

CS 35L

Discussion 1A

Week 6

C and Makefile



Contents

1. Overview of C

- Brief Intro
- C vs C++
- Syntax
- Dynamic Memory Management
- Libraries

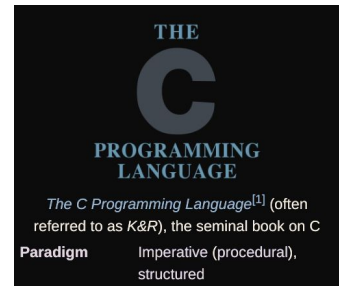
2. Makefile

- Contents
- Rules

1.1 C - Brief Intro

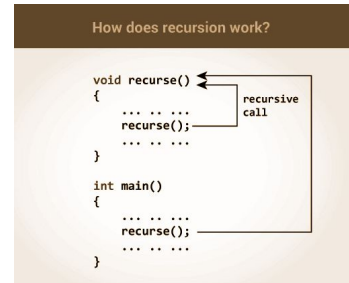
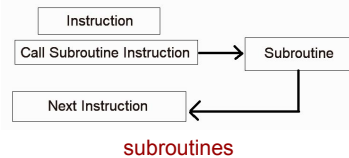
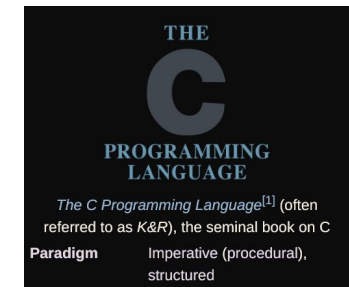
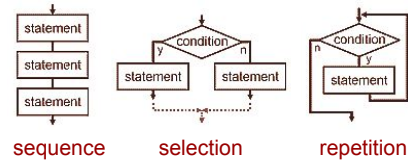
- C Programming Language

- C was originally developed at Bell Labs by Dennis Ritchie between 1972 and 1973 to construct running on Unix (improved from Programming Language B)
- It was applied to re-implementing the kernel of the Unix operating system
- It was designed to be complied to provide low-level access to memory and language constructs that map **efficiently** to machine instructions, all with minimal runtime support
- It was designed to encourage **cross-platform** programming: can be compiled for a wide variety of computer platforms and operating systems with few changes to the source code



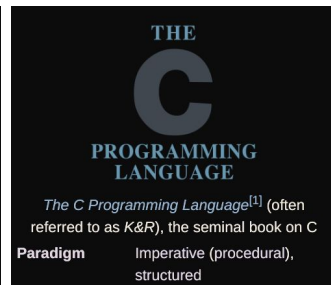
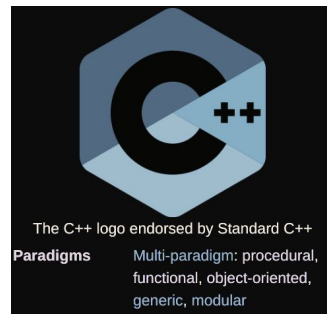
1.1 C - Brief Intro

- C is a general-purpose, procedural computer programming language
- C supports
 - Structured programming: (control structures)
 - Lexical variable scope (static scope)
 - Static scope v.s. Dynamic scope
 - (suppose a variable name's scope is a certain function)
 - Static: within the certain function text, the variable exists
 - Dynamic: while the function is running, the variable exists
 - Recursion
 - Solve a problem and the solution depends on solutions to smaller instance of the same problem
- C use the static type system
 - Static type check: verify the type safety of a program based on analysis of a program's text (source code)



1.2 C v.s C++

- Similar parts:
 - Manual memory management
 - Both are compiled languages
 - Basic data types
 - Structured programming languages: structure control, subroutines
- Differences:
 - C++: support exceptions; C: errors are notified by the returned value of the function, the exit value of the processes and error signals to the process
 - C++: Standard Template Library (STL) - provides four components called algorithms, containers, functions, and iterators.
 - C++: support stream operators like cin, cout, <<, >>
 - C++: support object-oriented properties
 - C++: generic programming (templates) - algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters



```
template <typename T>
T max(T x, T y) {
    return x < y ? y : x;
}
```

Template

Your code

```
std::cout << max(3, 7); // Outputs 7.
```

A function call

```
int max(int x, int y) {
    return x < y ? y : x;
}
```

Instantiated version

1.3 C Syntax - Hello World!

- **main** function
 - The entrance of our program
 - Return value is of type **int**
 - In general: **return 0** indicates that the program is successful
- Use libraries for some basic functions
 - **printf**: a basic function (print sth to stdout)
 - **stdio.h**: the header file where **printf** is defined
- Compile the c files
 - Compiler: **gcc**
 - **gcc sourceCode.c -o outexe**
 - Or with IDEs

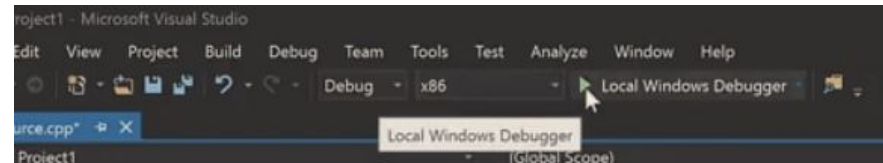


The image shows a code editor window with a file named `test.c`. The code is as follows:

```
1  #include <stdio.h>
2
3  int main()
4  {
5      printf("Hello World!\n");
6      return 0;
7  }
```

Below the code editor is a terminal window. It shows the following commands and output:

```
~/Des/c/testfolder > gcc test.c -o testout
~/Des/c/testfolder > ./testout
Hello World!
```



1.3 C Syntax - Data Structures

- Primitive data types
 - Numbers: integral, real and complex
 - Integral: integers
 - **signed** and **unsigned**
 - Default: **signed**
 - Different fixed sizes
 - Depend on your machine
 - Real/complex: floating point form
 - (complex: pair of numbers)

Floating-point types				
Type specifiers	Precision (decimal digits)		Exponent range	
	Minimum	IEEE 754	Minimum	IEEE 754
float	6	7.2 (24 bits)	±37	±38 (8 bits)
double	10	15.9 (53 bits)	±37	±307 (11 bits)
long double	10	34.0 (113 bits)	±37	±4931 (15 bits)

Specifications for standard integer types	
Shortest form of specifier	Minimum width (bits)
_Bool	1
char	8
signed char	8
unsigned char	8
short	16
unsigned short	16
int	16
unsigned int	16
long	32
unsigned long	32
long long ^[1]	64
unsigned long long ^[1]	64

```
vim test.cpp
#include <stdio.h>

int main()
{
    printf("sizeof char = %lu bytes\n", sizeof(char));
    printf("sizeof int = %lu bytes\n", sizeof(int));
    printf("sizeof float = %lu bytes\n", sizeof(float));
    printf("sizeof double = %lu bytes\n", sizeof(double));
    return 0;
}

yuxing@yuxing-home:~/tst 80x10
$ gcc test.cpp -o test_out
$ ./test_out
sizeof char = 1 bytes
sizeof int = 4 bytes
sizeof float = 4 bytes
sizeof double = 8 bytes
```

1.3 C Syntax - Data Structures

- Pointers

- In declarations the asterisk modifier (*) specifies a pointer type.

- Pointer values associate two pieces of info:

- Memory address
- Data type

- `int *ptr; // declaration of a pointer to integer`

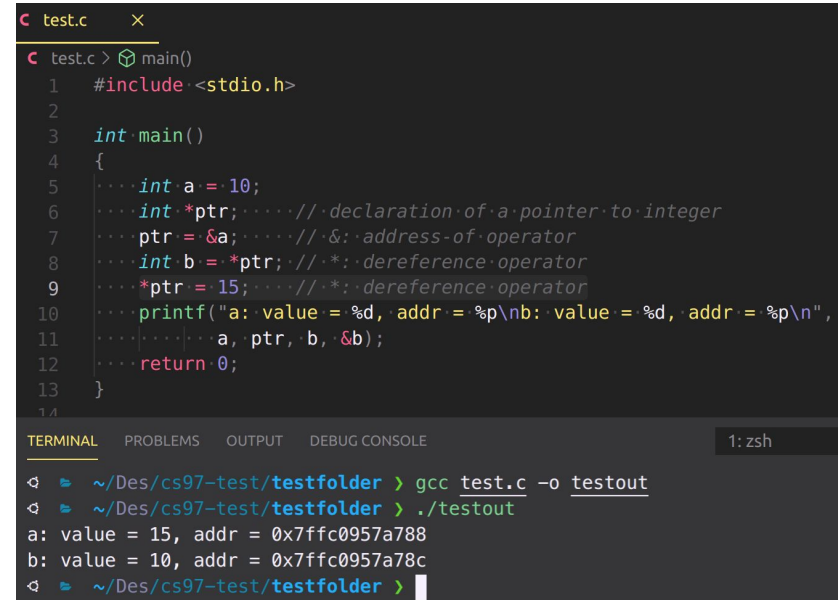
- Referencing: assign an address to the pointer before using it

- `ptr = &a; // &: address-of operator`

- Dereferencing

- The pointed-to data can be accessed through a pointer value.

- `*ptr = 15; // *: dereference operator`



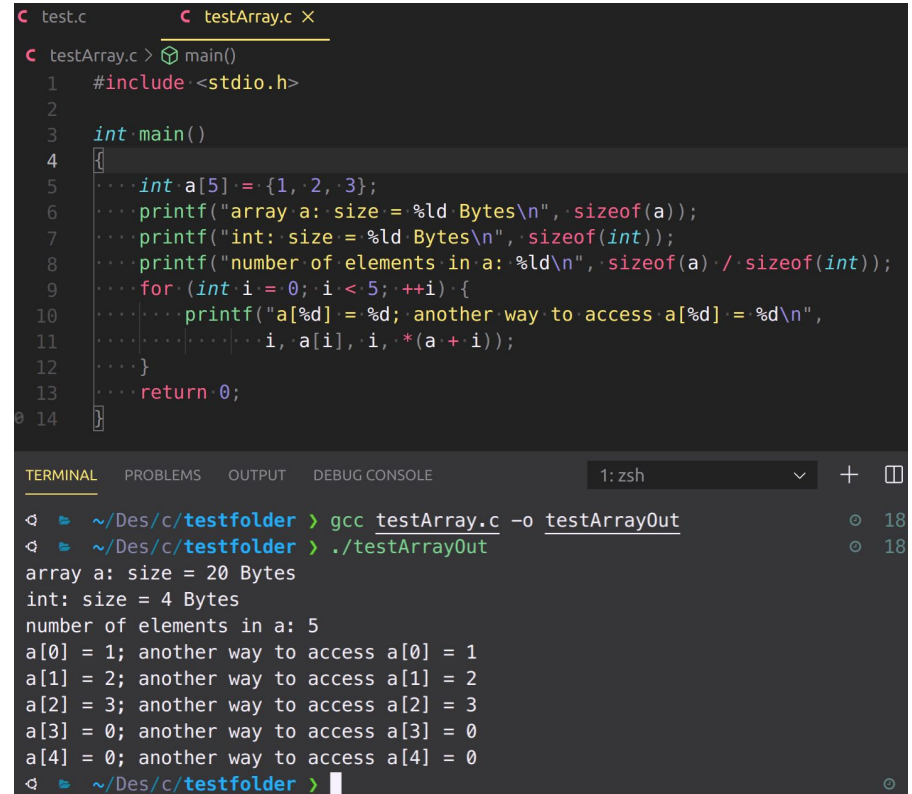
The screenshot shows a C program named `test.c` in a text editor. The code includes `<stdio.h>` and defines a `main` function. Inside `main`, it declares an integer `a` with value 10, a pointer `ptr` of type `int`, and an integer `b`. It then assigns `ptr` the address of `a` (`&a`), dereferences `ptr` to assign 15 to `b` (`*ptr = 15`), and prints the values and addresses of `a` and `b`. The program is compiled and run in a terminal, showing the output: `a: value = 15, addr = 0x7ffc0957a788` and `b: value = 10, addr = 0x7ffc0957a78c`.

```
C test.c x
C test.c > main()
1 #include <stdio.h>
2
3 int main()
4 {
5     int a = 10;
6     int *ptr; // declaration of a pointer to integer
7     ptr = &a; // &: address-of operator
8     int b = *ptr; // *: dereference operator
9     *ptr = 15; // *: dereference operator
10    printf("a: value = %d, addr = %p\nb: value = %d, addr = %p\n",
11           a, ptr, b, &b);
12    return 0;
13 }
14

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE 1: zsh
~/Des/cs97-test/testfolder > gcc test.c -o testout
~/Des/cs97-test/testfolder > ./testout
a: value = 15, addr = 0x7ffc0957a788
b: value = 10, addr = 0x7ffc0957a78c
~/Des/cs97-test/testfolder >
```


1.3 C Syntax - Data Structures

- Array
 - Represent structures of consecutive elements of the same type.
 - The definition of a **fixed-size** array
 - `int a[100];`
 - Primitive type: `int`
 - Number of values: `100`
 - Accessing elements
 - `a[i]` // `i` is an index number
 - `*(a+i)` // array name: `a` pointer to the array
 - Indexing: starting from 0
 - `sizeof` operator:
 - The total size of entire array (in bytes) (long unsigned int)



```
test.c      testArray.c X
C testArray.c > main()
1  #include <stdio.h>
2
3  int main()
4  {
5      int a[5] = {1, 2, 3};
6      printf("array a: size = %ld Bytes\n", sizeof(a));
7      printf("int: size = %ld Bytes\n", sizeof(int));
8      printf("number of elements in a: %ld\n", sizeof(a) / sizeof(int));
9      for (int i = 0; i < 5; ++i) {
10         printf("a[%d] = %d; another way to access a[%d] = %d\n",
11             i, a[i], i, *(a+i));
12     }
13     return 0;
14 }
```

```
TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE
1: zsh
~/Des/c/testfolder > gcc testArray.c -o testArrayOut
~/Des/c/testfolder > ./testArrayOut
array a: size = 20 Bytes
int: size = 4 Bytes
number of elements in a: 5
a[0] = 1; another way to access a[0] = 1
a[1] = 2; another way to access a[1] = 2
a[2] = 3; another way to access a[2] = 3
a[3] = 0; another way to access a[3] = 0
a[4] = 0; another way to access a[4] = 0
~/Des/c/testfolder >
```

1.3 C Syntax - Data Structures

- Strings

- String literals are surrounded by double quotes:
 - E.g.: "Hello world!"
- They are compiled to an **array** of specified **char** values, with an **additional null terminating character** (0-valued) code to mark the end of the string.
- char? int?
 - **char**: 8-bit integer
 - Store characters(include **control characters**)
 - To represent characters, computers have to map the characters to numerical code
 - ASCII (American Standard Code for Information Interchange)
<https://en.wikipedia.org/wiki/ASCII>)
- `#include <string.h>`
 - More string-related functions to manipulate C strings and arrays., e.g. **strlen**, **strcmp** ...
 - <https://www.cplusplus.com/reference/cstring/>

```
testString.c X
testString.c > main()
1 2
2 3 int main()
3 4 {
4 5     char someString[] = "Hello World!\n";
5 6     printf("%s", someString);
6 7     printf("array a: size = %ld Bytes\n", sizeof(someString));
7 8     printf("char: size = %ld Bytes\n", sizeof(char));
8 9     int sizeString = sizeof(someString) / sizeof(char);
9 10    for(int i = 0; i < sizeString; ++i) {
10 11        printf("someString[%d] = [%c] (int value = %d)\n",
11 12            i, someString[i], (int)someString[i]);
12 13    }
13 14    return 0;
14 15 }
```

```
TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE  1: zsh
~/Des/c/testfolder > gcc testString.c -o testStringOut
~/Des/c/testfolder > ./testStringOut
Hello World!
array a: size = 14 Bytes
char: size = 1 Bytes
someString[0] = [H] (int value = 72)
someString[1] = [e] (int value = 101)
someString[2] = [l] (int value = 108)
someString[3] = [l] (int value = 108)
someString[4] = [o] (int value = 111)
someString[5] = [ ] (int value = 32)
someString[6] = [W] (int value = 87)
someString[7] = [o] (int value = 111)
someString[8] = [r] (int value = 114)
someString[9] = [\] (int value = 92)
someString[10] = [d] (int value = 100)
someString[11] = [!] (int value = 33)
someString[12] = [
] (int value = 10)
someString[13] = [] (int value = 0)
~/Des/c/testfolder >
```

1.3 C Syntax

- Strings

- String literals
 - E.g.: `"hello world"`
- They are constant values, with (0-valued) characters
- char? int?

- `char`:
 - Store

- ASCII Information
 - https://en.cppreference.com/w/cpp/string/basic/basic_string_view

- `#include <string.h>`
 - More C string functions
 - https://en.cppreference.com/w/cpp/string/basic/basic_string_view

Backslashes may be used to enter control characters, etc., into a string:

Escape	Meaning
<code>\\</code>	Literal backslash
<code>\"</code>	Double quote
<code>\'</code>	Single quote
<code>\n</code>	Newline (line feed)
<code>\r</code>	Carriage return
<code>\b</code>	Backspace
<code>\t</code>	Horizontal tab
<code>\f</code>	Form feed
<code>\a</code>	Alert (bell)
<code>\v</code>	Vertical tab
<code>\?</code>	Question mark (used to escape trigraphs)
<code>%%</code>	Percentage mark, <code>printf</code> format strings only (Note <code>\%</code> is non standard and is not always recognised)
<code>\000</code>	Character with octal value <code>OOO</code> (where <code>OOO</code> is 1-3 octal digits, '0'-'7')
<code>\xHH</code>	Character with hexadecimal value <code>HH</code> (where <code>HH</code> is 1 or more hex digits, '0'-'9', 'A'-'F', 'a'-'f')

testString.c

```
    printf("hello World!\n");
};
printf("%ld Bytes\n", sizeof(someString));
printf("Bytes\n", sizeof(char));
printf("someString\n", sizeof(char));
printf("String\n", ++i) {
    printf("%c\n", (int) value = %d\n",
    i, (int) someString[i]);
}
```

1: zsh

testString.c -o testStringOut
testStringOut

2)
01)
08)
08)
11)
2)
7)
11)
14)
08)
100)
33)

1.3 C Syntax - Data Structures

- Structs
 - Data containers consisting of a sequence of named members of various types
 - The members of a structure are stored in consecutive locations in memory (compilers are allowed to insert padding between or after members)
 - The size of a structure = sum of the size of its members + the size of the padding
 - Declaration (in example)
 - Accessing members
 - Use a period .
 - Use -> (for pointer case)
 - **typedef** specifier
 - Create additional name for another data type
 - Eliminate the need for later **struct** keyword

```
testStruct.c X
testStruct.c > main()
1 #include <stdio.h>
2
3 struct StudentInfoStruct
4 {
5     /*.data.*/
6     char* name;
7     int age;
8     float height;
9 };
10 typedef struct StudentInfoStruct StudentInfo;
11
12 int main()
13 {
14     /*student.1*/
15     struct StudentInfoStruct s1;
16     s1.name = "Helen";
17     s1.age = 20;
18     s1.height = 5.5;
19     printf("student1: name = %s, age = %d, height = %f\n",
20           s1.name, s1.age, s1.height);
21
22     /*student.2*/
23     StudentInfo s2 = {"Harry Potter", 18, 6.0};
24     StudentInfo *s2Ptr = &s2;
25     printf("student2: name = %s, age = %d, height = %f\n",
26           s2Ptr->name, s2Ptr->age, s2Ptr->height);
27     return 0;
28 }
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE 1: zsh

```
> gcc testStruct.c -o testStructOut
> ./testStructOut
student1: name = Helen, age = 20, height = 5.500000
student2: name = Harry Potter, age = 18, height = 6.000000
```

1.3 C Syntax - Operators

- Type
 - Arithmetic operators
 - `a+b, a-b, -a, a*b, a/b, a%b, ++a, a++, --a, a--`
 - Comparison (relational) operators
 - `a==b, a!=b, a>b, a>=b, a<b, a<=b`
 - Logical operators
 - `!a, a&&b, a||b`
 - Bitwise operators
 - `~a, a&b, a|b, a^b, a<<b, a>>b`
 - Assignment operators
 - `a=b, a+=b, a-=b, a*=b, a/=b, a%=b, a&=b, a|=b, a^=b, a<<=b, a>>=b`
 - Member and pointer operators
 - `a[b], *a, &a, a->b, a.b`
 - Other operators
 - `a?b:c, sizeof, ...`
- Operator precedence

1.3 C Syntax - Control Structures

- Compound statement
 - Indicate by `{ }`
 - Define a scope
- Selection statement
 - If statement
 - Switch statement
- Iteration statement
 - Do-while statement
 - While statement
 - For statement
- Functions
 - Parameter: Pass by value

```
<return-type> functionName( <parameter-list> )  
{  
    <statements>  
    return <expression of type return-type>;  
}
```

```
{  
    <optional-declaration-list>  
    <optional-statement-list>  
}
```

```
if ( <expression> )  
    <statement1>  
else  
    <statement2>
```

```
switch ( <expression> )  
{  
    case <label1> :  
        <statements 1>  
    case <label2> :  
        <statements 2>  
        break;  
    default :  
        <statements 3>  
}
```

```
do  
    <statement>  
while ( <expression> ) ;  
  
while ( <expression> )  
    <statement>  
  
for ( <expression> ; <expression> ; <expression> )  
    <statement>
```

1.4 C - Dynamic Memory Management

- Recall: The definition of a **fixed-size** array
 - `int a[100];`
 - But now, let's consider a situation: no idea about the length of the text you need to store
- Runtime memory allocation and management

Sr.No.	Function & Description
1	void *calloc(int num, int size); This function allocates an array of num elements each of which size in bytes will be size .
2	void free(void *address); This function releases a block of memory block specified by address.
3	void *malloc(int num); This function allocates an array of num bytes and leave them uninitialized.
4	void *realloc(void *address, int newsize); This function re-allocates memory extending it upto newsize .

1.4 C - Dynamic Memory Management

- Recall: The definition of a **fixed-size** array
 - `int a[100];`
 - But now, let's consider a situation: no idea about the size
- Runtime memory allocation and management
 -

Sr.No.	Function & Description
1	void *calloc(int num, int size); This function allocates an array of num elements each of which size in bytes is size .
2	void free(void *address); This function releases a block of memory block specified by address.
3	void *malloc(int num); This function allocates an array of num bytes and leave them uninitialized.
4	void *realloc(void *address, int newsize); This function re-allocates memory extending it upto newsize .

```
C testMalloc.c X
C testMalloc.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(){
6
7      char name[100];
8      char *description;
9
10     strcpy(name, "Zara Ali");
11
12     /* allocate memory dynamically */
13     description = malloc(30 * sizeof(char));
14
15     if( description == NULL ){
16         fprintf(stderr, "Error - unable to allocate required memory\n");
17     } else {
18         strcpy(description, "Zara ali a DPS student.");
19     }
20
21     /* suppose you want to store bigger description */
22     description = realloc( description, 100 * sizeof(char) );
23
24     if( description == NULL ){
25         fprintf(stderr, "Error - unable to allocate required memory\n");
26     } else {
27         strcat( description, "She is in class 10th" );
28     }
29
30     printf("Name = %s\n", name);
31     printf("Description: %s\n", description);
32
33     /* release memory using free() function */
34     free(description);
35     return 0;
36 }

TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE  1: zsh
~/Desktop/cs97-test/testfolder > gcc testMalloc.c -o testMalloc
~/Desktop/cs97-test/testfolder > ./testMalloc
Name = Zara Ali
Description: Zara ali a DPS student.She is in class 10th
~/Desktop/cs97-test/testfolder >
```


1.4 C - Dynamic Memory Management

- Some common issues
 - Using Before Writing
 - Malloc memory contents initialize to garbage
 - We need to write to it before trying to use it
 - Forgetting to free
 - Causes memory leak
 - Double free (freeing the same address twice)
 - Causes undefined behavior and maybe a seg-fault
 - Using after free (dangling pointer)
 - Causes undefined behavior

1.5 C - Libraries

- Standard I/O and File I/O in C: <stdio.h>
 - https://www.tutorialspoint.com/c_standard_library/stdio_h.htm
- Standard I/O
 - **printf**: send formatted output to stdout
 - **%d**: int; **%f**: float, **%p**: pointer addrs; **%s**: string; **%c**: char ...
 - **getchar**: read a character from stdin
 - **gets**: read a line from stdin
 - **puts**: write a string to stdout
- File I/O:
 - Use pointers to access files in various modes
 - **r**: reading mode; **w**: writing mode; **a**: appending mode
 - **FILE *fopen(const char *filename, const char *mode)**: Opens the filename pointed to by filename using the given mode.
 - **int fclose(FILE *stream)**: Closes the stream. All buffers are flushed.
 - **fread**: Reads data from the given stream into the array
 - **fwrite**: Writes data from the array pointed to by ptr to the given stream.

Contents

1. Overview of C

- Brief Intro
- C vs C++
- Syntax
- Dynamic Memory Management
- Libraries

2. Makefile

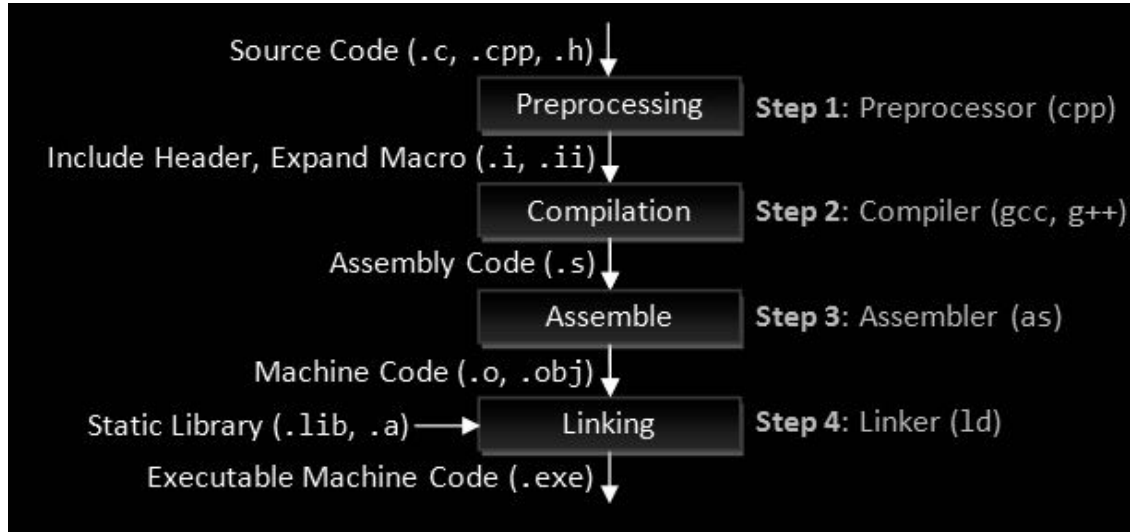
- Contents
- Rules

2.0 Compile C

- Source code --(compiler)--> machine-readable code
- We use gcc

• E.g.:

- `gcc -g -Wall -std=c99 source.c -o outExe`
- `-Wall` -> Display compiler warnings
- `-g` -> Make the program debuggable with gdb/valgrind
- `-std=c99` -> use C99 Standard
- `-o` -> specify output file name



2.0 Compile C

- Source code --(compiler)--> machine-readable code
- What if we don't want to type the long `gcc` line every time?
- What if I have multiple files in my project?
- What if I only modified `a.c`, do I have to recompile all `.c/.h` files in my project?

- We expect:
 - Compile all the files together
 - Recompile the modified files only
 - Type a simpler (shorter) command to compile

- E.g.:
 - `gcc -g -Wall -std=c99 source.c -o outExe`
 - `-Wall` -> Display compiler warnings
 - `-g` -> Make the program debuggable with gdb/valgrind
 - `-std=c99` -> use C99 Standard
 - `-o` -> specify output file name

2.1 Makefile Content

- A build automation tool that automatically builds executables and libraries from source code
- Specify how to derive the target program
- Will build objects as necessary and compiles only what is needed based on which file have been updated
- Content: explicit rules, implicit rules, variable definitions, directives, and comments.
 - **Explicit rule**: when and how to remake one or more files, called the rule's targets. It lists the other files that the targets depend on, and may also give a command list to use to create or update the targets.
 - **Implicit rule**: when and how to remake a class of files based on their names. It describes how a target may depend on a file with a ***name similar*** to the target and gives a command to create or update such a target.
 - **Variable definition**: specifies a text string value for a variable that can be substituted into the text later.
 - **Directive**: instruction for make to do something special such as reading another makefile.
 - **Comments**: '#' in a line starts a comment.

2.2 Makefile Rules

- Format

- Target:

- The name of a file that is generated by a program
 - Executables or object files
 - Or an action to carry out (e.g.: clean)

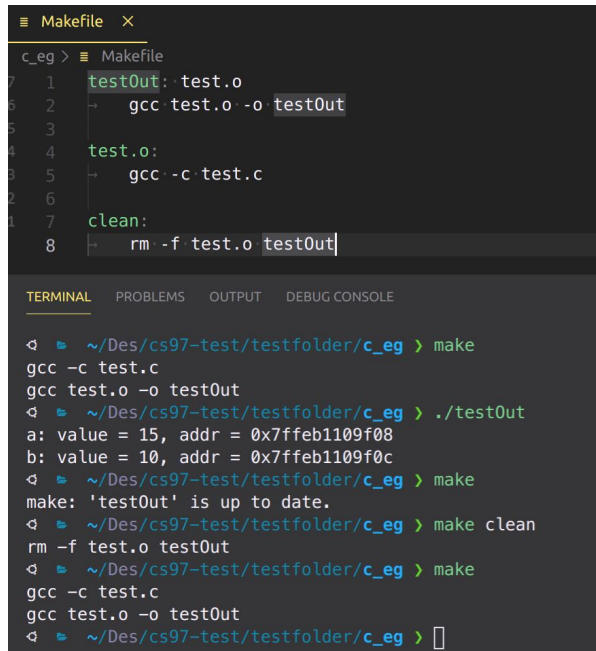
- Dependencies (prerequisite):

- Files used to create the target

- System commands (recipe):

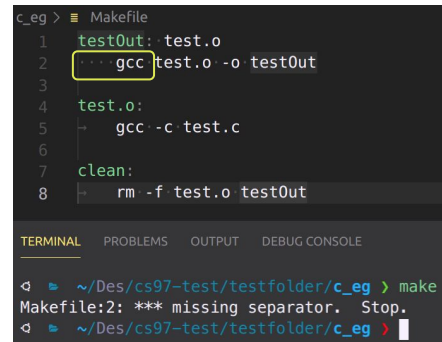
- An action that make carries out
 - can have more than one command

```
target: dependencies
system command(s)
```



```
Makefile X
c_eg > Makefile
1 testOut: test.o
2 gcc test.o -o testOut
3
4 test.o:
5 gcc -c test.c
6
7 clean:
8 rm -f test.o testOut

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE
c_eg > make
gcc -c test.c
gcc test.o -o testOut
c_eg > ./testOut
a: value = 15, addr = 0x7ffeb1109f08
b: value = 10, addr = 0x7ffeb1109f0c
c_eg > make
make: 'testOut' is up to date.
c_eg > make clean
rm -f test.o testOut
c_eg > make
gcc -c test.c
gcc test.o -o testOut
c_eg >
```



```
c_eg > Makefile
1 testOut: test.o
2 gcc test.o -o testOut
3
4 test.o:
5 gcc -c test.c
6
7 clean:
8 rm -f test.o testOut

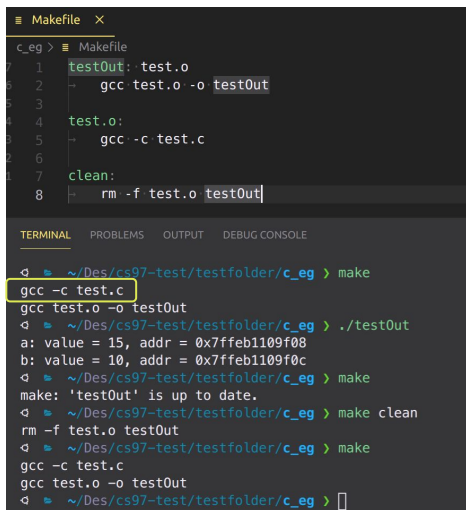
TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE
c_eg > make
Makefile:2: *** missing separator. Stop.
c_eg >
```

Note: we need a real tab instead of spaces in front of the recipe.

Otherwise we will have an error: **Makefile: *** missing separator. Stop.**

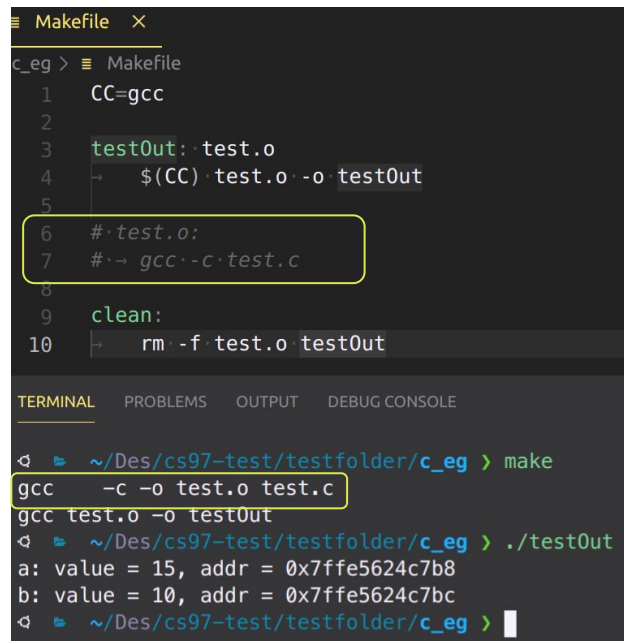
2.2 Makefile Rules

- More elegant makefiles:
 - Define constants
 - `CC=gcc`
 - Make will automatically generate .o files for us
 - Comment out `test.o` part



```
Makefile X
c_eg > Makefile
1 testOut: test.o
2 gcc test.o -o testOut
3
4 test.o:
5 gcc -c test.c
6
7 clean:
8 rm -f test.o testOut

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE
~/Des/cs97-test/testfolder/c_eg > make
gcc -c test.c
gcc test.o -o testOut
~/Des/cs97-test/testfolder/c_eg > ./testOut
a: value = 15, addr = 0x7ffeb1109f08
b: value = 10, addr = 0x7ffeb1109f0c
~/Des/cs97-test/testfolder/c_eg > make
make: 'testOut' is up to date.
~/Des/cs97-test/testfolder/c_eg > make clean
rm -f test.o testOut
~/Des/cs97-test/testfolder/c_eg > make
gcc -c test.c
gcc test.o -o testOut
~/Des/cs97-test/testfolder/c_eg > 
```



```
Makefile X
c_eg > Makefile
1 CC=gcc
2
3 testOut: test.o
4 $(CC) test.o -o testOut
5
6 # test.o:
7 # gcc -c test.c
8
9 clean:
10 rm -f test.o testOut

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE
~/Des/cs97-test/testfolder/c_eg > make
gcc -c -o test.o test.c
gcc test.o -o testOut
~/Des/cs97-test/testfolder/c_eg > ./testOut
a: value = 15, addr = 0x7ffe5624c7b8
b: value = 10, addr = 0x7ffe5624c7bc
~/Des/cs97-test/testfolder/c_eg > 
```


2.2 Makefile Rules

- Add more targets:
 - Default: a convention to specify the default target
 - **default: testOut**
 - All: a convention to specify all targets if there are multiple
 - **All: testOut, testMallocOut**

```
Makefile X
c_eg > Makefile
1 CC=gcc
2
3 default: testOut
4 testOut: test.o
5 → $(CC) test.o -o testOut
6
7 testMallocOut: testMalloc.o
8 → $(CC) testMalloc.o -o testMallocOut
9
10 clean:
11 → rm -f *.o testOut testMallocOut

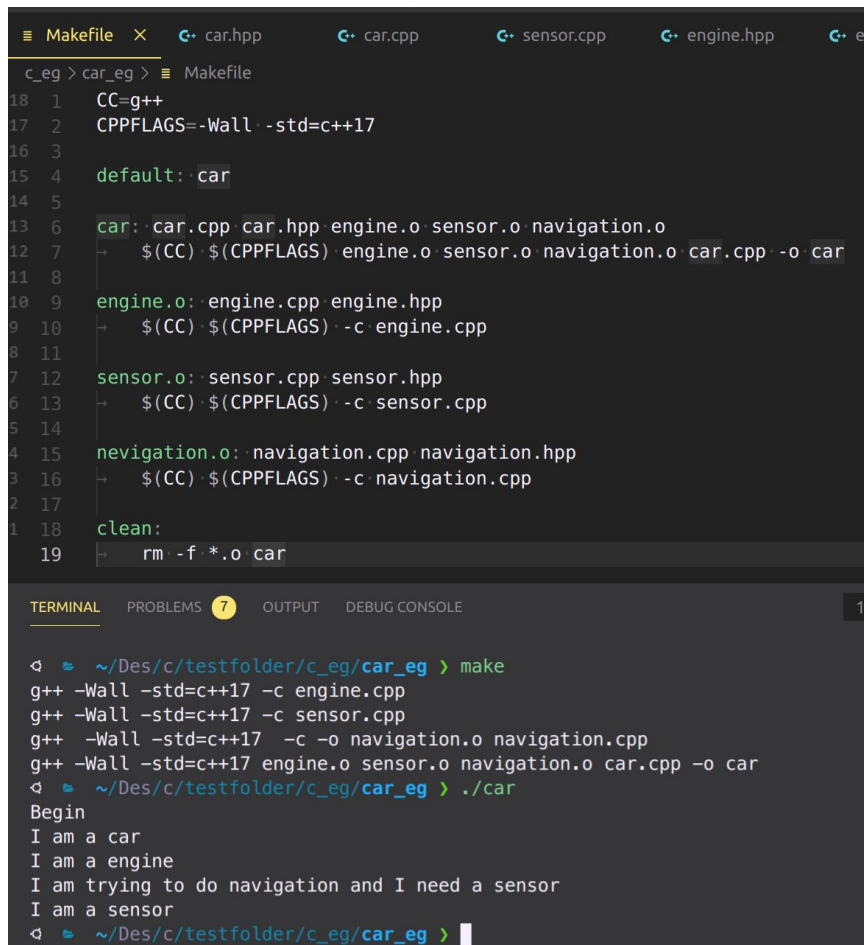
TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE
< ~ /Des/cs97-test/testfolder/c_eg > make
gcc -c -o test.o test.c
gcc test.o -o testOut
< ~ /Des/cs97-test/testfolder/c_eg > ./testOut
a: value = 15, addr = 0x7fff208d48d8
b: value = 10, addr = 0x7fff208d48dc
< ~ /Des/cs97-test/testfolder/c_eg > make testMallocOut
gcc -c -o testMalloc.o testMalloc.c
gcc testMalloc.o -o testMallocOut
< ~ /Des/cs97-test/testfolder/c_eg > ./testMallocOut
Name = Zara Ali
Description: Zara ali a DPS student.She is in class 10th
< ~ /Des/cs97-test/testfolder/c_eg >
```

```
Makefile X
c_eg > Makefile
2 1 CC=gcc
3
3 all: testOut testMallocOut
4 testOut: test.o
5 → $(CC) test.o -o testOut
6
7 testMallocOut: testMalloc.o
8 → $(CC) testMalloc.o -o testMallocOut
9
10 clean:
11 → rm -f *.o testOut testMallocOut

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE
< ~ /Des/cs97-test/testfolder/c_eg > make
gcc -c -o test.o test.c
gcc test.o -o testOut
gcc -c -o testMalloc.o testMalloc.c
gcc testMalloc.o -o testMallocOut
< ~ /Des/cs97-test/testfolder/c_eg > ./testOut
a: value = 15, addr = 0x7ffe84ebc3b8
b: value = 10, addr = 0x7ffe84ebc3bc
< ~ /Des/cs97-test/testfolder/c_eg > ./testMallocOut
Name = Zara Ali
Description: Zara ali a DPS student.She is in class 10th
< ~ /Des/cs97-test/testfolder/c_eg >
```

2.2 Makefile Rules

- Another example
 - “Car” Program written in C++
 - Car.cpp
 - (#include car.hpp, engine.hpp, sensor.hpp, navigation.hpp)
 - Navigation.cpp
 - (#include navigation.hpp, sensor.hpp)
 - Engine.cpp
 - (#include engine.hpp)
 - Sensor.cpp
 - (#include sensor.hpp)



The screenshot shows a code editor with a Makefile and its execution in a terminal. The Makefile is located in the 'c_eg' directory and defines the 'car' target. The terminal shows the execution of 'make' and './car', which produces the output: 'I am a car', 'I am a engine', 'I am trying to do navigation and I need a sensor', and 'I am a sensor'.

```
Makefile
18 1 CC=g++
17 2 CPPFLAGS=-Wall -std=c++17
16 3
15 4 default: car
14 5
13 6 car: car.cpp car.hpp engine.o sensor.o navigation.o
12 7     $(CC) $(CPPFLAGS) engine.o sensor.o navigation.o car.cpp -o car
11 8
10 9 engine.o: engine.cpp engine.hpp
9 10     $(CC) $(CPPFLAGS) -c engine.cpp
8 11
7 12 sensor.o: sensor.cpp sensor.hpp
6 13     $(CC) $(CPPFLAGS) -c sensor.cpp
5 14
4 15 navigation.o: navigation.cpp navigation.hpp
3 16     $(CC) $(CPPFLAGS) -c navigation.cpp
2 17
1 18 clean:
19     rm -f *.o car

TERMINAL
~/Des/c/testfolder/c_eg/car_eg > make
g++ -Wall -std=c++17 -c engine.cpp
g++ -Wall -std=c++17 -c sensor.cpp
g++ -Wall -std=c++17 -c -o navigation.o navigation.cpp
g++ -Wall -std=c++17 engine.o sensor.o navigation.o car.cpp -o car
~/Des/c/testfolder/c_eg/car_eg > ./car
Begin
I am a car
I am a engine
I am trying to do navigation and I need a sensor
I am a sensor
~/Des/c/testfolder/c_eg/car_eg >
```