(1) Recursion 0

We are used to functions calling other functions. For example, main calls foo0, and foo0 calls foo1, and so on.

```
main(){
    foo0(...);
}

foo0(){
    foo1(..);
}
```

There is another approach to designing programs: a function can be called, and then if certain condition is met, the function calls another instance of itself. This is called recursion.

```
#include <iostream>
using namespace std;
void printRecursive(int n){
    if(n==0)
        return;
    cout << "n = " << n << endl;

    printRecursive(n-1);
}
int main(){

    printRecursive(3);
    return 0;
}
```

(2) Recursion 1

Example problem: write a recursive function to solve n!

n! = n x n-1 x … x 1

```
int fact(int n){
    if(n>1)
        return n * fact(n-1);
    else
        return 1;

}
```

test: find 3!

(3) The Call Stack

We have implemented a generic data structure that follows the Stack ADT.

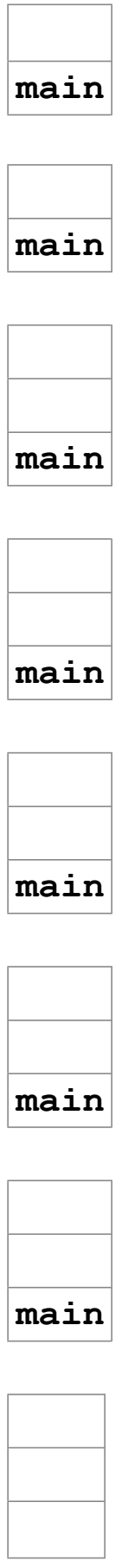
The Stack has many applications:

- One very important one is the **Call Stack**

The Call Stack is important to understand, as it governs the flow of control of the programs you write.

Example:

```
void foo1(){
    ...
}
void foo0(){
    foo1();
    ...
}
int main(){
    foo0();
    foo1();
    ...
}
```



(4) Recursion 2

Code:

```
int out = fact(3);
```

Call stack:

main()

```
fact(3)
if(n>1)
    return n*fact(n-1)
```

fact()	n=3
main()	

```
fact(2)
if(n>1)
    return n*fact(n-1)
```

f()	n=2
f()	n=3
main()	

```
fact(1)
if(n>1) false
else
    return 1
```

f(1)	n=1
f(2)	n=2
f(3)	n=3
main()	

```
fact(2)
if(n>1)
    return n*fact(n-1)
```

f(2)	n=2
f(3)	n=3
main()	

---

f(3)	n=3
main()	

---

main()	out = 6

```
int fact(int n){
    if(n>1)
        return n * fact(n-1);
    else
        return 1;

}
```