in this module we will be using C++ as the programming language and we will be covering algorithms and data structures

split into two terms, this term we will be covering
- C++, pointers, dynamic memory
- lists, stacks, queues, trees, sets, graphs

next term higher level algorithms are covered

Example code

- will be placed on git hub and code will be formatted according to the GNU coding standard
- \( \text{https://github.com/gaiusm/examples} \)
- to obtain all these examples, open up a terminal and type:

```
$ git clone https://github.com/gaiusm/examples
$ cd examples/c++
```

Data structures

- will be covered and implemented in C++
- will be adopting a functional programming approach (where it is practical)
  - using Dijsktra’s pre and post conditions where possible
  - recursion will be exploited to derive simple almost provable solutions

Example: Fibonacci sequence

- is a sequence of numbers: 1, 1, 2, 3, 5, 8, 13, 21, etc
  - the next value is the sum of the previous two
- could express this in pseudo code as:

```
if n<=2 then fib(n) = 1
else fib(n) = fib(n-1) + fib(n-2)
```
C++ implementation of the Fibonacci function

#include <cstdio>
static const int terms = 12;
/
* fibonacci - generate nth term in the classical sequen
* precondition: n > 0
* postcondition: returns the nth term
*/
static int fibonacci (int n) {
  if (n <= 2)
    return 1;
  else
    return fibonacci (n-1) + fibonacci (n-2);
}

Implementation notes
- notice that we can use printf within C++
- we can also declare int i within the for loop
- declare term as a const int. static means local to this file only.
- the rest looks like C

Compile the source file
- compile the single source file into an executable
  ```bash
g++ -O0 -g -Wall fib.cpp
```
- run the executable
  ```bash
gdb ./a.out
(gdb) run
(gdb) quit
```
- and again using valgrind
  ```bash
valgrind ./a.out
```
Functional coding style

- notice the functional coding use of recursion
- a criticism of this style is that it is slow
- however, this is not always true as compiler technology will often convert a recursive solution into an iterative one
- particularly tail recursive algorithms and small functions
- many of the algorithms we will look at during this term fit this pattern

Example performance test

```c++
#include <cstdio>
static const int terms = 45;
/*
 * fibonacci - generate nth term in the classical sequence.
 * precondition: n > 0
 * postcondition: returns the nth term
 */
static int fibonacci (int n)
{
    if (n <= 2)
        return 1;
    else
        return fibonacci (n-1) + fibonacci (n-2);
}
```
check runtime speed

```
$ time ./a.out
Fibonacci value for the first 45 are: ... 1134903170
real 0m3.143s
user 0m3.140s
sys 0m0.000s
```

much better, but still too slow, why?

After compiling and testing our program

- examine the code generated by the compiler
- open up fibspeed.lst and search for call
- which areas of code use calls?

we observe that the compiler has removed one recursive call to fibonacci \((n-2)\) but not the other call to fibonacci \((n-1)\) in the sequence

```
c++/fib/fibspeed.cc

static int fibonacci (int n) {
if (n <= 2) return 1;
else
  return fibonacci (n-1) + fibonacci (n-2);
}
```
Consider the function Sum

\[ x = \sum_{i=1}^{n} i \]

pseudo code

```c
sum (lower, upper)
    if lower <= upper then return lower
    else return lower + sum (lower+1, upper)
```

```c
#include <cstdio>
static const int low = 1;
static const int high = 1000000;
/*
 * sum - generate the sum of terms lower..upper.
 * precondition : lower <= upper.
 * postcondition: returns the sum of lower..upper.
 */
static int sum (int lower, int upper)
{
    if (lower == upper)
        return lower;
    else
        return lower + sum (lower + 1, upper);
}
```

```
/*
 * main - first user function executed.
 * precondition : none.
 * postcondition: returns 0 (silently).
 */
int main (int argc, char *argv[])
{
    printf("Sum of numbers from %d..%d is: ", low, high);
    printf("%d\n", sum (low, high));
}
```

compile and debug this via:

```
$ g++ -g -O0 sum.cpp
$ gdb ./a.out
(gdb)
r
```

the stack is being exceeded, when processing the recursive calls
Consider the function Sum

- let us try compiling with -O3

```
g++ -g -O3 sum.cpp
(gdb) run
(gdb) quit
```

Sum of numbers from 1..1000000 is: 1784293664

Consider the function Sum

- check the assembly language as before

```
g++ -Wall -S -fverbose-asm -g -O3 sum.cpp -o sum.s
as -allnd sum.s > sum.lst
```

- observe sum.lst and see the compiler has transformed the recursive algorithm into a very tight iterative loop!

Conclusion

- we have seen that a functional approach can be adopted

- sometimes the compiler is able to transform a recursive algorithm into an iterative solution (when tail recursion is used)

- other times it cannot - we need to be aware of these limitations and profile code accordingly