COMP1511 - Programming Fundamentals

Week 10 - Lecture 18

Programming is much more than just code

COMP1511 Programming Skills Topics

- History of Computing
- Problem Solving
- Code Style
- Code Reviews
- Debugging
- Theory of a Computer
- Professionalism

Problem Solving

Approach Problems with a plan!

- Big problems are usually collections of small problems
- Find ways to break things down into parts
- Complete the ones you can do easily
- Test things in parts before moving on to other parts

Code Style

Half the code is for machines, the other half for humans

- Remember . . . readability == efficiency
- Also super important for working in teams
- It's much easier to isolate problems in code that you fully understand
- It's much easier to get help if someone can skim read your code and understand it
- It's much easier to modify code if it's written to a good style

Code Reviews

No one has to work without help

- If we read each other's code . . .
- We learn more
- We help each other
- We see new ways of approaching things
- We are able to teach (which is a great way to cement knowledge)

Debugging

The removal of bugs (programming errors)

- Syntax errors are code language errors
- Logical errors are the code not doing what we intend

- The first step is always: Get more information!
- Once you know exactly what your program is doing around a bug, it's easier to fix it
- Separate things into their parts to isolate where an error is
- Always try to remember what your intentions are for your code rather than getting bogged down

Characters and Strings

Used to represent letters and words

- **char** is an 8 bit integer that allows us to encode characters
- Uses ASCII encoding (but we don't need to know ASCII to use them)

int a=1

int array[] = { }

- Strings are arrays of characters
- The array is usually declared larger than it needs to be
- The word inside is ended by a Null Terminator `'\0'`
- Using C library functions can make working with strings easier

Characters and Strings in code

```
// read user input
char input[MAX_LENGTH];
fgets(input, MAX_LENGTH, stdin); scanf
printf("%s\n", input);
          strings

// print string vertically
int i = 0;
while (input[i] != '\0') {
    printf("%c\n", input[i]);
    i++;
}
```

Structures

Custom built types made up of other types

- structs are declared before use
- They can contain any other types (including other structs and arrays)
- We use a `.` operator to access fields they contain
- If we have a pointer to a struct, we use `->` to access fields

Structs in code

```
struct spaceship {
    char name[MAX_NAME_LENGTH];
    int engines;
    int wings;
};

int main (void) {
    struct spaceship xwing;
    strcpy(xwing.name, "Red 5"); → char
    xwing.engines = 4;
    xwing.wings = 4;

    struct spaceship *myShip = &xwing;

    // my ship takes a hit
    myShip->engines--;  4-1
    myShip->wings--;    4-1
}
```

Memory

Our programs are stored in the computer's memory while they run

- All our code will be in memory
- All our variables also
- Variables declared inside a set of curly braces will only last until those braces close (*what goes on inside curly braces stays inside curly braces*)
- If we want some memory to last longer than the function, we allocate it
- **`malloc()`** and **`free()`** allow us to allocate and free memory
- **`sizeof`** provides an exact size in bytes so malloc knows how much we need

Memory code

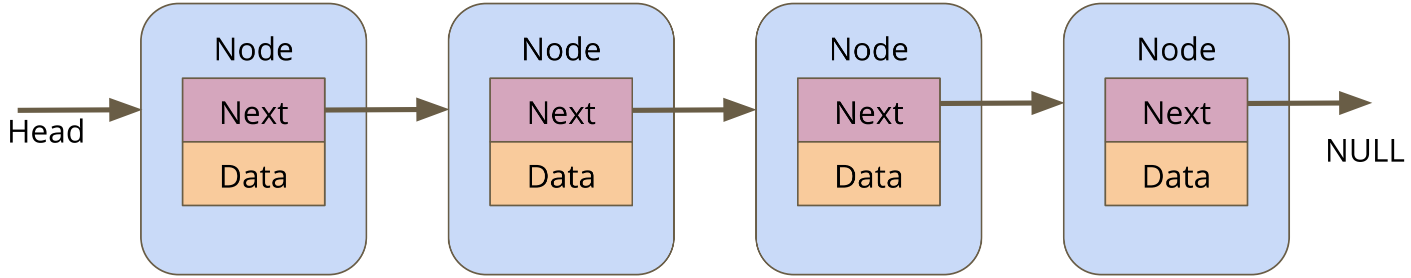
```c
struct spaceship {
    char name[MAX_NAME_LENGTH];
    int engines;
    int wings;
};

int main (void) {
    struct spaceship *myShip = malloc(sizeof (struct spaceship));
    strcpy(myShip->name, "Millennium Falcon");
    myShip->engines = 1;
    myShip->wings = 0;

    // Lost my ship in a Sabacc game, free its memory
    free(myShip);
}
```

新建个 □

Linked Lists

Structs for nodes that contain pointers to the same struct

- Nodes can point to each other in a chain to form a linked list
- Convenient because:
  - They're not a fixed size (can grow or shrink)
  - Elements can be inserted or removed easily anywhere in the list
- The nodes may be in separate parts of memory



Linked Lists in code

```
struct location {
    char name[MAX_NAME_LENGTH];
    struct location *next;
};

int main (void) {
    struct location *head = NULL;
    head = addNode("Tatooine", head);      head = newNode -> Null
    head = addNode("Yavin IV", head);
}

// Add a node to the start of a list and return the new head
struct location *addNode(char *name, struct location *list) {
    struct location *newNode = malloc(sizeof(struct location));
    strcpy(newNode->name, name);
    newNode->next = list;
    return newNode;        head.
}
```

Complications in Pointers, Structs and Memory

What's a pointer?

- It is a number variable that stores a memory address
- Any changes made to pointers will only change where they're aiming

What does * do?

- It allows us to access the memory that the pointer aims at (like following the address to the actual location)
- This is called "dereferencing" (because the pointer is a reference to something)

Complications in Pointers, Structs and Memory

What about `->` ?

- Specifically access a struct at the end of a pointer
- `->` must point at one of the fields in the struct that the pointer aims at
- It will dereference the pointer AND access the field

Pointers to structs that contain pointers to other structs!

- We can follow chains of pointers like `track->beat->note`

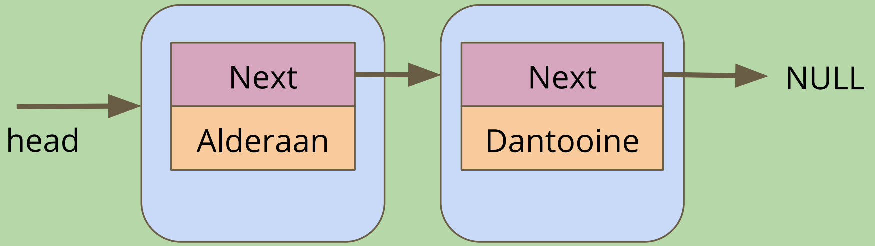
Complicated Pointer Code

```c
int main (void) {
    // create a list with two locations
    struct location *head = addNode("Dantooine", NULL);
    head = addNode("Alderaan", head);

    // create a pointer to the first location
    struct location *alderaan = head;

    // set head to a newly created location
    head = malloc((sizeof(struct location));

    // What has happened to the alderaan pointer now?
    // What has happened to the variable that the head and alderaan
    // both pointed at?
}
```

Pointer Arithmetic

A program's memory (not to scale)

Create a linked list of two locations with a head pointer aimed at the first location

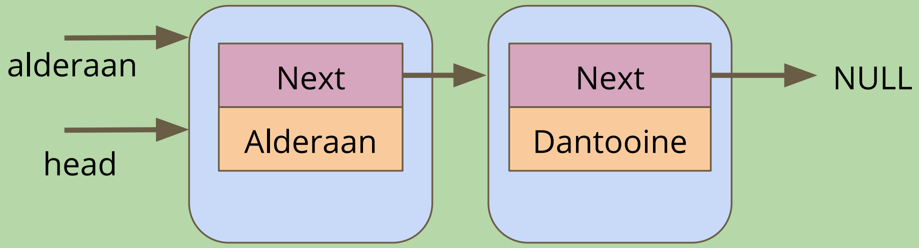


Pointer Arithmetic

A program's memory (not to scale)

```
struct location *alderaan = head
```

This line creates a new pointer that's a copy of the head pointer. It is given the same value as head, which means it's aimed at the same memory address

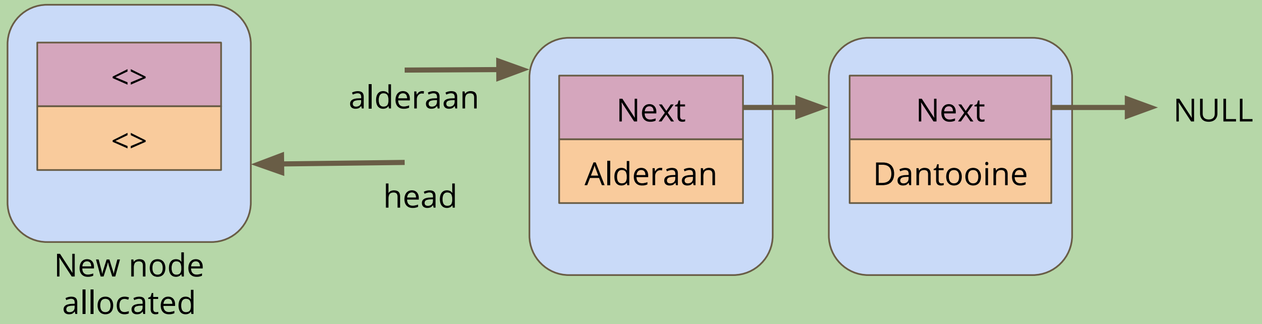


Pointer Arithmetic

A program's memory (not to scale)

```
head = malloc((sizeof(struct location));
```

This line allocates new memory and assigns the address of this new allocation to the head pointer.
Changing head doesn't change anything it was pointing at!



Keeping track of pointers

```
realm->castle->next->next->enemies = ????
```

- This is code that might work in most Castle Defense implementations

- **Remember:**
- Changing a pointer changes its value, a memory address
- Changing a pointer will change where it's aiming, nothing more!
- Once you use `->` on a pointer, you're now looking at a struct field
- This means you are not changing that pointer, you have dereferenced it and accessed a field inside the struct

Abstract Data Types

Separating Declared Functionality from the Implementation

- Functionality declared in a Header File
- Implementation in a C file
- This allows us to hide the Implementation
- It protects the raw data from incorrect access
- It also simplifies the interface when we just use provided functions

Abstract Data Types Header code

```
// ship type hides the struct that it is
// implemented as
typedef struct shipInternals *Ship;

// functions to create and destroy ships
Ship shipCreate(char* name);
void shipFree(Ship ship);

// set off on a voyage of discovery
Ship voyage(Ship ship, int years);
```

Abstract Data Types Implementation

```
struct shipInternals {
    char name[MAX_NAME_LENGTH];
};

Ship shipCreate(char* name) {
    Ship newShip = malloc(sizeof (struct shipInternals));
    return newShip
}
void shipFree(Ship ship) {
    free(ship);
}
// set off on a voyage of discovery
Ship voyage(Ship ship, int years) {
    int discoveries = 0, yearsPast = 0;
    while(yearsPast < years) {
        discoveries++;
    }
}
```

Abstract Data Types Main

- Including the Header allows us access to the functions
- The main doesn't know how they're implemented
- We can just trust that the functions do what they say

```c
#include "ship.h"

int main (void) {
    Ship myShip = newShip("Enterprise");
    myShip = voyage(myShip, 5);
}
```