COMP 1633: Intro to CS II

Recursion

Charlotte Curtis

March 27, 2024

Where we left off

- `friend` functions and stream operators
- A common abstract data type: stacks
- Designing a `SetInt` ADT

> *Textbook Sections 11.2, 13.2*

```
class StringStack {
public:
...
private:
    struct Node {
        std::string data;
        Node *next;
    };
    Node *head;
    int capacity;
    int size;
};
```

Today's topics

- Something completely different: **Recursion!**
- Note: due to Easter, I had to remove another example of an ADT from the schedule, but I've posted the content if you'd like to read about queues

> *Textbook Section 13.2, Chapter 14*

And now, recursion!

- Recursion is a **programming technique** that involves a function calling itself
- You may have seen a bit of this in COMP 1701, e.g.:

```python
def get_valid_input(valid_choices: list) -> str:
    choice = input('Enter your choice: ')
    if choice not in valid_choices:
        print('Invalid choice!')
        choice = get_valid_input(valid_choices)

    return choice
```

- What is actually happening here???

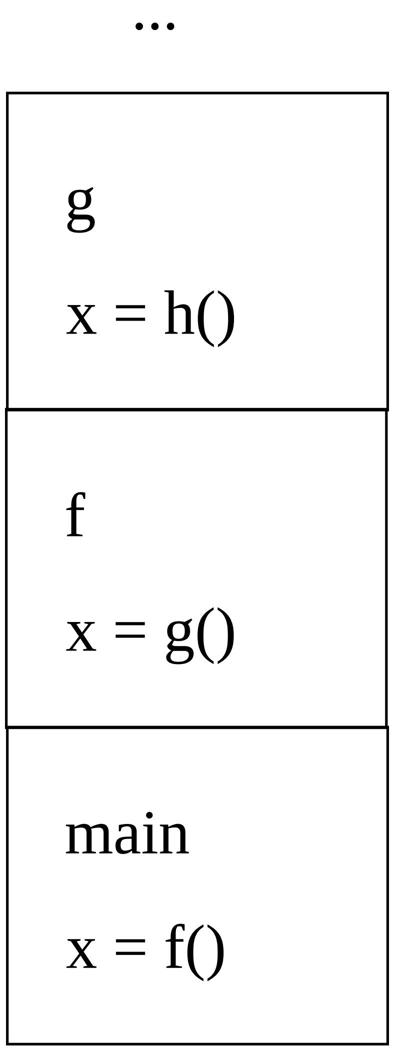
The call stack

```c
int f() {
    int x = g();
    return x;
}

int g() {
    int x = h();
    return x;
}

int h() ...
```

```c
int main() {
    int result = f();
    return 0;
}
```



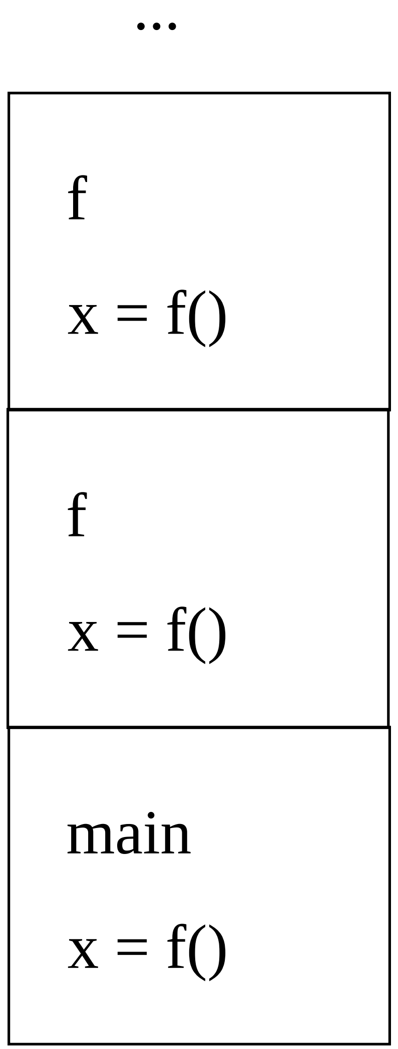
- Each function call adds a **stack frame** to the stack
- The stack frame contains the **local variables** of the function and the **return address** of the caller

Functions calling themselves

```
int f() {
    int x = f();
    return x;
}
```

```
int main() {
    int result = f();
    return 0;
}
```

- Each call adds an **independent stack frame**
- The local variables `x` do not interfere, and each call has a unique **return address**
- One big problem: **it never ends**!
- Let's see what happens on Python Tutor



Definition of Recursion

re·cur·sion

/rəˈkərZH(ə)n/

noun **MATHEMATICS • LINGUISTICS**

the repeated application of a recursive procedure or definition.

- a recursive definition.

  plural noun: **recursions**

- In programming, recursion involves a function calling itself repeatedly
- To be useful, it must stop at some point

Divide and conquer

- Just like with loops, recursion is a way to **repeat** a task
- We might have a big problem (such as deleting a linked list) that we can break down into smaller problems (deleting a node)
- Just like loops, we need a stopping condition - this is called the **base case**
- Everything else is the **recursive case**

Example: Factorial

- The **factorial** of a number is the product of all the integers from 1 to that number

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

- You could also think of it as $n! = n \times (n-1)!$ with a **base case** of $0! = 1$
- We could write this as a loop, but it's more fun as recursion:

```
int factorial(int n) {
    int result = 1;
    for (int i = 2; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

```
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

Tracing recursive functions

```
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}

int main() {
    cout << factorial(4) << endl;
}
```

Thinking recursively, step by step

1. What is the **base case**? This is the **simplest case** that must be solved directly.
   - For the factorial example, this is `factorial(0) = 1`
   - There may be more than one base case!
2. What is the **recursive case**? This is the case that depends on a prior case.
   - For the factorial example, this is `factorial(n) = n * factorial(n-1)`
   - There may be more than one recursive case!
3. How does the recursive case get closer to the base case?
   - For the factorial example, this is `n-1`
   - This is referred to as the **reduction step**

Typical structure of a recursive function

```
if (base case)
    solve the problem
else
    reduce the problem
    call the function again
```

- There's no requirement to check the base case first
- There *is* a requirement that the set of base and recursive cases must:
    - be **exhaustive** (cover all possible cases)
    - be **mutually exclusive** (no overlap between cases)
- There can also be more than one base case and/or recursive case

The Towers of Hanoi

- The Towers of Hanoi is a classic puzzle game with 3 pegs and `n` disks
- The goal of the game is simple: move all the disks from the 1st to the 3rd peg
- However, there are rules:
  - Only move one disk at a time
  - A larger disk cannot be placed on top of a smaller disk

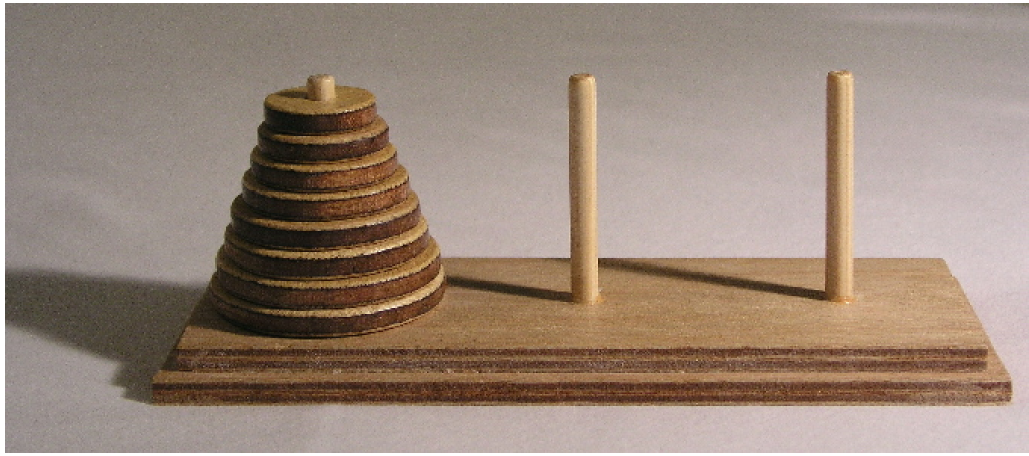


Image source: Wikimedia Commons

Recursion involving Linked Lists

- Linked lists are a natural fit for recursion!
- Operations performed on one element only need to know if it's `NULL` or not
  - base case: empty list
  - recursive case: non-empty list
  - reduction step: access `next` element

> *Example: computing the length of a linked list*

Printing a linked list

Given a list of `0 -> 1 -> 2 -> 3 -> NULL`, trace the following:

Iterative solution

```
void print(Node *head) {
    while (head) {
        cout << head->data << endl;
        head = head->next;
    }
}
```

Recursive solution

```
void print(Node *head) {
    if (head) {
        cout << head->data << endl;
        print(head->next);
    }
}
```

- What is the **base case**?
- There doesn't seem to be much advantage to the recursive solution, but...

Reversing the order of actions

Given a list of `0 -> 1 -> 2 -> 3 -> NULL`, trace the following:

```
void print(Node *head) {
    if (head) {
        print(head->next);
        cout << head->data << endl;
    }
}
```

- How would this be done in an iterative manner?
- This is one of few examples where the recursive solution is really the easiest!

Why *wouldn't* we use recursion?

- There are scenarios where recursion is easier to read and implement
- However, recursion comes at a cost:

```
int factorial(int n) {
    int result = 1;
    for (int i = 2; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

```
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

- The recursive solution just created `n` stack frames complete with `n` function return addresses and temporary variable allocations!

Tail recursion

- There is one way of reducing the overhead, but still using recursion
- **Tail recursion** is when the **very last thing** a function does is call itself
  - do not multiply the result by `n`
  - do not compare the result to anything
  - There can be no other operations between the recursive call and `return`
- Why? The compiler can **optimize** this to a loop!
- We avoid the overhead of all those stack frames

> *g++ may optimize other forms of recursion, but it's not guaranteed*

Tail recursion example

Back to the linked list deletion example:

```
void clear_list(Node *head) {
    if (head) {
        clear_list(head->next);
        delete head;
    }
}
```

- Can we make this tail recursive?
- Does it matter, or is this **premature optimization**?

Recursion check-in 1/2

Can any recursive function be implemented iteratively?

A. Yes

B. No

Recursion check-in 2/2

Trace the following code and write the result:

```
int mystery(int n) {
    if (n < 2)
        return n;
    else
        return mystery(n-1) + mystery(n-2);
}

int main() {
    cout << mystery(4) << endl;
}
```

Recursion with arrays

- Linked data structures are a natural fit for recursion, but what about arrays?
- It's doable! We need to consider:
  - What is the **base case**?
  - What is the **reduction step**?
- We can keep track of the "active" piece of the array with two indices, or...
- We can pass the **fill level** of the array as a parameter along with a pointer to the **start of the active portion**

Searching an array

- Consider the case of searching for a specific value in a **sorted** array
- A naive approach might be something like:

```
bool in_array(int *arr, int size, int value) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == value)
            return true;
    }
    return false;
}
```

- This is a **linear search** with an early return if the value is found
- If the value is not in the array we have to check every element!

Binary search

- Instead of checking every element, we can use a **binary search**:
  - Check the middle element
  - If it's the value we're looking for, we're done!
  - If it's less than the value we're looking for, search the **second half**
  - If it's greater than the value we're looking for, search the **first half**
  - Repeat until the value is found or the array is exhausted
- Each check eliminates half of the remaining elements!
- We could implement this iteratively, but it's a natural fit for recursion

Binary search with recursion

- We have multiple base cases and recursive cases
- Base cases:
    - The array is empty or has one element
    - The value is found
- Recursive cases:
    - The value is less than the middle element
    - The value is greater than the middle element
- Reduction step:
    - Chop the array in half and search the appropriate half

Coming up next

- Good Friday, Easter Monday
- Lab tomorrow: ADT implementation
- Lab Tuesday: Recursion
- Wednesday Lecture: Copying objects
- **Assignment 4** 🎉 due Monday, April 8th