CSCI 200: Foundational Programming Concepts & Design Lecture 03

Command Line Interface

Makefiles

Have VS Code Open

Previously in CSCI 200

- Variables comprised of
  - Data Type
    - Specifies amount of memory used to store value
    - How to store/retrieve value
  - Identifier
    - Name to refer to location in memory
- Both need to correspond to how real data appears in the world

CSCI 200 Style Guidelines

- Variables follow **`lowerCamelCase`**
- Constants follow **`UPPER_SNAKE_CASE`**

Static Declarations

- Need to declare data type up front so computer can allocate enough memory
- Data types take different amount of memory

Data Type	Size	Range	
`bool`	8 bits / 1 byte*	0 to 1	0 to 1
`char`	8 bits / 1 byte	-2^7 to $+2^7-1$	-128 to +127
`int`	32 bits / 4 bytes	-2^{31} to $+2^{31}-1$	-2,147,483,648 to +2,147,483,647
`float`	32 bits / 4 bytes	±1.18e-38 to ±3.4e38	~7 digits precision
`double`	64 bits / 8 bytes	±2.23e-308 to ±1.80e308	~16 digits precision

- *theoretically 1 bit, but in practice memory access is done by byte

Additional Modifiers

- `short int`
- `long int`
- `long long int`
  - Uses less or more memory

Data Type	Size	Range	
`short int`	16 bits / 2 bytes	-2^{15} to $+2^{15}-1$	-32,678 to +32,677
`int`	32 bits / 4 bytes	-2^{31} to $+2^{31}-1$	-2,147,483,648 to +2,147,483,647
`long int`	32 bits / 4 bytes	-2^{31} to $+2^{31}-1$	-2,147,483,648 to +2,147,483,647
`long long int`	64 bits / 8 bytes	-2^{63} to $+2^{63}-1$	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807

Additional Modifiers

- **unsigned**
  - Most significant bit part of value, not the sign

Data Type	Size	Range	
`signed short`	16 bits / 2 bytes	-2^{15} to $+2^{15}-1$	-32,678 to +32,677
`unsigned short`	16 bits / 2 bytes	0 to $2^{16}-1$	0 to 65,535
`signed int`	32 bits / 4 bytes	-2^{31} to $+2^{31}-1$	-2,147,483,648 to +2,147,483,647
`unsigned int`	32 bits / 4 bytes	0 to $+2^{32}-1$	0 to +4,294,967,295
`signed long long`	64 bits / 8 bytes	-2^{63} to $+2^{63}-1$	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
`unsigned long long`	64 bits / 8 bytes	0 to $2^{64}-1$	0 to +18,446,744,073,709,551,615

Note on Precision

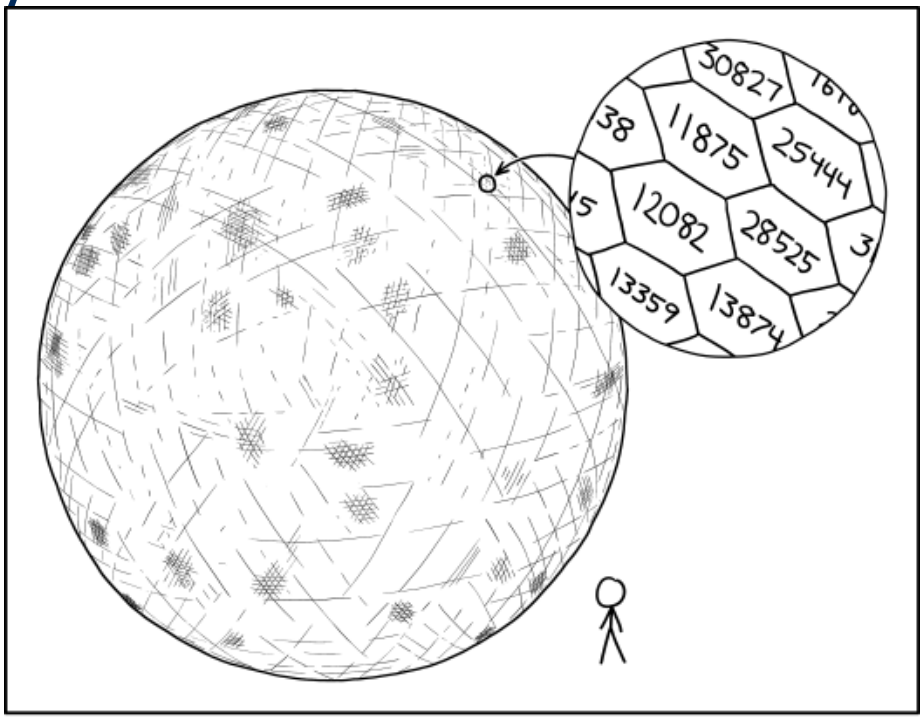
- For floating point values, the magnitude affects significance
- If 7 digits of precision:
  - 0.000abcdefg??
  - 0.abcdefg??
  - abcd.efg???
  - abcdefg????.???
- Trailing values may not be accurate

Precision Fun Fact

Precision	Bits	Bit Breakdown	# Digits Precision	Range
Half	16	1 sign 5 exponent 10 mantissa	~3	$\pm 10^{-5}$ to ± 65504
Single	32	1 sign 8 exponent 23 mantissa	~7	$\pm 10^{-38}$ to $\pm 10^{38}$
Double	64	1 sign 10 exponent 53 mantissa	~16	$\pm 10^{-308}$ to $\pm 10^{308}$
Quad	128	1 sign 14 exponent 116 mantissa	~33	$\pm 10^{-4932}$ to $\pm 10^{4932}$
Oct	256	1 sign 18 exponent 237 mantissa	~71	$\pm 10^{-78913}$ to $\pm 10^{78913}$

Random Numbers

- For the computer
  - Cannot generate purely random numbers
- Pseudo-Random
  - Kinda random
  - Good enough for us!



THE HARDEST PART OF SECURELY GENERATING RANDOM 16-BIT NUMBERS IS ROLLING THE D65536.

https://xkcd.com/2626/

Pseuo-Random Numbers

- Need to include the standard C library

```
#include <cstdlib>
```

- Simply call rand() and get a random number!
  - But…

Seed the RNG

- Random Number Generator (RNG)

- Initialize the sequence of generated random numbers using srand()
  - Use the same seed?
  - Get the same random sequence
  - Default seed: 1

Pseuo-Random Process

1. Include **`cstdlib`**
2. Set the random seed with **`srand()`**
3. Call **`rand()`** as needed

Practice

- What is min value rand() will generate?

  0

- What is max value rand() will generate?

  `RAND_MAX`

Practice

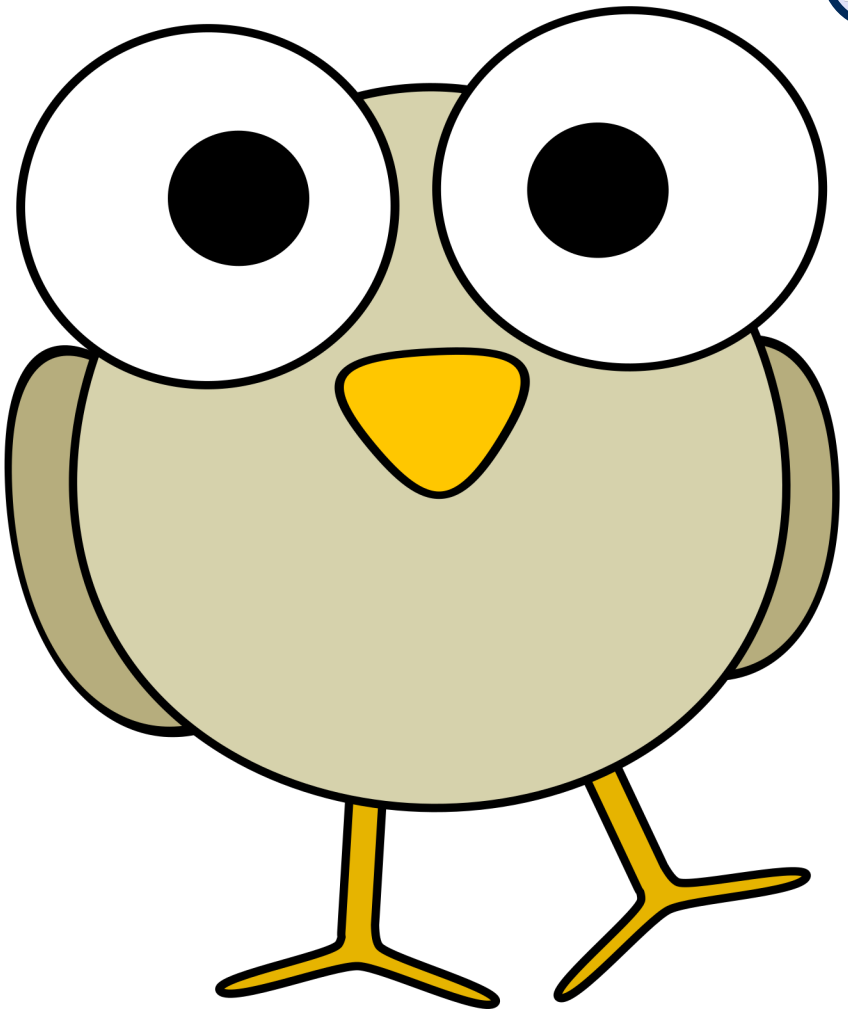
- How to generate a random integer between 2 & 10 inclusive?

```
rand() % (10 - 2 + 1) + 2
rand() % (max - min + 1) + min
```

- How to generate a random float between 2 & 10 inclusive?

```
rand() / (double)RAND_MAX * (10.0 - 2.0) + 2.0
rand() / (double)RAND_MAX * (max - min) + min
```

Questions?



Learning Outcomes For Today

- List common Linux terminal commands and choose the correct commands to work with a file system via the command line.
- Describe how a computer generates a program from code.
- Write and use a Makefile.
- Discuss the advantages of using Makefiles.

On Tap For Today

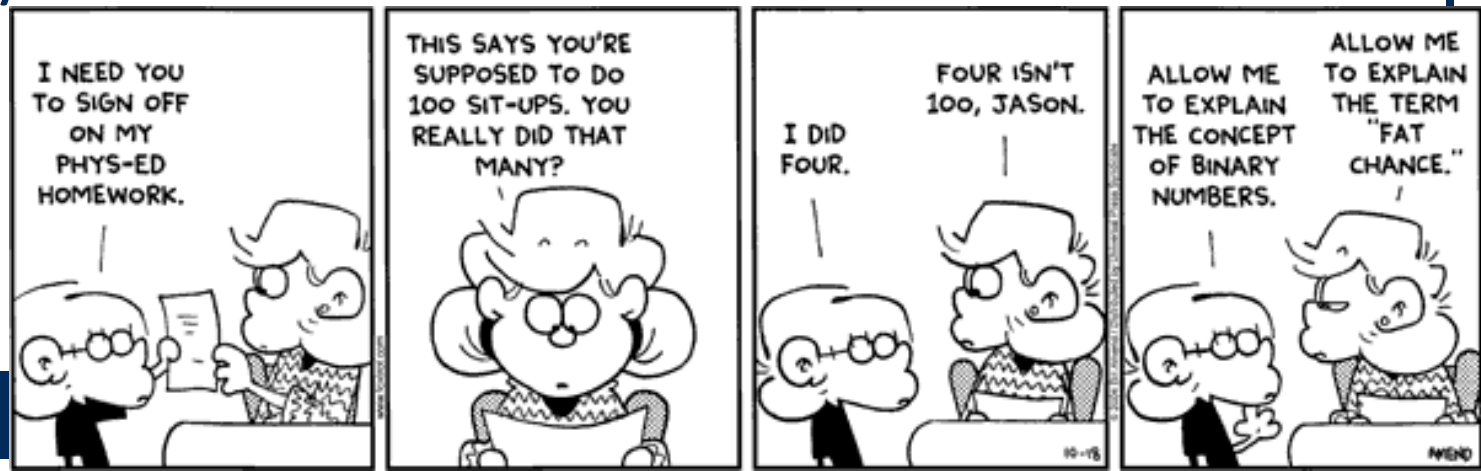
- Building a C++ Program
- Compiler Flags & Directives
- Makefiles
- Practice

On Tap For Today

- Building a C++ Program
- Compiler Flags & Directives
- Makefiles
- Practice

Programming Languages Review

- High Level Languages
  - C++
- Assembly Languages
  - x86
- Machine Language
  - Binary



Making a Program

- Three steps (in order)
  1. Compile
  2. Link
  3. Execute (run)

Step 1: Compiling

- The process of converting your code from C++ (a high level language) to binary (machine language)
  - Produces an “object file”
    - main.cpp → main.o
- g++ uses a compiler to translate your code

Object File

- Contains machine instructions (binary)
- Not executed
- Combined with other object files to make an executable or program

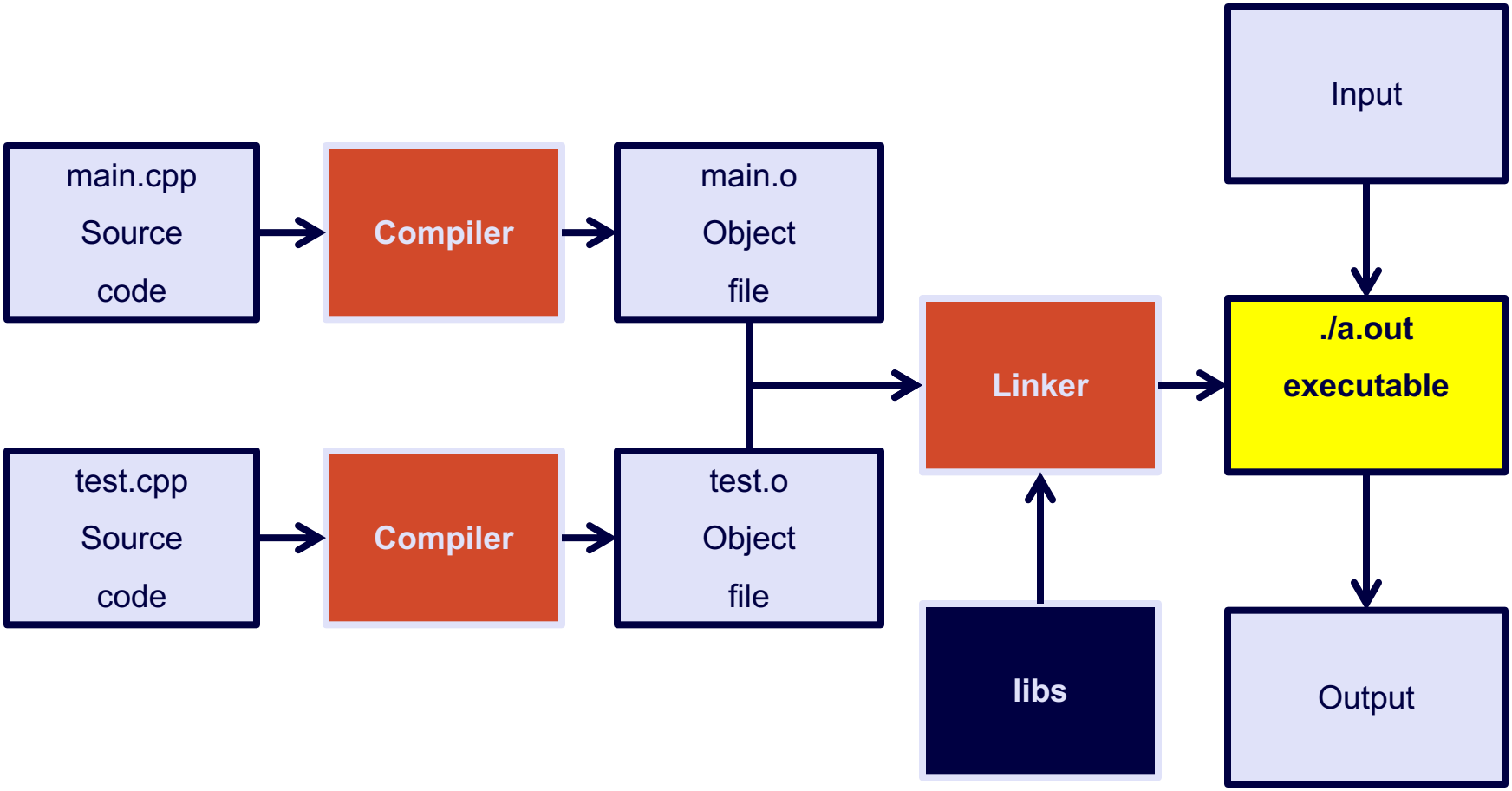
Step 2: Linking

- Your program relies on other libraries
  - iostream
  - cmath
- The linker combines all the necessary object files
  - g++ also uses a linker to link our program
- Produces an executable program
  - main.o + libiostream.a → a.out (a.exe on Win)

Step 3: Execution

- Executable programs are a file on disk
- To “execute” a program means to run it
  - Load the executable file into memory
  - Tell the computer where the first instruction is
  - Run the program!

Compile & Link Process



Command Line Interface (CLI)

- Textual representation to move through file system and directory structure

CLI Cheat Sheet

- Directory Operations
  - Show current directory
  - List files
  - Make directory
  - Change directory
    - Go up a directory
  - Copy a directory (**NO UNDO**)
  - Move a directory^ (**NO UNDO**)
  - Remove a directory^ (**NO UNDO**)
- *^Slight differences between Windows / OS X when copy/move/remove directory*

CLI Cheat Sheet

- File Operations
  - Create file^
  - Copy file (**NO UNDO**)
  - Move file (**NO UNDO**)
  - Remove file (**NO UNDO**)

- *^Slight differences between Windows / OS X when creating a file*

CLI Cheat Sheet

- Program Operations
  - Build program
  - Run program^

- *^Slight differences between Windows / OS X when running program*

On Tap For Today

- Building a C++ Program
- Compiler Flags & Directives
- Makefiles
- Practice

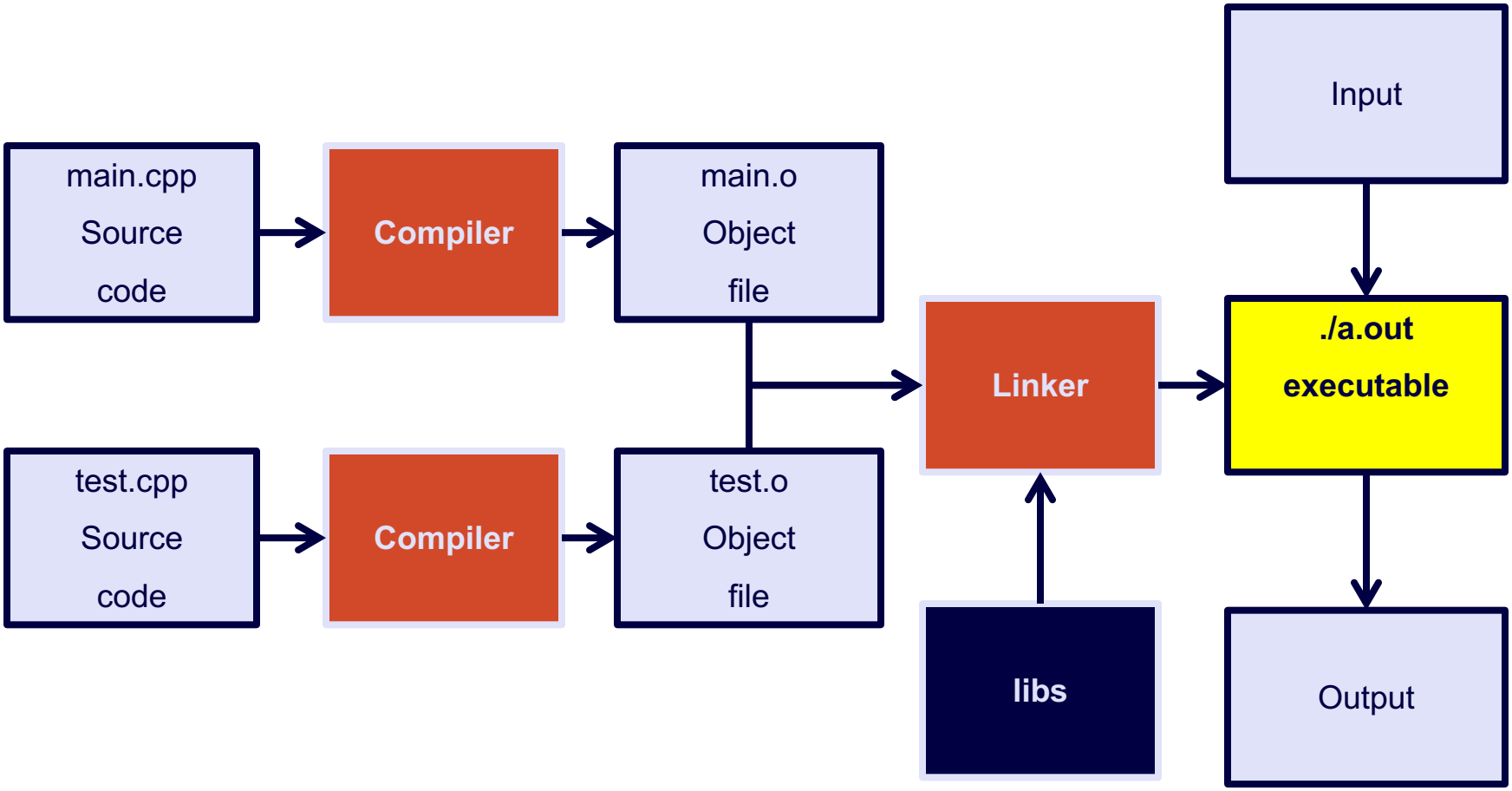
Build Command

- Currently

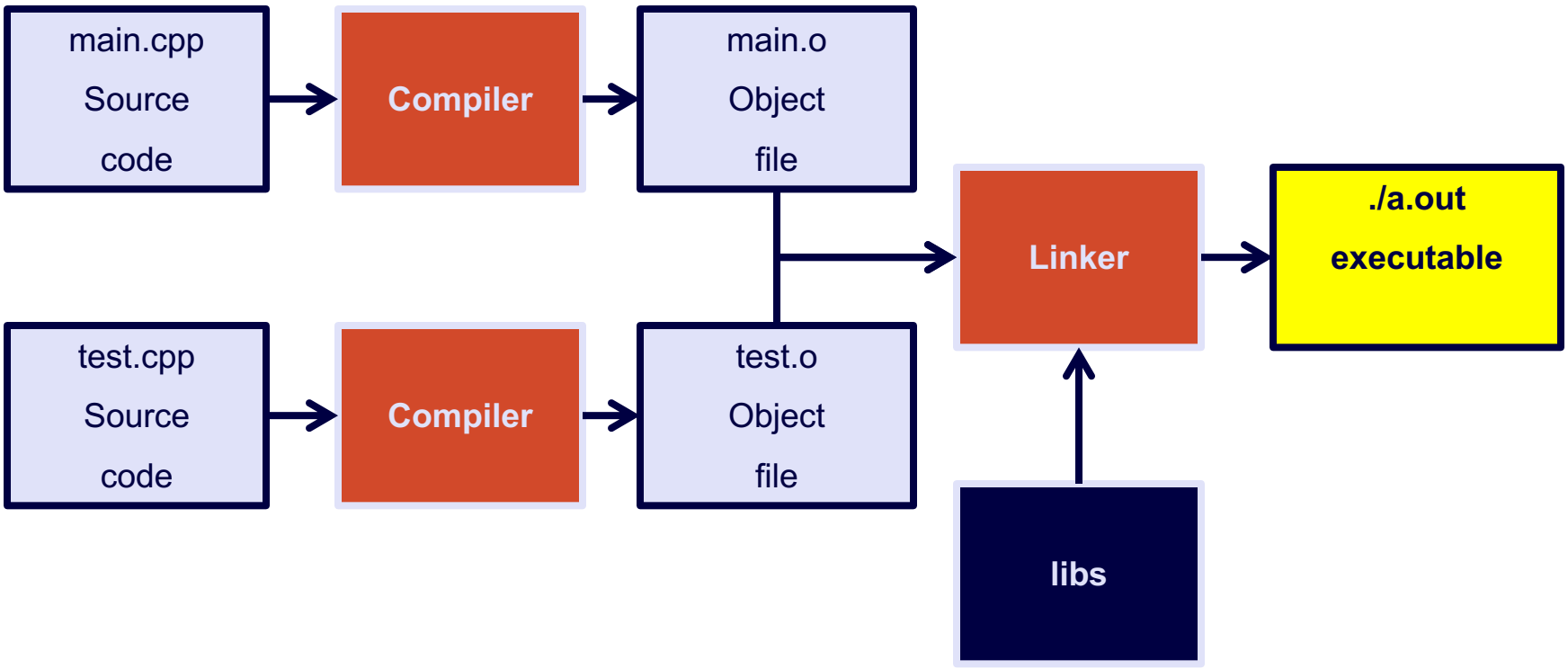
```
g++ main.cpp
```

- But doing several things behind the scenes

Compile & Link Process



g++ Compile & Link Process



Individual Steps

- First compile

```
g++ -o main.o -c main.cpp
```

- Then link

```
g++ -o a.exe main.o
```

- Then run

```
.\a.exe
```

Compiler Flags

- Options to the compiler to set

- Seen so far
  - `-o` what to name the output
  - `-c` which file to compile

More Complex Programs

- First compile all source code

```
g++ -o main.o -c main.cpp
g++ -o Square.o -c Square.cpp
```

- Then link all object files

```
g++ -o a.exe main.o Square.o
```

- Then run

```
.\a.exe
```

More Friendly Program Names

- First compile all source code

```
g++ -o main.o -c main.cpp
g++ -o Square.o -c Square.cpp
```

- Then link all object files

```
g++ -o SquareArea.exe main.o Square.o
```

- Then run

```
.\SquareArea.exe
```

Compiler Directives

- The C++ code also tells the compiler information

```
#include <iostream>
```

- C++ statements beginning with # are compiler directives

#include

```cpp
// main.cpp
#include <iostream>
using namespace std;

int main() {
  cout << "Hello World!" << endl;
  return 0;
}
```

```cpp
// iostream
// ...
cout = ...
endl = ...
```

#include

```
// main.cpp
#include <iostream>
using namespace std;

int main() {
  cout << "Hello World!" << endl;
  return 0;
}
```

```
// iostream
// ...
cout = ...
endl = ...
```

#include

```
// main.cpp
// iostream
// ...
cout = ...
endl = ...
using namespace std;


int main() {
  cout << "Hello World!" << endl;
  return 0;
}
```

Build Process Now

- First compile

```
g++ -o main.o -c main.cpp
```

```
// main.cpp
#include <iostream>
using namespace std;

int main() {
  cout << "Hello World!" << endl;
  return 0;
}
```

Build Process Now

- First compile

```
g++ -o main.o -c main.cpp
```

```
// main.cpp
// iostream
// ...
cout = ...
endl = ...
using namespace std;


int main() {
  cout << "Hello World!" << endl;
  return 0;
}
```

Build Process Now

- First compile

```
g++ -o main.o -c main.cpp
```

```
01010111001010101001010101001
10101010100100100101010101001
01010101010000111110010101001
10100100101010101010010101000
```

main.o

Build Process Now

- Then link

```
g++ -o HelloWorld.exe main.o
```

```
01010111001010101001010101001
10101010100100100101010101001
01010101010000111110010101001
10100100101010101010010101000
```

main.o

Build Process Now

- Then link

```
g++ -o HelloWorld.exe main.o
```

```
11000101010101000000101010100
01010101010101010101010101010
10100100101010101010010101000
11000001111111001010010010011
```

HelloWorld.exe

Build Process Growing

```
g++ -o main.o -c main.cpp
g++ -o Square.o -c Square.cpp
g++ -o Tri.o -c Tri.cpp
g++ -o Rect.o -c Rect.cpp
g++ -o Circle.o -c Circle.cpp

g++ -o Geometry.exe main.o
Square.o Tri.o Rect.o Circle.o
```

In Practice

- Need to remember which file(s) changed
  - Recompile them (or do all to be safe)
- Relink program
- Manually
- Every time

On Tap For Today

- Building a C++ Program
- Compiler Flags & Directives
- Makefiles
- Practice

Makefiles

- Tool to help build a program
  - Uses **`make`*** to automate commands via the terminal
- State what files make up the project
- Will be required to submit with all labs/assignments

- *Note:
  - On Windows, this program is called **`mingw32-make`**.
  - On OS X / Linux, this program is called **`make`**.

make

- Looks in the current directory for a file named **`makefile`** or **`Makefile`**

- Run in terminal via

  `make`

  `mingw32-make`

- Executes the corresponding **`Makefile`**

Makefile structure

- Generic format

```
variables = values

target: dependency1
        command1
        command2

dependency1:
        command3
```

- Important note! *command* lines are indented with a tab. Must be a tab.

Simplest Makefile

```
all:
	g++ main.cpp
```

Makefile

```
HelloWorld.exe:
        g++ -o main.o -c main.cpp
        g++ -o HelloWorld.exe main.o
```

Makefile

```
HelloWorld.exe: main.o
        g++ -o HelloWorld.exe main.o


main.o: main.cpp
        g++ -o main.o -c main.cpp
```

Dependencies

- **`make`** only executes dependency if timestamps have changed

```
HelloWorld.exe: main.o
        g++ -o HelloWorld.exe main.o


main.o: main.cpp
        g++ -o main.o -c main.cpp
```

- If **`main.cpp`** timestamp is newer than **`main.o`**, then **`main.o`** target commands run
- If **`main.o`** timestamp is newer than **`HelloWord.exe`**, then **`HelloWorld.exe`** target commands run

Makefile Variables

- Declaration and assignment

```
VAR_NAME = value
```

- Usage (Expansion)

```
$(VAR_NAME)
```

Makefile Special Symbols

- Given template

```
target: dependency1 dependency2
        command1
        command2
```

- `$@` expands to target name

  target

- `$<` expands to first dependency

  dependency1

- `$^` expands to all dependencies

  dependency1 dependency2

Makefile

```
TARGET = HelloWorld.exe


$(TARGET): main.o
	g++ -o $(TARGET) main.o


main.o: main.cpp
	g++ -o main.o -c main.cpp
```

Makefile

```
TARGET = HelloWorld.exe


CXX = g++


$(TARGET): main.o
	$(CXX) -o $(TARGET) main.o


main.o: main.cpp
	$(CXX) -o main.o -c main.cpp
```

Makefile

```
TARGET = HelloWorld.exe


CXX = g++


$(TARGET): main.o
        $(CXX) -o $@ $^


main.o: main.cpp
        $(CXX) -o $@ -c $<
```

Generalized Makefile

```
TARGET = HelloWorld.exe

SRC_FILES = main.cpp


CXX = g++

OBJECTS = $(SRC_FILES:.cpp=.o)


$(TARGET): $(OBJECTS)
        $(CXX) -o $@ $^


%.o: %.cpp
        $(CXX) -o $@ -c $<
```

Bigger Projects

```
TARGET = Geometry.exe

SRC_FILES = main.cpp Square.cpp Tri.cpp Rect.cpp Circle.cpp


CXX = g++

OBJECTS = $(SRC_FILES:.cpp=.o)


$(TARGET): $(OBJECTS)
        $(CXX) -o $@ $^


%.o: %.cpp
        $(CXX) -o $@ -c $<
```

Additional Tools

```
TARGET = HelloWorld.exe

SRC_FILES = main.cpp


CXX = g++

OBJECTS = $(SRC_FILES:.cpp=.o)


$(TARGET): $(OBJECTS)
        $(CXX) -o $@ $^


%.o: %.cpp
        $(CXX) -o $@ -c $<


clean:
        rm $(TARGET) $(OBJECTS)
```

Cross-Platform Makefile

```
TARGET = HelloWorld
SRC_FILES = main.cpp
# NO EDITS BELOW THIS LINE
CXX = g++
OBJECTS = $(SRC_FILES:.cpp=.o)
ifeq ($(shell echo "Windows"), "Windows")
            TARGET := $(TARGET).exe
            DEL = del
else
            DEL = rm -f
endif
all: $(TARGET)

$(TARGET): $(OBJECTS)
            $(CXX) -o $@ $^

%.o: %.cpp
            $(CXX) -o $@ -c $<

clean:
            $(DEL) $(TARGET) $(OBJECTS)
```

On Tap For Today

- Building a C++ Program
- Compiler Flags & Directives
- Makefiles
- Practice

To Do for Next Time

- zyBooks Ch. 3
  - Structured Programming: Conditionals