

Introduction to Software Engineering

in C++

INTRODUCTION

DARRYL HILL

About the Course

Two Main Aspects to the course:

- Software engineering – an introduction to bigger and more complex applications and the patterns and solutions we use to make them tenable
 - Program design
 - Patterns that are often seen or often used in bigger programs
 - Problems that have been solved and how to solve them
- Also a C++ course

Software Engineering

- Working code is the **most** important thing.
 - You can ship poorly written code that works and update it later.
 - You cannot ship code that doesn't work.
- You will continually update, fix bugs, etc, code.
 - All software ***will be modified***
 - Thus code must be **readable, extensible, maintainable**
 - Needs are continually changing and adapting
 - Once you stop updating, it is only a matter of time until it is obsolete
- Code also has to work with other code.
 - Certain designs allow for this better than others.
- In addition to having **working code**, these other needs must be addressed

Software Engineering

As programs get bigger, readability, extensibility, maintainability get more difficult.

Certain patterns and best practices help keep things manageable

- Testing – unit, systems, integration
- Documentation
- Modularity
 - Small pieces of code with clear purpose and well-defined interfaces
- Design patterns
 - The right design pattern in the right place allows for easier changes.
 - Design patterns and language feature are not solutions.
 - And often they are not a perfect fit – still useful.

About the Course

We also learn about C++.

Why C++?

- C++ is a systems level language with high-level constructs.
- Very fast, very powerful, very complex.
 - All the power and complexity make it easy to make mistakes in.

Python / Java



C



C++



High-level Languages

Python / Java / etc

- Expressive – say *what* you want done, the language handles the *how*.
 - High-level features – i.e., objects, generics.
 - Language enforced safety features.
 - Automatic memory management.

Less efficient

- less control over *how* things are done at a low level
- Do not know when memory is released
 - Releasing memory impacts performance

Systems Level Languages – C

A step above assembly – one very small layer of abstraction

Freedom to do anything

- Allows for clever tricks, hacks
- Can make code as fast as possible
- Low level management is good for OS's, devices

Very few high-level constructs

- not expressive
- need to “micromanage” your code

C++

Has the best of C:

- Low level control
- Can micromanage *how* your program does something

Also has High-level constructs

- Objects
- Templates

Best of both worlds

The price is complexity

- New Java specification – 844 pages
- New C++ specification – 2000+ pages

C++

The philosophy of C++ is you should have as much control as you want.

- The language has suggested best practices, but very few restrictions.
- Every rule has a way to break that rule.

The price is high complexity.

- Leads to bugs, memory leaks, segfaults, etc, when written poorly.

C++

- Why do we use it?
 - Things like triple A games push your machine to the limit –
 - Need both high-level abstractions and low level efficiency.
- Real time systems need fine control.
- If you cannot sacrifice efficiency for safety or expressiveness, then C++ is a good choice.

C++ and Software Engineering

The complexity and dangers of C++ can be managed by:

- Good understanding of the language.
 - Particularly handling memory.
- Good software engineering practices.
 - They help manage the complexity.
- This makes software engineering and C++ a good fit
- You can mitigate risks of dangerous tools by knowing how to use them well

Software Engineering

Software engineering came about because of pressures

- Need for software to be mutable

One goal of this course is to write bigger, more complex programs so that you feel these pressures.

Software engineering is a set of techniques that are a response to those pressures

- Kind of like evolution

Introduction to Software Engineering

This course is also about setting the foundation for COMP3004.

Medium sized programs of increasing complexity.

We will walk you through many techniques.

- You are not quite “free-range programmers”.
- That comes next in 3004.

Learning Outcomes

C++

- A foundational level of knowledge
 - Low-level C++
 - Compile it ourselves using “Make” files
 - You will begin to understand the compiler and how it works
 - C++ compilers are primitive (on purpose)
 - Learn to “think” like a compiler
 - No smart pointers – we will manage memory by having well-defined responsibilities for dynamic memory.
 - Whoever “owns” the memory should delete it when the time comes.
 - Every time you *allocate* a piece dynamic memory you should know where it is *deleted*.
- If you learn more C++ in the future, you will better understand how the new knowledge fits

Learning Outcomes Software Engineering

Object Oriented programming techniques:

- Testing.
- Code organization.
- Memory management.
- Inheritance, polymorphism.
- Refactoring

Visualizing (and designing) programs using UML

Some design patterns and architecture styles

- Design pattern or architecture style is a function of scale
 - Big – architecture style
 - Small(ish) – design pattern

Learning Outcomes Software Engineering

Don't fall into the “over-designing” trap

- Design patterns are very useful when used correctly, i.e. at scale
- Small programs typically need few design patterns
 - Object Oriented Programming is sufficient
 - The need for design patterns arises as programs grow and require modification.
 - You **refactor** code as it grows to introduce necessary design patterns.

Learning Outcomes Software Engineering

Good software engineering will:

- Protect you from making mistakes.
- Help you understand your own code better.
- Help you change your own code more easily.
- Find mistakes more easily.
- Etc.

Our Tools - Linux Operating System

Based on the Unix operating system

[Developed in Bell Labs by Ken Thompson and Dennis Ritchie](#)

MacOS is Unix-compliant

Linus Torvalds developed Linux as a hobby

- Roughly Unix compliant

Linux Operating System

Modular

- Can be easily extended

Open source – Free!

- Can modify any part of it
- This also has dangers

Designed for simplicity rather than user-friendly

- Though many user friendly iterations exist

Linux Operating System

Very developer / science friendly

- Initially written in assembly
- C was developed on and for Unix
- Unix was rewritten completely in C
- Linux is also written in C

“Unix is simple. It just takes a genius to understand its simplicity” --Dennis Ritchie

“Unix was not designed to stop its users from doing stupid things, as that would also stop them from doing clever things” --Doug Gwyn

(This quote also applies to C and C++)

Linux Operating System

Unix family of OS's:

- Unix
- Linux
 - Ubuntu, Arch, etc
- Mac OS
- BSD
- Solaris
 - Used by Oracle who maintain Java
- Android
 - Uses Linux kernel

Linux On the Web:

How many Linux servers?

- Depends on how the statistics are gathered
- since February 2010, Linux distributions represented six of the top ten, FreeBSD three of ten, and Microsoft one of ten, [\[118\]](#) with Linux in the top position.

Because Linux is often hacked by hobbyists, you could say it “evolved” rather than was traditionally “built”

Our Tools - Shell

A Shell is an interface to the OS

- Command line
- Similar to Windows command line

Allows you to run commands on the OS

- Look through file systems
- Execute programs

Shell

Common shells:

- Bourne shell (sh)
- Bourne-again shell (bash) - this is the default shell for Linux
- C shell (csh)

Differences between shells

- Command line shortcuts
- Environment variables
- Very little

Some Shell Commands

- `ls` list the files in the current directory
- `cd` change directory
- `chmod +x <file>` make <file> executable
- `./<file>` (from same directory as <file>) execute <file>
 - If I use <file> the shell will not search the current directory, but rather the directories in the PATH environment variable
- `cat` write the contents of a file to the console
- `grep` search a file for a string
- `etc`

Our Tools - Programming Tools

Text Editor

- Atom, vim, emacs, gedit
- Visual Studio Code, Sublime

Required to write

- Programs
- Makefiles

Programming Tools

Compiler

- GNU C++ compiler: g++
- We will run it from the shell / command line
- We will use the current Ubuntu version of the compiler.
 - g++ (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0 currently
 - Most Ubuntu versions will work, however
 - It **must** run in the course VM – please test it there before submitting.
- Options:
 - -o is specifies an output file
 - -c compile an object file (more on this later)

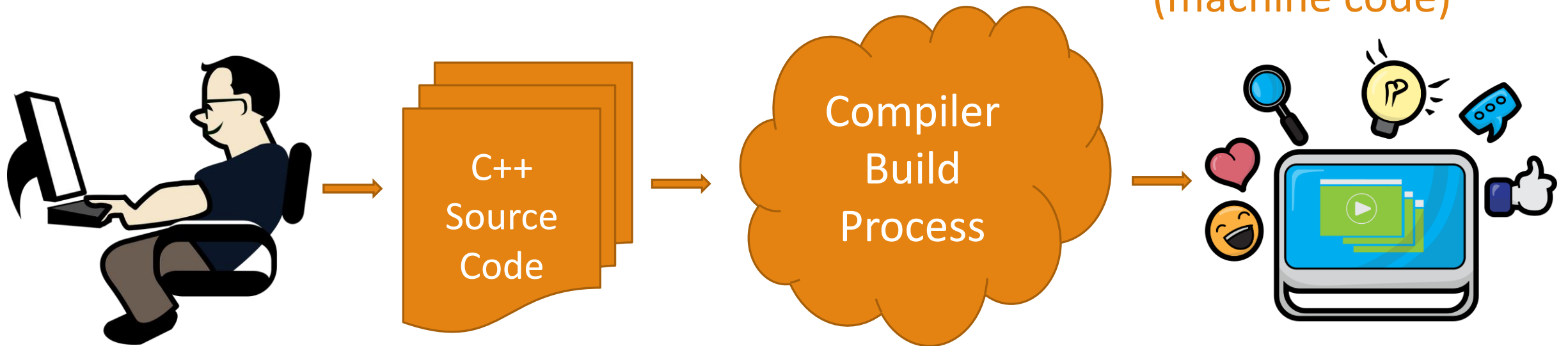
Programming Tools

Code must be compiled in the same environment it executes in

- Windows has multiple C++ compilers BUT
- Compiled on Windows will NOT run on Linux.
- In addition, Windows file systems are NOT case sensitive, but Linux is.
 - Room.cc and room.cc are the same file on Windows, but different on Linux.
 - Make sure all work you submit is **case-sensitive**.
- Programming example <p1>

Programming Tools

In the given program we compiled straight from source code to machine code



[This Photo](#) by Unknown Author is licensed under [CC BY-SA](#)

Programming Tools

As programs get bigger, we compile in pieces and then “link” the final product together

The pieces are called “object code”

Object code is machine code but perhaps with the addresses of certain function calls missing

Programming Example <p2>

Program Building

Program building is translating source code into machine code

- Source code (C++) – Instructions written in a high-level language resembling English
 - Cannot be understood by the CPU
- Machine code – instructions generated from the source code
 - Resembles a bunch of numbers
 - Can be understood by a CPU

Program building is the making of an executable from one or more source files.

Program Building

Program executable:

- The end result of program building
- A binary file that can be understood by your computer
- In general executables are NOT PORTABLE (unless you are running on the same OS and similar architecture)
 - A program compiled on one platform will not work on another platform
- A typical executable contains code from many sources
 - Your code, other people's code, libraries, etc
 - There is **one** jumping off place where it starts – corresponds to the **main** function

Program Building

Transforming C++ source into an executable:

Compilation

- turns C++ code into object code
- 1-to-1 correspondence between C++ source files and object files

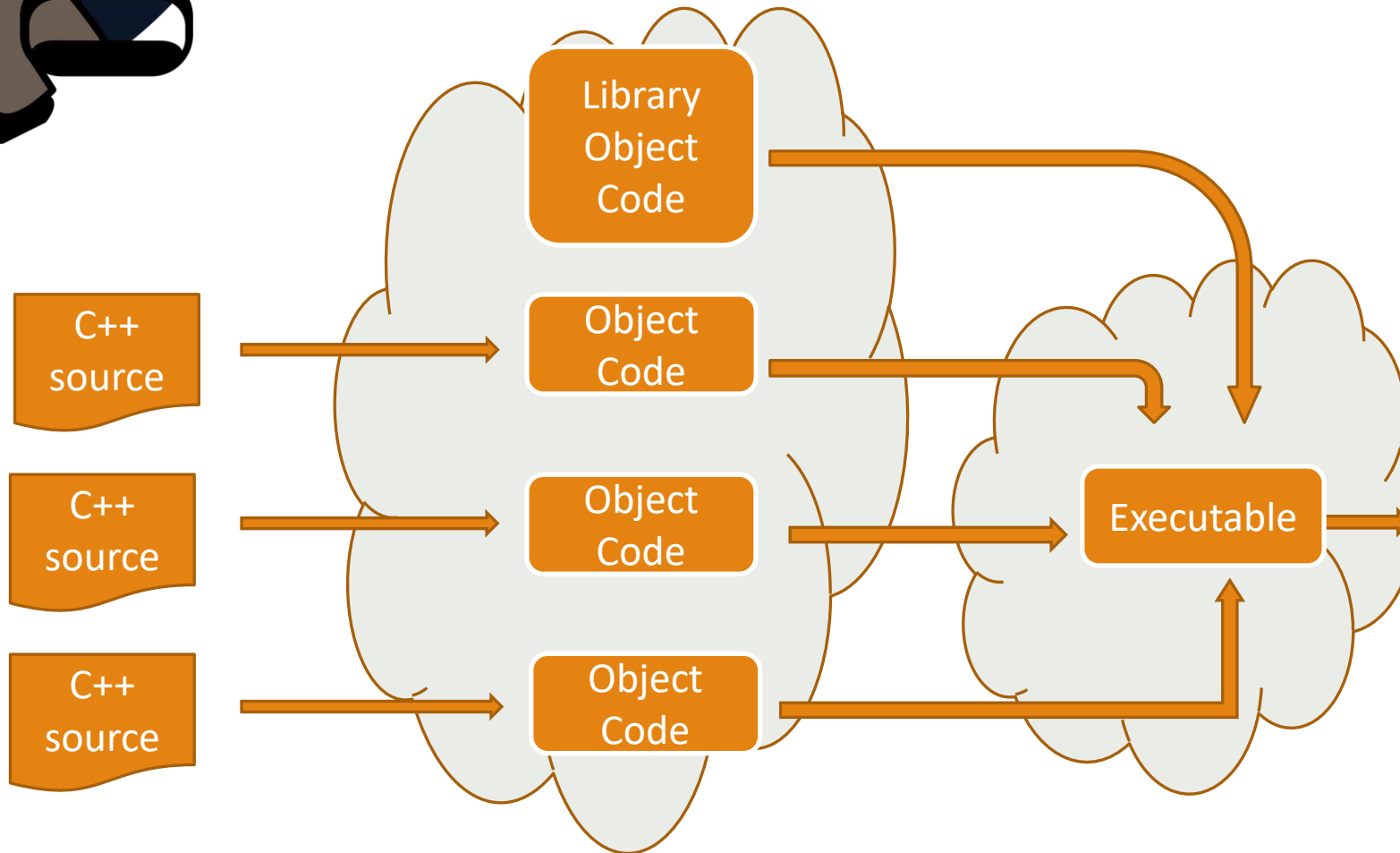
Linking

- Connects object code together
- Ex. Your source code may call a library function
- When compiling object code that calls other libraries, the compiler only needs the function signature – not the executable code
- Executable code for the library function is in a different object file
- “Linking” supplies the called function code to your compiled code



Compiler

Linker



Program



[This Photo](#) by Unknown Author is licensed under [CC BY-SA](#)

Compiling

Compiling a single file:

```
g++ -o p1 p1.cc
```

Compiling a single file using object files:

```
g++ -c p1.cc
```

```
g++ -o p1 p1.o
```

Compiling multiple files using object files:

```
g++ -c foo.cc
```

```
g++ -c foo2.cc
```

```
g++ -o foo foo.o foo2.o
```

Makefiles

Compiling and linking can get complicated quickly

Makefiles keep the process organized

A Makefile is a text file of compile commands and dependencies

- Files are only recompiled if they or a file they depend on has changed
- Sometimes dependencies can get difficult – sometimes “cleaning” and recompiling will fix a broken dependency
- If you get bugs you cannot find, try a **clean** step and recompiling from scratch.

Anatomy of a Makefile


```
foo: foo.o foo2.o  
    g++ -o foo foo.o foo2.o
```

```
foo.o: foo.cc  
    g++ -c foo.cc
```

```
foo2.o: foo2.cc  
    g++ -c foo2.cc
```

Anatomy of a Makefile

Make
command:
"make foo"



```
foo: foo.o foo2.o  
    g++ -o foo foo.o foo2.o
```


```
foo.o: foo.cc  
    g++ -c foo.cc
```

```
foo2.o: foo2.cc  
    g++ -c foo2.cc
```

Anatomy of a Makefile

```
foo: foo.o foo2.o  
    g++ -o foo foo.o foo2.o
```

Dependencies: the compile command is only given if these files have changed



```
foo.o: foo.cc  
    g++ -c foo.cc
```

```
foo2.o: foo2.cc  
    g++ -c foo2.cc
```


Anatomy of a Makefile

foo: foo.o foo2.o

g++ -o foo foo.o foo2.o



Command: this is the
command you would give
from the command line

foo.o: foo.cc

g++ -c foo.cc

foo2.o: foo2.cc


g++ -c foo2.cc

Anatomy of a Makefile

all: foo

foo: foo.o foo2.o
g++ -o foo foo.o foo2.o

May include an “all”
command, which makes
everything



foo.o: foo.cc
g++ -c foo.cc

foo2.o: foo2.cc
g++ -c foo2.cc

clean:
rm -f foo *.o

Anatomy of a Makefile

```
all: foo
```

```
foo: foo.o foo2.o
```

```
    g++ -o foo foo.o foo2.o
```

```
foo.o: foo.cc
```

```
    g++ -c foo.cc
```

```
foo2.o: foo2.cc
```

```
    g++ -c foo2.cc
```

```
clean:
```

```
    rm -f foo *.o
```

Should include a “clean”
command which removes
executables and object files

If you mess up the dependencies
(which can be really easy), doing a
“clean” first can sometimes fix your
issue

Example: <p3>