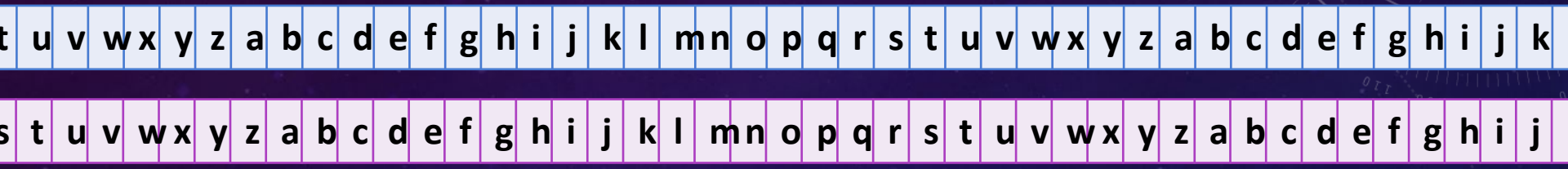
ENGR 101 – Chapter 17

Program Design in C++

3/13/21

Caesar Ciphers



File Structure

- We can split our code into separate modules:
  - `caesar.cpp` – Contains functions for encrypting/decrypting via a Caesar cipher. The functions are generally useful and could potentially be used in many different projects.
  - `encryptDocument.cpp` – The main driver program. It takes care of opening a file, calling functions from `caesar.cpp`, and writing output.

`encryptDocument.cpp`

```
#include <iostream>
// other library includes



using namespace std;

int main() {
  // use caesar functions here
  // to implement the main program
  encrypt_word( ... );
}
```

Error! The compiler encounters a call for `encrypt_word` here, but it hasn't been declared in this file!

`caesar.cpp`

```
#include <string>
// other library includes

using namespace std;

// Implementations for caesar
// functions.
string encrypt_word( ... ) {
  // do something useful
}
```

```
g++ encryptDocument.cpp caesar.cpp -o encryptDocument
```

Recall: Function Prototypes

- A **function prototype** *declares* a function before it is actually *defined*.
  - It is written as the function signature followed by a `;`.

```
int square(int n);

int main() {
  int x = 3;

  cout << "x = " << x << endl;
  cout << "x squared = " << square(x) << endl;
}

// Returns the square of the given number
int square(int n) {
  return n * n;
}
```

"Hey compiler, just FYI there will be a function called `square` that works like this, so don't worry if you see it used somewhere."

Don't forget to actually define the function later on in your file, outside of `main()`

File Structure

- Add a file called `caesar.h`
  - This is a "header" file (thus the `.h`), because we will include it at the top of other files.
  - Note the syntax for `#include` is different. Use < > for libraries, use " " for your own files.

caesar.h

```
// Function prototypes for
// each function in caesar.cpp.

string encrypt_word( ... );
char shift_letter( ... );
...
```

encryptDocument.cpp

```
#include <iostream>
// other library includes

#include "caesar.h"

using namespace std;

int main() {
  // use caesar functions here
  // to implement the main program
  encrypt_word( ... );
}
```

The compiler knows about `encrypt_word` due to the `#include`.

caesar.cpp

```
#include <vector>
// other library includes

#include "caesar.h"

using namespace std;

// Implementations for caesar
// functions.
string encrypt_word( ... ) {
  // do something useful
}
```

```
g++ encryptDocument.cpp caesar.cpp -o encryptDocument
```

Unit Testing

- Programs often use many functions working together.
- In **unit testing**, we test each function individually to make sure it behaves as it should according to its **interface**.

- Generally, this amounts to:
  - Running the function with a bunch of inputs
  - Verifying it produces the right output for each one

Unit Testing: Example

- Here's a single unit test for the built-in max function:

```
int main() {

  string original = "cat";
  int offset = 1;
  string expected = "dbu";

  string result = encrypt_word(original, offset);
  assert(result == expected);

  assert(encrypt_word("cat", 1) == "dbu"); // more concise
}
```

- `assert`
  - A built-in MATLAB function used for testing.
  - It ends the program with an error message if its input is not `true`.

Writing Unit Tests

- A unit test checks the behavior of an individual component or function.
- You can write unit tests in a separate file with its own main function.

caesar.h

```
// Function prototypes for
// each function in caesar.cpp.

string encrypt_word( ... );
char shift_letter( ... );
...
```

test_caesar.cpp

```
#include <iostream>
// other library includes

#include "caesar.h"

using namespace std;

int main() {
  assert(encrypt_word("cat", 1) == "dbu");
  assert(encrypt_word("cat", 3) == "fdw");
  assert(encrypt_word("zzz", 3) == "bbb");
  assert(encrypt_word("it", -2) == "gr");
}
```

caesar.cpp

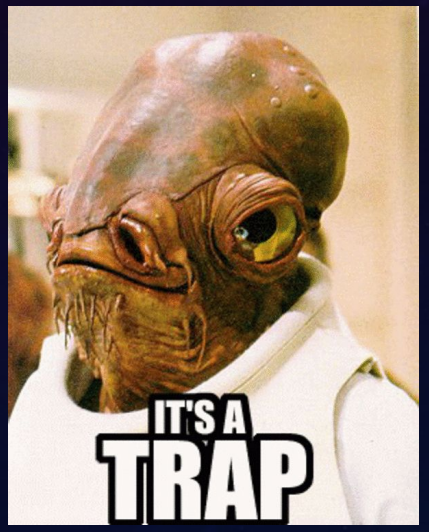
```
#include <vector>
// other library includes

#include "caesar.h"

using namespace std;

// Implementations for caesar
// functions.
string encrypt_word( ... ) {
  // do something useful
}
```

```
g++ test_caesar.cpp caesar.cpp -o test_caesar
```

Floating Point Precision

- Computers can't perform floating-point math perfectly.
  - Limited memory means limited precision

```
int main() {
   double x = 0.1;
   double y = 0.2;
   if(x + y == 0.3) {
     cout << "equal" << endl;
   }
   else {
     cout << "not equal" << endl;
   }
}
```



Comparing Floating Point Numbers (i.e. doubles)

- It's not safe to use == or != with floating point numbers.
  - The results of computations that *should* be equal may not turn out to be *literally* equal, due to limited precision.
- Instead, check whether the numbers are very close…

```
bool almostEqual(double x, double y) {
  double diff = x - y;
  if(diff < 0) {
    diff = -diff;
  }

  return diff < 0.0001;
}
```

This is often called an "epsilon value".

```
int main() {
  double x = 0.1;
  double y = 0.2;
  if( almostEqual(x + y, 0.3) ) {
    cout << "equal" << endl;
  }
  else {
    cout << "not equal" << endl;
  }
```

NO `assert(sqrt(18.0625) == 4.25);`

YES `assert(almostEqual(sqrt(18.0625), 4.25));`

This is how we check your numeric solutions on the autograder :)