COMP2221 — Sys. Prog.

Makefiles: Building for Hackathon Speed

Practical 02 • 20 October 2025

Instructor: Dr. Stuart James (stuart.a.james@durham.ac.uk)

Scenario

You are the build engineer for a 48-hour hackathon team. Your job is to deliver a **fast, reliable C build** so teammates can iterate without friction. In this practical you will: (1) compile basic C programs, (2) express correct dependencies between `.c`, `.h`, and executables, (3) remove duplication using variables, pattern rules, and automatic variables, (4) generate header auto-dependencies, and (5) exploit parallel builds. By the end, your Makefile will compile only what changed and provide developer-friendly targets.

Learning Objectives

- Compile and run simple C programs using `gcc`.
- Explain Makefile rules (target, prerequisites, recipe) and timestamp-driven rebuilds.
- Use variables (`CC`, `CFLAGS`, directories) and pattern rules (`%.o: %.c`) with automatic variables (`$@`, `$<`, `$^`).
- Organise a project into `src/`, `include/`, `obj/`, `bin/` with order-only prerequisites.
- Compare debug vs optimised builds and measure speed-ups using parallel builds (`-j`).

Prerequisites

- Access to a UNIX or UNIX-like system (Linux, macOS or `mira.dur.ac.uk` (See Ultra for details) – A general tutorial is provided in 2).
- Reviewed the lab info sheet 1.
- Ability to handle the shell and edit text files in a terminal.

Tasks

Task 1: Quick sanity check (gcc)

SOLO INTRO 10m

Outcome: Confirm a working toolchain with a single-file compile.

▶ Activity

Create `hello.c`:

hello.c

```c
#include <stdio.h>
int main(void) {
    puts("Ground station online.");
    return 0;
}
```

Compile and run:

```
gcc hello.c -o hello
./hello
```

? Quiz

If you omit `-o hello`, what is the default executable name and how do you run it?

Task 2: Your first Makefile (single-file)

SOLO INTRO 20m

Outcome: Automate the same build and introduce Makefile rules.

▶ Activity

Create a `Makefile` (capital M, no extension), write the Makefile rules that compiles hello.c (from prior task).
Build and run:

```
make
./hello
make       # run again — note “up to date”
```

Important

Spacing note: Make requires each rules line commands lines to start with an actual **tab character**, not spaces.

◇ **Reflection**

What advantage does `make` provide over typing raw `gcc` commands when files change frequently?

Task 3: Tiny log tools (multi-file + auto-deps)

SOLO

INTERMEDIATE 35m

Outcome: Separate compilation with a header; use variables, pattern rules, and auto-generated header dependencies.

▶ **Activity**

Create three files:

`reader.h`

reader.h

```
#ifndef READER_H
#define READER_H
int contains_token(const char *line, const char *token);
#endif
```

`reader.c`

reader.c

```
#include <string.h>
#include "reader.h"

int contains_token(const char *line, const char *token) {
    return (strstr(line, token) != NULL);
}
```

`main.c`

main.c

```
#include <stdio.h>
#include "reader.h"

int main(int argc, char **argv) {
    const char *token = (argc > 1) ? argv[1] : "OK";
    char buf[256];
    int count = 0;
    while (fgets(buf, sizeof buf, stdin)) {
        if (contains_token(buf, token)) count++;
    }
    printf("Lines containing '%s': %d\n", token, count);
    return 0;
}
```

▶ Activity

Create an improved `Makefile` which uses automatic variables.

▶ Activity

Create a copy of your Makefile (e.g. `Makefile2`), above each of the lines write a debug statement using `echo`. Now compile specific makefiles using `-f`.
Remember use `man make`, online manuals or Microsoft CoPilot to help you understand the `-f`.

? Quiz

Run `touch reader.h` then rebuild. What is rebuilt and why?

Task 4: Speed experiment (parallel builds) SOLO INTERMEDIATE 10m

Outcome: Measure speed benefits and change impact.

▶ Activity

From the `project/` directory, try:

```
make clean && time make -j1
touch src/reader.c && time make -j1

make clean && time make -j2
touch include/reader.h && time make -j2
```

Tip

You might also want to get from the system the number of cores on Linux this can be done using `nproc`.

Prompt: Compare timings and which files rebuilt. When does parallelism help most? What happens when only headers change?

◇ Reflection

Describe one concrete Makefile choice you made that reduced build time under change.

Task 5: AI assistance (optional) WITH AI OPTIONAL INTERMEDIATE 15m

Outcome: Critically assess an AI-generated Makefile.

► Activity

Use Microsoft CoPilot to produce a Makefile for `project/` with two build flavours and auto-deps. Compare it to yours:

- Variables and directory layout present?
- Pattern rules using automatic variables?
- Auto-deps with `-MMD -MP` included?
- Helpful `help` target? `.PHONY`s set?

Save a short reflection as `ai_makefile_reflection.txt`.

Important

Use AI to explore patterns—but *verify* every rule. You are responsible for correctness and maintainability.

Task 6: Assessed: System Diagnostics Suite ASSESSED HARD

45m

Develop a small C diagnostics suite that builds multiple related programs using a single, professional Makefile.

Scenario: Your hackathon team is producing a "System Diagnostics Suite" for a prototype rover. The suite includes several small utilities built from shared code: a status viewer, a configuration reporter, and a test harness.

Each program reuses one common library of helper functions (‘common.c‘, ‘common.h‘) that define the system name and a helper to print formatted output.

Project structure:

```
diagnostics/
|-- src/
|   |-- common.c
|   |-- status.c
|   |-- config.c
|   |-- tester.c
|-- include/
|   |-- common.h
|-- obj/
|-- bin/
|-- Makefile
```

▶ Activity

Files to create:

include/common.h

```c
#ifndef COMMON_H
#define COMMON_H
#define SYSTEM_NAME "Hackathon Rover"
void print_header(const char *title);
#endif
```

src/common.c

```c
#include <stdio.h>
#include "common.h"

void print_header(const char *title) {
    printf("=== %s ===\n", SYSTEM_NAME);
    printf("%s\n", title);
    printf("-------------------------\n");
}
```

► Activity

Files to create:

src/status.c

```c
#include <stdio.h>
#include "common.h"

int main(void) {
    print_header("Status Report");
    printf("Network: online\nDatabase: connected\nUI:
        responsive\n");
    return 0;
}
```

src/config.c

```c
#include <stdio.h>
#include "common.h"

int main(void) {
    print_header("Configuration");
    printf("Build mode: standard\nOptimisation: enabled\n");
    return 0;
}
```

src/tester.c

```c
#include <stdio.h>
#include "common.h"

int main(void) {
    print_header("System Self-Test");
    printf("All diagnostics passed.\n");
    return 0;
}
```

► Activity

Makefile requirements:

- Use variables: `CC`, `CFLAGS`, `SRCDIR`, `INCDIR`, `OBJDIR` (Object Directory), `BINDIR` (Binary Directory).
- Build three executables: `bin/status`, `bin/config`, `bin/tester`.
- Use pattern rules with automatic variables (`$@`, `$<`, `$^`).
- Each executable must depend on its own object file and `obj/common.o`.
- Provide `.PHONY` targets: `all`, `clean`, `run-status`, `run-config`, `run-tester`, `help`.
- Add a `debug` target that rebuilds everything with `-g -O0`.
- Name your file `Makefile-task6` for submission.

► Activity

You now need to prepare your project for deployment on Durham's NCC (NVIDIA CUDA Centre) cluster. Write a small batch submission script to compile and run your existing `diagnostics` project using the Makefile you created. Consult the NCC documentation (`https://nccadmin.webspace.durham.ac.uk/`) to identify the correct scheduler directives, compiler module, and resource requests (time, CPUs, memory). Your script should:

- Load the necessary compiler or toolchain environment.
- Request appropriate compute resources.
- Build and run your programs automatically.

Save your script as `run-task6.sh`.

Tip

Look up the NCC batch scheduler syntax and commands—make sure your script includes a shebang, resource requests, module loading, and calls to `make` and your executables.

Tip

In the documentation the `Recommended Headers` gives you an example format.

Important

Do NOT request a GPU. Make sure you use the cpu Partition. (Requesting resources you do not need is wasteful in shared environments).

Submission Checklist

Submission Checklist

- ☐ `Makefile-task3` — your improved Makefile from Task 3 using variables and automatic rules.
- ☐ (Optional) `ai_makefile_reflection.txt` — if you completed the AI assistance activity.
- ☐ `Makefile-task6` — your professional multi-target Makefile that builds all tools.
- ☐ `run-task6.sh` — your NCC batch submission script.

Place all the above files in a single folder named `Practical2` and submit it via the coursework submission system.

The weekly structure can be assumed to be:

```
Practical1/
|-- [Contents of Practical1]
Practical2/
|-- Makefile-task3
|-- ai_makefile_reflection.txt    # (optional)
|-- Makefile-task6
|-- run-task6.sh
Practical3/
Practical4/
Practical5/
Practical6/
Practical7/
Practical8/
Practical9/
README.md
```

1 Task Tag Reference

Task Tag Guide

Each task in this worksheet includes small coloured *tags* that give you extra context about the task requirements and expectations:

Approach:

SOLO	This task is intended to be attempted individually. Work independently to develop your own understanding and solutions.
PARTNER	This task is designed for collaboration with a partner. Share ideas, discuss approaches, and learn from each other.
WITH AI	This task explicitly incorporates AI assistance (e.g., Microsoft Copilot). Use AI as a coding partner for critique and optimization.
ASSESSED	This task contributes to your course grade. Follow requirements precisely and ensure high-quality submissions.
OPTIONAL	This task is optional and can be skipped if time is limited. Recommended for students wanting extra practice or deeper understanding.

Difficulties:

INTRO	Introductory level - focuses on basic concepts and simple implementations.
INTERMEDIATE	Intermediate level - requires combining concepts and problem-solving skills.
HARD	Advanced level - involves complex integration and production-ready considerations.

Guidance:

10m	Estimated completion time in minutes. Use this to plan your session effectively.

All tags are displayed on the same line as the task heading, aligned to the right for quick visual scanning and session planning.

2 Using the Terminal Across Operating Systems

In this module, you will use the Unix terminal to run commands, write shell scripts, and connect to the University's remote Linux system. While the practicals assume a Linux-style environment, there are several ways to access this from your own computer depending on your operating system.

Accessing Mira (Durham Remote Linux Server)

Durham University provides a remote Linux environment known as **mira**. You can log in securely using the `ssh` (Secure Shell) command, which is available on all modern systems.

- **On macOS or Linux:** Open your Terminal and type:

  ```
  ssh ab12cd@mira.dur.ac.uk
  ```

  Replace `ab12cd` with your CIS username.
- **On Windows (with WSL):** Open your WSL (e.g., Ubuntu) terminal and run the same command:

  ```
  ssh ab12cd@mira.dur.ac.uk
  ```

- **On other systems:** If you are unable to use a native terminal, you can connect using an SSH client such as **PuTTY** (available for Windows).

After entering your CIS password, you will likely be asked for your authenticator code, which you can get from your normal Microsoft Authenticator app. After which, you will be logged into Mira's Linux environment. This is the same setup used in the labs and by all teaching materials. Files saved on Mira remain available between sessions and can be accessed both on and off campus.

Mira GUI using X2Go

1. Log in to the Linux timeshare service (mira) using X2Go:
   - On Windows MDS: Go to `https://appsanywhere.durham.ac.uk/` and launch "X2go-Mira r3"
   - On your own computer: Download X2Go from `https://www.x2go.org`
   - Set "Host" to `mira.dur.ac.uk` and "Session Type" to "MATE"
   - Use your CIS username and password to login
   - Click "Yes" if asked about trusting the host key
2. Open a terminal window (click the screen icon at the top with tooltip "MATE Terminal")

Uploading and Downloading Files with Mira

Sometimes you will want to transfer files between your own computer and Mira — for example, uploading a script you have written locally or downloading output from your experiments. There are two easy command-line methods to do this: `scp` (secure copy) and `sftp` (secure file transfer).

Using `scp` The `scp` command lets you copy files directly over SSH. You can use it in either direction.

- **Upload a file to Mira:**

  ```
  scp myscript.sh ab12cd@mira.dur.ac.uk:~/COMP2221/
  ```

  This uploads `myscript.sh` from your current directory to the folder `COMP2221` in your Mira home directory.

- **Download a file from Mira:**

  ```
  scp ab12cd@mira.dur.ac.uk:~/results/output.txt .
  ```

  The final dot (`.`) means "save it to my current local directory".

You can also copy entire directories using the `-r` (recursive) flag:

```
scp -r ab12cd@mira.dur.ac.uk:~/data ./data_copy
```

Using `sftp` If you prefer an interactive approach, `sftp` works like a remote file browser within the terminal.

```
sftp ab12cd@mira.dur.ac.uk
```

Once connected, you can use simple commands:

```
ls        # list files on Mira
cd lab02  # change remote directory
get file.txt   # download a file
put script.sh  # upload a file
bye       # exit
```

Tip: Graphical Clients If you prefer a graphical interface, you can use software such as:

- **FileZilla** or **Cyberduck** (macOS/Windows) — connect via `sftp://mira.dur.ac.uk`
- ForkLift (macOS) — connect via `sftp://mira.dur.ac.uk`
- **VS Code Remote SSH** extension — edit and transfer files directly within VS Code.

All of these methods use the same secure SSH connection, so whichever you choose, your data remains protected.

Using the Terminal on macOS

macOS includes a built-in Unix terminal. You can find it under:

```
Applications → Utilities → Terminal
```

By default, macOS uses the **Z shell (zsh)**. Our lecture examples and practical tasks are written for the **Bourne Again Shell (bash)** for simplicity and consistency. To switch temporarily to `bash`, type:

```
bash
```

To make `bash` your default shell (not recommended), you can run:

```
chsh -s /bin/bash
```

Once set, opening a new terminal window will place you directly into the `bash` shell, matching the environment used in lectures and on Mira.

Using the Terminal on Windows

Windows users can obtain a Linux-style terminal by installing the **Windows Subsystem for Linux (WSL)**. This provides an environment very similar to Mira's, allowing the same commands and scripts to run.

- Install WSL by running the following in PowerShell (as Administrator):

  ```
  wsl --install
  ```

- After restarting, open the "Ubuntu" application from the Start Menu.
- Inside this terminal, you can use the same commands as in the practicals, including:

  ```
  ssh ab12cd@mira.dur.ac.uk
  ```

Compatibility note: While WSL provides good compatibility, certain Linux system behaviours (such as process management or file permissions) may differ slightly. You should therefore test your scripts on **Mira** before submission to ensure they behave as expected in the assessment environment.

Summary

System	Recommended Shell	Notes
Mira (Durham)	bash	Fully supported Linux environment
macOS	bash (instead of zsh)	Switch shell for alignment with teaching
Windows	bash via WSL	May have minor compatibility differences

Regardless of your operating system, using a **bash-compatible shell** will ensure your experience matches the lecture content and Mira environment.