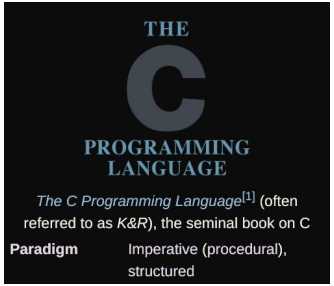
CS 35L
Discussion 1A
Week 6

C and Makefile

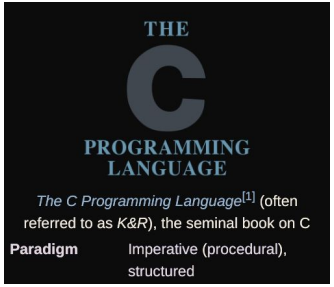
Contents

1.1 C - Brief Intro

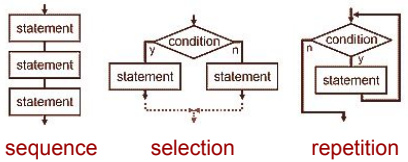


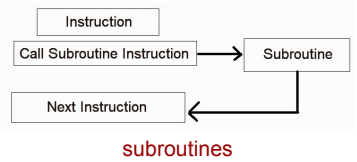
- C Programming Language
  - C was originally developed at Bell Labs by Dennis Ritchie between 1972 and 1973 to construct running on Unix (improved from Programming Language B)
  - It was applied to re-implementing the kernel of the Unix operating system
  - It was designed to be complied to provide 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

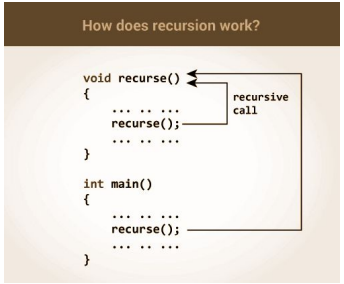


- C is a general-purpose, procedural computer programming language
- C supports
  - Structured programming: (control structures)



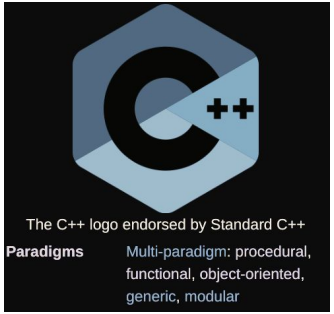


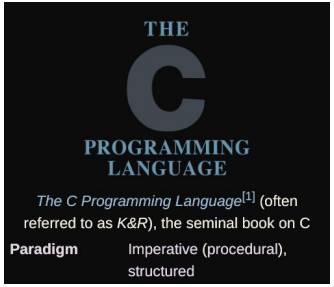
  - Lexical variable scope (static scope)
    - Static scope v.s. Dynamic scope
      - (suppose a variable name's scope is a certain function)
      - Static: within the certain function text, the variable exists
      - Dynamic: while the function is running, the variable exists
  - Recursion
    - Solve 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.
```

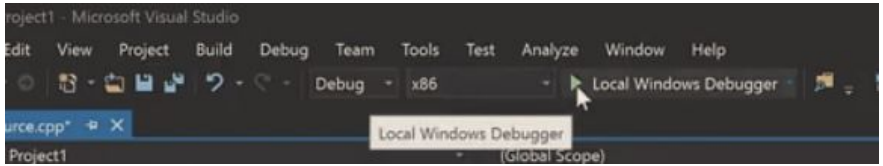
A function call

```
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



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)

Specifications for standard integer types

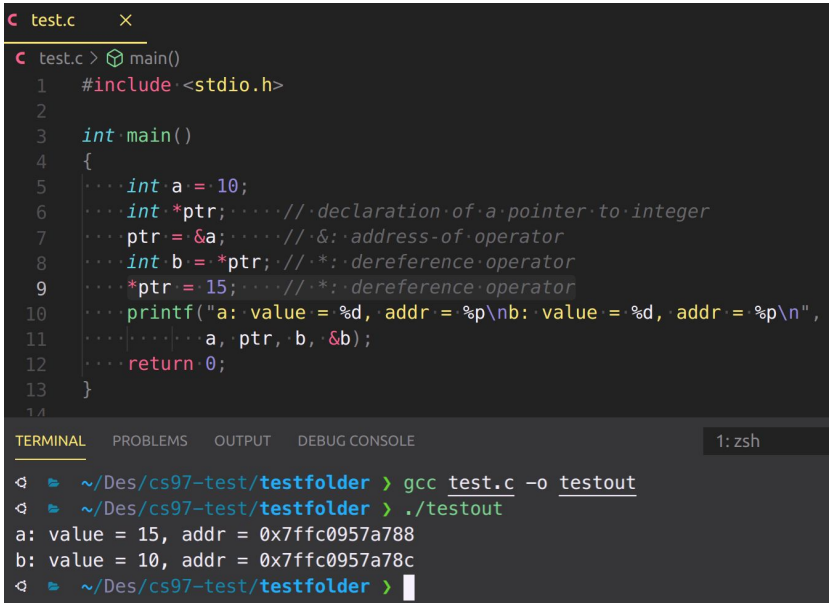
Shortest form of specifier	Minimum width (bits)
`_Bool`	1
`char`	8
`signed char`	8
`unsigned char`	8
`short`	16
`unsigned short`	16
`int`	16
`unsigned int`	16
`long`	32
`unsigned long`	32
`long long` [1]	64
`unsigned long long` [1]	64

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)

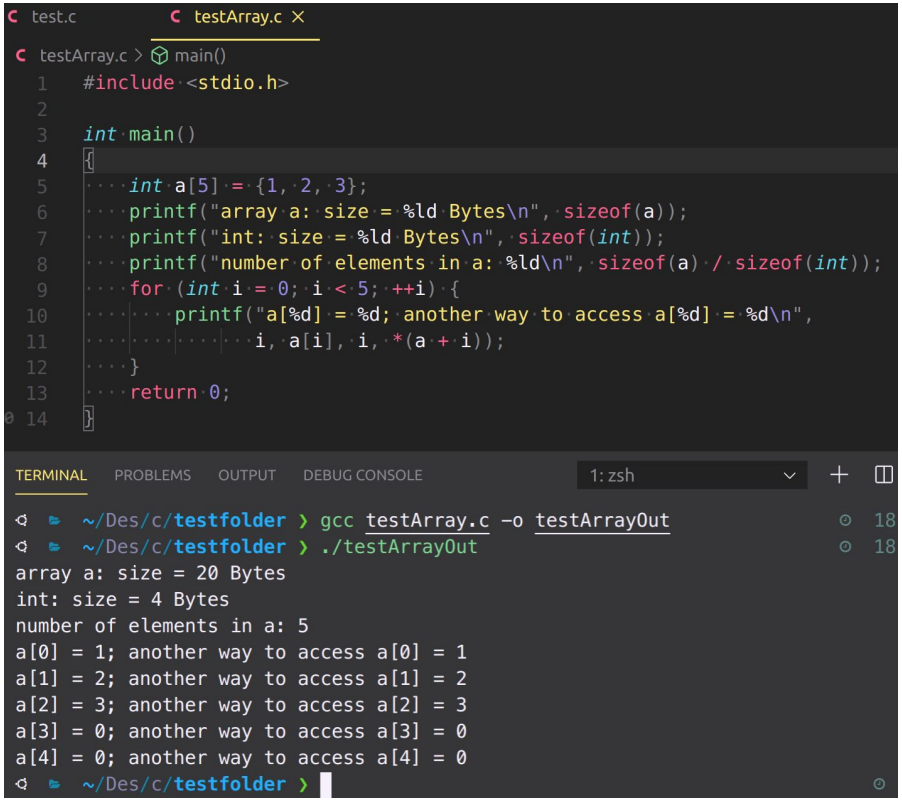
1.3 C Syntax - Data Structures

- 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`



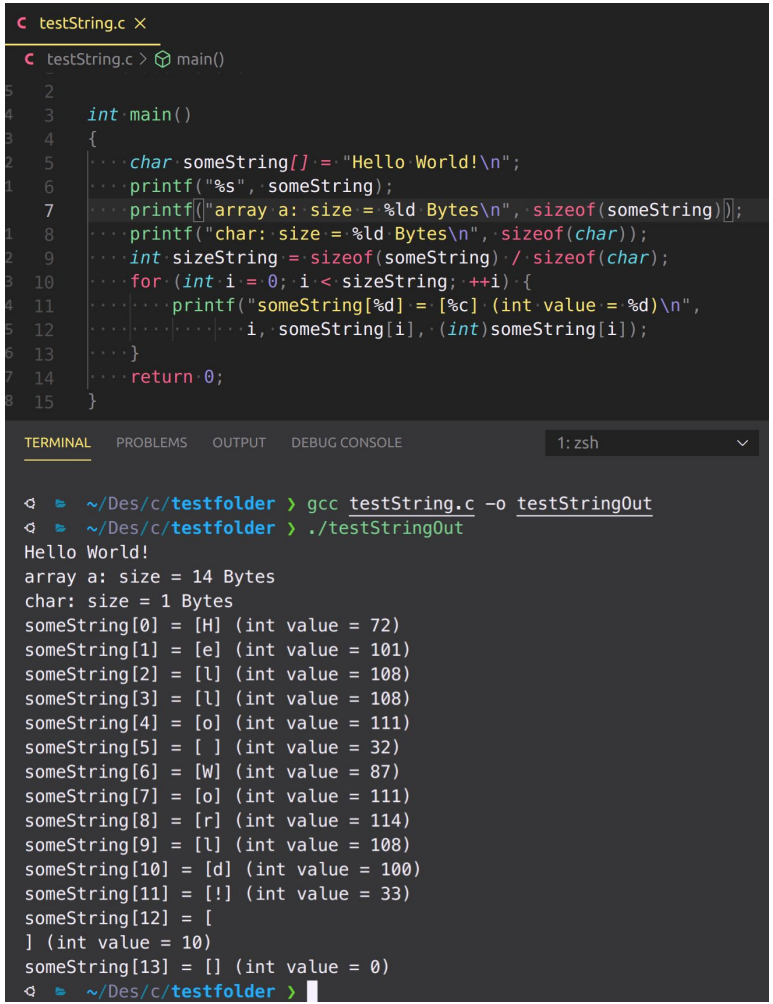
1.3 C Syntax - Data Structures

- 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)



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/

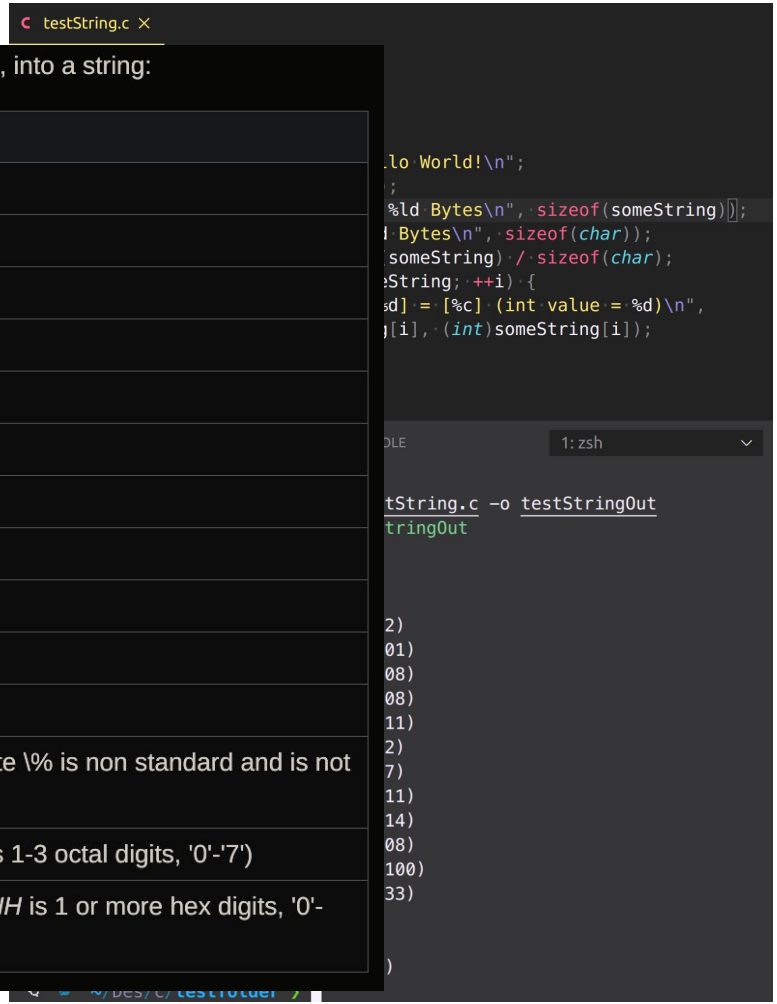


1.3 C Synt

- Strings
  - String litera
    - E.g.:
  - They are co
    values, with
    (0-valued) c
  - char? int?
    - char:
    - Store
      -
    - ASCI
      Infor
      https
  - #include <
    - More
      C str
      - https://

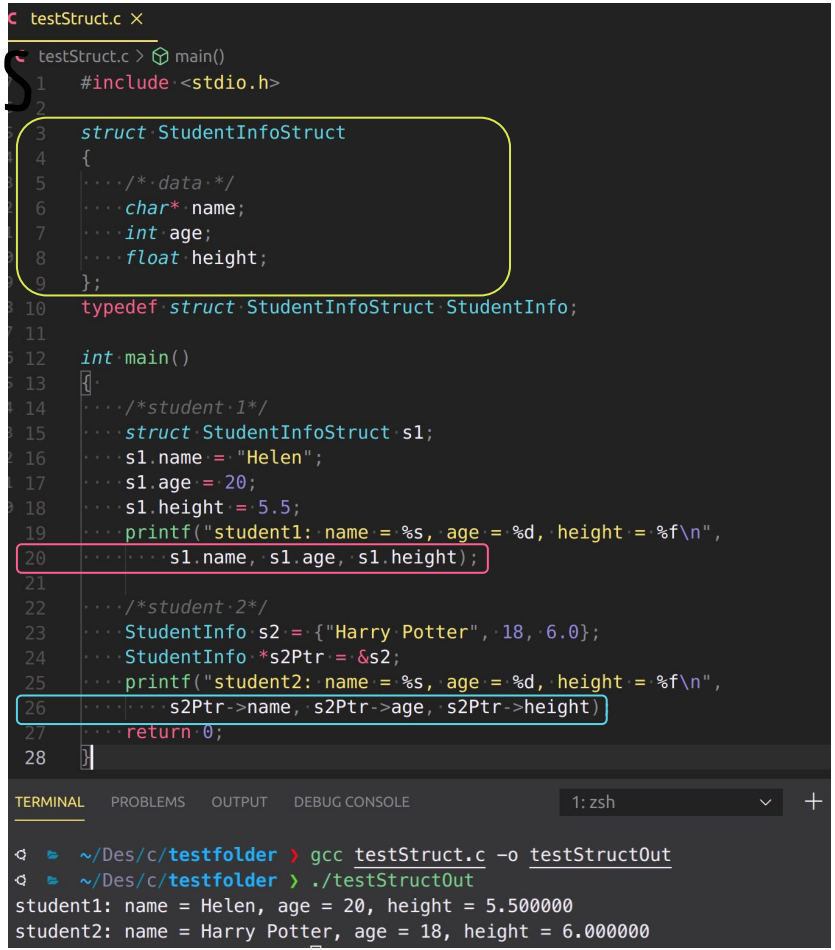
Backslashes may be used to enter control characters, etc., into a string:

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



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



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

https://en.wikipedia.org/wiki/Operators_in_C_and_C

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>
```

https://en.wikipedia.org/wiki/C_syntax#Control_structures

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**.

https://www.tutorialspoint.com/cprogramming/c_memory_management.html

1.4 C - Dynamic Memory Mar

- Recall: The definition of a **fixed-size** array
  - `int a[100];`
  - But now, let's consider a situation: no idea about th
- 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 w
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**.

testMalloc.c

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

int main() {

   char name[100];
   char *description;

   strcpy(name, "Zara Ali");

   /* allocate memory dynamically */
   description = malloc( 30 * sizeof(char) );

   if( description == NULL ) {
      fprintf(stderr, "Error - unable to allocate required memory\n");
   } else {
      strcpy( description, "Zara ali a DPS student.");
   }

   /* suppose you want to store bigger description */
   description = realloc( description, 100 * sizeof(char) );

   if( description == NULL ) {
      fprintf(stderr, "Error - unable to allocate required memory\n");
   } else {
      strcat( description, "She is in class 10th");
   }

   printf("Name = %s\n", name );
   printf("Description: %s\n", description );

   /* release memory using free() function */
   free(description);
   return 0;
}
```

```
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 ❯
```

Example from:
https://www.tutorialspoint.com/cprogramming/c_memory_management.html

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

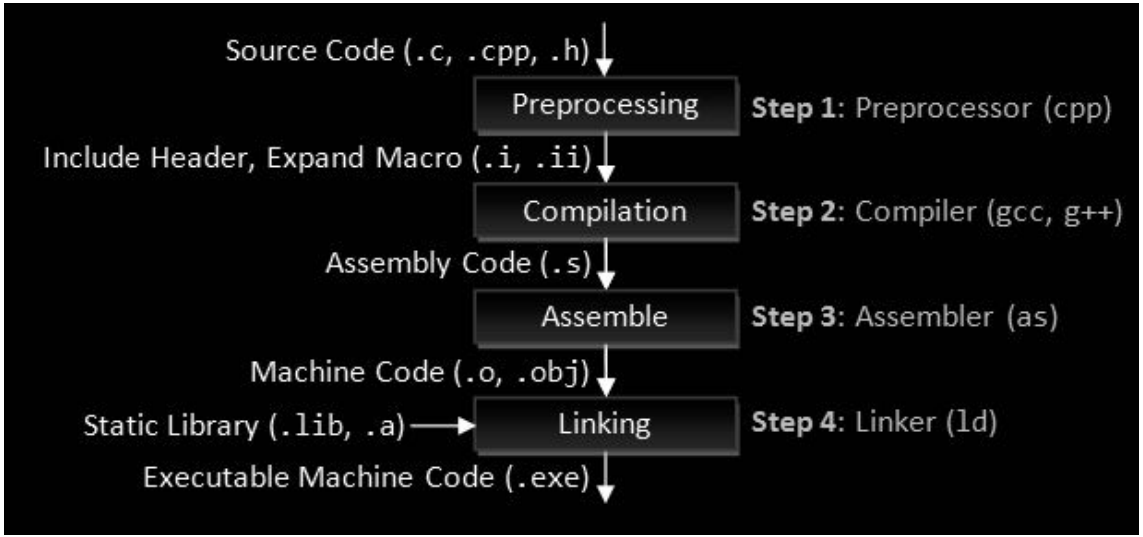


Image credit: https://medium.com/@earthtojhuang/gcc-main-c-9f6ee2e6894b

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?

- 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

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

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

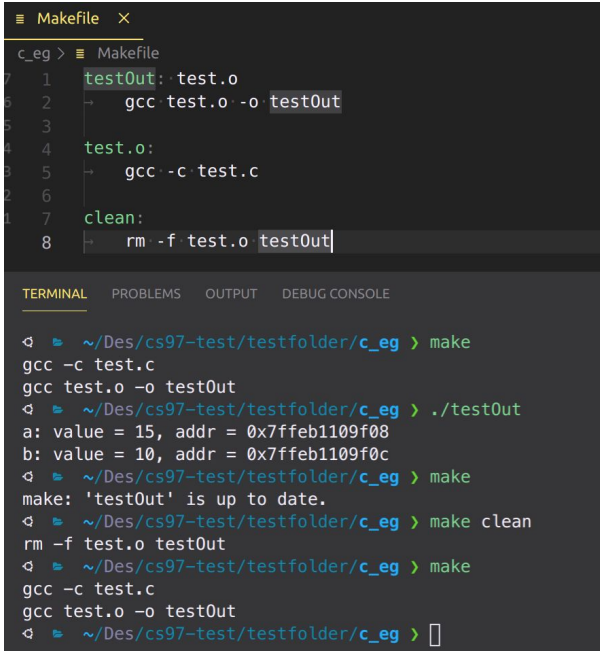
- Content: explicit rules, implicit rules, variable definitions, directives, and comments.
  - Explicit rule: when and how to remake one or more files, called the rule's targets. It lists the other files that the targets depend on, and may also give a command list to use to create or update the targets.
  - 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

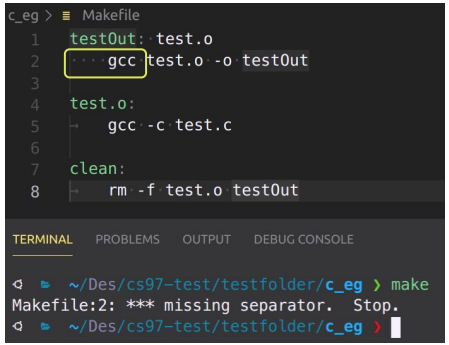
```
target: dependencies
    system command(s)
```

  - 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



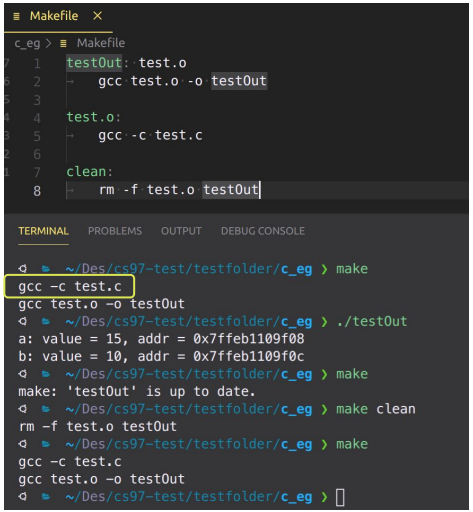
Note: we need a real tab instead of spaces in front of the recipe.

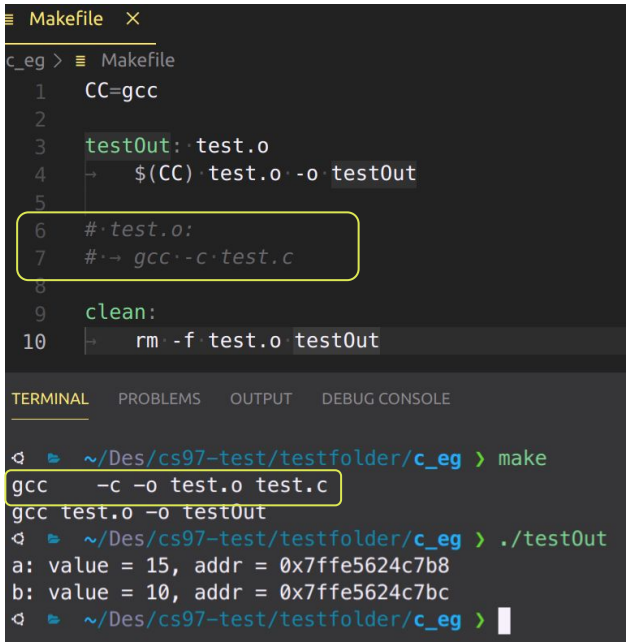
Otherwise we will have an error: **`Makefile: *** missing separator.  Stop.`**



2.2 Makefile Rules

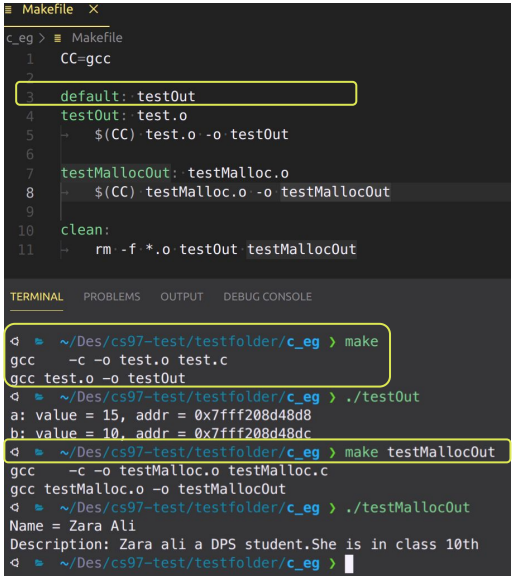
- More elegant makefiles:
  - Define constants
    - `CC=gcc`
  - Make will automatically generate .o files for us
    - Comment out `test.o` part

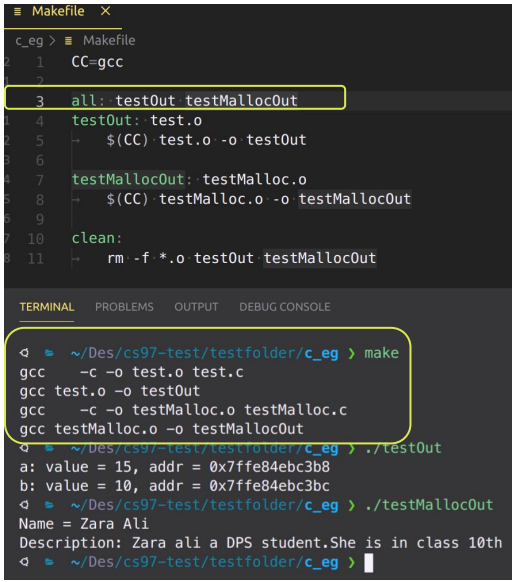




2.2 Makefile Rules

- 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`





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)