Topics for Today

- Revision
  - Templating
  - Iterators
- CS Theory
  - Half-closed intervals
  - Recursion
- This week’s lab
  - Recursive Factorial
  - Recursive vs Iterative Fibonacci
  - Adding Iterators to MyVector

Templates

Templates allow us to make functions reusable for different types.

In the line `template<typename T>`, `T` is our name for our generic type, it's like a stand-in for `int`, or `std::string`, or `Node*`, etc; whatever it needs to be in an implementation.

```
template<typename T>
T TemplatedMax(T a, T b)
{
    return (a > b) ? a : b;
}

int main()
{
    int max = TemplatedMax<int>( a: 10,  b: 42);
}
```

The compiler will make/call this for us

```
int TemplatedMax(int a, int b)
{
    return (a > b) ? a : b;
}
```

Half-closed Intervals

Half-closed intervals is the name given to an indexing convention commonly used in programming.

In mathematical notation we can write it as [low, high)
This means that the boundary at low is *included*, and the boundary at high is *excluded*.

It basically means that for your range, you start at low, and stop one before high.
This is how we usually deal with arrays anyway.
i.e.

```
for (int i=0; i < array.size(); i++)
```

The range is equivalent to *[0, array.size)*
So we start at the inclusive 0 index, and stop when we reach the excluded high index.

Divide and Conquer

There are a few programming paradigms we will be looking at in this course. Today we are looking at *Divide and Conquer*, which is about breaking problems down into smaller parts, and solving the sub-problems and recombining them into an overall solution.
This makes divide and conquer naturally well suited to recursive programming, but it does not necessarily need to be programmed that way.

Some D&C algorithms we'll be studying include:

- Mergesort
- Quicksort

Recursion

Most of the algorithms we have written so far dealing with sequences have done so in an *iterative* manner. Today we are going to use an alternative methodology, and solve them *recursively*.

Recursion is about functions that call themselves. Each function call either creates another function call on a simpler input, or when the input is adequately simple, it can immediately return a value, which is then passed up into the previous function calls, constructing the total answer to the problem.

Recursion can be a much easier and natural way to implement certain algorithms than iteration.

Recursion suits dealing with nodes very well, which to do iterative requires numerous nested loops, so recursion can result in much code.

Recursive Factorial

Factorial (!) is a mathematical operator that returns the product of all integers between the supplied number and one.

3! = 3 * 2 * 1 = 6

4! = 4 * 3 * 2 * 1 = 24

Previously in week 2, we solved factorial iteratively using the code:

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

n = 3, result = 1

i = 2, (2 <= 3), result = 2

i = 3, (3 <= 3), result = 6

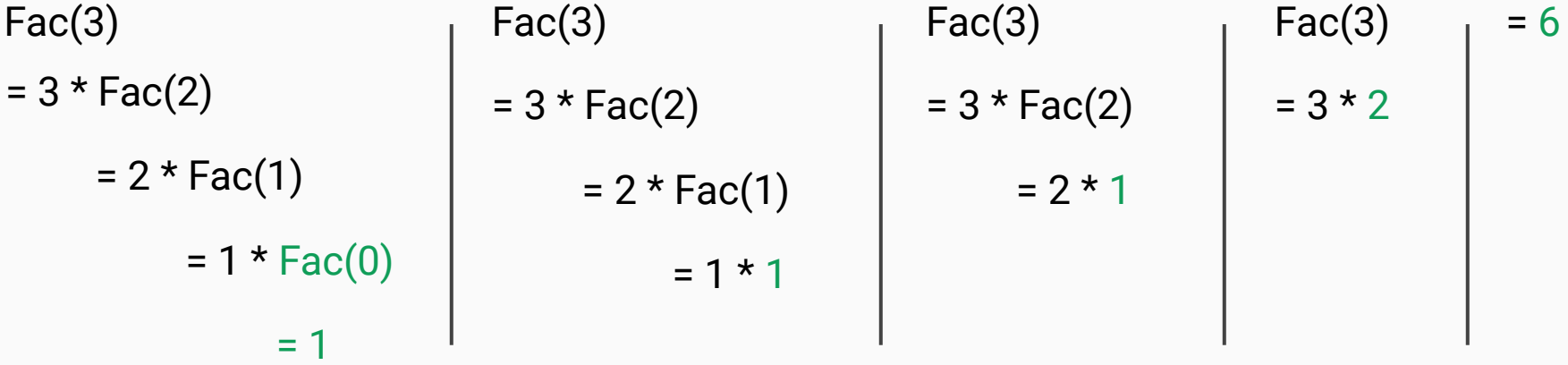
i = 4, (4 <= 3), no-operation

return result

Recursive Factorial

This week, we want to write this again recursively, that is by having the function simplify its input and call itself, instead of by using a for-loop and iterating towards the answer.

To calculate recursively, we need a point when our input gets so simple, we can immediately return the solution. For us, this case is that 0! = 1, so when n = 0, return 1.



Fibonacci Numbers

Do we remember the Fibonacci Numbers?
They follow this intuitive formula:

$$F(0) = 0,\ F(1) = 1$$

$$F(n) = F(n-1) + F(n-2) \text{ for all } n > 1$$

These numbers form a sequence, each element being the sum of the two prior.
F = 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, …
n = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, …

We are going to build a function to get the nth number in the sequence in both iterative and recursive styles, to show the compare the two.

To write it recursively, either call your function again twice with n-1 and n-2, or if it's small enough, return either 0 or 1.

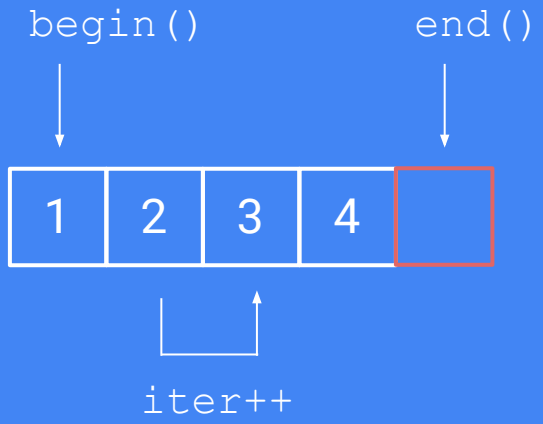
Recursion

Unfortunately when we get to the execution and hardware level, recursion does not tend to do as well as iteration. There is a greater overhead associated with function calls which are the nature of recursive design. They also consume more memory because they hold so many unfinished functions at a given time.

That said, it is still a good option.
It lends itself nicely to node based and graph theory data structures, and this is what you will be working on in assignment 2

Iterators

Iterators are objects that allow us another way of iterating through an array/list like data structure.
An iterator is a pointer to a place in memory within our list. We can increment and decrement them (traverse the list),
and dereference them to see what is inside.




```cpp
// Declare a vector that contains three ints {1, 2, 3, 4}
std::vector<int> vec{ 1,2,3,4 };

for (auto iter = vec.begin(); iter != vec.end(); ++iter) {
    std::cout << *iter << " ";
}
std::cout << "\n";
// Alternatively
std::list<int>::iterator iter;
iter = lis.begin();
while (iter != lis.end()) {
    std::cout << *iter << " ";
    iter++;
}
```

```
1 2 3 4
```

```
1 2 3 4
```

When our iterator reaches the end address, it is no longer in the valid range of our item, and we stop before trying to access the value stored at this address.

Adding Iterators to MyVector

Last week we ran out of time to template MyVector, and we are starting with that already done. Ask me if that causes any confusion. It just means that MyVector can store different types now, like you can have a `Vector<int>` or `Vector<float>`, etc.

Today we are making an Iterator class for our MyVector class. Iterator class supplied in the `.hpp` file.
I recommend you work in the following order:

In this activity we will be writing four member functions for the iterator class:

- `MyVector<T>::Iterator::Iterator(T* input)`- Constructor, make an iterator on *input*
- `T& MyVector<T>::Iterator::operator*()`- Return the value stored at the point the iterator is on
- `typename MyVector<T>::Iterator& MyVector<T>::Iterator::operator++()` Move the iterator one position forward
- `typename MyVector<T>::Iterator& MyVector<T>::Iterator::operator--()` Move the iterator one position backward

And two functions for the MyVector class:

- `typename MyVector<T>::Iterator MyVector<T>::begin()` return an iterator for the first element of the vector
- `typename MyVector<T>::Iterator MyVector<T>::end()` return an iterator after the last element of the vector