1. Write a recursive and an iterative version of the Fibonacci function, with the following signature:

   ```
   int fib(int b);
   ```

   Recursive:

```
int fib(int n) {
    if (n <= 1) {
        return n;
    }

    return fib(n - 1) + fib(n - 2);
}
```

   Iterative:

```
int fib(int n) {
    int a = 0;
    int b = 1;
    for (int i = 0; i < n; i++) {
        int tmp = a;
        a = b;
        b += tmp;
    }

    return a;
}
```

2. What string X would make the following code:

   ```
   printf(X, "Alice", 123, 10.0);
   ```

   Print:

   ```
   Hello Alice, your order code is 00123, it will cost £10.00.
   ```

```
"Hello %s, your order code is %05d, it will cost you £%.2f."

We could use "%05u" instead of "%05d" since we know we are
probably only dealing with unsigned integers.
```

3. What is a null-terminated string?

An array of characters that ends with the null character '\0' (which marks the end of the string).

To store a string that contains 10 characters, we need 11 bytes (+1 due to null character)

4. Consider the following declarations:

```
int **test;

int arr[2][3]; -> an array of arrays
```

Describe what the following expressions mean:

- **test
- *test = &another
- &*test
- (*test) + 1
- test[1][2]
- arr[1][2]

- **test dereferences *test, which is a pointer to an int, so **test is basically the int value that is pointed to by *test (and test points to *test)

- *test = &another makes the pointer that "test" points to (i.e. *test) point to the address of "another"

- &*test = address of *test = test

- *test points to the address of **test, so (*test) + 1 will point to the next int in memory (address of **test + sizeof(int))

  When we do pointer arithmetic in C, the "+1" will be in the units of the size of the type that the pointer references

- test[1] = *(test + 1), so test[1][2] = *(*(test + 1) + 2)

  Note that *(test + 1) != *(test) + 1

- arr[1][2] = the element in the second row, third column of arr

  arr[x][y] = *(arr + x * <width> + y) -> e.g. if we declared int arr[2][3], then arr[1][2] = *(arr + 1 * 3 + 2)

5. Write an implementation of:

```
int strcmp (const char *str1, const char *str2);
```

Recall that strcmp should return:

- 0: if the strings are equal.
- < 0: if str1 lexicographically comes before str2.
- > 0: if str1 lexicographically comes after str2.

```
int strcmp(const char *str1, const char *str2) {
    while (*str1 && (*str1 == *str2)) {
        str1++; // str1 += sizeof(char)
        str2++;
    }

    return (unsigned char) *str1 - (unsigned char) *str2;
}
```

6. Give a struct or class definition for a binary tree node, that is templated on arbitrary types for the key and value.

```
template <typename T>
class TreeNode {
public:
    T val;
    TreeNode* left;
    TreeNode* right;

    // Constructor
    TreeNode(T value) : val(value), left(nullptr),
right(nullptr) {}
};

template<typename K, typename V>
struct tree_node {
     K key;
```

```
    V value;
    tree_node *left, *right;
};
```

7. What is __attribute__((packed)), and how does it help OS developers?

```
__attribute__((packed)) is a GCC/Clang compiler feature that
affects struct memory layout. It removes padding between
members of a struct.

Example…

struct Normal {
    char a; // 1 byte + 3 bytes of padding
    int b; // 4 bytes
};

struct __attribute__((packed)) Packed {
    char a; // 1 byte (no padding)
    int b; // 4 bytes
};

Using packed structs might result in slower runtime since data
becomes unaligned (no longer in multiples of 4).

However packed structs do help to reduce the amount of memory
used -> useful when dealing with stuff like communication
protocols, where we want the size of the data to be as small
as possible.
```

8. Implement the member functions insert and remove for the following circularly linked list node definition:

```
template<class T>
class list_node {
public:
    explicit list_node (T data)
        : prev_(this)
        , next_(this)
        , data_(data) {}
    void insert(list_node *n) {
```

```
            . . .
        }
        void remove() {
            . . .
         }
        const T& data() const { return data_; }
    private:
        list_node *prev_, *next_;
        T data_;
    };
```

```
void insert(list_node* n) {
     n->prev_ = this;
     n->next_ = this->next_;
     this->next_->prev_ = n;
     this->next_ = n;
}

void remove() {
     this->next_->prev_ = this->prev_;
     this->prev_->next_ = this->next_;
     this->prev_ = this;
     this->next_ = this;
}
```