

COMP1511 Week 10

adt, stacks and recursion

Joanna Lin

myExperience Survey!

What we'll cover today

- Command line arguments
- Multifile projects.
- Abstract data types (ADT)
 - Hiding away complex logic.
- Stacks
 - The *concept* of a stack.
 - Representing stacks in code.

Command Line Arguments

Command Line Arguments

- the user can supply some arguments when running the program.
- for example: `./program hello world!`
- to access what the user enters into the command line, we change the `main` function signature from `int main(void)` to `int main(int argc, char *argv[])`
 - `argc` stores the count of the command line arguments (the length of `argv`)
 - `argv` stores the command line arguments as an array of strings
 - `argv[0]` is always the name of the program.
- in the above example, `argc` would be 3 and `argv` would store the strings `"./program"`, `"hello"` and `"world!"` in that order.

Command Line Arguments

array of strings... or 2D array?

We can think of **argv** as either

```
{{ '\.', '/', '\p', '\r', '\o', '\g', '\r', '\a', '\m', '\0' },  
{ '\h', '\e', '\l', '\l', '\o', '\0' },  
{ '\w', '\o', '\r', '\l', '\d', '\0' }}
```

```
{ "./program", "hello", "world" }
```

```
#include <stdio.h>  
int main(int argc, char *argv[]) {  
    int i = 0;  
    while (i < argc) {  
        printf("%s\n", argv[i]);  
        i++;  
    }  
    return 0;  
}
```

Array of strings

OR

```
#include <stdio.h>  
int main(int argc, char *argv[]) {  
    int i = 0;  
    while (i < argc) {  
        int j = 0;  
        while (argv[i][j] != '\0') {  
            putchar(argv[i][j]);  
            j++;  
        }  
        putchar('\n');  
        i++;  
    }  
    return 0;  
}
```

2D Array

use **argc** to
write the
condition on the
loop when
looping through
argv

loop through each
character in each
string if you want to
have finer-grain
control.

Multifile Projects

Multifile Projects

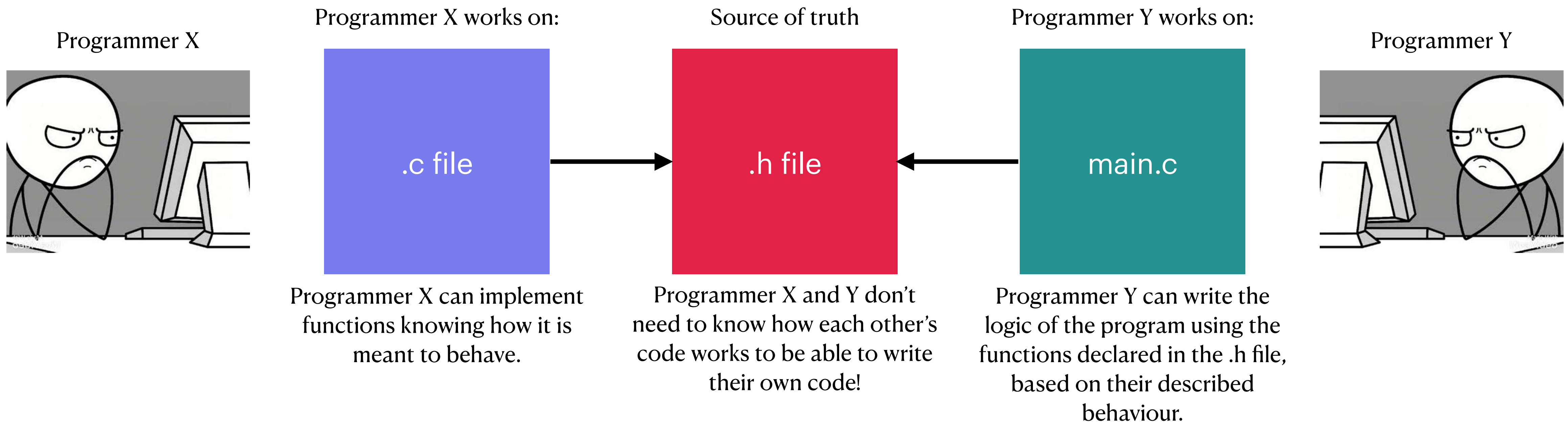
Structure

- `main.c`
 - contains the `main` function
- `.h` files (header files)
 - contains function prototype and function comments.
 - `#include`'d in the file that needs it (e.g. `#include "my_file.h"`)
 - Note double quotes are used for your own header files, whereas angular brackets are used for C standard library header files. (e.g. `#include <stdio.h>` vs `#include "my_file.h"`)
- `.c` files
 - contains function definition (the implementation of the functions)
 - compiled alongside `main.c` (e.g. `gcc my_file.c main.c -o program`)

Multifile Projects

Benefits

- spread code across multiple files to make navigating our code easier
- make collaborating on projects easier.
- reusability of code across different projects



Abstract Data Types

Abstract Data Types

- ADTs are defined by purely by their behaviour and the operations that can be done on them.
- They are 'abstract' because the details of how these operations are implemented are hidden from the user. The only part of the ADT that is exposed to the user are the necessary operations.
- We can compare an ADT to a car:

Operation	Behaviour
Accelerate	The car moves faster
Steer	The car turns
Brake	The car stops

- A manufacturer builds (implements) a car based on the defined operations, careful to expose only the necessary parts (the accelerator, steering wheel and brake) to the user to perform these operations, hiding (abstracting) away all of the complicated engineering behind a car.

Why ADTs?

- Until now, we've implemented everything with full control over every piece of data in our code. Think back to how, when we worked with linked lists, we had direct access to the data field and the next field simply by dereferencing our struct node pointers.
- However, having to keep track of every piece of data is not practical or efficient when you want to work on larger projects that require use of several data structures with complicated implementations. Remember, data types are a means to an end — not the end!
- Having full control over every bit of information makes it easy for a data type to become inconsistent leading to undefined behaviour.
- Imagine if, to operate a car, you needed to manually set the engine rotation speed and man the combustion chambers. It wouldn't even be practical to use a car!
- Furthermore, allowing so much access to the inner workings of the car means one is much more likely to break it by accident. For example, if you operate the engine incorrectly, pressing the accelerator might not work anymore!

Stacks

Stacks

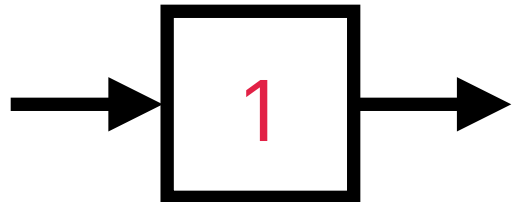
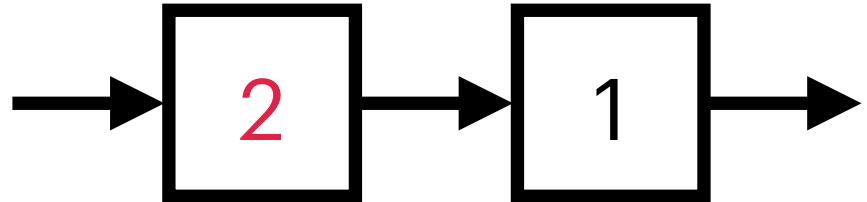
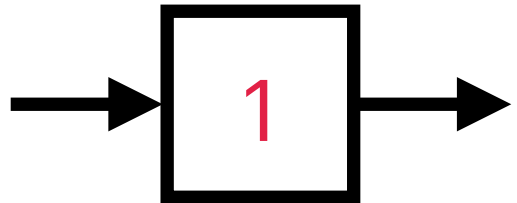

An ADT

- Consider a stack of books: the last book you placed onto the stack has to be the first book you take off of it.
- A stack in programming also has this last in, first out (LIFO) behaviour.
- Some operations (**bold** ones are the most pertinent to stacks)

Operation	Behaviour
Creating a stack	Gives the user the stack data structure
Pushing	An element is added to the stack.
Popping	The last element that was added in is taken out of the stack. The user is informed of the value that was removed.
Check if empty	Tells the user whether the list is empty

Stack (Linked List Version)

- **Implementation:** When the user pushes a number, we insert a node at the start of the list. When the user pops, we store the value inside the node the start of the list, then remove that node.
- Example usage:

What the user does	Our implementation	What the user sees
Push the number 1 into stack		—
Push the number 2 into stack		—
Pop from the stack		The number 2
Pop from the stack		The number 1

Stack (Array Version)

- **Implementation:** We keep track of the size of the stack, starting from 0. When the user pushes a number, we add a value at index given by the size of the stack and increment the size counter. When the user pops, we return the element at index size - 1, and then decrement the size of the list.
- Example usage (notice how the ‘what the user sees’ section is exactly the same as the linked list version):

What the user does	Our implementation	What the user sees
Push the number 1 into stack	<div><div><div>1</div><div>0</div><div>0</div><div>...</div></div><div>size: 1</div></div>	—
Push the number 2 into stack	<div><div><div>1</div><div>2</div><div>0</div><div>...</div></div><div>size: 2</div></div>	—
Pop from the stack	<div><div><div>1</div><div>2</div><div>0</div><div>...</div></div><div>size: 1</div></div>	The number 2
Pop from the stack	<div><div><div>1</div><div>2</div><div>0</div><div>...</div></div><div>size: 0</div></div>	The number 1