Outline

- Quiz - 7 questions x 3 minutes per question = 21 minutes
- Communication is like files.
- Memory: Stack and Heap
- How hardware stores and executes software.

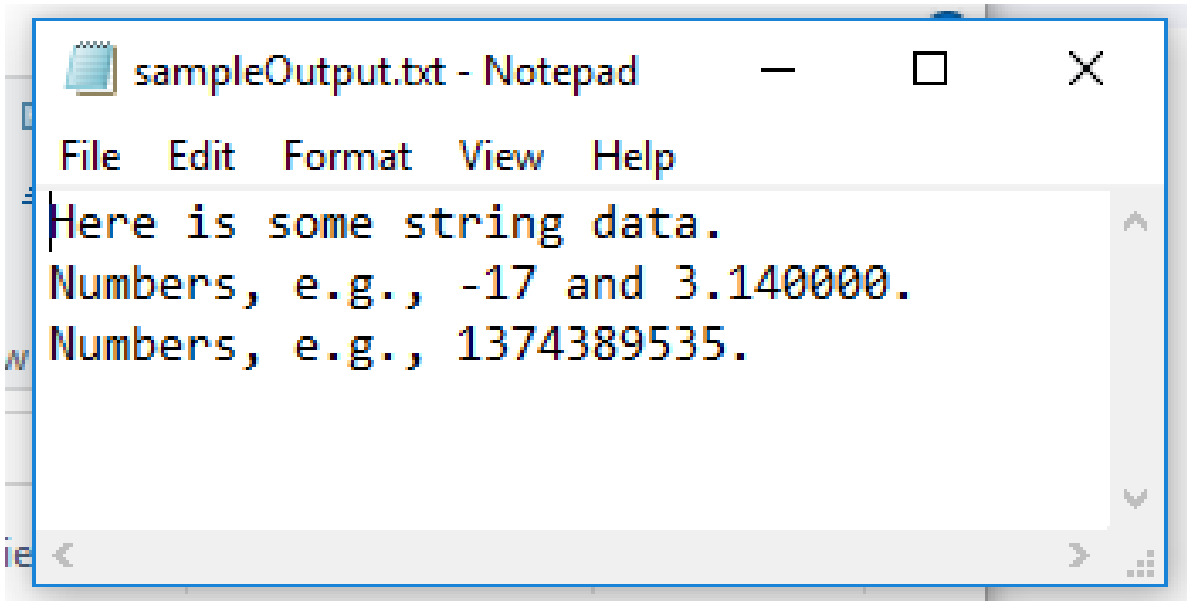
C supports input and output with files. I

- We have seen an example of reading from a text file.
- We will see an example of writing to a text file.
- There are also binary files.

Here's an example of writing to a file.

```
bool testFileOutput()
{
    puts("starting testFileOutput\n");
    fflush(stdout);
    bool ok = false;
    FILE* fp = fopen("sampleOutput.txt", "w");
    fprintf(fp, "Here is some string data.\n");
    fprintf(fp, "Numbers, e.g., %d and %f.\n", -17, 3.14);
    fprintf(fp, "Numbers, e.g., %d.\n", 3.14);
    fclose(fp);
    return ok;
}
```

Here's the file content.

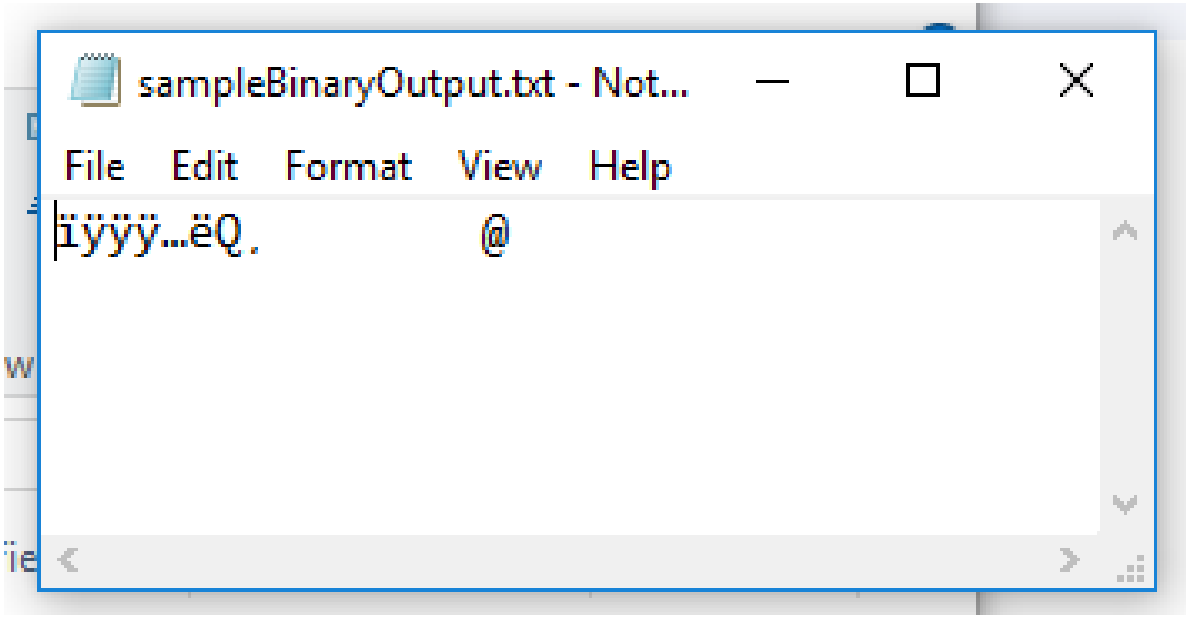


Message: use format specifiers with care.

C standard library supports binary files.

```
FILE* fp = fopen("sampleBinaryOutput.txt", "w");
int a = -17;
double pi = 3.14;
fwrite(&a, sizeof(int),1, fp);
fwrite(&pi, sizeof(double),1, fp);
fclose(fp);
```

Binary files are not generally readable with text editors.



Binary files can be read using C.

```
FILE* fp = fopen("sampleBinaryOutput.txt", "r"); //<- read
int a = -2;
double pi = 8.1;
fread(&a, sizeof(int),1, fp);
fread(&pi, sizeof(double),1, fp);
fclose(fp);
printf("Retrieved %d and %f from file.\n", a, pi);
```

We can see that the variables were set from the file.

```
starting testBinaryFileOutput

starting testBinaryFileInput

Retrieved -17 and 3.140000 from file.
```

Other communication is like files.

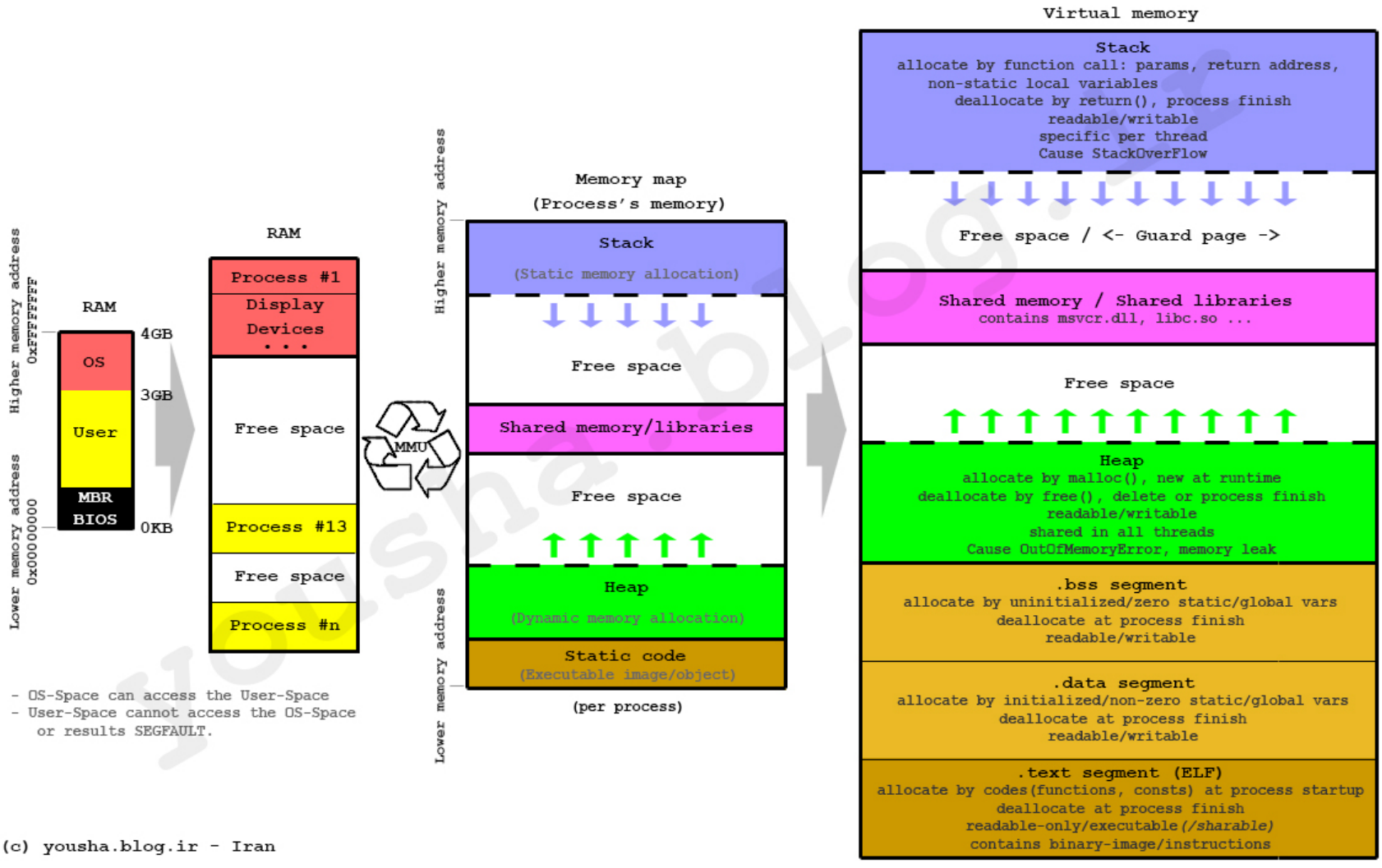
- Input from the user is formatted with scanf.
- Output to the user can be formatted with printf.
- There are format specifiers.
- Examples are %d for an integer, %f for a double, %c for a character, and %s for a string.
- Notice how fread can put a value into a variable, using its address and datatype.
- We can put a value into a distinguished machine address (as a robot might have).

```
in the .h file:
#define ROBOT_EFFECTUATOR (int*) 0X1357642  // a device ad
in the .c file:
int signal = 13;
int* deviceP = ROBOT_EFFECTUATOR;//we know the value of th
//we use a pointer variable to hold that information
*deviceP = signal;
```

Main memory (RAM) is subdivided into regions.

- One such region is the stack for the user's process.
- One such region is the heap for the user's process.
- There are other regions.

There is a device called the memory management unit (MMU).



Function invocations make use of the stack.

- When we write code invoking a function,
- we often designate a variable to receive a return value,
- and we often specify values to be used by the function (if it is parameterized).
- The stack is usually used in sending these values to be used by the function,
- and to convey the results returned by the function.

A reservations of memory (malloc) makes use of the heap.

- When we write code invoking malloc,
- we specify how many bytes we wish.
- If they are available in a contiguous group, we receive a pointer to the first of them.
- When we are through using that memory, we should free it (otherwise we create a memory leak).

Here's some example code using malloc.

```
int* iP = (int*) malloc (300 * sizeof(int));
if (iP != NULL)
{
    //this means we succeeded
    ok = true;
    free(iP); //when we malloc, we free
}
```

We can consider how hardware stores and executes software. I

- Recall that code (i.e., instructions) are stored in main memory.
- It can be that the whole code is too big to be in memory at once.
- In that case, the "paging system" brings in the estimated most relevant instructions,
- while the remaining instructions stay out on the disk.
- Instructions must be brought from main memory into the instruction execution unit.
- Instruction words have fields: they can have address data, they can have operands, and other fields.
- The addresses may participate in operations, that is, the addresses can be modified:
- Addition to a base address is a common example.

We can consider how hardware stores and executes software. II

- Addresses are also used to fetch data.
- Data gets operated on using the "Arithmetic Logic Unit" (ALU) for operations including add, subtract, shift, logicals such as bit-wise AND, sometimes multiply, etc.
- The ALU can also serve in calculations involving addresses.
- Calculations on floats and/or doubles are often supported by floating point hardware.
- A great deal of trouble has been taken to increase the throughput of processors.
- Frequently used books on this subject include one by Hennesey and Patterson, and another by Patterson and Hennessey.

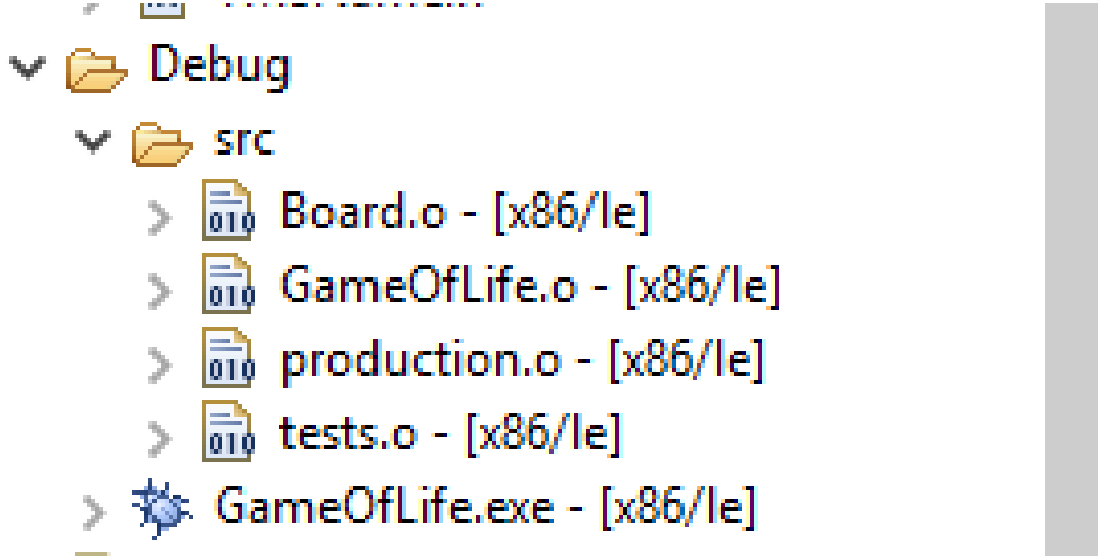
C code gets processed into instructions. I

- There are several stages in the conversion of C code we write into instructions the machine executes.
- As we saw from the memory diagram, processes share the memory used by the processor.
- The memory used by the processor uses physical addresses.
- The memory used by the process is virtual memory.
- The memory management unit converts between the two.
- The compiler, with the help of the assembler, translates the C code into instructions.
- These are ".o" files.
- Some of the code comes from libraries.
- The linker merges the library code with the assembled code.
- The positioning of the program onto the machine is carried out by the loader.

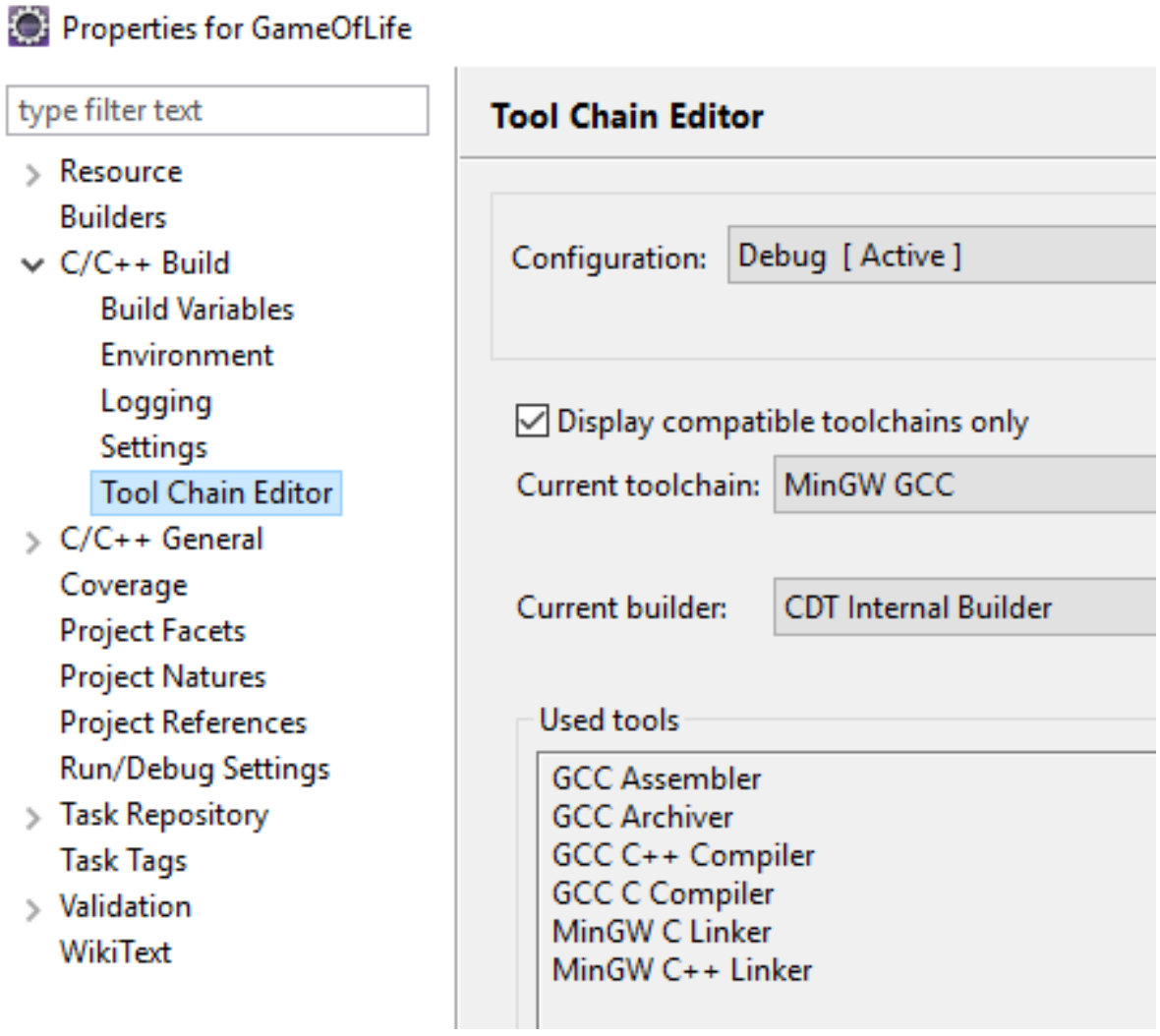
There are efficiencies to be achieved in compiling and linking.

- When we modify code, we do not always change every file.
- Some files remain unchanged.
- Unchanged files do not need to be converted into assembly code again.
- Those ".o" files would be no different from the previous ones.
- There is a language for specifying what does need to be done, in response to changes to any given file.
- This is the language of the "make file".
- Because these dependencies can be determined by the IDE (including Eclipse), the IDE can create these files.

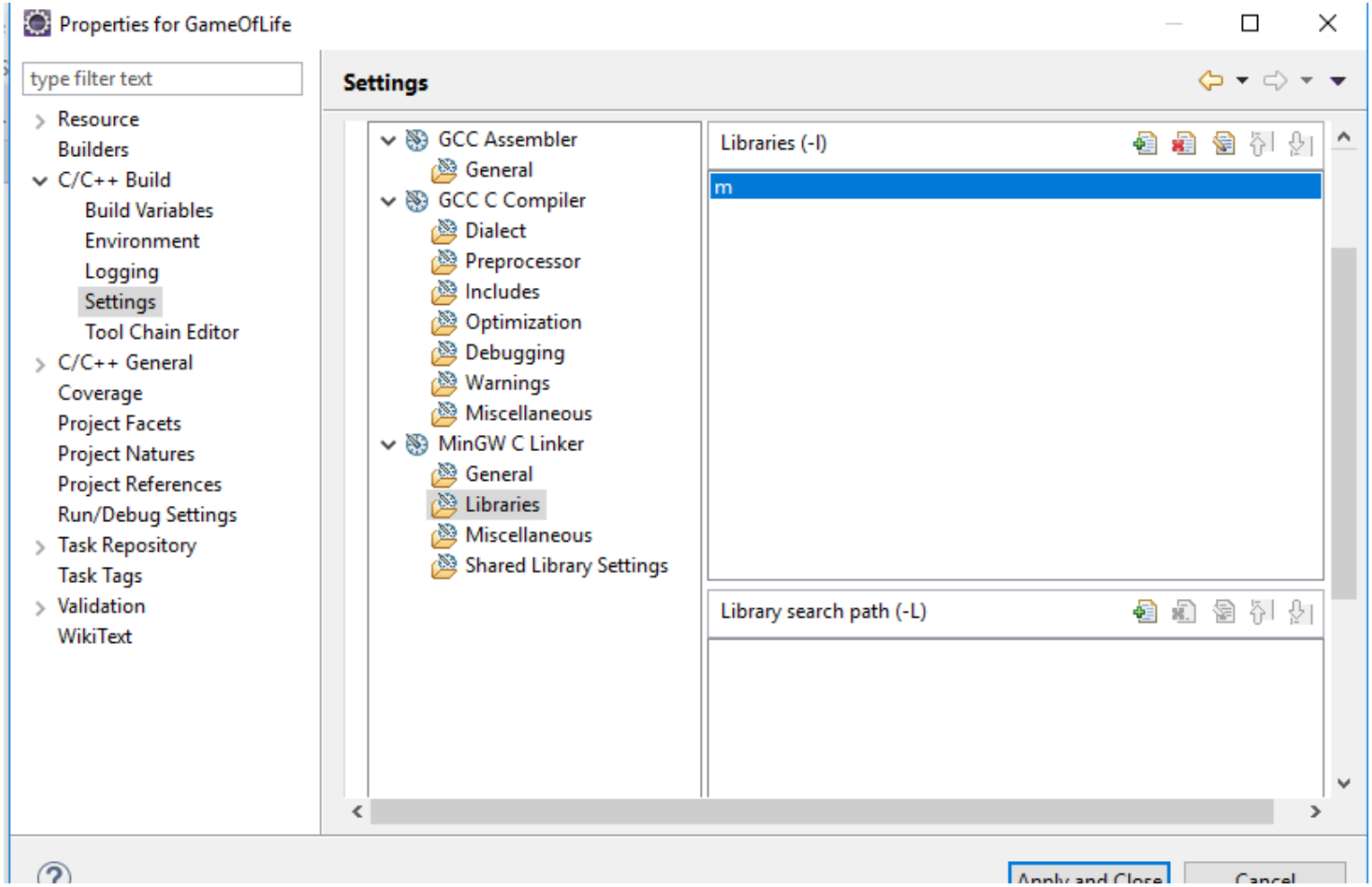
We can see the ".o" files.



We can see the tools participating in compilation, assembly and linking.



We can see the libraries used when we call system functions.



Eclipse creates make files for us.

- We do not need to.

```
CC=gcc
CFLAGS=-I.
DEPS = hellomake.h

%.o: %.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAGS)

hellomake: hellomake.o hellofunc.o
	$(CC) -o hellomake hellomake.o hellofunc.o
```