Introduction to Modern C++

Lab Session 4
Headers, Linked Lists, Recursion, and Dynamic programming

This lab session will explore functions and some interesting constructions with them. Note that from now on you will need to work with real files (not just the website). So please make sure you know how to write, compile, and run programs on the Ubuntu machine.

1 Headers

Task 1. Consider the following two codes:

```cpp
int function1(int x) {
  return x + 2;
}

int function2(int x) {
  return function1(x) + 2;
}

int main() {
  int y = function2(42);
}
```

```cpp
int function2(int x) {
  return function1(x) + 2;
}

int function1(int x) {
  return x + 2;
}

int main() {
  int y = function2(42);
}
```

What is the difference? Which one of these codes works? Why?

Task 2. Create a file `myheader.h` containing the following:

```cpp
int function1(int x);
int function2(int x);
int main();
```

These are called the *signatures* of our functions. Consider again the two codes of Task 1, but this time add

```cpp
#include "myheader.h"
```

as the first line. Now the order of `function1` and `function2` doesn't matter anymore!

Remark: In C++, the `.h` files are called "headers" or "interfaces". They describe how functions (or data structures) should be used, i.e. their signature.

Task 3. What happens if we `#include` a header file twice? Add `int c = 42;` to the header file. Does it cause an error? Why? Add the following command as the first line of your header file:

```cpp
#pragma once
```

What does it do? Does it fix the problem?

2 Recursion

The factorial of a number n, denoted $n!$ is defined as $n! = 1 \times 2 \times \cdots \times n$.

Task 4. Implement a function `factorial` that computes the factorial using a `for` loop:

```cpp
int factorial(int n) {
  // Your code with a for loop
}
```

Task 5. Observe that $n! = n \times (n-1)!$ and recall that $0! = 1$. Use these observations to rewrite the `factorial` function in a recursive fashion *without any loop construct*.
Hint: You have to distinguish two cases; the case $n = 0$ and the case $n > 0$.

Task 6. The famous Fibonacci sequence is 1, 1, 2, 3, 5, 8, 13, ... (the next number is the sum of the two last numbers). Let's call F_k the k-th Fibonacci number (e.g. $F_5 = 5$), then we have the recurrence $F_k = F_{k-1} + F_{k-2}$. Write a recursive function `Fib` that takes k as an argument and returns F_k.
Hint: $F_0 = 0$ and $F_1 = 1$.

Task 7. Try using your program to compute F_{100}. What happens? Why? Try to compute F_{40}. How many calls do you make to `fib`?

Remark: Writing programs in a recursive fashion makes it easier to check and analyse them. But it is not always the most efficient way to code!

3 Memoization and Dynamic Programming

We will make the Fibonacci program much faster now, by using a technique known as "memoization". It simply means that we store the answers. Next time the same question appears, instead of computing again, we just look at the saved answer. This avoids computing things twice. Memoization is a core element of dynamic programming, a powerful problem-solving approach that you will see in a homework.

● **Task 8.** Complete the following code. Use `mem_fib` to save and retrieve the values of F_k.

```
#include <iostream>
#include <vector>

std::vector<int> mem_fib;

int fib(int k) {
  // Your new code here
}

int main() {
  std::cout << "F_40 = " << fib(40) << std::endl;
}
```

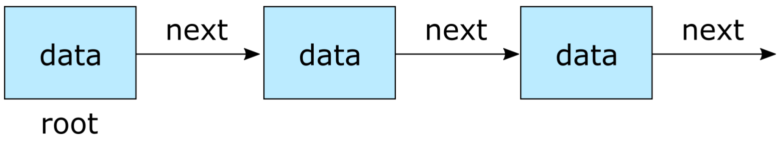
Reminder: You can use `mem_fib[k]` to read or write the value of the k-th element in `mem_fib`; you can use `mem_fib.size()` to find the number of elements stored; you can use `mem_fib.resize(n)` to set the size of `mem_fib` to a value n.

How many calls to `fib` do you make in total when computing F_{40} now?

4 ● Linked lists

A linked list is an efficient way to store a list of data. Elements of that list are called "nodes". There is a root node, where the list begins.

Each node has some information (the data), and a pointer to the next node in the list. When there is no next node, this pointer is null (`nullptr`).



One interest of this data structure is that it makes it very easy to add and remove nodes from anywhere, just by playing with pointers. It is thus much faster than `std::vector` for these operations.

Of course, there is already a standard library implementation of linked lists (`std::forward_list`). But it is interesting to see exactly how it works.

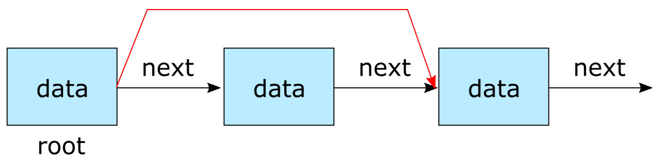
Consider this stub implementation of linked lists:

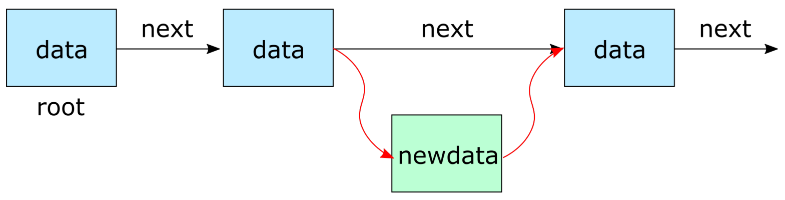
```cpp
#include <iostream>

struct Node {
  int   data;  // Data of the node
  Node* next;  // Pointer to the next node
};

int main() {
  Node root {42, nullptr};

  Node new_element1 {43, nullptr};
  Node new_element2 {44, nullptr};
  Node new_element3 {45, nullptr};

  Node* end = &root;

  (*end).next = &new_element1;
  end = (*end).next;

  (*end).next = &new_element2;
  end = (*end).next;

  (*end).next = &new_element3;
  end = (*end).next;


  std::cout << root.data << std::endl;
  std::cout << (*root.next).data << std::endl;
  std::cout << (*(*root.next).next).data << std::endl;
  std::cout << (*(*(*root.next).next).next).data << std::endl;

  return 0;
}
```

Implement functions:

- `push_back`: This function adds a new Node at the end
- `erase`: This function removes a Node from the list
- `insert`: This function inserts a Node in the list
- `size`: This function tells how many elements are in the list

Hints:


data
next
data
next
data
next
root


data
next
data
next
data
next
root
newdata