COMP1521

TUTORIAL 1

ICEBREAKERS!

- Name
- Degree and Year
- Boring Fact About Yourself

Assessment Structure

- 9 Sets of Weekly Labs (15%)
- 8 Weekly Tests (10%)
  - First is in week 3 (there is also one during flex week)
  - Best 6 of 8 is taken for total test mark
- 2 Assignments (Each 15%)
  - Assignment 1 is typically a game translated from C to MIPS
  - Assignment 2 is written in C and typically involves bitwise operations and files
- Final Exam (45%)
  - Typically, this has a hurdle of at least 40% mark in final

Getting Full Marks in Labs

- 9 weeks of labs (2 marks each)
  - Normal exercises are worth 1.6
  - Challenge exercises are worth 0.4
- All non-challenge exercises will add up to 14.4/15

- To get 15/15 typically need to complete 2 weeks of challenge exercises
  - Or about 4 challenge exercises total over the term
  - Pick your battles with the challenge exercises some are much more difficult than others!

Submission

- All work is submitted through give using the command line
  - commands are provided for labs/tests/assignments

```
give cs1521 lab01_no_vowels no_vowels.c
```

- Autotests are also provided where applicable (basically everywhere)

```
1521 autotest no_vowels
```

Course Website and Forum

Course Website: https://cgi.cse.unsw.edu.au/~cs1521/25T2/

Course Forum: https://discourse02.cse.unsw.edu.au/25T2/COMP1521/

- Join the forum! Tutors are almost always around to help, and answer questions

Help Sessions

https://cgi.cse.unsw.edu.au/~cs1521/25T2/help-sessions/

- Start in week 3/4 around assignment 1 release
- Swing by for help debugging, lab or assignment help

- Usually, they are in K17 in CSEHELP, or SEM113
  - But always check the room on the schedule!

C REVISION!

Write a c program count_chars.c that uses getchar() to read in characters until the user enters Ctrl-D and then print the total number of characters entered.

Use man 3 getchar to get the manual entry

```c
#include <stdio.h>

int main(void) {

    int ch;
    int count = 0;

    while ((ch = getchar()) != EOF) {
        count += 1;
    }

    printf("Characters entered = %d\n", count);

}
```

In the following program print_arguments.c, what are argc and argv?

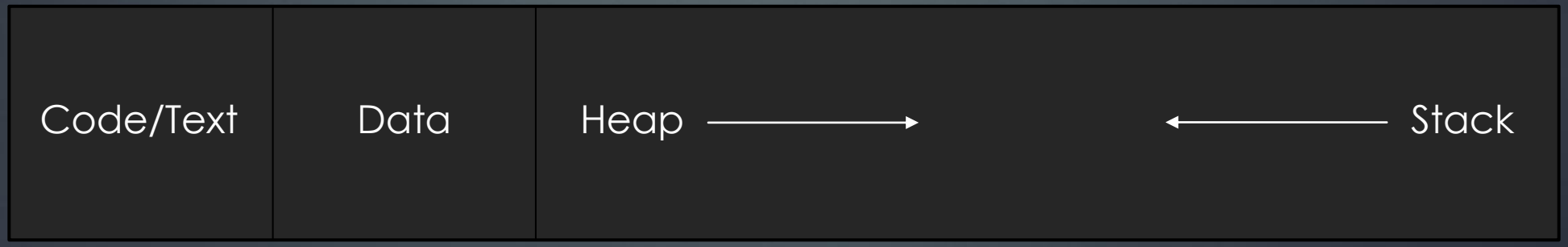
```c
#include <stdio.h>

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;
}
```

```
./print_arguments I love MIPS
```

Argc = 4
Argv[0] = print_arguments
Argv[1] = I
Argv[2] = love
Argv[3] = MIPS

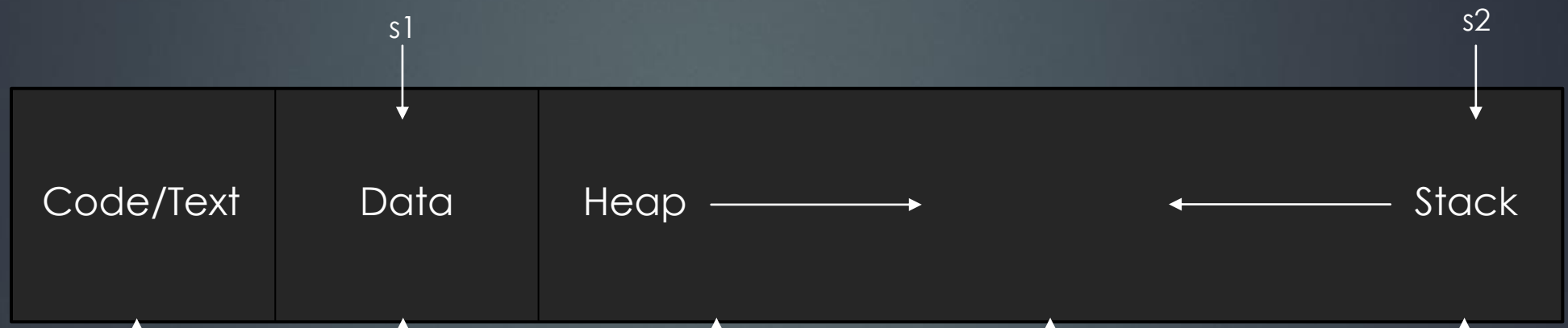
```c
#include <stdio.h>

char *s1 = "abc";

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

- What is different about the s1 and s2 variables?
- What does the memory layout of a typical program look like?
- Where are global variables located?
- Where are local variables located?
- What is a string literal? Where are they located?



When our computer executes our code, the operating system spawns a process, essentially a task the computer will run.

Each process will be allocated a section of memory by the operating system, which typically resembles the diagram below.



- Binary representation of our code
- Global variables
- String literals (read-only data)
- Malloc'd memory
- Must be manually freed
- Flexible storage the heap/stack grow into
- Local variables for functions
- Return addresses
- Memory automatically cleaned

Example: get_num_ptr.c

What's wrong with this code?

Assume we still want get_num_ptr() to return a pointer, how can we fix this code?

How did the change made affect each variable's location in memory?

```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;
}
```

Solution: get_num_ptr.c

What's wrong with this code?

- Stack use after return bug, x is located on the stack which is automatically cleaned when get_num_ptr() returns, which invalidates the address

Assume we still want get_num_ptr() to return a pointer, how can we fix this code?

- We need to ensure the value lives longer, so we should malloc the memory instead, and we can control the lifetime

How did the change made affect each variable's location in memory?

- The num and x pointer does not change location as they still live on the stack, however, the value x and num point to is allocated on the heap where malloc'd memory is stored

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

int *get_num_ptr(void);

int main(void) {
    int *num = get_num_ptr();

    printf("%d\n", *num);

    free(num);
}

int *get_num_ptr(void) {
    int *x = malloc(sizeof(int));
    *x = 42;

    return x;
}
```

Example:c_strings.c

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

- When compiled with DCC it will error as str[2] is uninitialized
- With other compilers it will compile without errors
- As printf expects a string to be null terminated, so it will keep executing, and str[2] may often contain '\0', or it may print garbage values, or even or it may index outside of the array, and stop with an error

```c
#include <stdio.h>

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

What does this look like in memory?

- char str[10] allocates 10 bytes on the stack, the first two bytes are 'H' and 'I', then 8 bytes of uninitialised values

H	i	?	?	?	?	?	?	?	?

How could we fix this program?

- We can null terminate the string by setting str[2] = '\0'

H	i	\0	?	?	?	?	?	?	?

```c
#include <stdio.h>

int main(void) {
    char str[10];
    str[0] = 'H';
    str[1] = 'i';
    str[2] = '\0';
    printf("%s", str);
    return 0;
}
```

RECURSION!

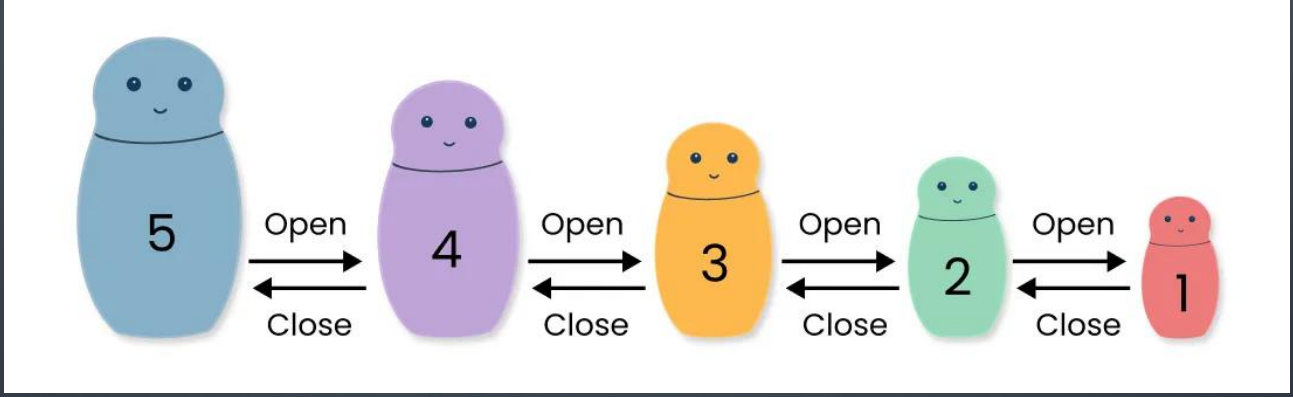
What is recursion?

Recursion is a programming method where a function solves a problem by calling itself with smaller versions of the same problem. Think of a Russian nesting doll, you open each to find a smaller, identical doll until you reach the final solid one.

To avoid infinite loops recursive functions, have two key parts:

1. The base case: The simplest possible version of the problem, that doesn't need recursion to be solved, the condition we return a simple answer from

2. The recursive case: The part of the function that breaks the problem down into smaller pieces, moving it closer to the base case



Example: Sum from 1 to n (sum.c)

This function takes in a number n and returns the sum of all integers from 0 to n

Iterative Version

```c
int sum(int n) {
    int result = 0;

    for (int i = 0; i <= n; i++) {
        result += i;
    }

    return result;
}
```

Recursive Version

```c
int sum(int n) {
    if (n == 0) {
        return 0;
    }

    return n + sum(n-1);
}
```

What is the difference in function calls between the loop and recursive versions?

- When a loop is used to calculate the sum there is only one call to the sum function.

What happens in memory when this program runs?

- After a function is called its arguments and local variables are stored on the stack.
- In the recursive version, there are n calls to the sum function. Each function call creates a new section of memory (or frame) on the stack to store its arguments and local variables.

Example: Length of Linked List (linked_list_*.c)

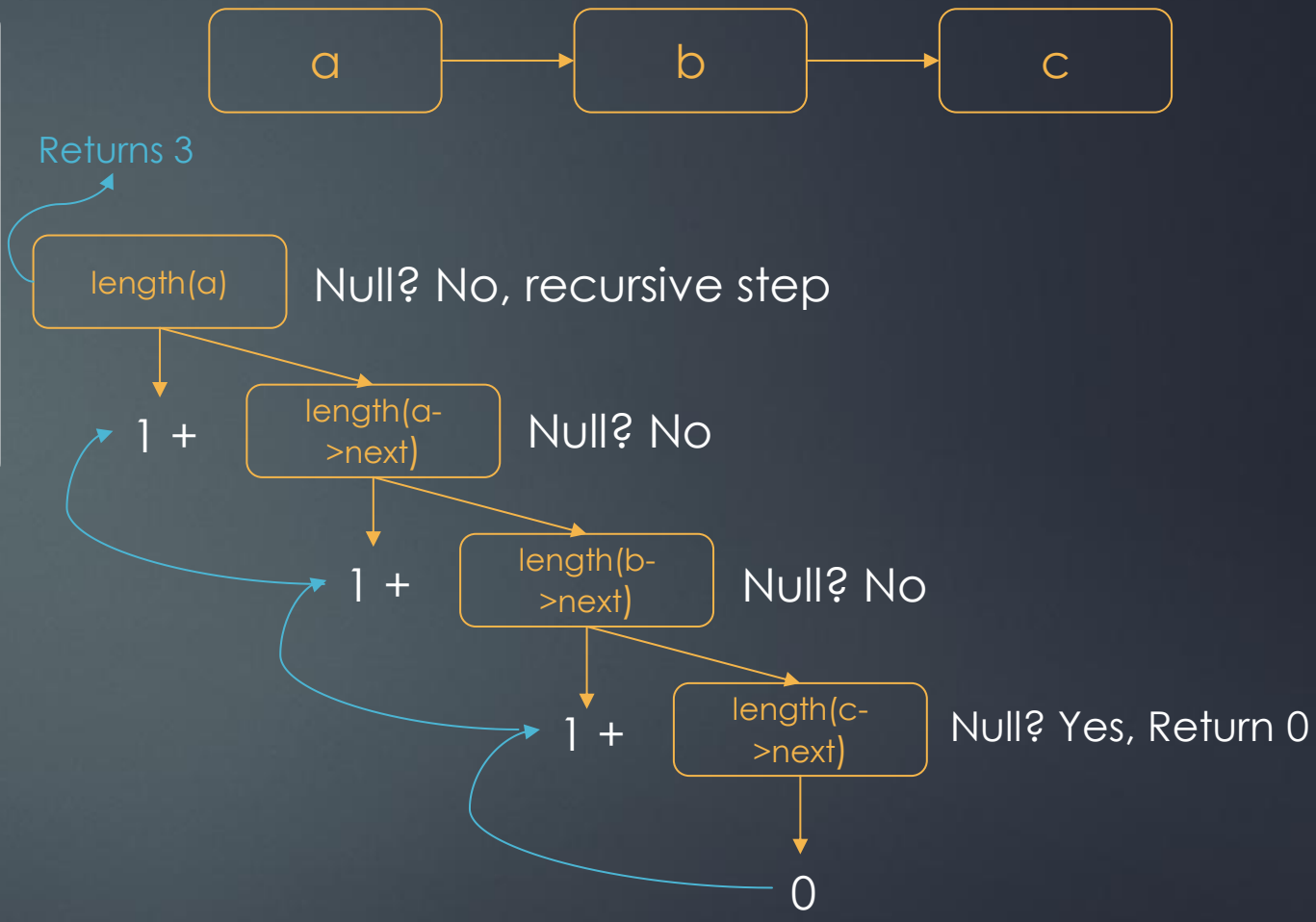
Iterative Version

```c
int length(struct node *head) {
    int count = 0;
    struct node *current = head;

    while (current != NULL) {
        count++;
        current = current->next;
    }

    return count;
}
```

Recursive Version

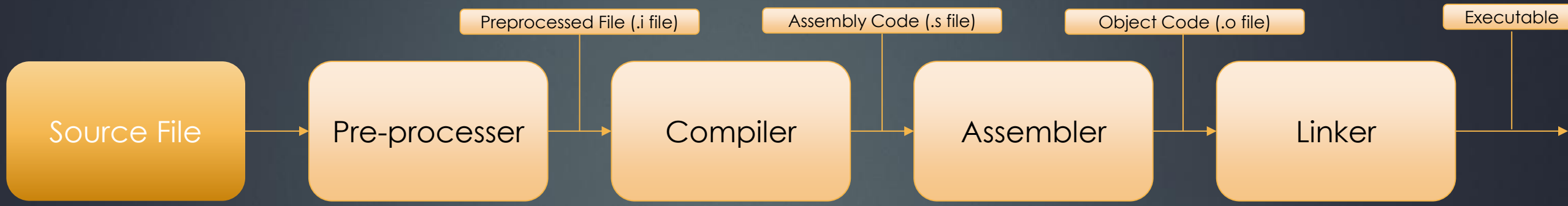
```c
int length(struct node *head) {
    if (head == NULL) {
        return 0;
    } else {
        return 1 + length(head->next);
    }
}
```

Example List: {a, b, c}



COMPILATION!

From source to execution



- Takes care of #defines
- Takes care of #includes
- Removes comments
- Other # directives out of scope of the course

- Turns code into assembly matching CPU architecture
- Most Windows PCs would produce x86, an m1 Mac would produce ARM
- Assembly is basically another programming language that aligns closely to the CPU hardware

- Turns the compiled assembly into binary code (text section of a process)
- Still requires a way to include library functions (printf)
- Will leave placeholders for the linker to resolve

- Resolves links library function definitions to their calls in binary
- Super complicated, but generally either links to definitions elsewhere in ram, or copy-pasted definitions into the executable.

Producing each step of the compiler

clang -E x.c

- Executes the C pre-processor and writes modified C code to stdout containing the contents of all #include'd files and replacing all #define'd symbols

clang -S x.c

- Produces a file x.s containing the assembly code generated by the compiler for the C code in x.c, the type of assembly code is architecture dependent

clang –c x.c

- Produces a file x.o containing relocatable machine code for the C code in x.c. Also, architecture dependent. This is not a complete program, even if it has a main() function: it needs to be combined with the code for the library functions (by the linker *ld*)

clang x.c

- Produces an executable file called a.out, containing all the machine code needed to run the code from x.c on the target machine architecture. The name a.out can be overridden by specifying a flag -o *filename*.

ASSEMBLY/MIPS INTRO!

What is Assembly Code?

As we saw in the compilation section, assembly code is produced by the compiler.

Think of it like human readable translation of machine code, the lowest level language before you're writing 1's and 0's, it is like writing a set of instructions for the CPU.

Comparing assembly to C

C Code (High-Level): Say you write a line of code like a = b + c

Assembly (Low-Level): This translates into several simple instructions, such as:

- load b into one place
- load c into another place
- add b and c together
- store the result into a

Simply put, assembly is the bridge between your C code and the computer's processor. Assembly's instructions align directly with the processor's specific architecture.

What assembly language do we use? MIPS!

There are many different types of assembly language, one for each kind of CPU architecture (like x86 on your PC or ARM on your phone or MacBook). For this course, we will use MIPS.

MIPS is a clean, simple, and elegant architecture that is often used for teaching. Its simplicity makes it perfect for learning the fundamental concepts of how a computer works without the extra complexity of modern commercial processors.

MIPSY 🛠️

Since your computer likely doesn't run on a MIPS processor, we use an emulator mipsy, that lets us run MIPS assembly code on any machine. If you want to get ahead on learning MIPS, this website is the ultimate MIPS resource for COMP1521.

https://cgi.cse.unsw.edu.au/~cs1521/25T2/resources/mips-guide.html

Additionally for reference, we can run the MIPS files (*.s) on cse servers using:

- 1521 mipsy test.s
- Or we can use mipsy-web, which we will look at more next week!

  https://cgi.cse.unsw.edu.au/~cs1521/mipsy/

What does a MIPS program look like?

C Version

```c
int main(void) {
    int b = 12;
    int c = 30;

    printf("%d\n", b + c);
}
```

MIPS Version

```
.text

main:
    li      $t0, 12
    li      $t1, 30

    add     $t2, $t0, $t1   # Add the two numbers

    li      $v0, 1          # Syscall value for print integer

    move    $a0, $t2        # Move the result to $a0 as the syscall expects it there
    syscall                 # Run syscall, that currently exists in $v0

    li      $v0, 0
    jr      $ra             # return 0 from main
```

We'll learn a lot more about the instructions, different types of syscalls and how to use them all over the next few weeks, it's completely normal if this is a little confusing at first!