CMSC 240

SOFTWARE SYSTEMS DEVELOPMENT

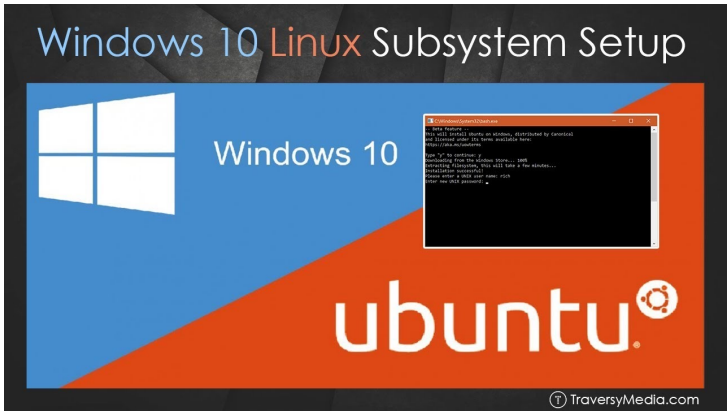
Homework

- https://facultystaff.richmond.edu/~dszajda/classes/cs240/Spring_2023/index.html
- Read:
  - Syllabus
- Tutorial:
  - *Unix Tutorial for Beginners* (Intro, Tutorial 1, Tutorial 2)

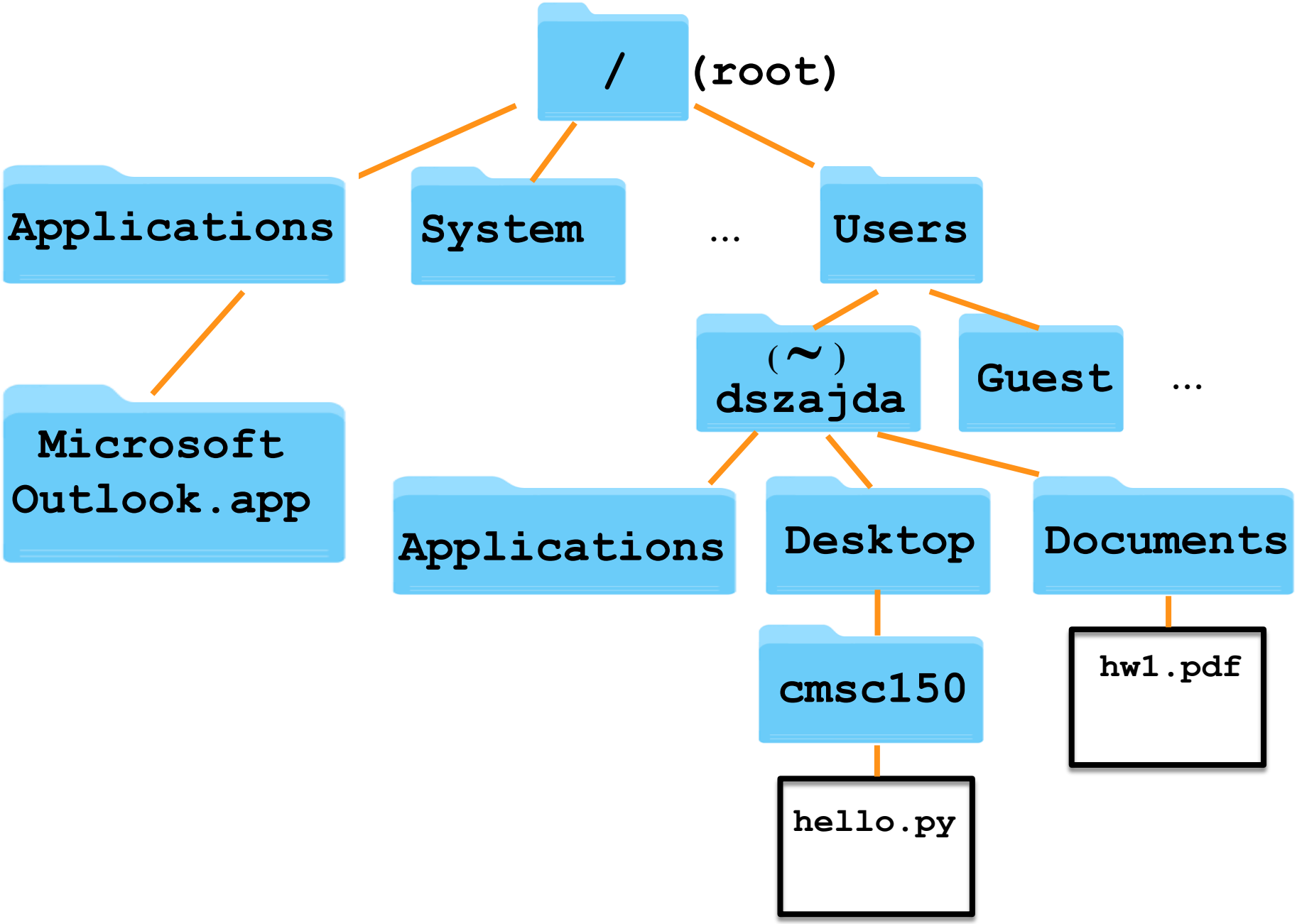
Linux / Unix

- Macs (G02, G14, G04, 225): Unix
- Linux Boxes (225): Linux
- Windows Linux Subsystem (G03)

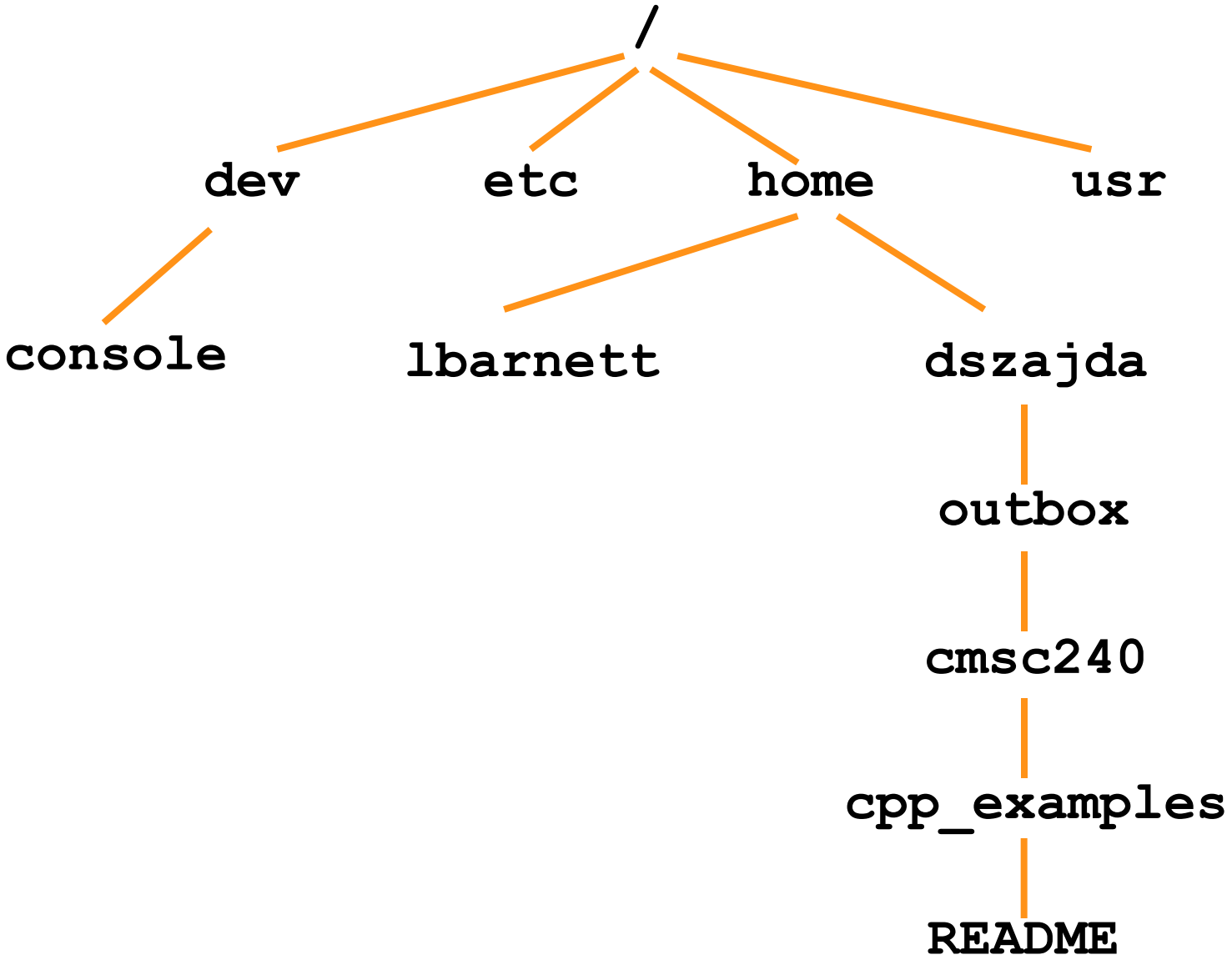
- *nix primarily uses a command prompt
  - Interact via commands typed in window
  - Similar to DOS command prompt



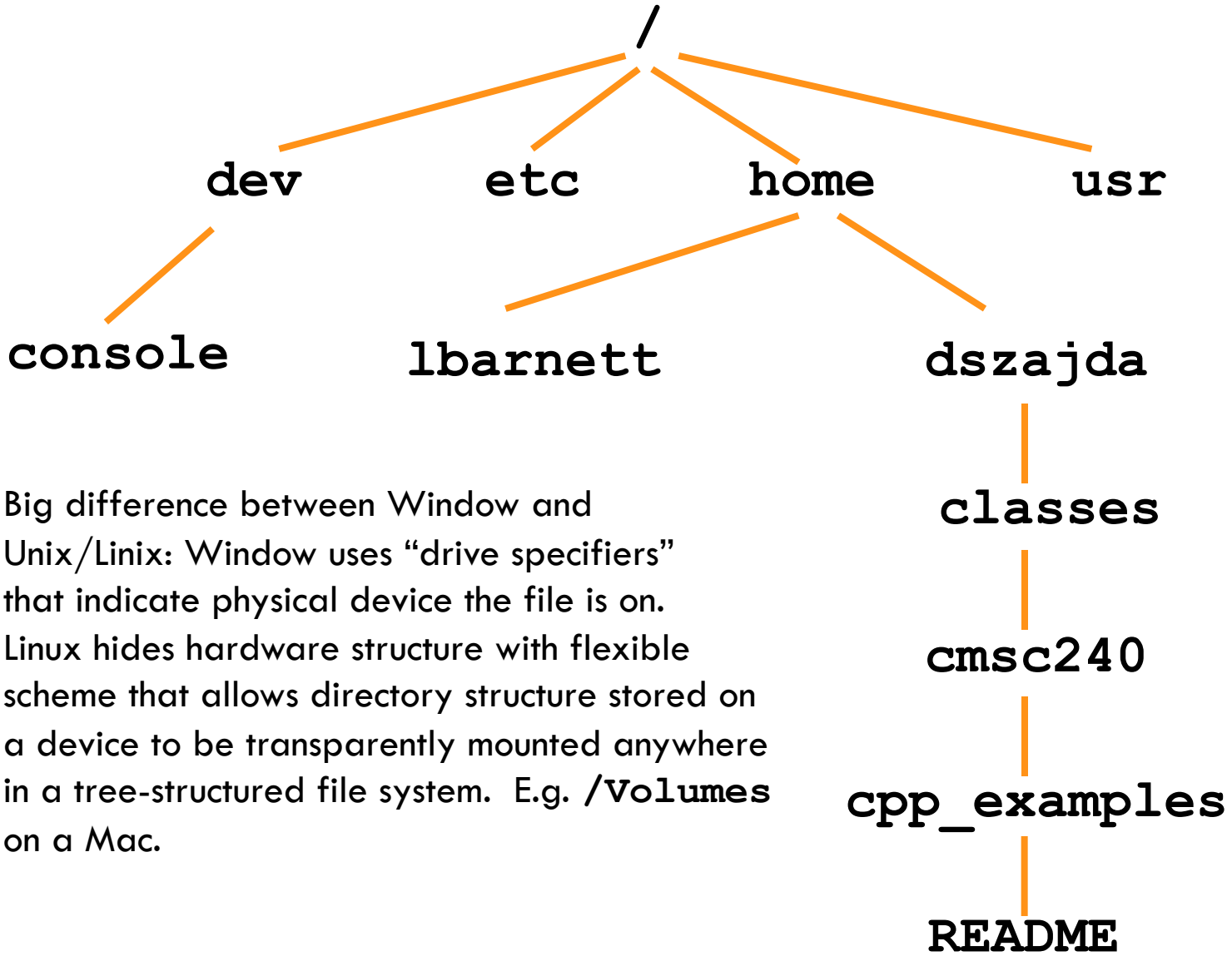
Example Unix File System (on Mac)



Example Unix File System (on Linux)



Example Unix File System (on Linux)



Big difference between Window and Unix/Linix: Window uses “drive specifiers” that indicate physical device the file is on. Linux hides hardware structure with flexible scheme that allows directory structure stored on a device to be transparently mounted anywhere in a tree-structured file system. E.g. **`/Volumes`** on a Mac.

Unix/Linux File System

- Special directory names:
  - Root directory: /
  - Current directory: .
  - Parent directory: .. (allows you to go up)
  - User's home directory: ~
  - Some other user's home: ~sb4tc

- Two primary operations for navigating/locating:
  - `cd <name>` change directory to "name" (relative)
  - `ls` list all files/directories in current directory

Unix/Linux File System

- Two primary operations for navigating/locating:
  - `cd <name>` change directory to "name" (relative)
  - `ls` list all files/directories in current directory
- Special cases:
  - cd (with no arg) goes to your home directory
  - cd . does nothing
  - cd .. moves to the parent of the current directory
  - ls allows wildcards, e.g., ls *.cpp lists all files ending in .cpp

Example Terminal Commands

`cd ~`	change to home directory
`mkdir cmsc240`	make a new cmsc240 directory
`cd cmsc240`	cd to the cmsc240 directory
`pwd`	print the working directory
`echo "Hi!" > myFile.txt`	redirect output to a new file
`cat myFile.txt`	display contents of file
`cp myFile.txt yourFile.txt`	make a copy of the file
`mv yourFile.txt ourFile.txt`	rename the new file
`mkdir tmpDir`	make another new directory
`mv ourFile.txt tmpDir`	move the file copy to new dir
`ls`	list current directory contents
`cd ..`	change to parent directory

Need Help? Use "man" pages…

- man ls
- man cd

- Navigating a manual page:
  - <return> advances line at a time
  - <space> advances page at a time
  - b reverses page at a time
  - /*keywd* searches for *keywd*
  - q quits

Writing Your First C++ Program

```
#include <iostream>
using namespace std;

int main() {
    cout << “Hello!” << endl;
    return 0;
}
```

Writing Your First C++ Program

```
#include <iostream>
using namespace std;


int main() {
    std::cout << “Hello!” << std::endl;
    return 0;
}
```

More on namespaces later

Writing Your First C++ Program

```
#include <cstdio>

int main() {
    int count{5};
    printf("Count is %d\n.", count);
    return 0;
}
```

Why? Because some C++ code uses this "legacy" style.

C Format Specifiers

- %d Integer format specifier
- %f Float format specifier
- %c Character format specifier
- %s String format specifier
- %u Unsigned int format specifier
- %ld Long int format specifier
- %x or %X Hex format specifier (lower or UPPER)
- Augmenting options control field width, etc:
  - %8.5f would cause " 1.23400" (note leading space)

Writing Your First C++ Program

```
#include <cstdio>

void myfunc(char *mystring) {
    printf(mystring);
    return 0;
}
```

Yes, you can do this. **DON'T!** It's a security vulnerability: "format string bug"

Writing Your First C++ Program

```
#include <cstdio>

void myfunc(char *mystring) {
    printf("%s", mystring);
    return 0;
}
```

Do this instead. Or better yet, just use `cout`.

Compile & Execute Your Program

- `g++ hello.cpp –o hello`
- `ls -l *`
- `./hello`

indicates that the executable resides in the current directory

in this example indicates where the "executable" should be stored. In general, indicates what output of operation should be called

Compile & Execute Your Program

- `g++ hello.cpp –o hello`

Compilation in C++ directly creates platform-specific executable code (or object code – more later).

Unlike Java, there is no compilation first to platform-independent byte code.

Hence, a C++ executable created on a Mac will not run on Linux or Windows.

Compile & Execute Your Program

- `g++ -std=gnu++2a hello.cpp –o hello`
- `ls -l *`
- `./hello`

indicates where the “executable” should be stored as output

indicates that the executable resides in the current directory

Allows the Mac default clang compiler to compile ``modern'' C++

Compile & Execute Your Program

- `g++ -std=c++17 hello.cpp –o hello`
- `ls -l *`
- `./hello`

indicates where the "executable" should be stored as output

indicates that the executable resides in the current directory

No longer necessary on cluster!

Allows the Linux cluster machines to compile ``modern'' C++

Compile & Execute Your Program

- `g++ -o hello -std=gnu++2a hello.cpp`
- `./hello`

Note order of config parameters not usually important. I tend to put executable first.

“Modern C++”

- This is the third year of the full-unit CMSC 240
- In this course you will learn “Modern C++”
  - So if you consult notes from before 2020, code might look very different from what we do
- C++ “versions”: C++98, C++03, C++11, C++14, C++17, C++20
  - I learned C++ in 1995-97.
    - I learned **modern** C++ during Summer 2020. There is still much I have to think twice about(it’s a very powerful, very feature rich, very complex language)
  - The change is not “minor” – complete language revision.
    - We will learn modern C++ from the start

“Modern C++”

- C++ is a powerful language that provides low level access to memory
  - Which is great for system programming
  - But which also leaves a person open to pernicious bugs if they are not careful…

“Modern C++”

- C++ is a powerful language that provides low level access to memory (and won’t prevent bad judgement)

```
#include <cstdio>

int main(int argc, char** argv) {

  int myarray[10];

  printf("%x\t%x\n", myarray[-8], myarray[10]);

}
```

Command-Line Parameters (a.k.a. Arguments)

```
#include <iostream>

int main(int argc, char* argv[])
{
   if (argc != 2) // argc counts the num of CLPs
   {
      std::cerr << "Usage: " << argv[0]
                << " <first name>" << std::endl;
      exit(0);
   }

   std::cout << "Hello " << argv[1] << std::endl;
   return 0;
}
```

The *insertion* operator (when used with streams)

Command-Line Arguments

```
#include <iostream>

int main(int argc, char* argv[])
{
      if (argc != 2)
      {
            std::cerr << “Usage: “ << argv[0]
                  << “ <first name>” << std::endl;
            return 0;
      }

      std::cout << “Hello ” << argv[1] << std::endl;
      return 0;
}
```

So what is argv?

Converting Command-Line Args

```
...
int bar(int param)
{
  return(8675309 + param);
}

int main(int argc, char* argv[])
{
  // check for correct # of CLAs!!
  char* offsetAsString { argv[1] };
  int   offset         { std::stoi(offsetAsString) };
  std::cout << bar(offset) << std::endl;
  return 0;
}
```

char* is a type:
C-style string

stoi(): converts
string to integer

Compile & Execute Your Program

- `g++ -o hello hello.cpp`
- `ls -l *`
- `./hello`
- `./hello Lilly`

C-Style Arrays

```
...
int main()
{
        std::cout << "Enter 10 integers: " << std::endl;

        int array[10];
        for (int i = 0; i < 10; i++) {
                std::cin >> array[i];
        }

        std::cout << "In reverse, you entered: ";
        for (int i = 9; i >= 0; i--) {
                std::cout << array[i] << " ";
        }
        std::cout << std::endl;

        return 0;
}
```

The name of the array is "array"

The *extraction* operator (when used with streams)

C-style arrays
do not
have a length field

C-Style Arrays

```
...
int main()
{
```

Problems with C-style arrays:

1) doesn't know its own size
2) converts to a pointer to its first element
   (more on pointers later)

```
}
```

C++ Arrays: std::array for fixed size

```
...
int main()
{
    std::array<int, 10> array;

    std::cout << "Enter " << array.size() << " integers: " << std::endl;
    for (int i = 0; i < array.size(); i++) {
        std::cin >> array[i];
    }

    std::cout << "You entered: ";
    for (int val : array)
    {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    // omission of return 0; ==> implicitly returns 0
}
```

This is a *range-based* loop

C++ Arrays: std::vector for dynamic size

```
...
int main()
{
    std::vector<int> vector;

    std::cout << "Enter a non-negative integer, or -1 to quit: ";
    int num = 0;
    while (std::cin >> num) {
        if (num == -1) break;
        vector.push_back(num);
        std::cout << "Enter a non-negative integer, or -1 to quit: ";
    }

    std::cout << "You entered: ";
    for (auto val : vector)
    {
        std::cout << val << " ";
    }
    std::cout << std::endl;
}
```

C++ Arrays: std::vector for dynamic size

```
...
int main()
{
    std::vector<int> vector;

    std::cout << "Enter a non-negative integer, or -1 to quit: ";
    int num = 0;
    while (std::cin >> num) {
        if (num == -1) break;
        vector.push_back(num);
        std::cout << "Enter a non-negative integer, or -1 to quit: ";
    }

    std::cout << "You entered: ";
    for (auto val : vector)
    {
        std::cout << val << " ";
    }
    std::cout << std::endl;
}
```

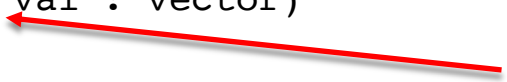
The overloaded operator>> function returns a reference to the stream itself, and the stream has an overloaded operator that allows it to be used in a boolean condition to see if the last operation went okay or not. Part of the "okay or not" includes end of file reached, or other errors.

C++ Arrays: std::vector for dynamic size

```
...
int main()
{
    std::vector<int> vector;

    std::cout << "Enter a non-negative integer, or -1 to quit: ";
    int num = 0;
    while (std::cin >> num) {
        if (num == -1) break;
        vector.push_back(num);
        std::cout << "Enter a non-negative integer, or -1 to quit: ";
    }

    std::cout << "You entered: ";
    for (auto val : vector)
    {
        std::cout << val << " ";
    }
    std::cout << std::endl;
}
```



In using "auto", C++ will automatically determine the type (within reason – be careful about this).

C++ Arrays: std::vector for dynamic size

```
...
int main()
{
    std::vector<int> vector;

    std::cout << "Enter a non-negative integer, or -1 to quit: ";
    int num = 0;
    while (std::cin >> num) {
        if (num == -1) break;
        vector.push_back(num);
        std::cout << "Enter a non-negative integer, or -1 to quit: ";
    }

    std::cout << "You entered: ";
    for (auto val : vector)
    {
        std::cout << val << " ";
    }
    std::cout << std::endl;
}
```

vector is almost always preferred over an array. If you know exactly how many elements you'll have, array has slightly less overhead. But vector is more flexible and has more operations. It should be your primary sequential container.

Functions in C++: With Prototype

```
#include <iostream>

int foo(); // declaration (AKA function prototype)

int main()
{
  std::cout << foo() << std::endl; // call foo()
  return 0;
}

int foo() // definition
{
  return(8675309);
}
```

It's like declaring variables before using them: either define functions before calling them , or you have to "declare" them.

Functions in C++: Without Prototype

```
#include <iostream>

int foo() // definition (occurs before call)
{
  return(8675309);
}


int main()
{
  std::cout << foo() << std::endl; // call foo()
  return 0;
}
```

It's like declaring variables before using them: either define functions before calling them , or you have to "declare" them.

Functions in C++: With Prototype

```
#include <iostream>

int foo(char); // declaration (AKA function prototype)

int main()
{
  std::cout << foo() << std::endl; // call foo()
  return 0;
}

int foo(char initial) // definition
{
  return(8675309);
}
```

Also, declaration provides interface: no need for formal parameters in declaration.

Functions in C++: With Prototype

```
#include <iostream>

int foo(char initial) // definition
{
  return(8675309);
}
```

Note: In most cases, no difference in return statements with or without parens. However,
one exception can be found here:
https://stackoverflow.com/questions/4762662/are-parentheses-around-the-result-significant-in-a-return-statement.
You'll never have to worry about that in this class. But if you decide to become a C++ master, then you'll likely want to go down this rabbit hole (and many others).

Functions in C++: Parameters

```
#include <iostream>

int bar(int param)
{
  return(8675309 + param);
}

int main()
{
  int offset {10};  // (almost) same as int offset = 10;
  std::cout << bar(offset) << std::endl; // call bar
  return 0;
}
```

= versus {}-list initialization

- From Stroustrup (*A Tour of C++*, 2nd Ed., Section 1.4.2):

The = form is traditional and dates back to C, but if in doubt, use the general {}-list form. If nothing else, it saves you from conversions that lose information:

```
int i1 = 7.8; // i1 becomes 7 (surprise?)
int i2 {7.8}; // error: floating-point to integer conv.
```

- This is a feature introduced in C++11, so you will need to indicate that in your compile command (by making sure your compiler "knows" it is compiling modern C++)

I highly recommend Stroustrup book. AFTER you have learned C++.

C++: Input and Output

- Standard input / standard output streams
  - `#include <iostream>`
  - `std::cin / std::cout`

- File input / file output streams
  - `#include <fstream>`
  - `std::ifstream / std::ofstream`

- Streams for console and files all work the same
  - >> reads *whitespace-delimited* words from input stream
  - << prints text/variables to output stream

  - to read entire line, use `getline(cin, stringvar)`

Example of Opening an Input File

- Remember to `#include <fstream>`
- To open an input file for reading:

```
string fname { “phone.txt"};

ifstream infile {fname};

if (!infile)
{
   cerr << "Could not open file: " << fname << endl;
  return 0;
}
```

- Then use `infile` where you would use `cin`, with >>
- Make sure to close when done…

```
infile.close();
```

File Reading Example

(a) enter this C++ program as `fileInput.cpp`
(b) compile the program:
`g++ fileInput.cpp -o fileInput`
(c) create a text file containing two numbers
(d) run, passing in the name of your text file:
`./fileInput numbers.txt`

```
#include <fstream>
#include <iostream>
#include <string>

void error(std::string msg, std::string arg)
{
    std::cerr << msg << arg << std::endl;
    exit(0);
}

int main(int argc, char* argv[])
{
    // appropriately check for # of CLAs!!
    std::ifstream infile {argv[1]};
    if (!infile) {
        error("Unable to open file: ", argv[1]);
    }

    std::string number1;
    std::string number2;
    infile >> number1 >> number2;
    std::cout << number1 << " : " << number2 << std::endl;
    infile.close();
}
```

Example of Opening an Output File

- Remember to `#include <fstream>`
- To open an output file for reading:

```
string fname {"myOutput.txt"};
ofstream outfile {fname};
if (!outfile)
{
   // cerr appropriate error message
   return 0;
}
```

- Then use `outfile` where you would use `cout`, with <<
- Make sure to close when done…

```
outfile.close();
```

File Writing Example

(a) enter this C++ program as `fileOutput.cpp`

(b) compile the program:

```
g++ fileOutput.cpp -o fileOutput
```

(c) run, passing in the name of a text file:

```
./fileOutput newNumbers.txt
```

(d) dislay the contents of the new output file:

```
less newNumbers.txt
```

```
#include <fstream>
#include <iostream>

void error(std::string msg, std::string arg)
{
    std::cerr << msg << arg << std::endl;
    exit(0);
}

int main(int argc, char* argv[])
{
    // appropriately check for # of CLAs!!
    std::ofstream outfile {argv[1]};
    if (!outfile)
    {
        error("Unable to open file: ", argv[1]);
    }

    for (int i = 0; i < 10; i++)
    {
        outfile << (866 + 1) << " - " << (530 * 10 + 9 + i);
        outfile << std::endl;
    }

    outfile.close();
}
```

Week 1 Assignment

- Write, compile, and test 7 short C++ programs from these slides
- Name as follows, submitting to `hw1` in shared Box folder

1.	`args.cpp`	Slide 24 on command-line arguments
2.	`array.cpp`	Slide 31 on std::array in C++
3.	`vector.cpp`	Slide 32 on std::vector in C++
4.	`prototype.cpp`	Slide 34 on functions w/ prototype
5.	`params.cpp`	Slide 38 on function parameters
6.	`fileInput.cpp`	Slide 42 on file input example
7.	`fileOutput.cpp`	Slide 44 on file input example

- For #7, #8, include checks for number of CLAs (see logic on slide 25)