COMP1511 - Programming Fundamentals

Week 9 - Lecture 16

What did we learn last lecture?

Abstract Data Types

- An extension/explanation of the use of Multiple File Projects
- Presenting an idea. A structure with a set of rules
- But hiding the implementation
- A Stack as an example

What are we covering today?

Recursion

- An interesting inversion on the order of program execution
- Functions that call themselves
- Using the program call stack to determine the order of operations

Understanding Recursion

Recursion is a little bit backwards

- Now that you understand Recursion, you can use it
- In order to understand Recursion you must already understand Recursion
- It's good that you knew Recursion before we started

We need to think a little bit in reverse here, but let's step through an example first . . .

Add up all the numbers in a linked list

Loop through and add them up . . . we can already do this

```
// Loop through a list of nodes, adding their values together
int sum_list(struct node *n) {
    int total = 0
    while (n != NULL) {
        total += n->data;
        n = n->next;
    }
    return total;
}
```

What about a different way?

Let's look at what might happen if we have a function that can call itself

A function that says:

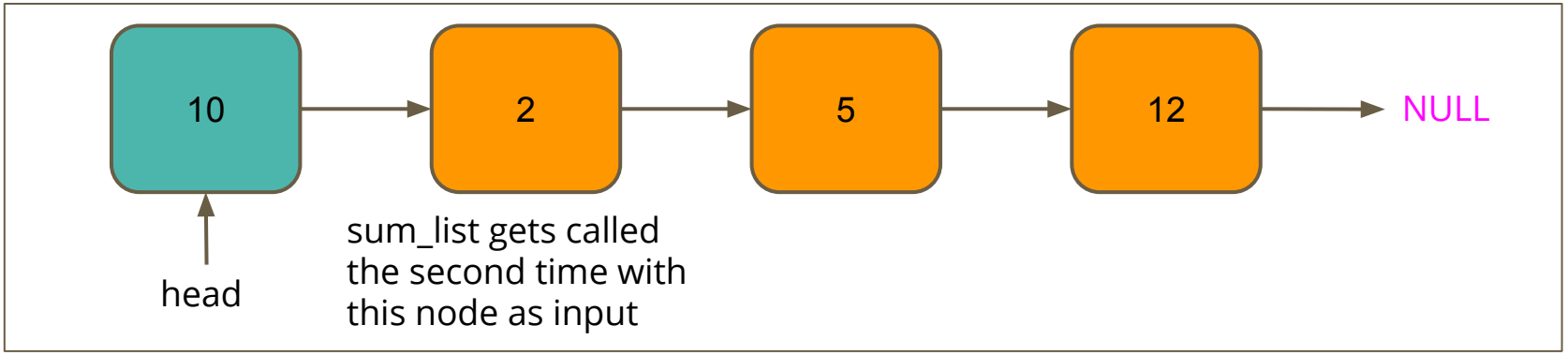
```
// sum_list calls itself again, but on a different
// part of the list
int sum_list(struct node *head) {
    int total = head->data + sum_list(head->next);
    return total;
}
```

The function can call itself? What happens here?

Functions calling themselves

```
sum_list(head) = head->data + sum_list(head->next);
```

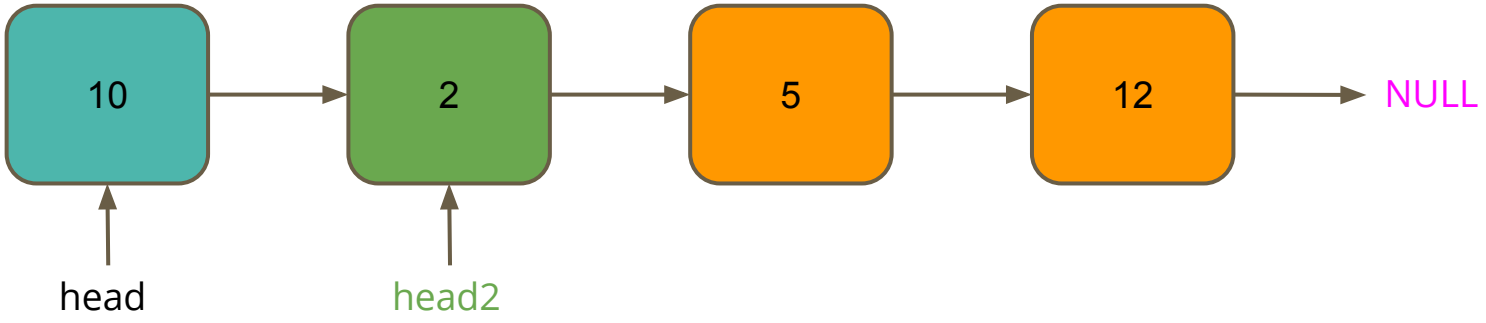
The total is equal to the value of the head added to the sum_list function called on the rest of the list



10 + whatever the rest of the list happens to add up to . . .

The second function

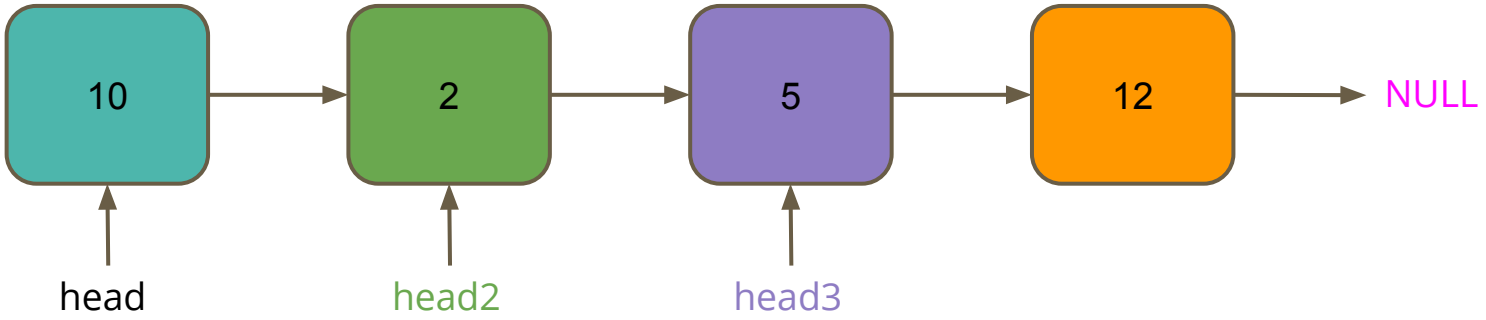
```
sum_list(head) = head->value + sum_list(head->next);

sum_list(head) = head->value + head2->value +
                 sum_list(head2->next);
```



10 + 2 + whatever the rest of the list adds up to

It keeps going…

```
sum_list(head) = head->value + head2->value + head3->value +
                 sum_list(head3->next);
```



10 + 2 + 5 + whatever the rest of the list adds up to

Is this endless?

Like loops, Recursive function calls still need to know when to stop

In the previous example:

- What happens if we reach the end of the list?
- What happens if the list was empty to begin with?

We need a "stopping case" where the function won't call itself again

Two Cases

Keep going or stop

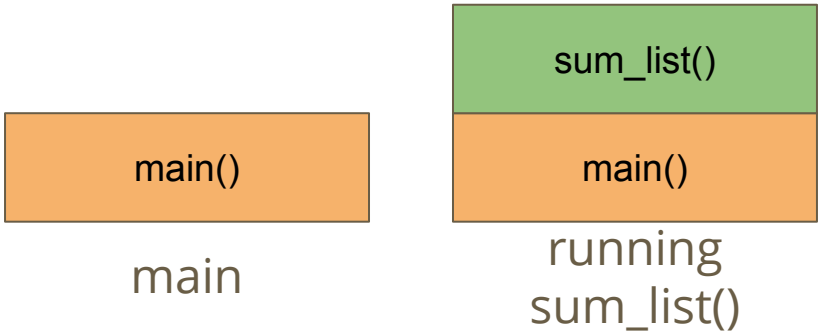
- We've already got the "keep going" case
- How do we stop?
- Let's test for the situation where we wouldn't want to add more elements

```
// sum_list calls itself again, but stops if there's
// nothing to add
int sum_list(struct node *head) {
    if (head == NULL) {
        return 0;
    } else {
        int total = head->data + sum_list(head->next);
        return total;
    }
}
```

Functions and Stacks

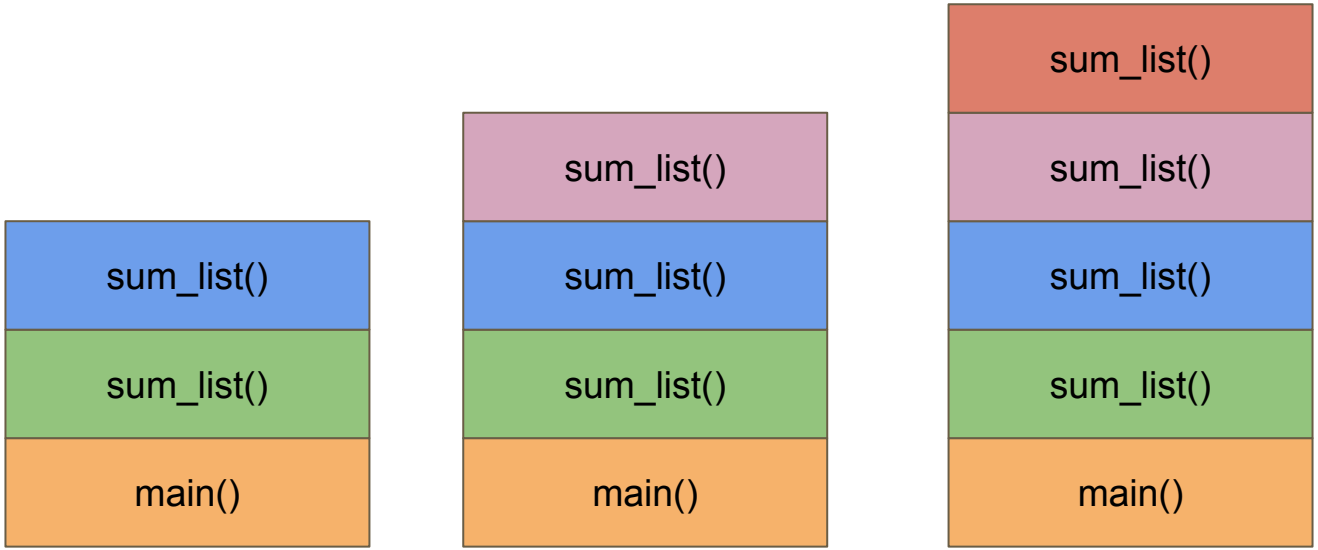
The Function Call Stack during recursion

Initially we have a main function that calls **`sum_list()`**



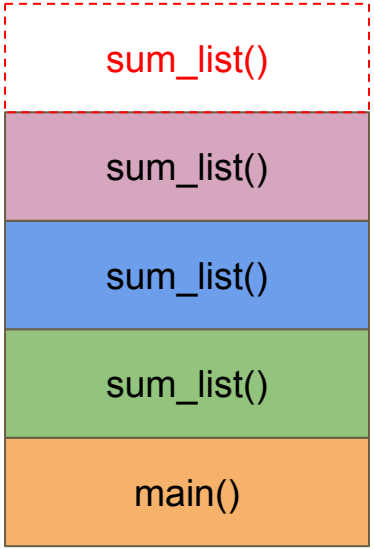
The Call Stack as Recursion continues

As the function "recurses", it adds more function calls

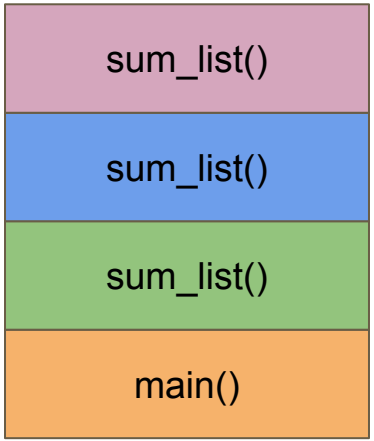


Reaching a stopping case

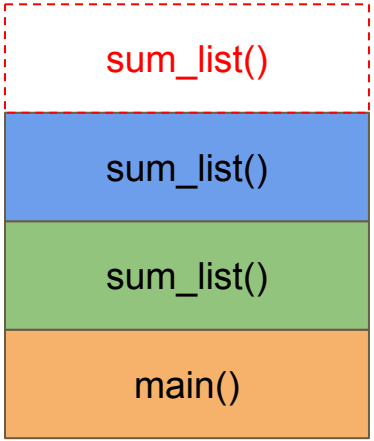
Returning from a recursive function to the previous call



Returning because of the stopping case



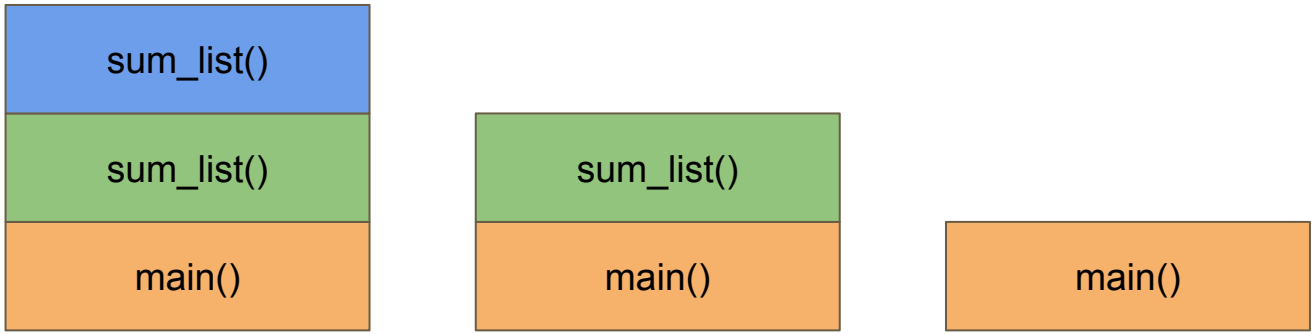
Return to previous function call



This also returns

Completion of a recursive function

Eventually the chain of returns will finish



Functions return one after the other until . . .

We're back to the main

Code Example - Reverse Print Names

Returning to our Battle Royale Example

Say we had a list of people who had been knocked out of the game and we want to "replay" the order they were knocked out?

- We have a linked list of names
- It's currently in the reverse order of when they were knocked out
- So we want to print out their names in the opposite of their order

Our List

We have a standard linked list node

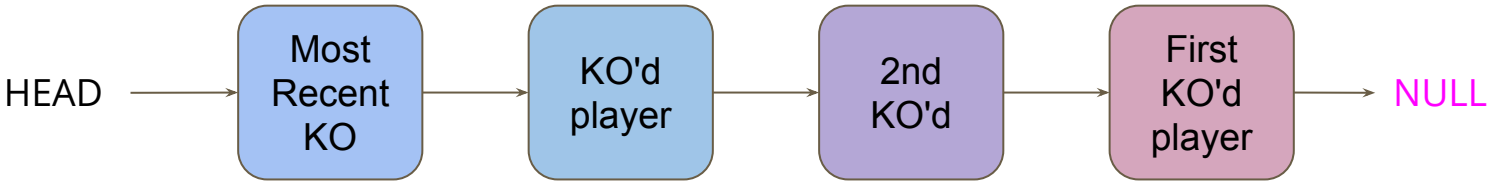
- Contains a name and a pointer

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

How it was created

Let's say that during our game, we built a list of players

- Each time a player is knocked out, we add them to the head of a list



We want to be able to print this out in the order that they were knocked out

How do we do this without Recursion?

A "procedural" implementation

1. Loop to end of the list and print out the name
2. Have some way of remembering which player we've already printed from
3. Start a new loop, going until just before the one we printed previously
4. Print out that name
5. Keep repeating until there are no names left

Code to print out an element before a pointer

```
struct player *print_before(struct player *player_list, struct player *after)
{
    // loop until you see the after pointer
    struct player *curr = player_list;
    struct player *prev = NULL;
    while (curr != after) {
        prev = curr;
        curr = curr->next;
    }
    if (prev != NULL) {
        // element exists, print its name
        fputs(prev->name, stdout);
        putchar('\n');
    }

    return prev;
}
```

Code for a procedural reverse_print()

```
// Print out the names stored in the list in reverse order
// This is a procedural programming implementation
void reverse_print(struct player *player_list) {
    struct player *end = NULL;
    int finished = 0;

    // Loop once for each name in the list
    while (!finished) {
        end = print_before(player_list, end);
        if (end == NULL) {
            finished = 1;
        }
    }
}
```

Break Time

Where to find further information about programming?

- There are a lot of online resources that can help with programming
- Teaching yourself can help to go beyond course content
- Stack Overflow is a question and answer site
  - It can sometimes be useful but sometimes be confusing or argumentative
- There are several free online courses that will teach you different languages
  - Too many to list!
- Experimentation will always teach you something!
- Pick an idea of something you want to make and see what you can build!

That was exhausting

What did we need to do?

Outer Loop

- Loops once for each element of the list
- Keeps track of the last element printed

print_before() function

- Loops until the given element pointer
- prints out the one before that (if it exists)
- returns a pointer to the element that was printed

Recursion instead

Let's try this recursive and see how it works

Stopping case

- there are no elements, so print out nothing

Otherwise

- printReverse() the rest of the list
- After that print out the current head.

Code for reverse_print_recursive()

```
// Print out the names stored in the list in reverse order
// This is a recursive programming implementation
void rev_print_rec(struct player *player_list) {
    if (player_list == NULL) {
        // stopping case (there are no elements)
        return;
    } else {
        // there are element(s)
        rev_print_rec(player_list->next);
        fputs(player_list->name, stdout);
        putchar('\n');
    }
}
```

Wait is that it?

Yes.

- Recursion often takes a lot of thinking and not much code

Still, let's look deeper to get a stronger understanding

- What order are things happening?
- What happens if we change the order?

What's the order of execution?

A single call of our recursive function:

1. Check if we're stopping, if so return
2. Otherwise, call the function again with the tail (all remaining elements)
3. Then print the name of the current head of the list

Order of execution

More recursive function calls

1. Check if we're stopping, if so return
2. Otherwise, call the function again with the tail (all remaining elements)
    a. Check if we're stopping, if so return
    b. Otherwise, call the function again with the tail (all remaining elements)
        i. Check if we're stopping, if so return
        ii. Otherwise, call the function again with the tail (all remaining elements)
        iii. Then print the name of the current head of the list
    c. Then print the name of the current head of the list
3. Then print the name of the current head of the list

Changing the order

What happens if we change the order in a recursive function?

1. Check if we're stopping, if so return
2. Then print the name of the current head of the list
3. Otherwise, call the function again with the tail (all remaining elements)

Having swapped 2 and 3, will the function behave differently?

Changing the order in code

```
// Changing the order of operations in a recursive function
// This is a recursive programming implementation
void rev_print_rec(struct player *player_list) {
    if (player_list == NULL) {
        // stopping case (there are no elements)
        return;
    } else {
        // there are element(s)
        fputs(player_list->name, stdout);
        putchar('\n');
        // the recursion is now after the print
        rev_print_rec(player_list->next);
    }
}
```

Interesting results

We're now printing in order . . .

How did this happen?

Let's look at the order of execution again

Order of execution

If we change when the recursive function call is made . . .

1. Check if we're stopping, if so return
2. Otherwise, print the name of the current head of the list
3. Then call the function again with the tail (all remaining elements)
    a. Check if we're stopping, if so return
    b. Otherwise, print the name of the current head of the list
    c. Then call the function again with the tail (all remaining elements)
        i. Check if we're stopping, if so return
        ii. Otherwise, print the name of the current head of the list
        iii. Then call the function again with the tail (all remaining elements)

What did we learn today?

Recursion

- If you know recursion, you can return now
- Otherwise, you can learn recursion by learning recursion

- Functions calling themselves can set up interesting patterns
- It lets us use and manipulate the function call stack
- Ordering of when and how we recurse can change behaviour