

BSM-308 System Programming

Sakarya University-Computer Engineering

<https://web.eecs.utk.edu/~huangj/cs302/notes/Some%20fundamentals.htm>

Content

a) .h files

b) Meaning of the keywords extern and static

c) linking vs compilation

d) how a process is organized in memory (chunk, stack, etc.)

e) makefiles

• <http://web.eecs.utk.edu/~huangj/cs302/notes/Some%20fundamentals.htm>

• <http://web.eecs.utk.edu/~jplank/plank/plank/classes/cs360/360/notes/CStuff-1/lecture.html>

What is a program?

❑ Of course, we all know that when a program is running in an operating system, it is called a <proses>.

❑ The program itself can be in many forms, i.e. machine code, assembly code, C code, C++ or Fortran, Java, ...

❑ Processors ONLY execute **machine code**. Machine code is written in binary machine words consisting of opcodes (Opcode) and operands (Operands).

❑ The opcodes represent different instructions, for example an arithmetic addition. **The operands are** executed by the instruction. A machine code might look like this:

10001000110101001010101101101011

❑ Most sane people don't want to write code in this format. That's why assembly language was developed;

ADD AX, BX

Assembly

❑ Mnemonics in assembly language are one-to-one with machine code, which is **processor dependent**. But you still need an "**assembler**" for conversion!

❑ If you switch from Intel x86 to Motorola 68k, a new program must be written.

❑ Here is a real assembly program running on a Motorola 68k:

❑ By the way, you can see how your C program looks like Assembly if you want to see it, use "cc -S xxx.c" or "gcc -S xxx.c".

try it. These commands will show the build version of your C produces .s files.

```
ORG $1000
N EQU 5
CNT1 DC.B 0
CNT2 DC.B 0
ARRAY DC.B 2,7,1,6,3

                ORG $1500
MAIN LEA ARRAY,A2
        MOVE.B #N,D1
        CLR D6
        CLR D7
        JSR SORT
        STOP #$2700
```

Assembly

- ❑ In the program above, the **addresses are** physical addresses corresponding to individual bytes in your memory banks.
- ❑ If you hard-coded these addresses, you'd better run only one program at a time on your processor!
- ❑ Programming in assembly language can be boring! Unless you are making system calls, it can take 30 lines of code to put a character in your console.
- ❑ Every time you want to reuse your own piece of code you have to deal with it. You build all the structures, Imagine using **someone else's code**.
- ❑ Languages like C/C++ were developed to overcome such problems.

C language

- ❑ Your source code is run through a **compiler and it** translates your C program into binary machine code and creates a module, often called 'object-object'.
- ❑ Inside each object file that the compiler outputs, you basically get a machine code module in a platform-dependent format.
- ❑ However, the important point is that there **is no physical address encoded** there. This makes the module **relocatable** in main memory.
- ❑ In this case the object file xxx.o is written to disk.

C language

- ❑ a separate **linker-linker** program is then called to look in your object file for external references (external ref.), function calls or variables.
- ❑ For example, you called the function `"printf("Hello World\n");"`. Your compiler does not know how this call works because it is provided in the system library.
- ❑ **It is your linker's job** to find a `printf` machine code module from the system library and put it together with `xxx.o`.
- ❑ The result of your linking session is an executable-executable file on disk.

C language

- ❑ After typing a.out on the command line, the **loader-loader** in the operating system reads a.out from disk and writes it to main memory.
- ❑ This determines the physical address of each line in a.out.
- ❑ Then, the operating system finds the entry point of your program (the main function in xxx.c) and starts executing it from there.

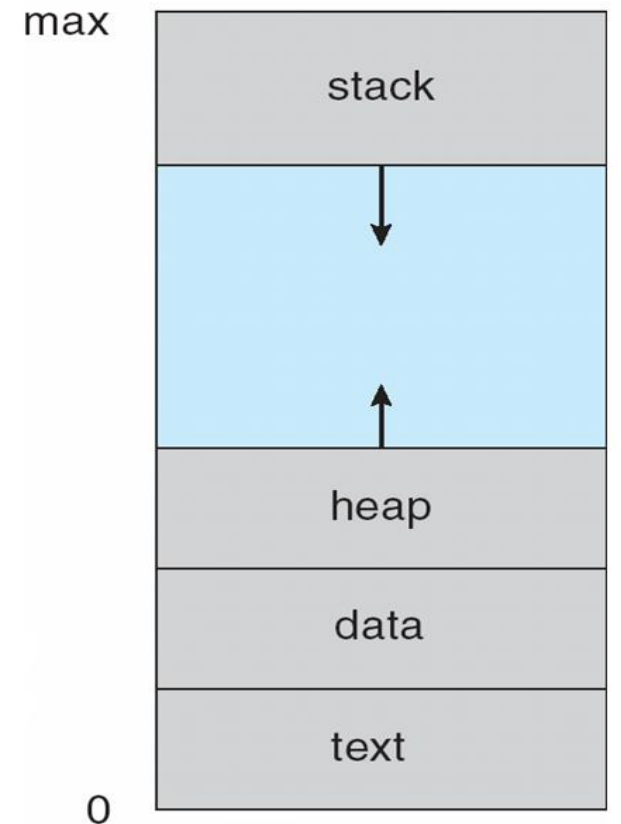
Why do we need a .h file?

- ❑ It takes many people to develop a serious application using **libraries-libraries** in the operating system and sometimes libraries provided by others.
- ❑ Modular development is the norm.
- ❑ Compilers need to do type checks and make sure that your calls to the library and other people's functions are correct.
- ❑ Whether or not you have access to the C source of the function body you are calling, **pasting a function body into your code is a bad idea** (why?).
- ❑ Using .h files, which usually contain **data types, function declarations and macros**, solves this problem.

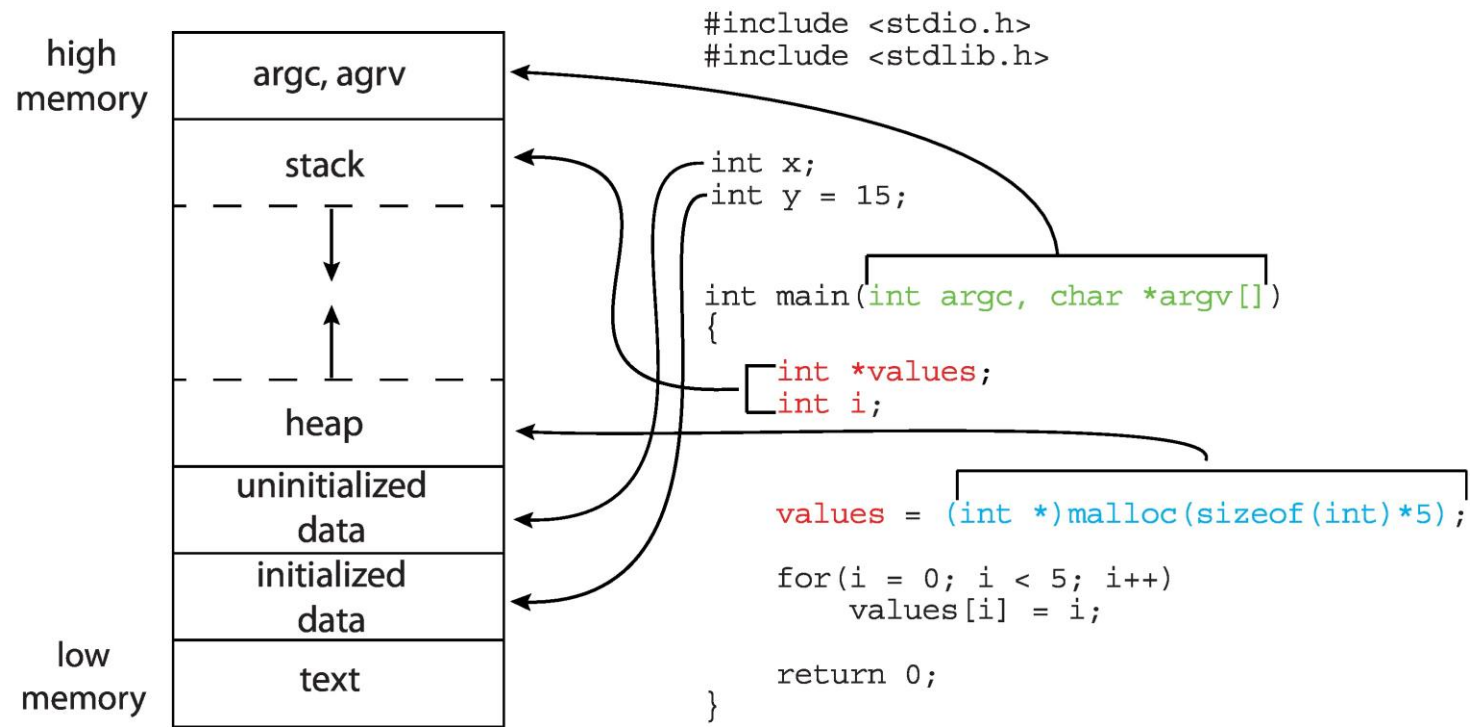
Placement of the process in memory

Sections;

- **program code** (text section),
- **Current activities**, including **current activities**, **program counter** and other recorders.
- The stack contains temporary data.
 - Function parameters, return values, local variables
- The data section contains global variables.
- **Memory heap** - The heap contains memory dynamically allocated at runtime.



Memory Layout of a C Program



Functions and Stack section

1. the OS pushes the current address in main program onto stack
2. pushes all necessary data to recover running the main program, including CPU registers, onto stack
3. pushes all arguments in that function call onto stack one-by-one
4. finds the head address of the text segment of that function and start executing
5. inside the function, allocated more memory space in the stack for local variables
6. if there are malloc() calls, allocate memory from heap space
7. finish computing in function
8. if had dynamic memory allocations, maybe you need to deallocate (well-behaving) ?
9. pop all local variables off stack
10. pop all function arguments off stack
11. recover original running state from stack
12. start executing again from the address just got from stack

Scope and Visibility

- ❑ File scope-File scope;
- ❑ Identifier names with file scope are often called "**global-global**" or "**external-external**".
- ❑ It appears outside any block or parameter list and can be accessed from anywhere after its declaration.

- ❑ Function Scope-Function scope;
- ❑ Defined within a function and Tag names must be unique within a function.

Scope and Visibility

- ❑ Block scope-Block scope;
- ❑ It only appears from the point of the declaration or definition to the end of the block containing the declaration or definition.
- ❑ Its scope is limited to that block and any nested blocks in that block, and ends at the bracket closing the associated block. Such identifiers are sometimes called "**local variables**".
- ❑ Function Prototype-Function-prototype scope;
- ❑ Appears in the list of parameter declarations in a function prototype.

Extern and static

- ❑ A variable declared with **static**. You can explicitly initialize the static variable with a constant expression. If you omit the initializer, the variable is initialized to 0 by default.

```
static int k = 16;  
static int k;
```

Extern and static

- ❑ We can use the **static** specifier with local variables.
- ❑ Normally, whenever we call a function that has a local variable defined in it, the local variable value is refreshed.
- ❑ However, if we define this local variable to be static, each time we call the function, the local variable will retain its last value from the previous function call.
- ❑ As a result, we can only give a value to a static local variable once on the first call of the function.

The example below illustrates external declarations:

```

/*****
                                SOURCE FILE ONE
*****/

extern int i;                  /* Reference to i, defined below */
void next( void );            /* Function prototype          */

void main()
{
    i++;
    printf( "%d\n", i );      /* i equals 4 */
    next();
}

int i = 3;                    /* Definition of i */

void next( void )
{
    i++;
    printf( "%d\n", i );      /* i equals 5 */
    other();
}

/*****
                                SOURCE FILE TWO
*****/

extern int i;                  /* Reference to i in      */
                                /* first source file      */

void other( void )
{
    i++;
    printf( "%d\n", i );      /* i equals 6 */
}

```

Extern and static

```
#include <stdio.h>

void fonk(void);
void fonk_sta(void);

int main(void)
{
    fonk();
    fonk_sta();
    printf("\n");
    fonk();
    fonk_sta();
    return 0;
}
```

```
void fonk(void) {
    int id = 1;
    printf("fonk() id variable value: %d\n", id);
    id = id + 21;
    printf("fonk() id variable value: %d\n", id);
}

void fonk_sta(void) {
    static int id = 1;
    // Only works on the first call to the function.
    printf("fonk_sta() id variable value: %d\n", id);
    id = id + 21;
    printf("fonk_sta() id variable value: %d\n", id);
}
```

Extern and static

```
fonk() id variable value: 1  
fonk() id variable value: 22  
fonk_sta() id variable value: 1  
fonk_sta() id variable value: 22  
fonk() id variable value: 1  
fonk() id variable value: 22  
  fonk_sta() id variable value: 22  
fonk_sta() id variable value: 43
```

Extern and static

- ❑ We can also use the **static** specifier with global variables.
- ❑ When we define a **static** global variable, only the functions in the file in which it is defined can use it.
- ❑ In addition, we can define two global variables with the same name, one normal and one static, in different files of the same program.

Extern and static

```
// trial1.c
#include <stdio.h>
void fonk1(void);
void fonk2(void);
static int gid = 21; // Static global int variable declaration
int main(void) {
    fonk1();
    fonk2();
    return 0; }
void fonk1(void)
{ printf("trial1.c gid variable value: %d\n", gid); }
```

```
// trial2.cpp
#include <iostream>
using namespace std;
int gid = 35; // global int variable declaration
void fonk2(void)
{
    printf("trial2.c gid variable value: %d", gid);
}
```

experiment1.c gid variable value: 21
trial2.c gid variable value: 35

Extern

Using **extern** at the beginning of a variable indicates that the variable is defined in one of the code files in the project and that we want to access that variable from the file in which we use extern.

```
// trial1.c
#include <stdio.h>
void fonk(void);
int gid = 287; // global int variable declaration
int main(void) {
    printf("trial1.c gid variable value: %d\n", gid);
    fonk(); // call to the fonk() function in experiment2.c
    return 0;
}

// trial2.c
extern int gid; // global int variable declaration
void fonk(void)
{ printf("trial2.c gid variable value: %d", gid); }
```

```
experiment1.c gid variable value: 287
experiment2.c gid variable value: 287
```

Lifetime

- ❑ "Lifetime" is the period during the execution of a program in which a variable or function exists.
- ❑ The storage time of the identifier determines its lifetime, either static lifetime (global lifetime) or automatic lifetime (local lifetime).
- ❑ An identifier declared without a static storage class specifier has automatic storage duration if declared within a function or block.
- ❑ **Global lifetime:** All functions have a global lifetime. Therefore, they always exist during program execution.
- ❑ **Local lifetime:** If a local variable has an initializer, the variable is initialized each time it is created (unless declared statically).

Header files

- ❑ As in C++, we add the standard header files with `#include`.
- ❑ We append the file name and the `.h` extension to the lowercase/uppercase signs.

```
#include <iostream>  
using namespace std;
```

- ❑ In C;

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


Data types in C

- ❑ In C there are three types of types that variables can have -
 - ❑ scalars, aggregates, and pointers
 - ❑ Scalar types
 - ❑ char -- 1 byte
 - ❑ short -- 2 bytes
 - ❑ int -- 4 bytes
 - ❑ long -- 4 or 8 bytes, depending on the system and compiler
 - ❑ float -- 4 bytes
 - ❑ double -- 8 bytes
 - ❑ (pointer -- 4 or 8 bytes, depending on the system and compiler)

Data types in C

- ❑ In C, if you want to verify or use the size of a type, you use the sizeof() macro.
- ❑ For example, sizeof(long) returns 4 or 8 depending on how big a long is on your system.

```
#include <stdio.h>
#include <stdlib.h>
int i;
int main(int argc, char **argv) {
int j; /* Copy argc to j to i and print i */
j = argc;
i = j;
printf("Argc: %d\n", i); /* Print the size of a long. */
j = sizeof(long);
printf("Sizeof(long): %d\n", j); /* Print the size of a pointer. */
j = sizeof(int *);
printf("Sizeof(int *): %d\n", j); return 0; }
```

```
UNIX> bin/p1
Argc: 1
Sizeof(long): 8
Sizeof(int *): 8
UNIX> bin/p1 using many arguments
Argc: 4
Sizeof(long): 8
Sizeof(int *): 8
UNIX>
```

Data types in C

- Some machines allow you to compile in 32-bit mode, which forces pointers and lengths to be four bytes;

```
UNIX> gcc -m32 -o bin/p1-32 src/p1.c
```

```
UNIX> bin/p1-32
```

```
Argc: 1
```

```
Sizeof(long): 4
```

```
Sizeof(int *): 4
```

```
UNIX>
```

Data types in C

- ❑ Aggregate (Cluster-Additive types);
- ❑ **Arrays and structs** are set types in C.
- ❑ They are more complex than scalars. You can statically declare an array as a global or local variable.

```
#include <stdio.h>
#include <stdlib.h>
char s1[15];
int main(int argc, char **argv) {
char s2[4];
...
```

Data types in C

- ❑ If an array is statically defined, you cannot assign it to another array.
- ❑ The statement `s2 = "Jim"` is invalid in C because `s2` is statically declared. If you try to compile this program, gcc will give you an error:

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

char s1[15];

int main(int argc, char **argv)
{
    char s2[4];

    s2 = "Jim"; // This line will not compile.
    return 0;
}
```

```
UNIX> gcc -o bin/p2 src/p2.c
src/p2.c:16:6: error: array type 'char [4]' is not assignable
s2 = "Jim"; // This line will not compile.
~~ ^
1 error generated.
UNIX>
```

Pointer

- ❑ A pointer is actually a number, an integer representing an address in memory.
- ❑ On the one hand, many programming languages have no concept of pointers.
- ❑ But one of the biggest advantages of C, namely flexibility, comes from the use of pointers.
- ❑ On the other hand, array is a concept that most programming languages have.
- ❑ We have the same thing in C.

```
int myarray [30];
```

- ❑ The variable **myarray** is really a pointer of type (int *). In C, array elements are accessed using pointers:

```
myarray[5] can actually be translated as *(myarray +  
5)
```

Pointer

- ❑ We can also point to the pointer;

```
int myarray[30];  
main()  
{  
    int * myptr;  
    int ** mydblptr;  
    myptr = myarray;  
    mydblptr = &myarray;  
}
```

- ❑ The Bss segment also has 30 integers allocated. These 120 bytes in Bss is where you actually put your 30 integers and myarray is equal to the head address of these 120 bytes.

Example

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

/* This sets all lower-case letters in a to upper case. */

void change_case(char a[20])
{
    int i;

    for (i = 0; a[i] != '\0'; i++) {
        if (a[i] >= 'a' && a[i] <= 'z') a[i] += ('A' - 'a');
    }
}

/* This initializes a 19-character string of lower-case letters, and then calls change_case(). */

int main()
{
    int i;
    char s[20];

    /* Set s to "abcdefghijklmnopqrs". */

    for (i = 0; i < 19; i++) s[i] = 'a' + i;
    s[19] = '\0';

    /* Print, call change_case() and print again. */

    printf("First, S is %s.\n", s);
    change_case(s);
    printf("Now, S is   %s.\n", s);

    return 0;
}
```

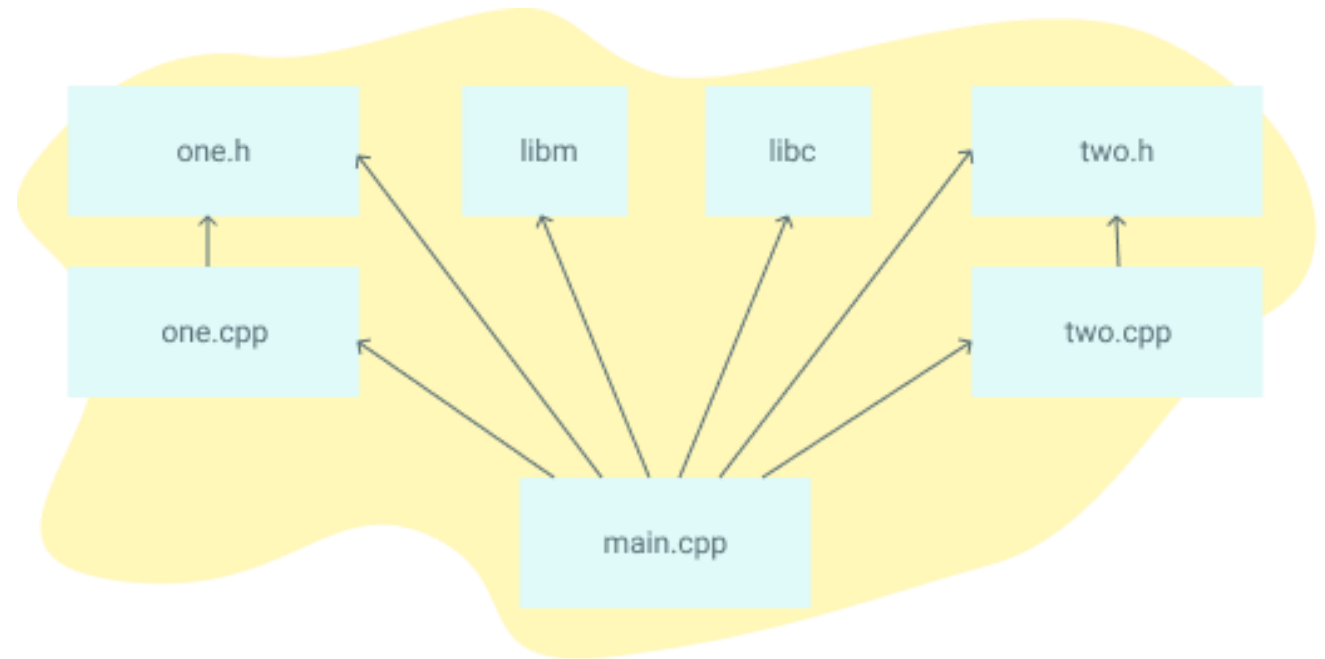
- ☐ **When you pass arrays as parameters, pointers are passed, not arrays.**
- ☐ Although you declare array A as an array of 20 characters, you will notice that the pointer passed to the procedure is just a pointer.
- ☐ Therefore change_case() works on the array, not on a copy:

UNIX> **bin/p2a** First, S is abcdefghijklmnopqrs.
Now, S is ABCDEFGHIJKLMNOPQRS.

UNIX>

makefile

- ❑ **make** is a generic tool that can produce many different outputs.
- ❑ Through a generated **rule** file, the specified system commands are executed successively.
- ❑ On this occasion, you can easily do very complex tasks with a simple **make** command from the command line.



make

- ❑ What alternatives are there for **make**?
- ❑ Popular C/C++ alternative compilation systems are SCons, CMake, Bazel and Ninja.
- ❑ Some code editors, such as Microsoft Visual Studio, have their own built-in **build** tools.
- ❑ For Java there is Ant, Maven and Gradle.
- ❑ Other languages like Go and Rust have their own **build** tools.

makefile

- ❑ Nowadays, open source software comes with a **makefile file** to be used for the compilation process as well as the source files.
- ❑ You can compile any Linux distribution or open source software yourself with the **make** program.
- ❑ If you are a C programmer and you **have a project with multiple source files**, it will take a lot of time to manually compile and link them.
- ❑ Every time you make a change you have to manually enter a lot of commands and filenames through the gcc compiler.

make

- ❑ Here, with the make program and the rule file, these operations can be done automatically only on the console screen:

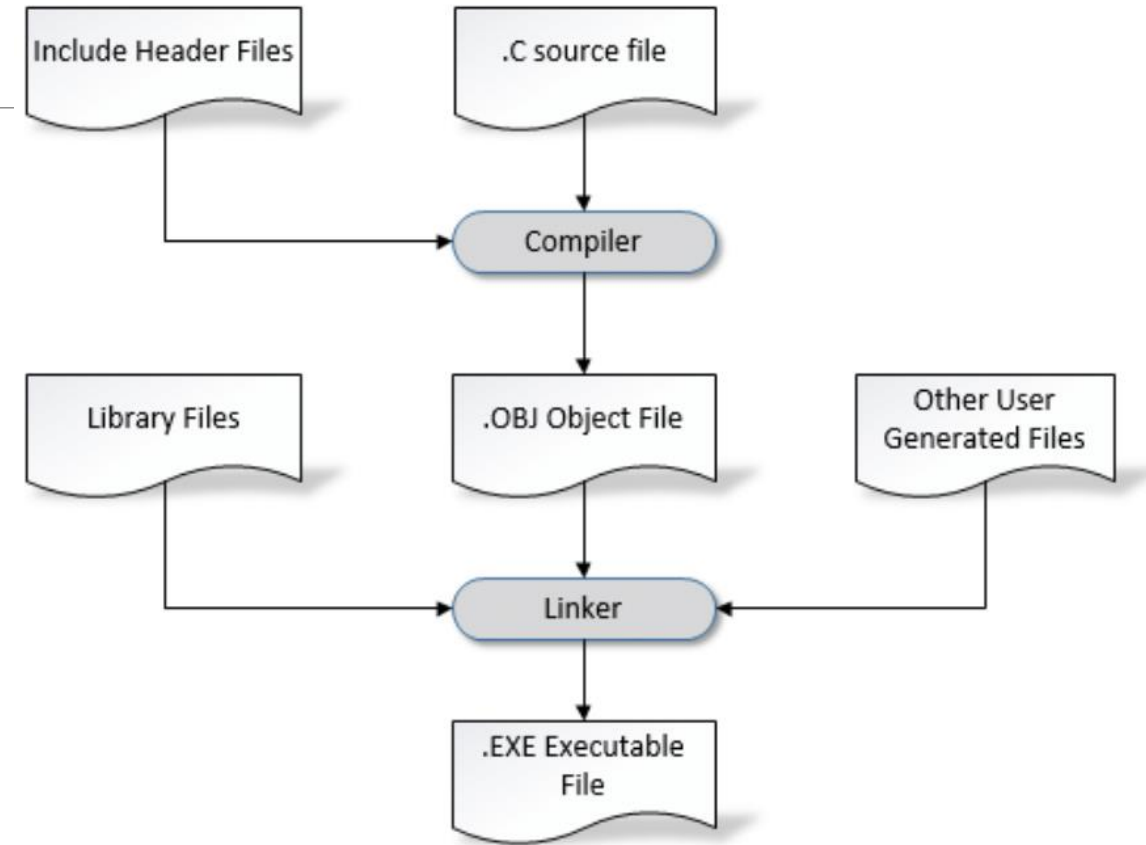
```
# make
```

by typing the command.

- ❑ With the definitions under the tag or tags in the rule file, the **make** program does the necessary compilation and linking for us.
- ❑ If you make changes in the source files, **the** make program first compiles the modified file and then links it with other object files to create the executable file.

Compilation-Binding

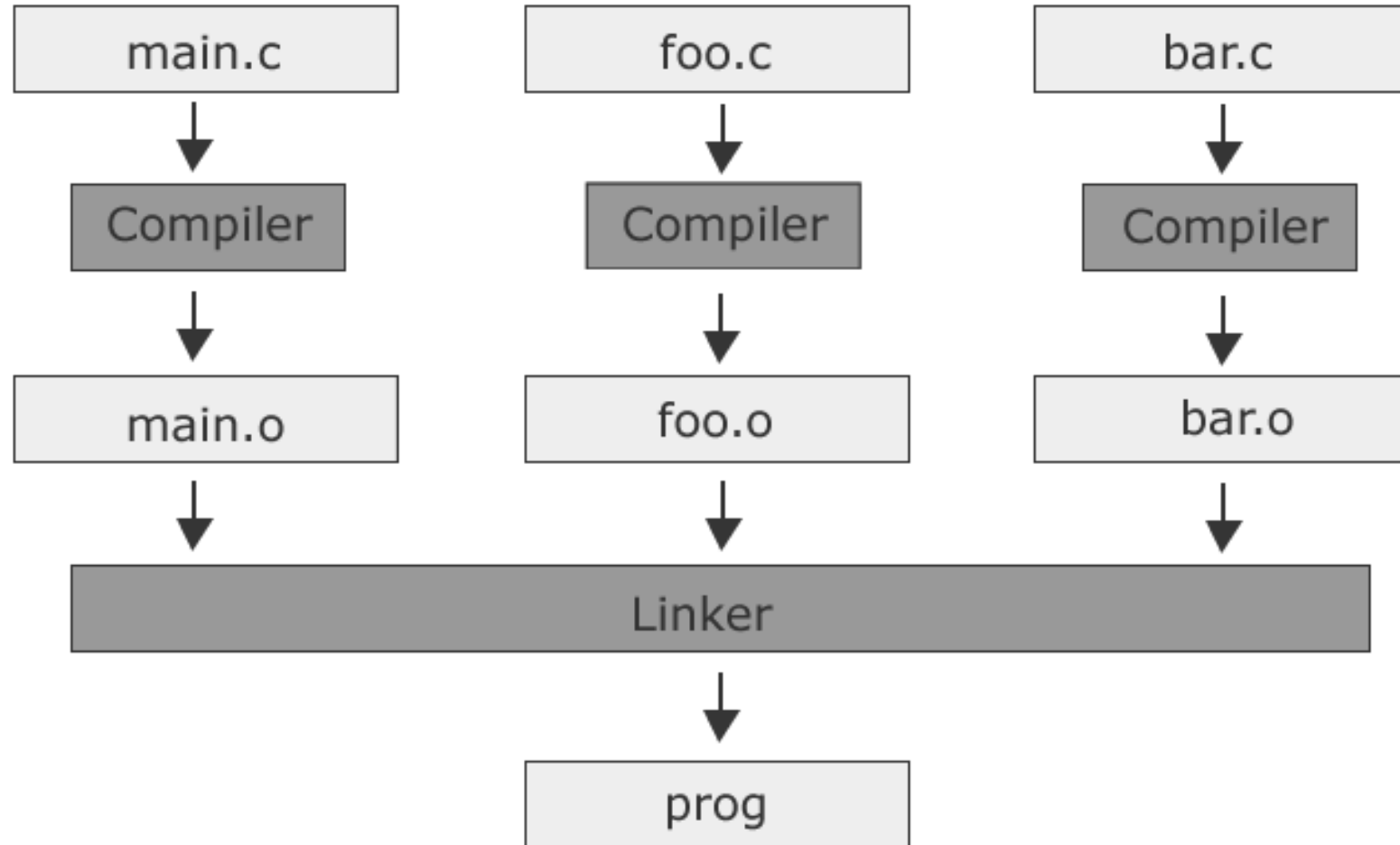
□ It can be run simply from a .c file the process of creating a file is as follows happens :



https://www.researchgate.net/figure/3-Compiler-and-Linker-Relationship_fig2_320142963

makefile

❑ Multiple sources with Makefile compile file



<https://www.c-howto.de/tutorial/makefiles/>

makefile

□ Makefiles are composed of clauses called rules. The general form of a rule is as follows :

```
Target : Additions
```

```
<tab>Command
```

```
...
```

```
-----
```

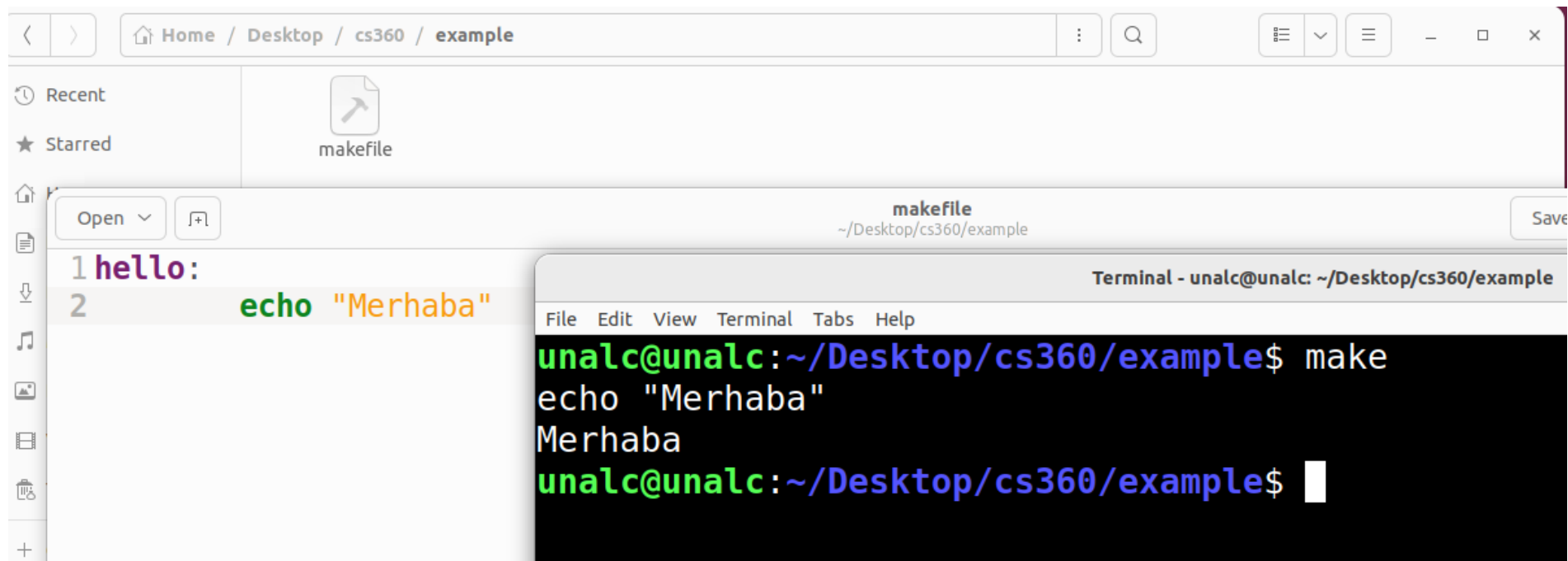
```
Target1 Target2 : Dependency1 Dependency2
```

```
<tab>Command
```

```
...
```

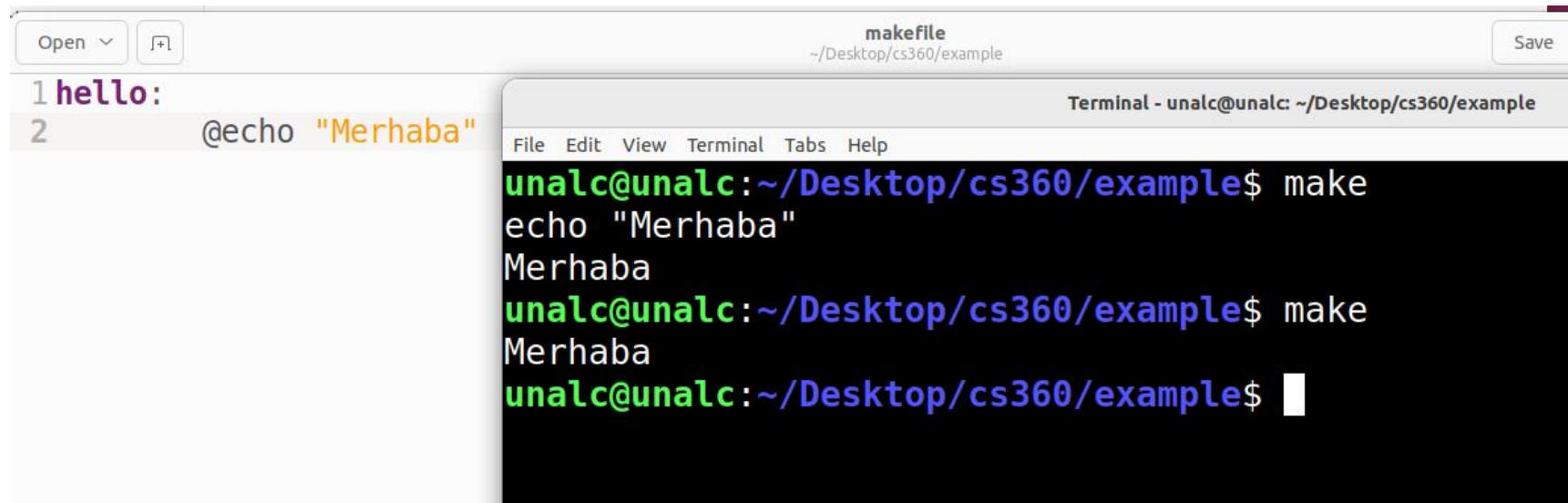
makefile

- ❑ Let's start by printing the classic "Hello World" in the terminal.
- ❑ Create an empty myproject directory containing a Makefile with this content:



make

- ❑ Going back to the example above, when **make** was executed, the full "Hello" **echo** command was displayed, followed by the actual command output.
- ❑ Most of the time we don't want this. We need to start **echo** with **@** to avoid echo the actual command:



The screenshot shows a code editor window titled 'makefile' with the following content:

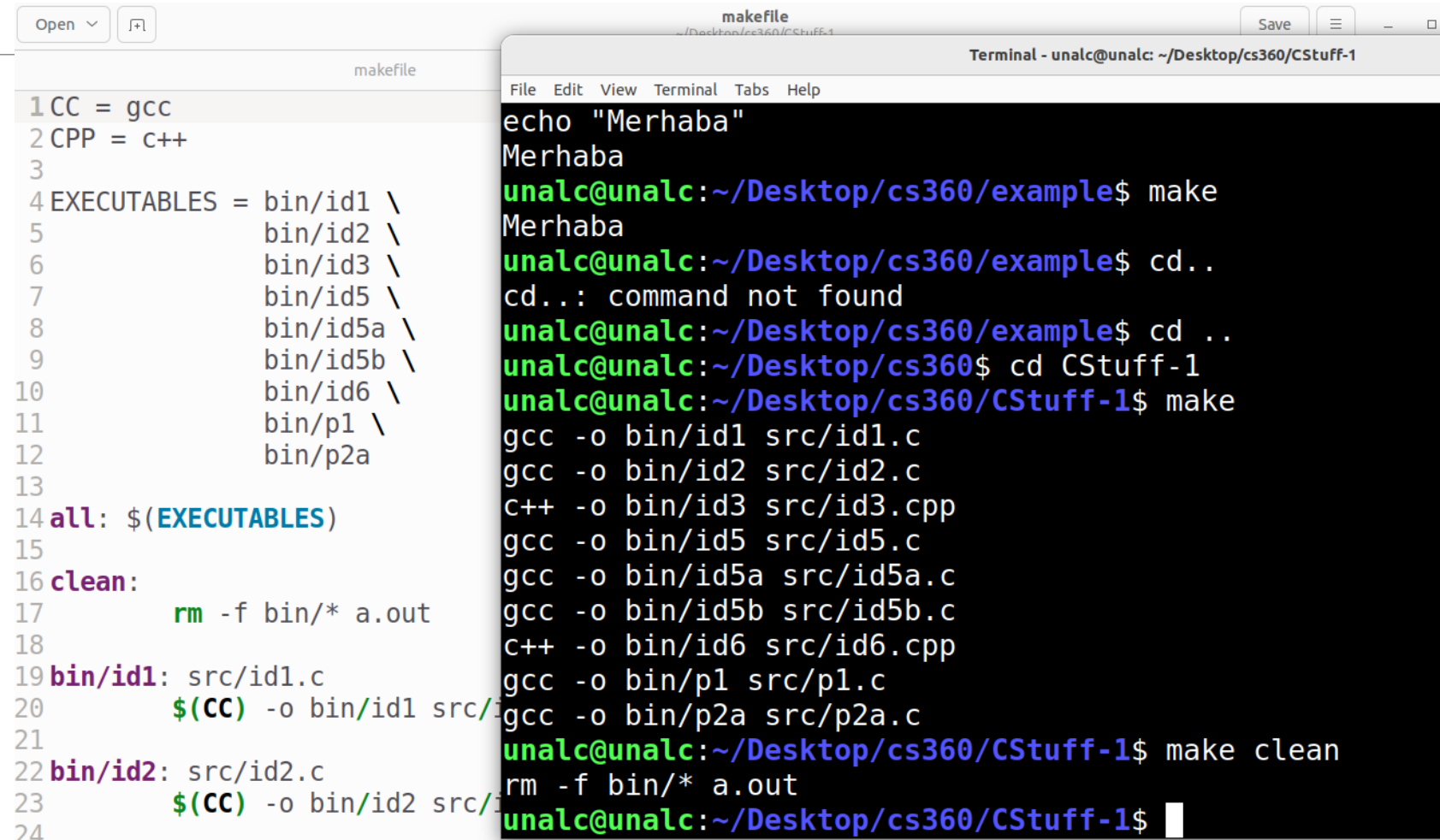
```
1 hello:
2 @echo "Merhaba"
```

Overlaid on the code editor is a terminal window titled 'Terminal - unalc@unalc: ~/Desktop/cs360/example'. The terminal shows the output of running 'make' twice:

```
unalc@unalc:~/Desktop/cs360/example$ make
echo "Merhaba"
Merhaba
unalc@unalc:~/Desktop/cs360/example$ make
Merhaba
unalc@unalc:~/Desktop/cs360/example$
```

makefile

- ❑ If we write more than one destination we need to write its name



The image shows a text editor window titled 'makefile' and a terminal window titled 'Terminal - unalc@unalc: ~/Desktop/cs360/CStuff-1'.

The text editor contains the following Makefile content:

```
1 CC = gcc
2 CPP = c++
3
4 EXECUTABLES = bin/id1 \
5               bin/id2 \
6               bin/id3 \
7               bin/id5 \
8               bin/id5a \
9               bin/id5b \
10              bin/id6 \
11              bin/p1 \
12              bin/p2a
13
14 all: $(EXECUTABLES)
15
16 clean:
17     rm -f bin/* a.out
18
19 bin/id1: src/id1.c
20     $(CC) -o bin/id1 src/id1.c
21
22 bin/id2: src/id2.c
23     $(CC) -o bin/id2 src/id2.c
24
```

The terminal window shows the following commands and output:

```
echo "Merhaba"
Merhaba
unalc@unalc:~/Desktop/cs360/example$ make
Merhaba
unalc@unalc:~/Desktop/cs360/example$ cd..
cd..: command not found
unalc@unalc:~/Desktop/cs360/example$ cd ..
unalc@unalc:~/Desktop/cs360$ cd CStuff-1
unalc@unalc:~/Desktop/cs360/CStuff-1$ make
gcc -o bin/id1 src/id1.c
gcc -o bin/id2 src/id2.c
c++ -o bin/id3 src/id3.cpp
gcc -o bin/id5 src/id5.c
gcc -o bin/id5a src/id5a.c
gcc -o bin/id5b src/id5b.c
c++ -o bin/id6 src/id6.cpp
gcc -o bin/p1 src/p1.c
gcc -o bin/p2a src/p2a.c
unalc@unalc:~/Desktop/cs360/CStuff-1$ make clean
rm -f bin/* a.out
unalc@unalc:~/Desktop/cs360/CStuff-1$
```

makefile

❑ Make multiple source files;

```
project : main.o io.o data.o /usr/lib/libc.a
```

```
    gcc -o project main.o io.o data.o /usr/lib/libc.a/libc.a
```

```
main.o : data.h io.h main.c
```

```
    gcc -c main.c
```

```
io.o : io.h io.c
```

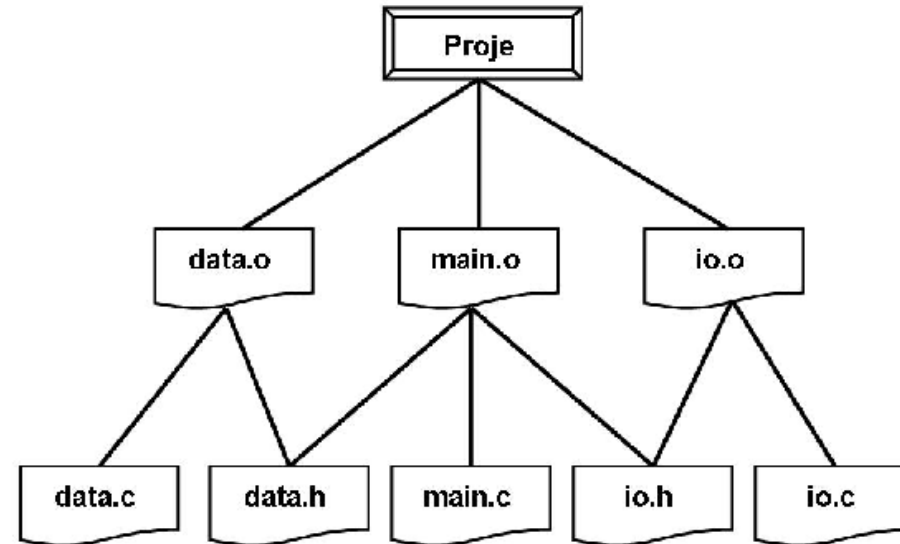
```
    gcc -c io.c
```

```
data.o : data.h data.c
```

```
    gcc -c data.c
```

```
clean :
```

```
    /usr/lib/rm *.o -c io.c
```



make

- ❑ The **make** program ignores lines starting with #. We can write a comment here
- ❑ For the warning message;

```
project : main.o io.o data.o /usr/lib/libc.a
cc -o project main.o io.o data.o /usr/lib/libc.a
@echo == Successful compilation!... ==
```

Make macro

- ❑ Macros are used in **Makefile** files to avoid some long typing. A macro is sort of like a symbolic constant in C. The basic syntax of a macro is :

name : values\s Example usage: \${MAKRO_NAME}

- ❑ If we used a macro, our file would be ;

```
LIBCDIR = /usr/lib/
```

```
COMPILER = cc
```

```
OBJS = main.o main.o io.o data.o
```

```
EXE = project
```

```
${EXE} : ${OBJS} ${LIBCDIR}libc.a
```

```
${COMPILER} -o ${EXE} ${OBJS} ${LIBCDIR}libc.a
```

```
@echo == Successful compilation!... ==
```