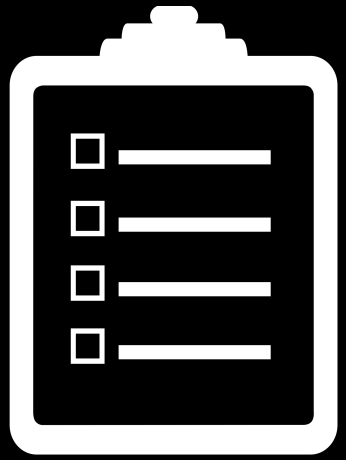
Final Topics

Ali Malik
malikali@stanford.ed

Game Plan



Command Line Compilation

Makefiles

What is a thread?

Race Conditions

Locks

Command Line Compilation

CL Compilation

So far, we have let QtCreator deal with compiling our code.

Today we will briefly cover how to do this manually in the terminal.

First we should understand how C++ compilation works.

C++ Compilation model

1. Preprocessor - Deals with `#include`, `#define`, etc directives
2. Compiler - Converts C++ source code into assembly
3. Assembler - Turns assembled code into object code (.o files)
4. Linker - Object files are linked together to make an executable program

Preprocessor

Responsible for everything starting with a **#**

- `#include`
- `#define`
- `#ifndef`
- `#pragma`

Compilers

Converts each .cpp source file into assembly.

This process is localised to each file.

Outputs .s files

Assembler

Turns previously generated assembly code into object code.

Outputs .o files.

Still no intercommunication between separate cpp files.

Linker

Combines all the separate object files into one executable file.

In previous phases we only looked at one file at a time.

The linker is the first place where files are combined.

Linker

Linker checks that every declared function has an implementation.

This is why you get errors like:

- `Linker error: symbols not found for architecture x86`
- `Linker error: duplicate symbols found for architecture x86`

CL Compilation

We will use `g++` as our compiler.

Basic usage:

```
g++ main.cpp otherFile.cpp -o execFileName
```

CL Compilation

We will use three common compiler flags:

-std=c++11

Enable C++11 support

-g

Add debugging information to the output

-Wall

Turn on most compiler warnings

Makefiles

Basics of Make

Make is a tool to streamline the commands needed to compile big projects.

Basics of Make

```
all: main.cpp obj.cpp obj.h
     g++ -o myprogram main.cpp obj.cpp


target_name: prerequisites
   recipe
```

Basics of Make

```
target_name: prerequisites
    recipe
```

Makefiles consist of a series of targets

You can run any target by running `make target_name`

If you just run make, it'll run the first target in the file

Basics of Make

```
target_name: prerequisites
    recipe
```

Each target can have prerequisites.

If any of the prerequisites have been modified since the last time make was run, they will be made first.

Basics of Make

```
target_name: prerequisites
    recipe
```

Each target has a recipe which is just a series of shell commands.

Simple Makefile

```
all:
	g++ -o myprogram main.cpp obj.cpp obj.h
```

This basic makefile says that it doesn't have any prerequisites, and that running `make all` should simply run that command.

But there are issues with this...

Goals

We want a fast, easy to use build process.

Most of the time, we only actually modified one or two files.

Can we find a way to minimize the work in this case?

A Link to the Past

Files are distinct until the linker runs.

We should be able to reuse the output of the assembler if the source file didn’t change.

We can use the dependencies for this!

A Faster Makefile

```
all: hello

hello: main.o factorial.o hello.o
    g++ main.o factorial.o hello.o -o hello

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

factorial.o: factorial.cpp
    g++ -c factorial.cpp

hello.o: hello.cpp
    g++ -c hello.cpp

clean:
    rm *.o hello
```

Multithreading

What is a thread?

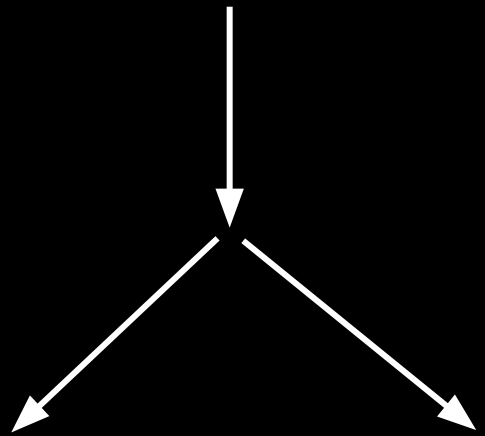
Code is usually sequential.

Threads are ways to parallelise execution.

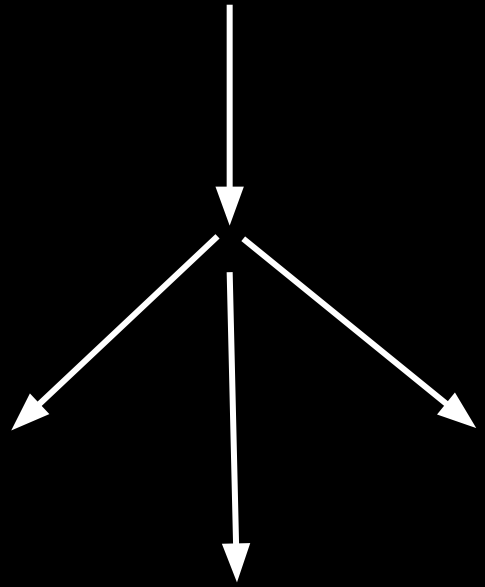
What is a thread?

What is a thread?

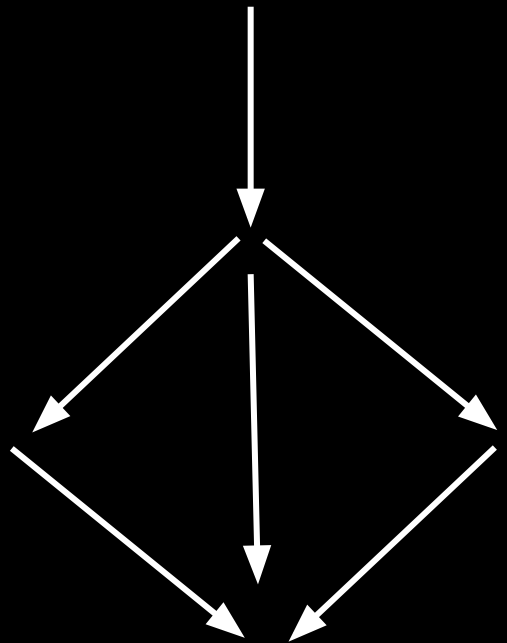
What is a thread?



What is a thread?



What is a thread?



Threads, Locks, and RAII

Let’s switch to code + whiteboard:

`Threading.pro`