# COMP1521

Week 1 - Memory & C revision

## Introductions

Name (optionally pronouns)

Degree

Highlight from the break

Preference for chocolate: Twirl, Cadbury Milk or Crunchie?

## Overview

Admin

Programming tools

Memory (Q2, Q3, Q4)

Compilation (Q8)

C revision (Q5, Q6, Q7)

MIPS intro

## Admin

Assessment:

- Weekly labs (15%)
- Weekly tests (10%) (beginning in Week 3)
- Assignments (2 * 15%)
- Final (45%)

Challenge lab exercises:

- Complete 2 weeks worth to get full marks for the lab component

  (Assuming you get full marks for everything)

Lab marking:

- Style is not marked for labs

## Admin

Submission of work

- All work is submitted on the command line using give
- All required commands are provided in lab/assignment specs
- No work is submitted via Edstem (I think 1511 might use it, but not in 1521)

## Admin

Course website tour: https://cgi.cse.unsw.edu.au/~cs1521/24T2/

## Tools

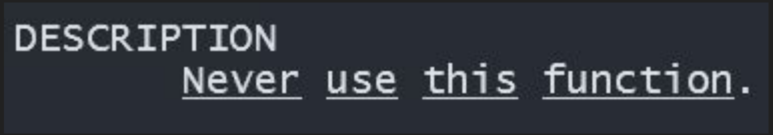
# Man pages (manual pages)

$ man 3 [name]

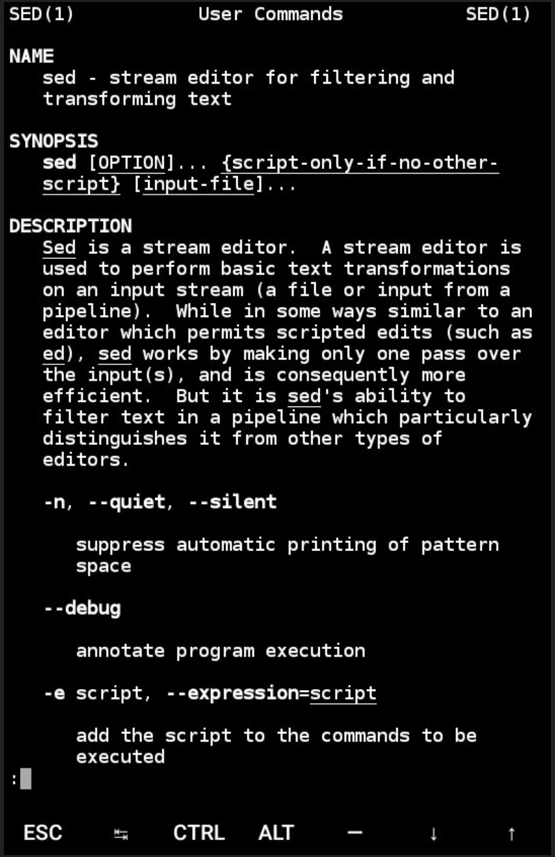
(3 = C library functions, see “man man” for more detail)

Contains:

- Arguments
- Return values
- What it does
- Sometimes example of usage



```
DESCRIPTION
       Never use this function.
```



```
SED(1)            User Commands            SED(1)

NAME
   sed - stream editor for filtering and
   transforming text

SYNOPSIS
   sed [OPTION]... {script-only-if-no-other-
   script} [input-file]...

DESCRIPTION
   Sed is a stream editor.  A stream editor is
   used to perform basic text transformations
   on an input stream (a file or input from a
   pipeline).  While in some ways similar to an
   editor which permits scripted edits (such as
   ed), sed works by making only one pass over
   the input(s), and is consequently more
   efficient.  But it is sed's ability to
   filter text in a pipeline which particularly
   distinguishes it from other types of
   editors.

   -n, --quiet, --silent

      suppress automatic printing of pattern
      space

   --debug

      annotate program execution

   -e script, --expression=script

      add the script to the commands to be
      executed
:
```

## Makefiles

Instructions stored in Makefile

Convenient way to compile C programs.

Compile a single program:

“make program_name”

Compile all programs:

“make”

Pretty much just runs dcc for you

Not assessable, but useful for the exercises (we provide Makefiles for labs).

## Memory in C

## What is RAM?

Just a big 1D array that your computer can read and write to.

## What is a process?

A smaller array, stored inside your computer's RAM.

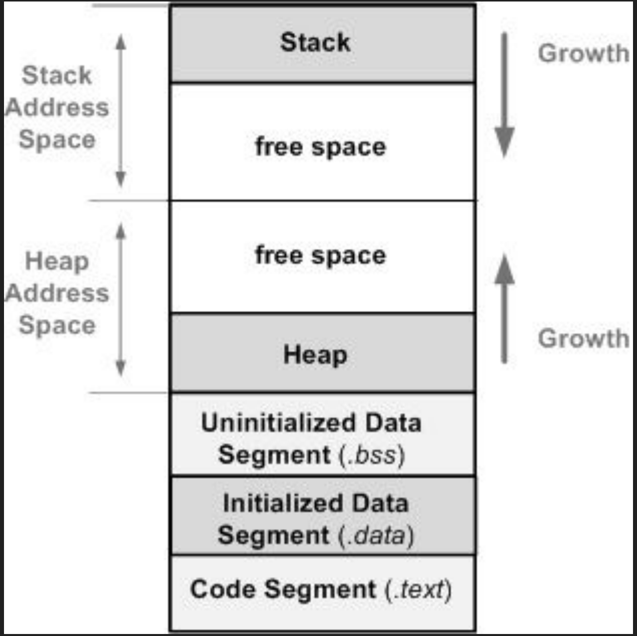
This array has some different segments:

Stack (local vars)

Heap (malloc or other dynamic allocated)

Data (global variables)

Text (code)



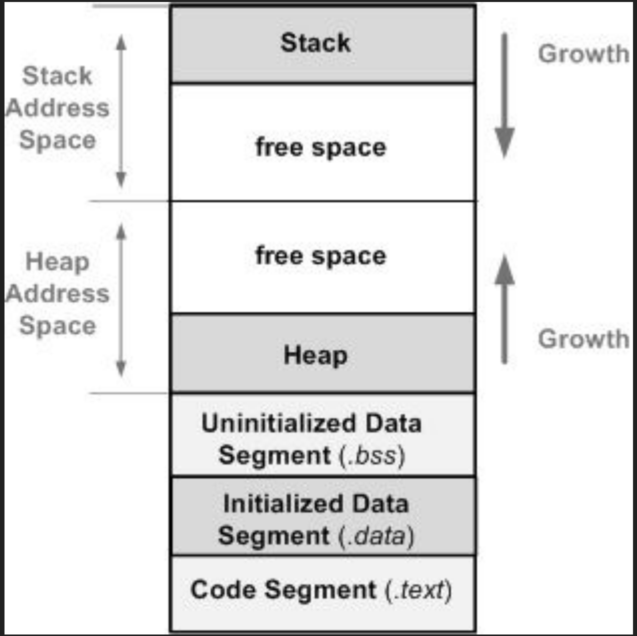
## Stack segment

A stack frame describes a small part of the process array which is used to store information about a function call, like local variables.

A stack frame is always located inside the stack segment.

Every time you call a function, the computer creates a new stack frame to keep track of its variables.

Whenever you return from a function, its stack frame is DESTROYED :(

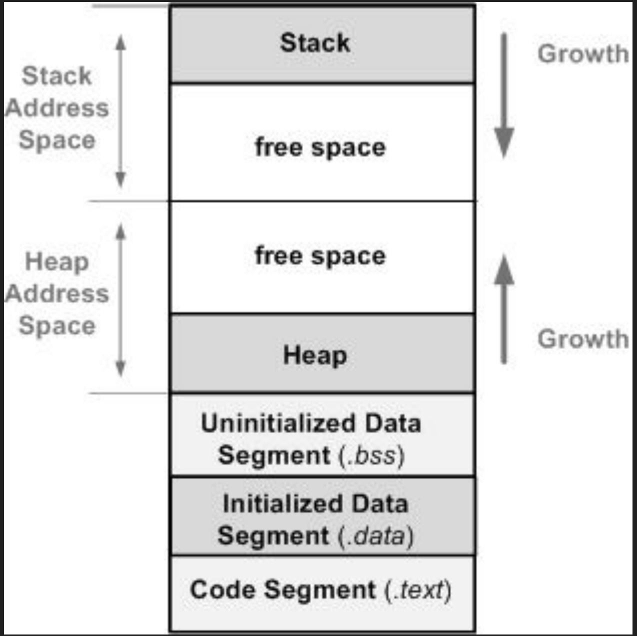


## Heap segment

The heap is a segment used to contain memory which is allocated by the programmer.

If you call malloc, the computer will set aside some heap memory and give you a pointer to it.

Heap memory is only destroyed at the programmer’s request (like with free)

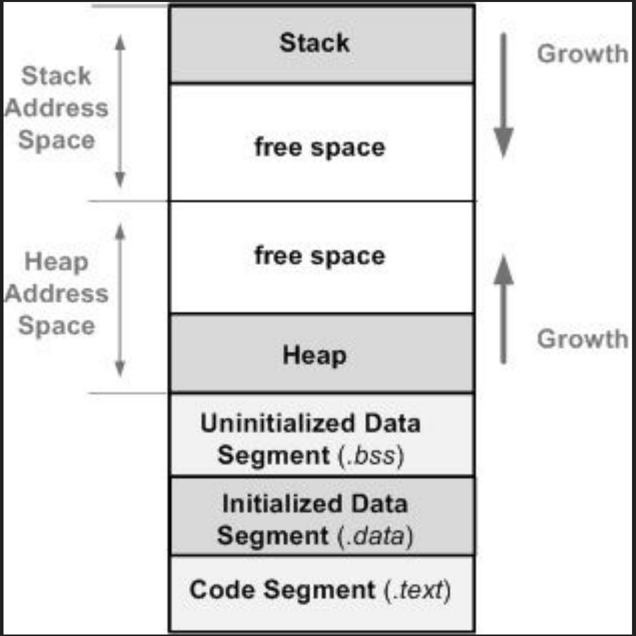


## Data segment

The data segment mainly stores global variables.

Kinda like the heap, but the data segment is static instead of dynamic

(so the data segment won't let you add more stuff at runtime, just whatever globals you defined originally)

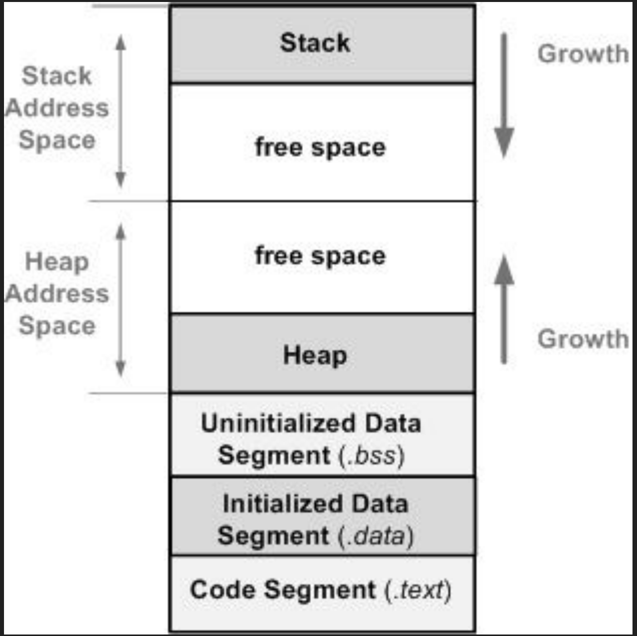


## Text segment

Stores a binary representation of your process’s code.

This binary representation is produced by a compiler.

There is an “entry” address inside the text segment where the CPU will start reading the binary code from your program, and running it using the CPU hardware.

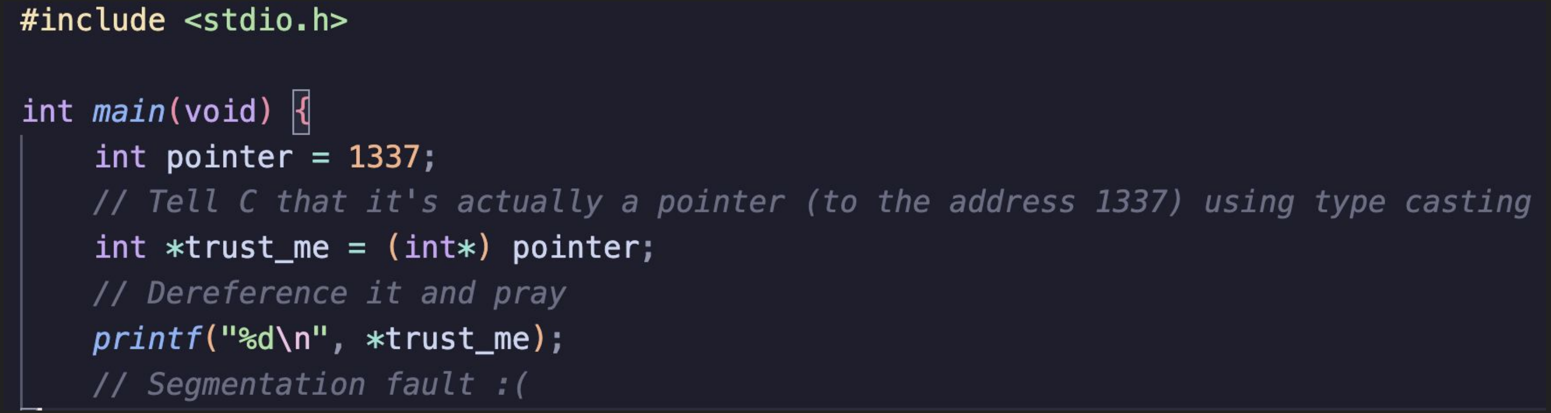


## What is a pointer, exactly?

A pointer is simply an index into the process array!

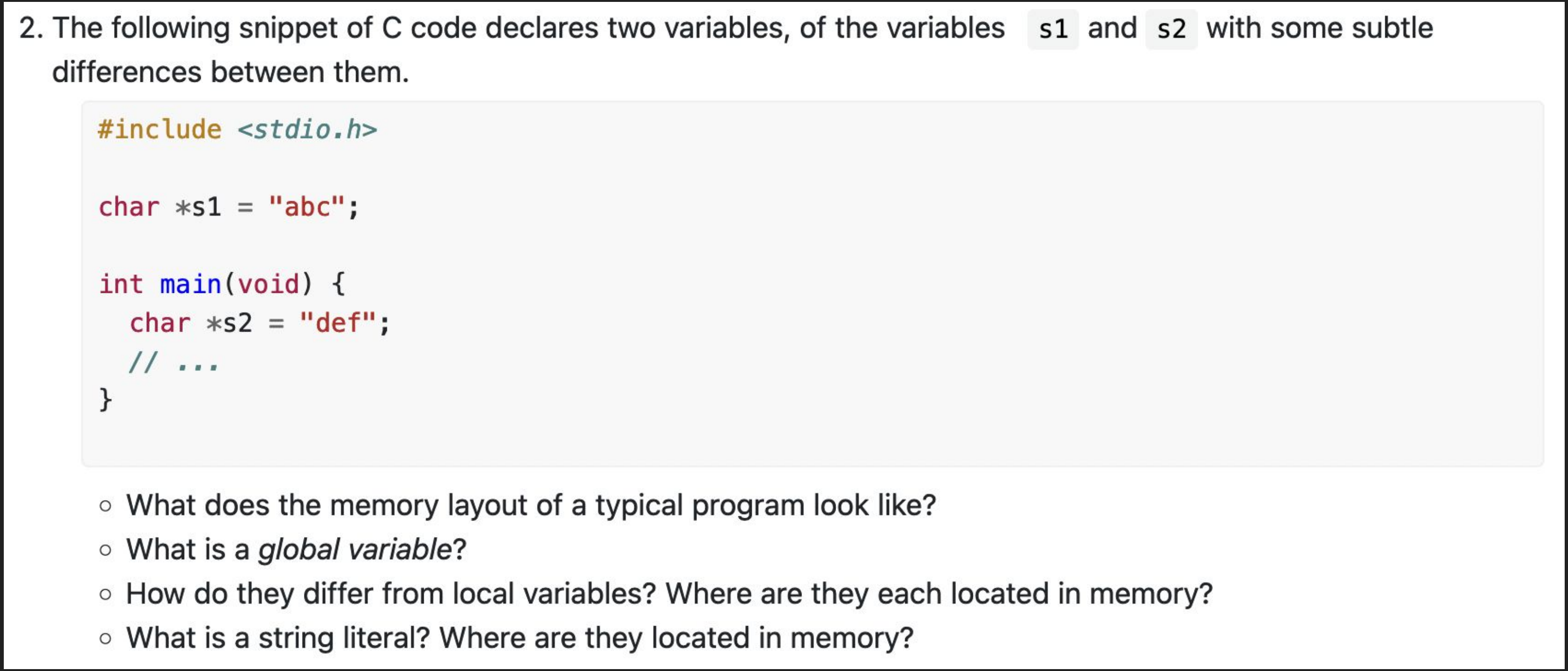
NULL is the index 0.

This is valid C code:



```c
#include <stdio.h>

int main(void) {
    int pointer = 1337;
    // Tell C that it's actually a pointer (to the address 1337) using type casting
    int *trust_me = (int*) pointer;
    // Dereference it and pray
    printf("%d\n", *trust_me);
    // Segmentation fault :(
```

# Tute Questions

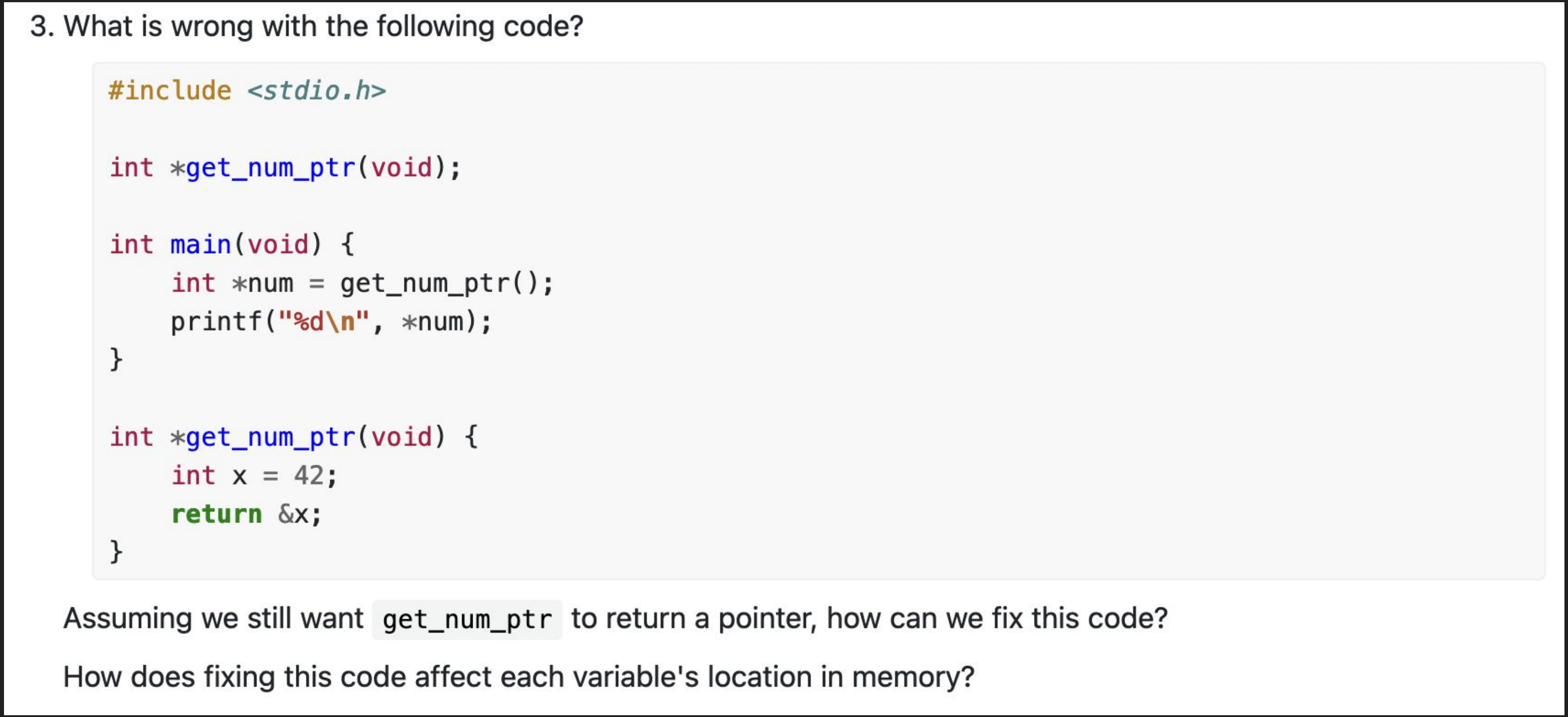


2. The following snippet of C code declares two variables, of the variables `s1` and `s2` with some subtle differences between them.

```c
#include <stdio.h>

char *s1 = "abc";

int main(void) {
  char *s2 = "def";
  // ...
}
```

- What does the memory layout of a typical program look like?
- What is a *global variable*?
- How do they differ from local variables? Where are they each located in memory?
- What is a string literal? Where are they located in memory?

## Tute Questions



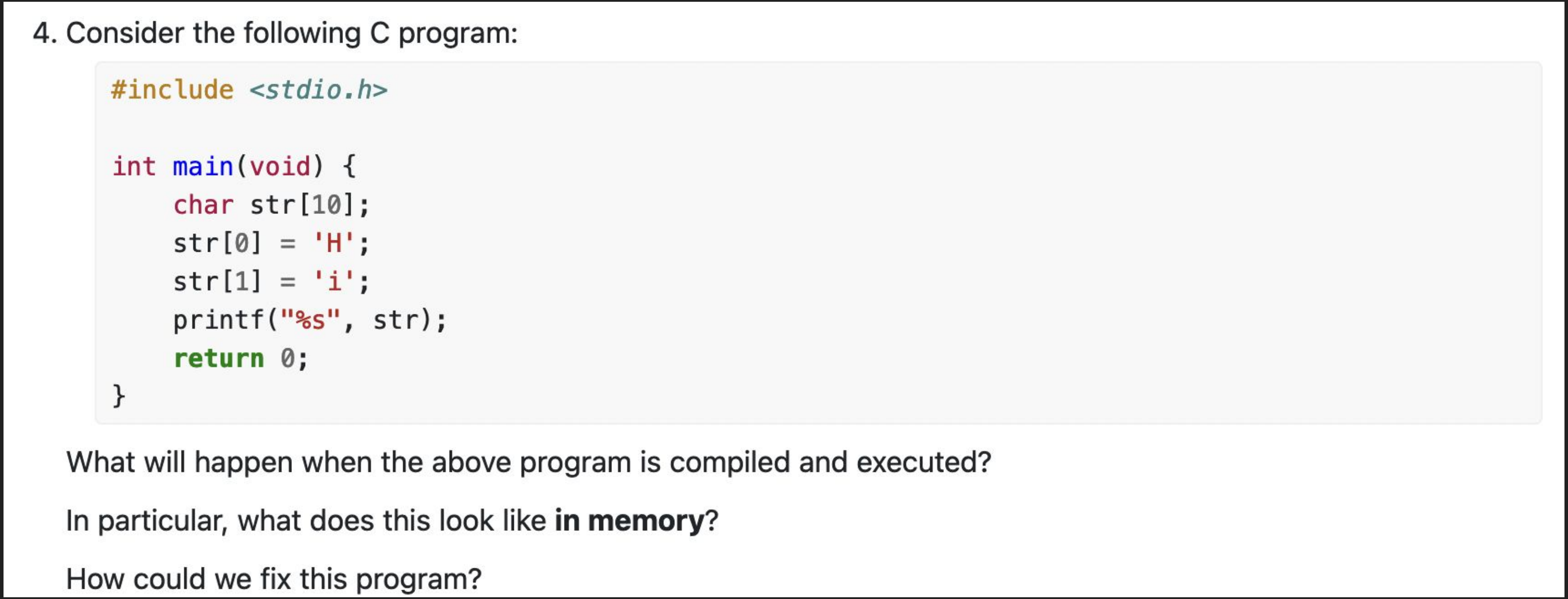
3. What is wrong with the following code?

```c
#include <stdio.h>

int *get_num_ptr(void);

int main(void) {
    int *num = get_num_ptr();
    printf("%d\n", *num);
}

int *get_num_ptr(void) {
    int x = 42;
    return &x;
}
```

Assuming we still want `get_num_ptr` to return a pointer, how can we fix this code?

How does fixing this code affect each variable's location in memory?

# Tute Questions



4. Consider the following C program:

```c
#include <stdio.h>

int main(void) {
    char str[10];
    str[0] = 'H';
    str[1] = 'i';
    printf("%s", str);
    return 0;
}
```

What will happen when the above program is compiled and executed?

In particular, what does this look like **in memory**?

How could we fix this program?

## Compiling

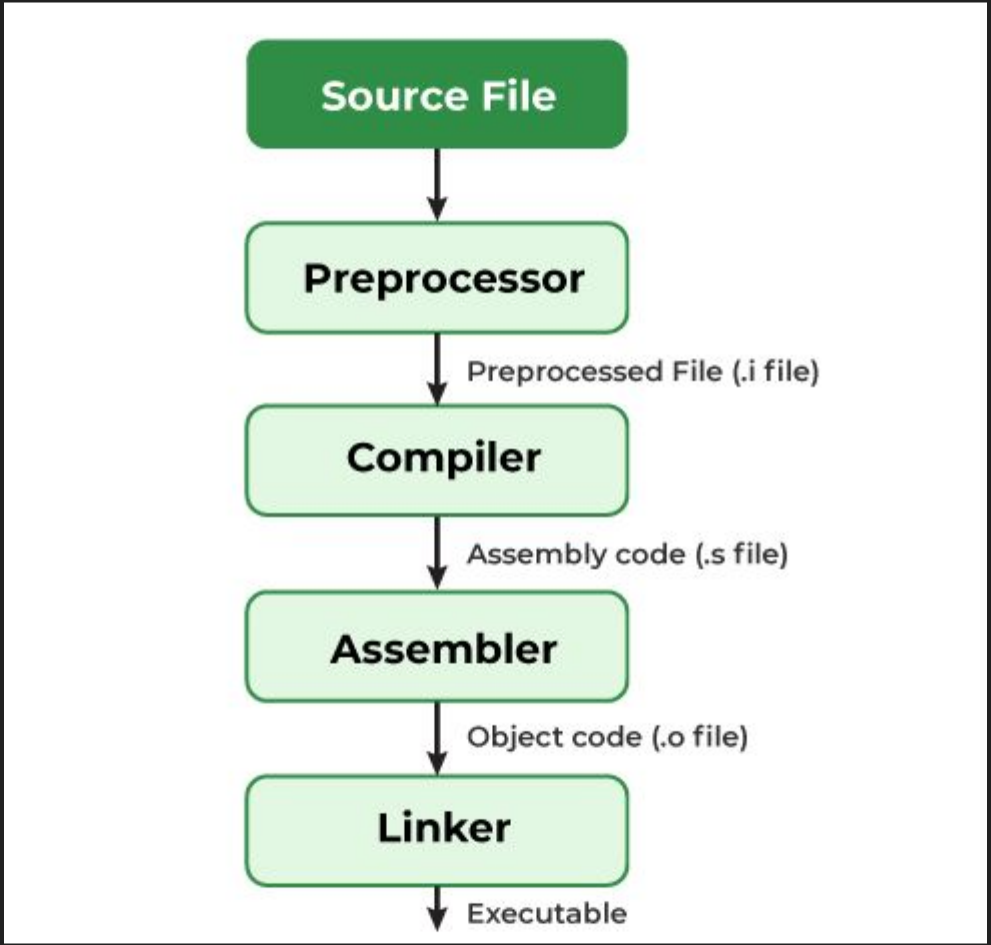
## Steps of a compiler

Preprocessing

Compiling (no way)

Assembling

Linking (dark arts)



## Preprocessing

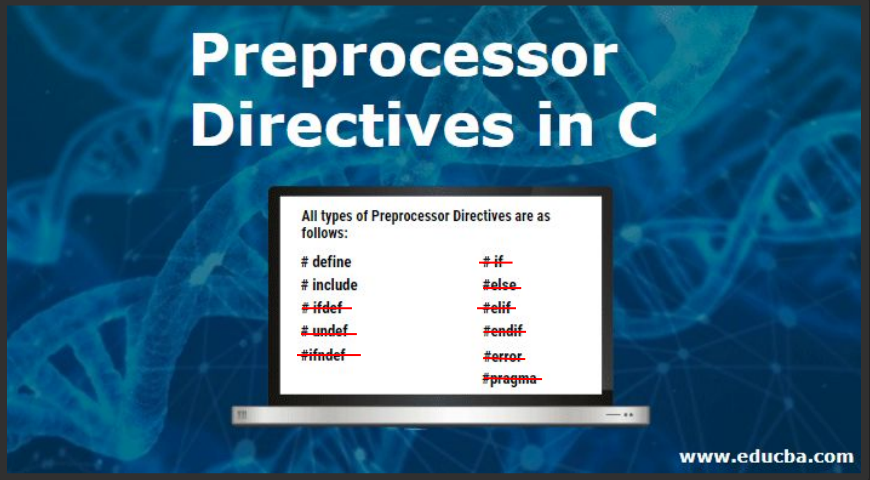
Take care of #defines and macros.

Take of #includes (literally just copy paste the entire included file into this file)

Remove comments (goodbye style)

Other directives which start with ‘#’

(you don’t need to know weird C preprocessor directives for this course)


Preprocessor Directives in C
www.educba.com

All types of Preprocessor Directives are as follows:

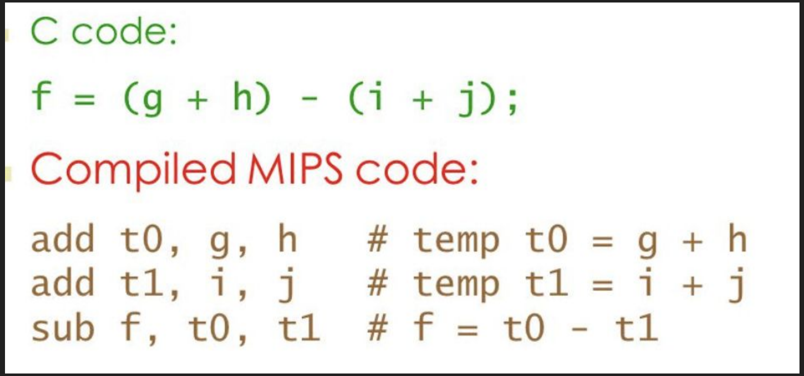
| | |
|---|---|
| # define | ~~# if~~ |
| # include | ~~#else~~ |
| ~~# ifdef~~ | ~~#elif~~ |
| ~~# undef~~ | ~~#endif~~ |
| ~~#ifndef~~ | ~~#error~~ |
| | ~~#pragma~~ |

## Compiling

Turns C code into assembly code which matches your CPU

(for example, produce x86 assembly code for a windows PC, ARM assembly code for a macbook, or MIPS assembly code if you are doing COMP1521)

("Assembly" is just a weird programming language that maps nicely to the hardware inside a CPU.)


C code:

```
f = (g + h) - (i + j);
```

Compiled MIPS code:

```
add t0, g, h    # temp t0 = g + h
add t1, i, j    # temp t1 = i + j
sub f, t0, t1   # f = t0 - t1
```

## Assembling

Turns the compiled assembly code into the binary code we mentioned earlier (for .text section).

It doesn't work yet though, because we don't have a function definition for library functions like "printf".

We need to include the definition of library functions like "printf" in our binary file somehow.

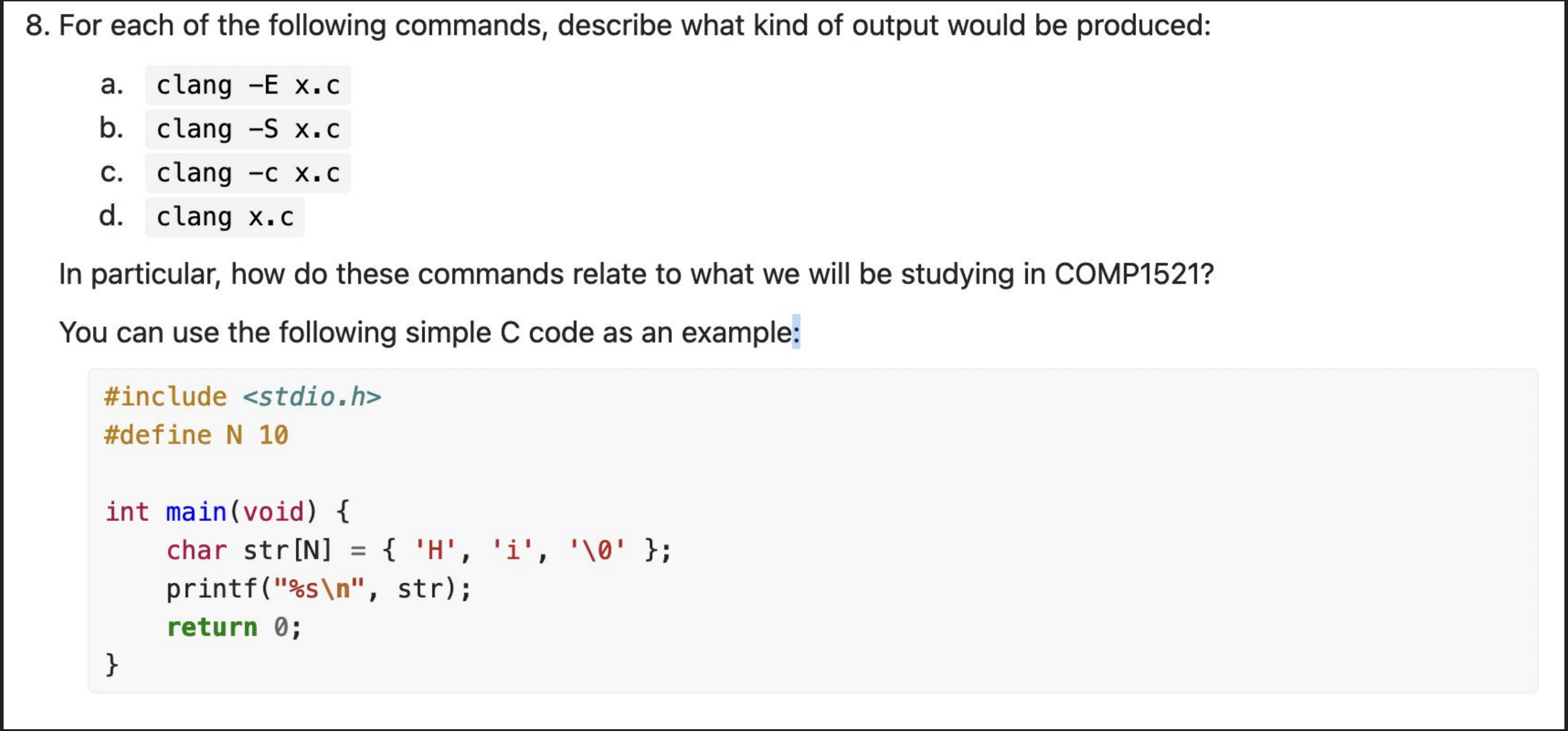
## Linking

Link definitions of library functions to their calls in the binary code.

This is super complicated so we won’t spend time going through exactly how it works.

To generalise, we either link the functions to definitions which exist elsewhere in RAM, or we copy-paste in the definitions themselves into this executable.

## Tute Questions



8. For each of the following commands, describe what kind of output would be produced:

   a. `clang -E x.c`
   b. `clang -S x.c`
   c. `clang -c x.c`
   d. `clang x.c`

   In particular, how do these commands relate to what we will be studying in COMP1521?

   You can use the following simple C code as an example:

```c
#include <stdio.h>
#define N 10

int main(void) {
    char str[N] = { 'H', 'i', '\0' };
    printf("%s\n", str);
    return 0;
}
```

## C revision/new stuff

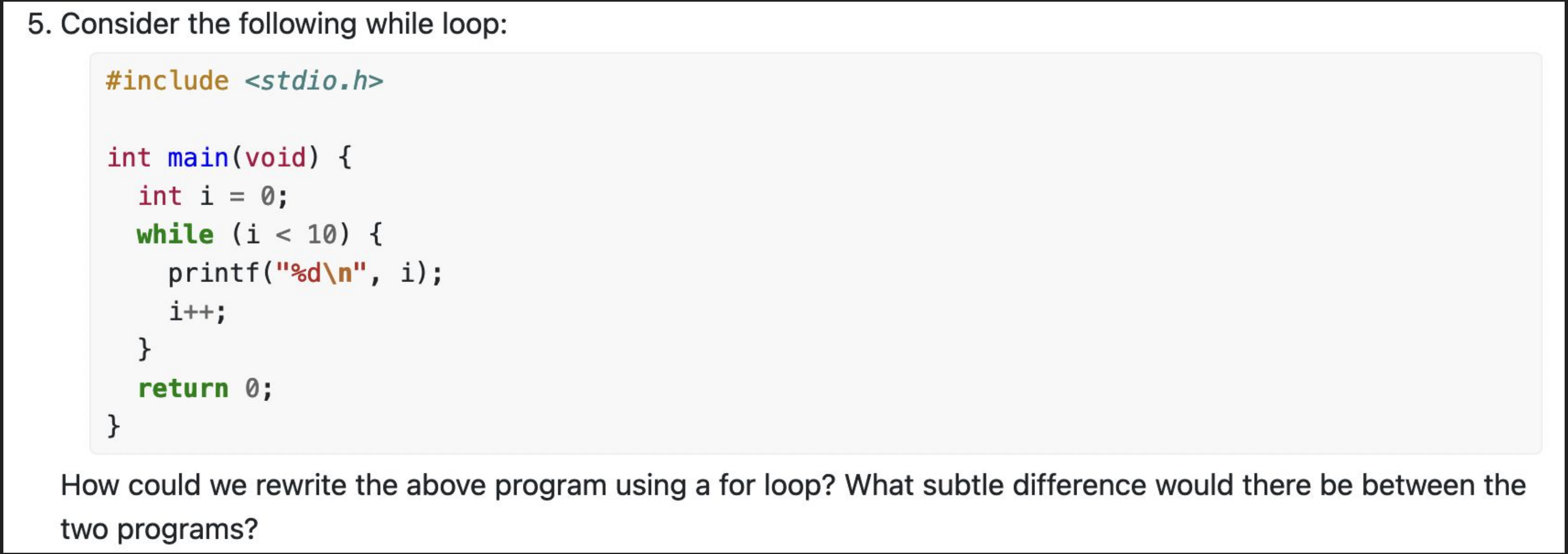
## For loops

```
/* initialize stuff */
while (/* loop condition */ ) {

  /* stuff */


  /* run this at the end of each loop */
}
```

versus

```
for (/* initialize stuff */ ; /* loop condition */ ; /* run this at the end of each loop */) {
  /* stuff */
}
```

## Tute Questions



5. Consider the following while loop:

```c
#include <stdio.h>

int main(void) {
  int i = 0;
  while (i < 10) {
    printf("%d\n", i);
    i++;
  }
  return 0;
}
```

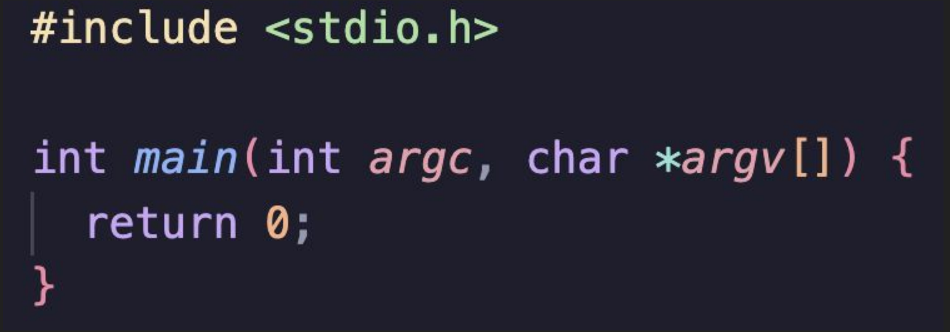
How could we rewrite the above program using a for loop? What subtle difference would there be between the two programs?

# Argc and Argv

Argc = number of command line arguments

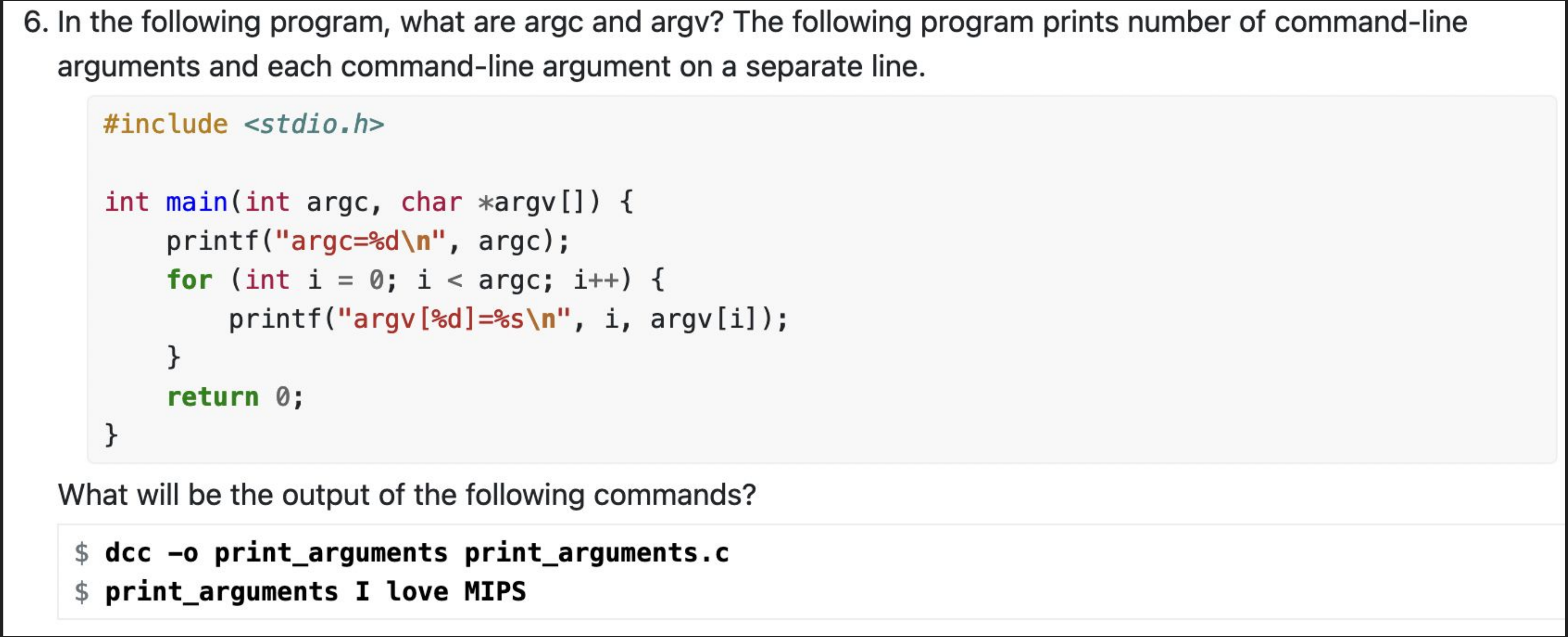
Argv = array containing command line arguments (array of strings)

Note that both include the name of the program as the first argument!!!!!!!!!!



```c
#include <stdio.h>

int main(int argc, char *argv[]) {
  return 0;
}
```

## Tute Questions



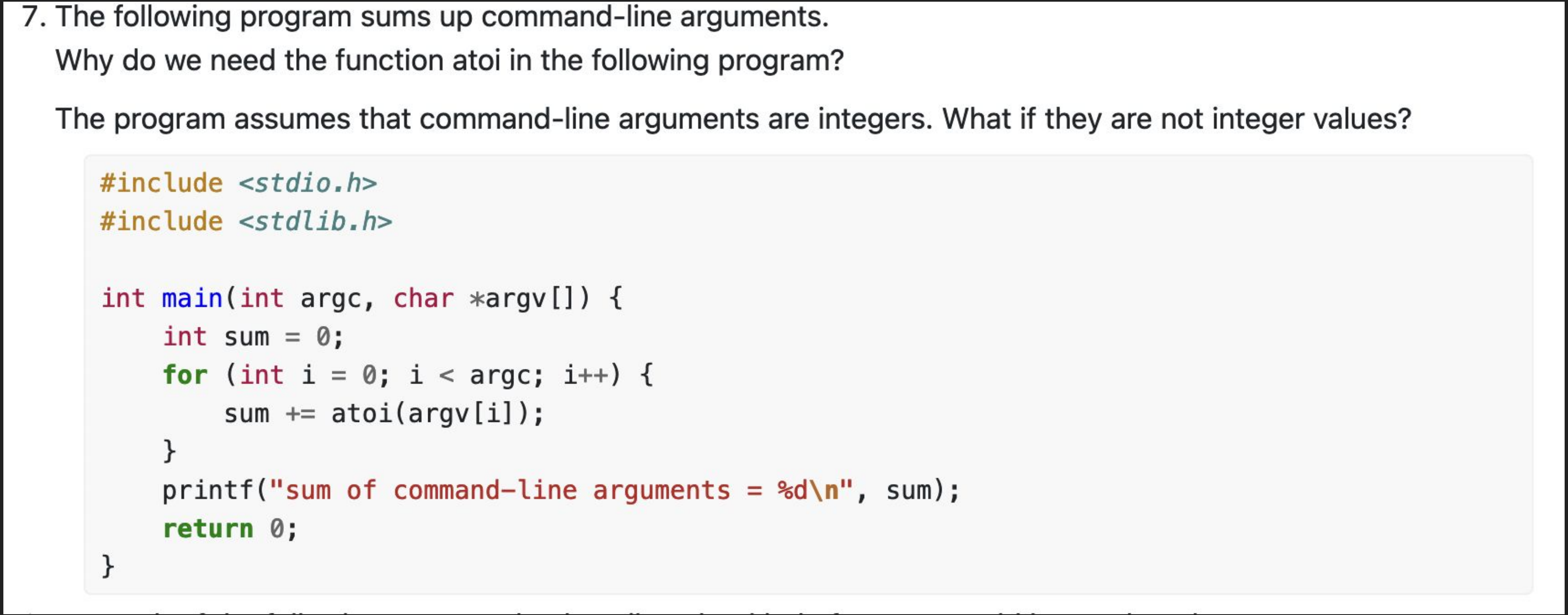
6. In the following program, what are argc and argv? The following program prints number of command-line arguments and each command-line argument on a separate line.

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("argc=%d\n", argc);
    for (int i = 0; i < argc; i++) {
        printf("argv[%d]=%s\n", i, argv[i]);
    }
    return 0;
}
```

What will be the output of the following commands?

```
$ dcc -o print_arguments print_arguments.c
$ print_arguments I love MIPS
```

# Tute Questions



7. The following program sums up command-line arguments.

   Why do we need the function atoi in the following program?

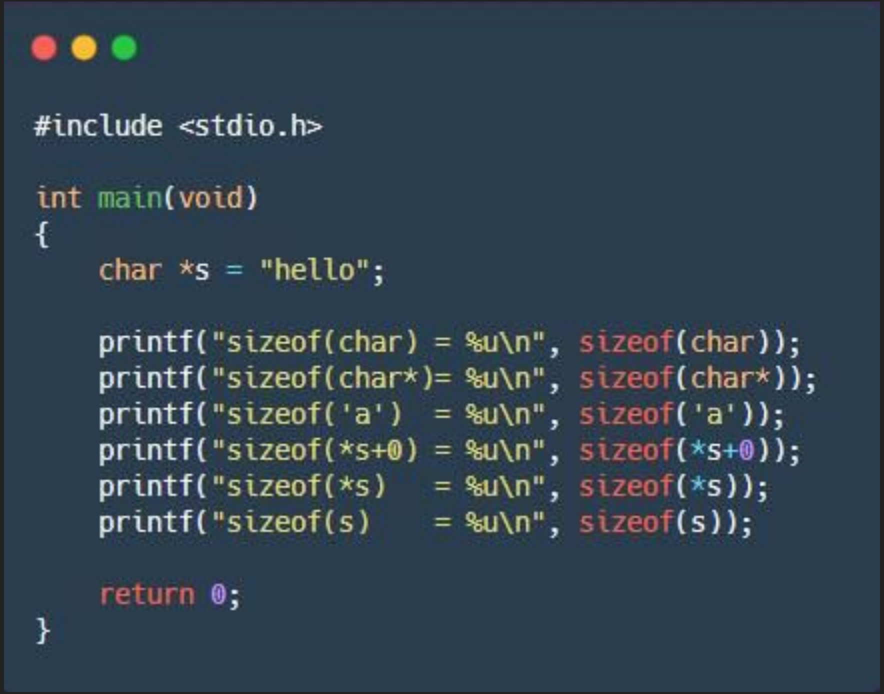
   The program assumes that command-line arguments are integers. What if they are not integer values?

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

int main(int argc, char *argv[]) {
    int sum = 0;
    for (int i = 0; i < argc; i++) {
        sum += atoi(argv[i]);
    }
    printf("sum of command-line arguments = %d\n", sum);
    return 0;
}
```

## Sizeof

Given a variable or a type, sizeof will tell you how many bytes large it is.

Avoid using sizeof on arrays, as it can create confusing behaviour - sizeof will sometimes treat arrays as pointers and incorrectly return the size of the pointer type instead of the size of the entire array.



```c
#include <stdio.h>

int main(void)
{
    char *s = "hello";

    printf("sizeof(char) = %u\n", sizeof(char));
    printf("sizeof(char*)= %u\n", sizeof(char*));
    printf("sizeof('a')  = %u\n", sizeof('a'));
    printf("sizeof(*s+0) = %u\n", sizeof(*s+0));
    printf("sizeof(*s)   = %u\n", sizeof(*s));
    printf("sizeof(s)    = %u\n", sizeof(s));

    return 0;
}
```

## Recursion

Recursion allows programmers to write more simple or concise code to solve certain types of problems.

Sometimes this code can be really confusing to read, compared to an iterative approach (like using a loop). Regardless, in a lot of cases recursion can massively simplify the logic for a program.

## Recursion

How to write a recursive solution.

1: Find a “base case”, or a few. This is where your program will stop “recursing” and typically return a very simple answer.

2: Find a way to solve the other cases. It can be pretty easy, if you can recursively call a “smaller” case and use that information to trivially get a solution.

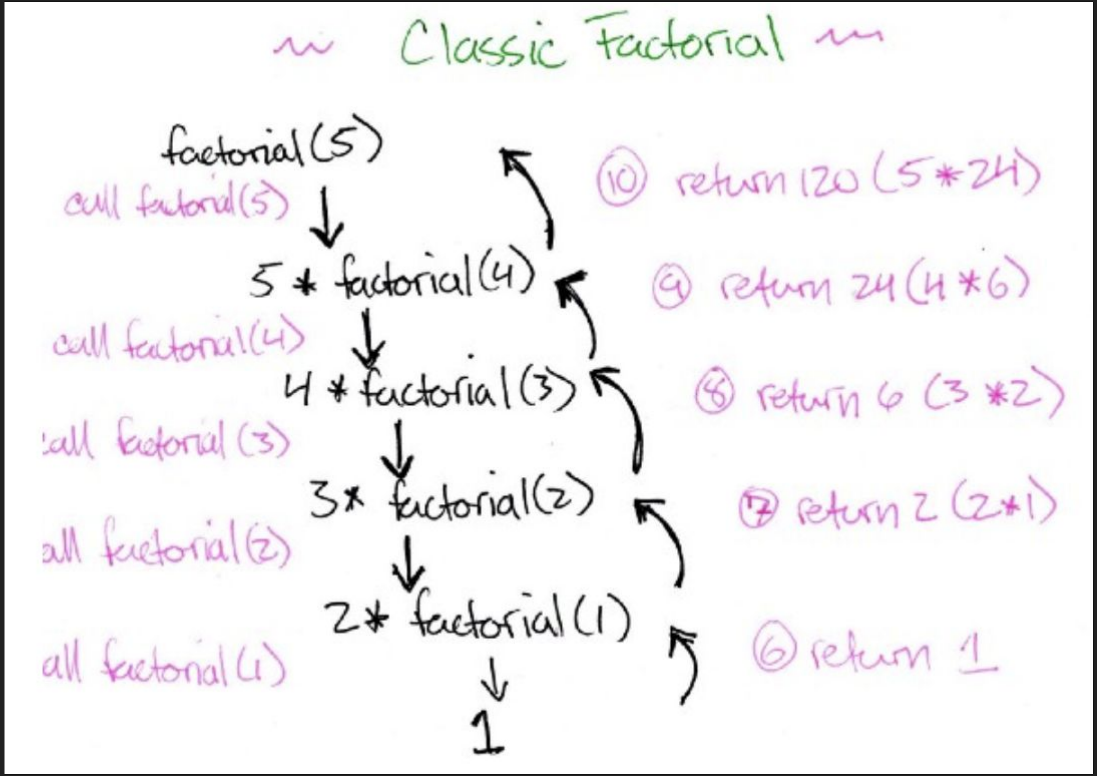
For example, solving “5!”, I can trivially call “4!” and multiply the result by 5.

The “base case” is “0!” - here we stop recursing and we return “1” to whoever called us (likely someone who wanted to calculate “1!”).

# Recursion

Intuition:

We “walk” down to the smallest case, and figure out the solution on the way up.



# MIPS intro

## What is MIPS

Recall that second step of a compiler: transform C code into assembly code for a certain processor.

MIPS is an assembly language which was used in old CPUs, and is now mostly used to teach people about assembly (and program old CPUs).

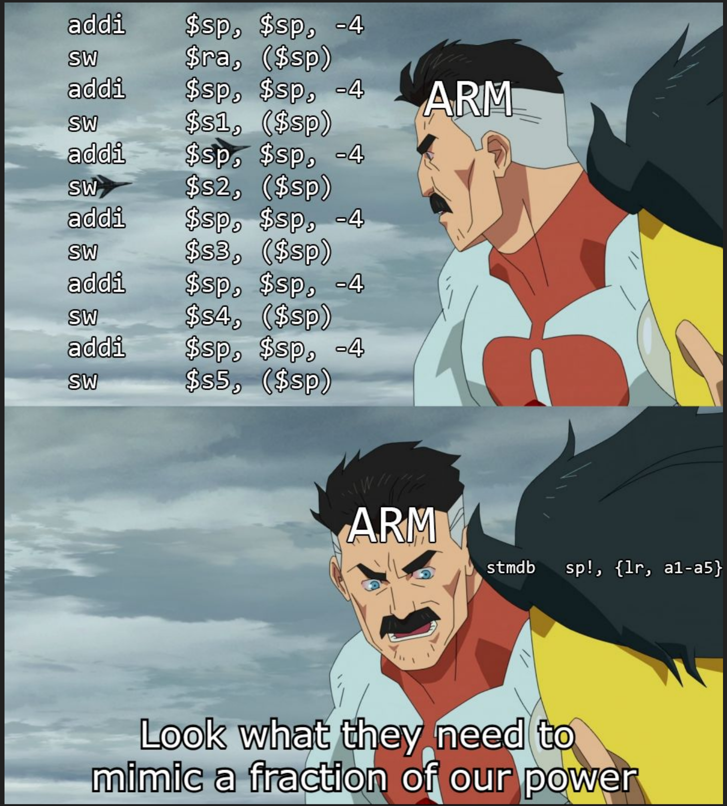
For the first half of this course we will learn to become a compiler, and convert C code into MIPS assembly code by hand.

## Why MIPS, why not something that is used more?

MIPS has great educational resources

Stuff like x86 and ARM can be much more complicated

By learning MIPS first you will have a much easier time learning x86 or ARM if that’s something you are interested in.



```
addi    $sp, $sp, -4
sw      $ra, ($sp)
addi    $sp, $sp, -4
sw      $s1, ($sp)
addi    $sp, $sp, -4
sw      $s2, ($sp)
addi    $sp, $sp, -4
sw      $s3, ($sp)
addi    $sp, $sp, -4
sw      $s4, ($sp)
addi    $sp, $sp, -4
sw      $s5, ($sp)
```

```
stmdb   sp!, {lr, a1-a5}
```