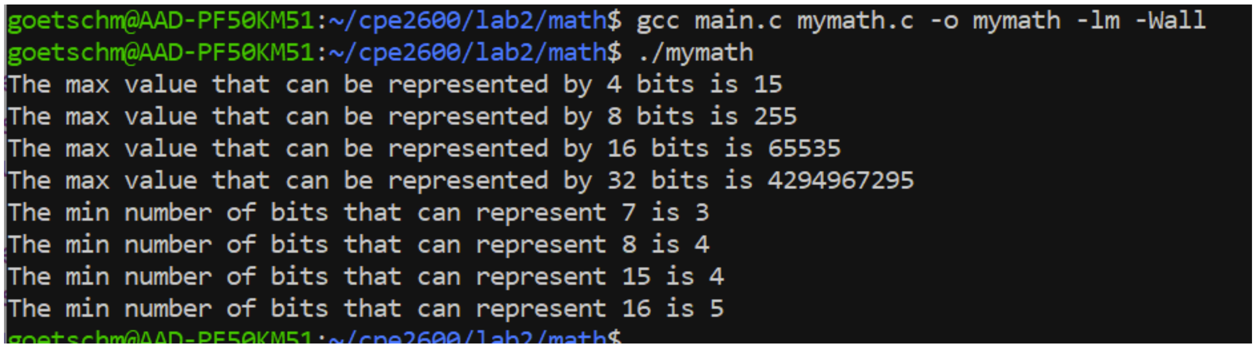
```c
/**
* @file mymath.c
* @brief contains two functions for math operations
*
* Course: CPE2600
* Section: 121
* Assignment: Lab 2
* Name: Leigh Goetsch
*
*/

long int maxvalue(int bits);

int numbits(int value);
```

6.1.3 mymath.c

```
/**
* @file mymath.c
* @brief contains two functions for math operations related to calculating bit ranges
*
* Course: CPE2600
* Section: 121
* Assignment: Lab 2
* Name: Leigh Goetsch
*
*
*/
#include "mymath.h"
#include <math.h>

#define BASE_2 2.0

long int maxvalue(int bits){
   return (long int) pow(BASE_2, (double) bits) - 1;
}

int numbits(int value) {
    return ((int) log2((double) value)) + 1;
}
```

[24]

[24] Include screen caps of all of your files



```
goetschm@AAD-PF50KM51:~/cpe2600/lab2/math$ gcc main.c mymath.c -o mymath -lm -Wall
goetschm@AAD-PF50KM51:~/cpe2600/lab2/math$ ./mymath
The max value that can be represented by 4 bits is 15
The max value that can be represented by 8 bits is 255
The max value that can be represented by 16 bits is 65535
The max value that can be represented by 32 bits is 4294967295
The min number of bits that can represent 7 is 3
The min number of bits that can represent 8 is 4
The min number of bits that can represent 15 is 4
The min number of bits that can represent 16 is 5
goetschm@AAD-PF50KM51:~/cpe2600/lab2/math$
```

Figure 16: result of compiling and running the mymath program

[25]

[25]as well as the output of your program with your submission.

Sources

- GCC CPP Manual
- GCC Manual 14.2.0
- Red Hat docx