1 Testing

- Must do before you start coding!
- Not the same as debugging - we must first test before debugging.
- Testing cannot guarantee correctness - it can only prove wrongness.
- Test suites are used to show/test expected behaviour.
- The difficulty in debugging is that there is no real process to test every single program to make sure your program works.
- There's also a psychological barrier, as people don't like seeing their code not working.
- IDEALLY (not in this course, unfortunately), the developer and the tester should be DIFFERENT PEOPLE.
- Black box testing vs. white box testing (black is test w/o knowing implementation, white is test specific parts).
- Human testing:
  - Humans look over code and find flaws.
  - One person should execute a program by hand to make sure the logic is correct.
  - Typically this method only spots logic and syntax errors - stuff that is easily spotted.
  - We don't have the ability to do this on assignments.
- Machine testing:
  - What we will do.
  - Run the program using test cases (white and black). The point of the dual due dates is to force you to do black box testing FIRST, then do white box testing after.
  - There is ”grey box” testing, which is like a hybrid of the two - w/ some knowledge of program implementation.
  - We should always start with black, supplement with white.
  - Note that we are not expected to test for invalid inputs, unless the problem states that you should consider said invalid input.
- Testing strategies:
  - Functional testing: does the program execute correctly?
  - Performance testing: is the program efficient (enough)?
  - Regression testing: do new changes to the program break old test cases? NEVER remove test cases - always just add more!
  - Volume testing: can the program handle both small and large inputs?

2 Basics of C++

- main is an int, as per C.
- std::cout (or just cout if you use namespace std) is our std. output.
- Use “using namespace std;” if you are planning on using just cout, endl, cin, etc.
- Like C, by default, main will return 0.
- Note that we can use bash’s echo on C++ programs, so if we make our main return, say, 314, then “echo $?” will return 314.
- We compile C++ using the command “g++ -std=c++14 program.cc -o program-name”.
- Note we are using C++14 - even though 17 is out, we’re not using it for now.
- If we do not use a “-o”, it will automatically compile to a file called “a.out”, or the like. It’s more convenient to name our files when testing.

3 I/O in C++

- Include “iostream”.
- We have three types of I/O streams:
- cout: printing to stdout.
- cin: recording from stdin.
- cerr: for printing to stderr.
- We use double chevrons to direct to cout or cin.
- ¡¡ means “put to”. For example, “cout ¡¡ x” will put x to cout to output.
- ¿¿ means ”get from”. For example, “cin ¿¿ x” will put the value from cin to x.
- We can interpret the direction of the chevrons as the direction of the flow of information (prof’s words, not mine).
- We can do something like “cin ¿¿ x ¿¿ y” to get two inputs, stored into x first, then y. Think of it like “cin ¿¿ x; cin ¿¿ y;”
- cin ignores whitespaces, and will stop at the first one. So like with the previous example, input of “3 0 5” would give x = 3, y = 0.
- What happens if cin fails or ends?
- If a cin read fails, cin.fail() will return true.

- If eof is entered, cin.eof() and cin.fail() will both return true AFTER the attempted read fails.
- There is an implicit conversion from cin to bool. cin is regarded as true if it completes successfully - we can simply say “(cin)” to see if the previous read was sucessful (true if successful).
- We can go even further beyond - “if (!(cin ¿¿ x)) break;” is a very easy way to break out of a cin loop.
- The “¿¿” and “¡¡” are also bit shift operators. It depends what is on the left - cin or cout is the expected I/O behaviour, but if it is a number, it will cause a bitwise shift.
- We use “cin.clear()” to reset our cin stream.
- We use “cin.ignore()” to get our program to ignore the current value that was just inputted that is waiting - otherwise, that value will get stuck in the stream! We must clear AND ignore.

4 Hex and decimal manipulators

- We use <iomanip .
- The following code would print “5109”

```
int i = 5;
int j = 190;
cout << i << j << endl;
```

- We use “setw” to specify the minimum spaces for the next numeric or string value.i
- The following code would print “___5__109” - four spaces for the 5, five spaces for the 109:

```
cout<<setw(4)<<i<<setw(5)<<j<<endl
```

- By default, C++ will print numberical values in decimal notation.
- If we want to print something in, say, hexadecimal notation, then we must redirect into it:

```
cout<<95<<endl; //Prints in decimal
cout<<hex<<95<<endl; //Prints in hexadecimal
cout<<95<endl; //Prints in hexadecimal
```

- As noted above, the effects of switching numerical formats will persist until you switch to another format!

5 Strings in C++

- C++ has much better support for strings than C does.
- Stings grow as needed in C++. There is no need for manual dynamic memory management as you add or remove letters!
- It is also much easier to manipulate strings. A simple "s = "hello";" is fine, there is no need to worry about null terminating.
- String operations:
  - =, != both work on strings as expected.
  - ¡. ¡=, ¿, ¿= work on strings as expected (lexographic order).
  - To get the length of a string, use "s.length()". This will be like strlen, returning the length of the true string.
  - To fetch individual characters, simply use "s[n]", where n is an integer.
  - To concatanate strings, we can simply do "s3 = s1 + s2;". "s3 += s4;" is also valid syntax.
  - See the previous Spring course notes for more examples.
- To use these operations, you must include the ¡string¿ library. Note that string - the type - is built into std, but string MANIPULATIONS require the string library. Yep.
- To declare a variable of type string, we would do something like the following:

```
#include <string>
using namespace std;
int main() {
    string s;
    cin >> s; //read into s, skip leading whitespace, stop at next ws.
    //IOW, read a word
    cout << s << endl;
}
```

- We note that cin would only get us individual words. Short of jury-rigging some loop, we need something else to get multi-word strings. We use "getline(cin, s);" to read EVERYTHING one enters into stdin until a newline (Enter) is given, and stores it into s.

6 Reading from a file

- We note that so far, we've only been reading from stdin.
- We use ifstream/ofstream to read/write from/to a file.

- We have to include the library "fstream" to do so. It allows us to "stream" to stream from and to files - like how iostream lets us stream from stdin and stdout.
- We can see fstream in action, the code prints every word until we hit eof.

```
#include<iostream>
#include<fstream>
using namespace std; //required for std::string, std::ifstream...
int main() {
    ifstream file{"suite.txt"};
    string s;
    while (file >> s) { //Read until we hit eof from file (suite.txt)
        cout << s << endl;
    }
    //We don't have to close file, the scope of file will mean it will cl
}
```

7 String stream

- Useful to build up a string, then output it.
- Requires the library sstream.
- We use "ostringstream" to write to a given string.
- We use "istringstream" to read from a given string.
- An example of us "throwing" values into our string stream can be shown:

```
int main() {
    ostringstream ss;
    int low = 1;
    int high = 1;
    ss << "Enter a number between " << low << " and " << high << ":";
    string s = ss.str();
    cout << s << endl;
    int input;
    cin >> input;
}
```

- Let us make a program that reads in numbers. Remember how in C, if you read a character instead of a number, things went to hell? We'll try to resolve that in another method (you can do this with just cin, as above):

```
int main() {
    while (true) {
        cout << "Enter a number:" << endl;
        string s;
        cin >> s;
        istringstream ss{s};
        if (ss >> n) break;
        cout << "I said to only"; //...enter a number.
Loops back to top.
    }
    cout << " You entered the number " << n << endl;
}
```

- Basically, “ss ¿¿ n” tries to convert ss to an int. If it fails, it will return false. If it works, it returns true. Obviously we can’t convert strings or chars to ints, so it’ll fail for any non-int.

8 Function predeclaration

- Like C, there where be dependancy errors if you just make two functions that are dependent on each other.
- You must declare the function before the dependent function Otherwise, it won’t properly compile!
- We can only DEFINE a function once, but DECLARE more than once (as of now).

9 Pointers

- Good news - they’re the exact same as C pointers! Bad news - they’re the exact same as C pointers. May references say my soul.
- Like C, C++ is pass by copy. Passing a value like foo(n) and incrementing n in foo won’t do anything to the original value of n.
- Remember aliasing, as it’s the same behaviour here.
- Also remember that a pointer will point to the first element in an array, like C.
- The null pointer in C++ that we will use is “nullptr”. Don’t use NULL, even though it is in C++.
- For example, to initalize a node struct, we would do something like “Node n1 5, nullptr” if we don’t want it to be linked to any other node. NOTE THAT FOR A STRUCT INIT, WE DO NOT NEED THE “struct” part!
- An old example, but to increment a value that is passed:

```
using namespace std;
void inc(int *n) {
    ++(*n);
}

int main() {
    int x = 5;
    inc(&x);
    cout << x << endl; //Prints 6
}
```

10 Default function parameters

- Consider the following bit of code:

```
void printFile(string name = "test.txt") {
    //Print the contents of file "name"
}

int main() {
    printFile("test.txt");
    printFile(); //These two lines of code will give the same output!
}
```

- Basically as it has a default value, not calling with an argument will default to whatever was set.

11 Overloading

- Basically Java's overloading.
- We can define a function more than once as long as the parameters are different.
- This makes sense if we think of it at a 251 level. The machine code doesn't care what the function's names are, nor their parameters. It just cares about how many and the type.
- Like Java, which function is called will depend on the arguments. The compiler is smart enough.
- Consider, say, you pass the integer "0" to a function that is overloaded to a bool parameter and an int parameter. By default, this will call the int function. If you want a bool, pass a variable of type bool.
- Consider the following example:

```
void f(int a, int b=1) {
    //...
}
void f(int a){
    //...
}
```

- What happens in the case above if we call f(5)? Well, since BOTH cases are completely valid, the compiler may just outright refuse to run.
- IOW, if you're confused, the compiler will also be confused, so long story short, don't write shitty code like this.
- Overloading is useful as we can reuse function names if the process is still the same, but it needs different arguments.

12 Constants

- What is the difference between something like:

  ```
  int n = 5;
  const int m = 5;
  ```

- We can easily see that the difference is that n is mutable and stored in stack/-global, m is immutable, and stored in read-only memory.
- As in C, make variables constant if needed.
- Consider the earlier example, where we initalied “Node n1 = 5, nullptr;”. If we say ”const Node n2 = n1;”, then now the values of n2 are immutable. We can make it point to other Nodes though (need to doublecheck, but should be!).
- Recall constant interactions with pointers. “const int *p = n;” gives us a pointer that we can mutate what it points to, but NOT what it points at (so we can say p = m, but NOT *p= 100).
- Typically, we read from right to left. So “int * const p = n” reads as “p is a constant pointer to an int”. The above reads as “p is an pointer to an integer that is constant”. We cannot change what it points to but we canchange the VALUE of what it points to with aliasing.
- Finally, “const int * const p = n” is ”a pointer that is constant that points to an integer that is constant”. We cannot change what p points to or the value of whawt it points to.

13 References

- C++ has another pointer-like type – references.
- This is how something like "cin ¿¿ x" works. It does NOT use a pointer.
- A reference is defined as
- For example:

  ```
  int y = 10;
  int &x = y; //x is of type "reference to integer". Functionally, it is t
  ```

- We define x as an lvalue reference.
- An lvalue is any value that can be on the left side of an assignment. A rvalue is any value that cannot be on the left side of an assignment. For example, x is an lvalue, y is an lvalue, 10 is an rvalue, "this is a string" is an rvalue. It's pretty obvious, tbh.
- We could say that the reference x to y is similar to saying "int *const z = y;". Basically, a reference cannot change what it points to, but we CAN change the value of what it points to.
- Consider changing the value of y through x:

  ```
  int y = 10;
  int &x = y;
  x = 999;
  cout << y << endl; //will print 999
  ```

- Bascially, a reference will automatically dereference for you, if you still want to think of it like a pointer. You could essentially define a reference as a pointer that automatically dereferences and cannot point at anything other than what it is initalized to (for now).
- There are some things that we should keep in mind regarding lvalue references:
  - Do NOT leave a reference uninitalized. It's like leaving a const pointer uninitalized. We cannot just declare a reference (so no "int x;").
  - Do NOT initalize a reference with an rvalue. It MUST be initalized with an lvalue (so no "int x = 5;", this is invalid!).
  - Do NOT create a pointer to a reference. "int *x=...;" would be invalid. However, a reference TO a pointer is fine, so "int *x = ...;" is fine.
  - A reference to a reference is legal, but don't do it as it means something different (so avoid "int x = ...;") for now.
  - An array of references is NOT allowed.
- We can pass references as function parameters:

```
void inc(int &val) {
    val += 1;
}

int main() {
    int x = 1;
    inc(x); //This will be fine, as it'll essentially do int &val = x
    cout << x << endl; //prints 2
}
```

- We can see a reference as a much safer pointer.
- A reference holds an address, so it's 8 bytes (???)
- Check to see which one is better for large data.
- (Mentioned in tutorial) we cannot call, say, f(5) if the function f is defined to accept only an integer reference. We CAN however call g(5), if the function g is defined with parameters of a CONSTANT integer reference. This is as 5 is an rvalue, NOT a lvalue.
- We have three main types of "pass by"'s in C++ (at least so far): pass by copy/-value (C behaviour), pass by reference, and pass by constant reference.

14 Dynamic Memory

- Similar to C in 136.
- In C, if we wanted to use memory that persisted outside of stack, we would allocate memory to the heap. However, there is the issue of memory leaks, so you must free all heap memory.
- C++ follows the same concept, but the syntax differs.
- For example, we don't use malloc nor free.
- Example:

```
struct Node {
    int data;
    Node *next;
};

Node *np = new Node;  //No need to call malloc - new will AUTOMATICALLY a

Node *myNodes = new Node [10]; //Creates an array of 10 Node structures.

delete np;
delete [] myNodes;
```

- Make no mistake - we're doing the same idea of allocing memory in the heap. We just don't have to manually malloc anymore.
- As such, we must still somehow "free" this memory, as it's heap memory.
- As a side note, most IDEs will NOT detect memory leaks (this is really obvious if you try it on something like VS and purposely make shitty code). Always run your code on the student environment first to make sure it works!
- Delete/free heap memory with "delete name". For an array, we simply use "delete [] name;".

15 Returning by Value/Ptr/Reference

- Consider the following three functions:

```
Node getMeANode() {
    //Expensive, returns a copy of n.
    Node n;
    return n;
}

Node *getMeANode() {
    //Undefined behaviour; n will be erased after we pop the stack.
    //Reference points at something that may be overwritten after the sta
    //Do not use.
    Node n;
    return &n;
}

Node *getMeANode() {
    //What we would probably use. Returns a pointer to something stored
    //Thus, the memory the pointer points at will persist post-stack-pop.
But we MUST FREE/DELETE.
    Node *np = new Node;
    return np;
}
```

16 Operator overloading

- Suppose we have the following structure:

```
struct Vec {
    int x, y;
};
```

```
...

Vec v1{1. 2};
Vec v2{3, 4};
//Vec v3 = v1 + v2; //This doesn't compile, as + is defined for numbers a
Unless...

//We add above:
Vec operator+(const Vec &v1, const Vec &v2) {
    Vec v {v1.x + v2.x, v1.y + v2.y};
    return v;
}

Vec v3 = v1+v2; NOW this compiles.
```

- Let's consider ANOTHER example:

```
Vec operator*(const Vec &v1, const int k) {
    return { k * v1.x, k * v1.y };
}

Vec operator*(const int k, const Vec &v1) {
    return v1 * k;
}

Vec v1 = {1, 2};
Vec v2 = 2 * v1; //Valid!
Vec v3 = v1 * 3; //Valid!
```

- Another example:

```
struct Grade {
    int theGrade;
};

int main() {
    Grade g;
    while (cin >> g) { //This won't work!
        cout << g << endl; //This won't work!
    }
}
```

- So how can we get this to work? Consider the following:

```
struct Grade {
    int theGrade;
};
```

```
ostream & operator<<(ostream &os, const Grade &g) {
    //Think of cout << a << b - this is like (cout << a) << b ->
    //cout << b, so cout << a returns a REFERENCE to cout.
    os << g.theGrade << "%";
    return os; //Returns a reference to our output stream.
}

int main() {
    Grade g;
    while (cin >> g) { //This won't work!
        cout << g << endl; //This will work now!
    }
}
```

- We can do something similar for the /textmore /textmore operator:

```
//Previous structure declaration omitted for simplicity.

istream &operator>> (istream &in, const Grade &g) {
    in >> g.theGrade;
    if (g.theGrade < 0) g.theGrade = 0;
    if (g.theGrade > 100) g.theGrade = 100;
    return in;
}

int main() {
    Grade g;
    while (cin >> g) { //This will work!
        cout << g << endl; //This will work now!
    }
}
```

17 Preprocessor

- include
  - “include ...”
  - Recall that for standard libraries, just use include ¡lib¿.
  - We can also include the full file path, like include ¡/.../.../lib¿.
  - For a non-standard library (anything you write), use include ”mylib”.
  - We cannot use an absolute/relative path for a non-standard library. Place your library/file into the same directory as your cpp file (this isn’t really invalid, but whatever...).

- define
  - “define VAR VALUE”
  - Allows us to assign values pre-compliation.
- define w/o any assigned value
  - “deifne FLAG”
- ”Example:”

  ```
  #define BBOS 1
  #define IOS 2
  #define OS BBOS
  #if OS == BBOS
      long long int publickey;
  #elif OS == IOS
      short int publickey;
  #endif
  ```

- We can use defines to debug:

  ```
  #define X

  int main() {
      cout << X << endll
  }
  ```

- We could modify this by saying compiling with “g++14 -DX = 15 ¡FILE¿ ¡COMPILE_TO¿”.
- We could debug something like double loops:

  ```
  #ifdef Debug1
      //loop
  #endif
  #ifdev Debug2
      loop
  #endif.
  ```

- Call with “g++ -DDebug1 -DDebug2 ¡FILE¿ ¡COMPILE_TO¿”

18 Seperate compilation

- Split your program into modules, where each module includes:
  - An interface file (.h): type declarations, and the prototypes of functions.
  - An implementation file (.cc): full details/definition for every function.

- Recall that a decleration asserts existence, while a definition provides the full details and allocates space.
- An example of a module:

```
//vector.h
struct Vec {
    int x;
    int y;
};

Vec operator+(const Vec &v1, const Vec &v2);


//vector.cc
#include "vector.h"
Vec operator+(const Vec &v1, const Vec &v2) {
    //...
}
```

- To use:

```
//main.cc

#include "vector.h"

int main() {
    Vec v{1,2};
    v = v+v;
}
```

- But how does C++ know to link these files together?

```
#Seperately compile, do not link, do not build executable, produce an obj
g++14 -c vector.cc
g++14 -c main.cc

#Produce an execuable, linking the object files, called ``name''.
Omitting the name and -o flag defaults to ''a.out''
g++14 vector.o main.o -o ``name''
```

- Remember to use extern if you are passing a global variable, which ensure that it is only declared, but not defined.
- We can declare multiple times, but not define. A constant must be defined.
- We note the following example:

```
//linalg.h
#include "vector.h"

//linalg.cc
#include "vector.h"
#include "linalg.h"

//main.cc
#include "vector.h"
#include "linalg.h"
```

- This will NOT compile. This is as we call headers too many times, possibly causing duplicate variables and structures.
- To prevent files from being used more than once, we use include guards as so.
- Note that this is not "include guard", the very practice is called "include" guards. Fuck naming conventions. This is NOT CODE.

```
//vector.h
//This is an example of an #include guard.
#ifndef VEC\_H   //See if it is already defined. If not, define it.
If it is defined, skip.
#define VEC\_H
#endif
```

- You MUST include these, or else you will lose marks.
- Never put "namespace std" into a .h file. This will compile, but will FORCE the client into using the "namespace std" convention, which they may not be using, as it's "techincally" bad style to use it in the case std changes or conflicts... but hey it's an intro C++ course, who cares?