# CM3 - C Language

## Representations, programs, and builds

Louis Ledoux

2026-07-04

# Course Shape

# The rule for this deck

Horizontal movement continues the lecture path.

Vertical movement opens details, examples, or exercises on the current topic.

In RevealJS, use right/left for the main path and down/up for deeper material.

# Today as a C course

- C programs are data plus representation choices.
- Source files are translated, linked, and executed.
- Tools matter: compiler warnings, `make`, debugger, memory checker.
- The deck itself is a testbed for teaching code well.

# What this version demonstrates

| Feature | RevealJS | Beamer PDF | Typst handout |
|---|---|---|---|
| Vertical detail slides | interactive | printed as pages | printed as sections |
| Progressive line highlights | stepped | static fallback | static fallback |
| Code annotations | hover/selectable | below code | below code |
| Scrollable long code | live only | avoid for final | avoid for final |
| Executed C example | executed at render | executed at render | executed at render |

# Sources and adaptation

This version combines:

- your imported CM3 material on files, compilation, `make`, tools, and style;
- the Berkeley CS61C lecture you added, especially number representations, C vs Java, `printf`, and undefined behavior;
- Quarto/RevealJS features that are useful for teaching code.

The Berkeley material is used as inspiration for topic order and examples, not as a slide import.

# Values And Bits

# Representation is a contract

A bit pattern has no meaning by itself.

`0b11111111` can mean:

- unsigned integer: `255`
- signed 8-bit integer: `-1`
- character byte: implementation dependent
- raw mask: all bits set

The program works only if writer and reader agree on the representation.

**read_byte.c** C

```c
#include <stdint.h>
#include <stdio.h>

int main(void) {
    uint8_t byte = 0xff;
    printf("%u\n", byte);
}
```

# Fixed-width integers

## When the size matters, say it.

types.c

```c
#include <stdint.h>
#include <stdio.h>

int main(void) {
    uint8_t  byte = 0xff;
    int32_t  temp = -12;

    printf("byte=%u temp=%d\n", byte, temp);
    return 0;
}
```

# Unsigned arithmetic wraps

wrap.c

```c
#include <stdint.h>
#include <stdio.h>

int main(void) {
    uint8_t x = 255;
    x = x + 1;
    printf("%u\n", x);
}
```

The result is `0` because unsigned arithmetic is modulo `2^N`.

▶ **Quick check**

▶ **Answer**

# Signed overflow is different

overflow.c

```c
#include <limits.h>
#include <stdio.h>

int main(void) {
    int x = INT_MAX;
    printf("%d\n", x + 1);
}
```

This is not the same story as unsigned wraparound. Signed overflow is undefined behavior in C.

> ⚠ **Warning**
>
> For teaching: this is a good moment to separate “what my machine did today” from “what C promises”.

# First Program

# Anatomy of `main`

hello.c

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("hello, CM3\n");
    return 0;
}
```

# `argv` is just data from the shell

**session.sh** SHELL

```
$ ./show_args 1 hi
argv[0] = ./show_args
argv[1] = 1
argv[2] = hi
```

**show_args.c** C

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    for (int i = 0; i < argc; i++) {
        printf("argv[%d] = %s\n", i, argv[i]);
    }
    return 0;
}
```

# `printf`: integer first

printf.c

```c
int n = 42;
printf("value = %d\n", n);
```

`%d` asks `printf` to read the next argument as a signed decimal integer.

# `printf`: then floating point

printf.c

```c
double x = 42.0;
printf("value = %f\n", x);
```

`%f` asks `printf` to read the next argument as a floating-point value.

# Format strings are contracts

wrong-format.c

```c
#include <stdio.h>

int main(void) {
    double x = 42.0;
    printf("as int: %d\n", x);
    printf("as double: %f\n", x);
}
```

The compiler may warn, but `printf` itself trusts the format string.

# Common `printf` specifiers

| Specifier | Meaning | Typical argument |
|---|---|---|
| `%d` | signed decimal integer | `int` |
| `%u` | unsigned decimal integer | `unsigned int` |
| `%x`, `%X` | hexadecimal integer | `unsigned int` |
| `%f` | floating point | `double` |
| `%s` | string | `char *` |
| `%p` | address | `void *` |
| `%%` | literal percent sign | none |

# Compilation

# C is compiled for a target machine

C:

- source is translated before execution;
- output depends on architecture and system ABI;
- compiler warnings are part of the feedback loop.

Java/Python comparison:

- Java source becomes bytecode, then JIT/runtime work happens;
- Python usually compiles to bytecode at runtime;
- portability is paid for by a runtime.

# The build pipeline

build.sh SHELL

```
1 $ gcc -E hello.c -o hello.i
2 $ gcc -S hello.i -o hello.s
3 $ gcc -c hello.s -o hello.o
4 $ gcc hello.o -o hello
```

- Preprocess: expand directives like `#include`.
- Compile: turn C into assembly.
- Assemble: turn assembly into an object file.
- Link: combine objects and libraries into an executable.

# Quarto can execute a C build at render time

The generated RevealJS/Beamer/Typst output is static, but this cell compiles and runs C while the document is rendered.

```
$ gcc -Wall -Wextra hello.c -o hello
$ ./hello
hello from compiled C
```

# Static trace is still useful

Use a plain `bash` code block when the goal is a readable transcript rather than live execution.

session.sh SHELL

```
$ gcc -Wall -Wextra hello.c -o hello
$ ./hello
hello from compiled C
```

This is deterministic, portable, and easy to print.

# Multi-File Projects

# A header is the public contract

**counter.h** C

```c
#ifndef COUNTER_H
#define COUNTER_H

extern int counter;
int next_value(void);

#endif
```

**counter.c** C

```c
#include "counter.h"

int counter = 0;

int next_value(void) {
    return ++counter;
}
```

# Include declarations, not duplicate definitions

main.c

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

int main(void) {
    printf("%d\n", next_value());
    return 0;
}
```

The header lets each `.c` file compile independently while agreeing on shared declarations.

# Longer example with scrolling

This slide intentionally scrolls in RevealJS so we can discuss a complete small project without splitting context.

**counter.h** C

```c
#ifndef COUNTER_H
#define COUNTER_H

extern int counter;
int next_value(void);

#endif
```

**counter.c** C

```c
#include "counter.h"

int counter = 0;
```

```c
int next_value(void) {
    return ++counter;
}
```

**report.c** C

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

void report_value(const char *label) {
    printf("%s: %d\n", label, counter);
}
```

**main.c** C

```c
#include "counter.h"

void report_value(const char *label);

int main(void) {
    report_value("initial");
    next_value();
    report_value("after one step");
    next_value();
    report_value("after two steps");
```

```
    return 0;
}
```

# Build Automation

# Separate compilation

build.sh SHELL

```
$ gcc -Wall -Wextra -c main.c -o main.o
$ gcc -Wall -Wextra -c counter.c -o counter.o
$ gcc -Wall -Wextra -c report.c -o report.o
$ gcc main.o counter.o report.o -o demo
$ ./demo
```

Object files let us rebuild only what changed.

# A Makefile records dependencies

Makefile MAKE

```
1  CC = gcc
2  CFLAGS = -Wall -Wextra -g
3  OBJ = main.o counter.o report.o
4  EXEC = demo
5
6  all: $(EXEC)
7
8  $(EXEC): $(OBJ)
9  ____$(CC) $(OBJ) -o $@
10
11 main.o: main.c counter.h
12 ____$(CC) $(CFLAGS) -c $< -o $@
13
14 counter.o: counter.c counter.h
15 ____$(CC) $(CFLAGS) -c $< -o $@
16
17 report.o: report.c counter.h
18 ____$(CC) $(CFLAGS) -c $< -o $@
```

# Full Makefile with tests

This is the kind of reference slide where vertical scrolling can be useful during live teaching.

**Makefile** MAKE

```
CC = gcc
CFLAGS = -std=c17 -Wall -Wextra -Wpedantic -g
CPPFLAGS = -Iinclude
LDFLAGS =

SRC_DIR = src
TEST_DIR = tests
BUILD_DIR = build

APP = $(BUILD_DIR)/counter-demo
TEST_APP = $(BUILD_DIR)/counter-tests

APP_SRC = $(SRC_DIR)/main.c \
          $(SRC_DIR)/counter.c \
          $(SRC_DIR)/report.c
```

```
TEST_SRC = $(TEST_DIR)/test_counter.c \
           $(SRC_DIR)/counter.c

APP_OBJ = $(APP_SRC:%.c=$(BUILD_DIR)/%.o)
TEST_OBJ = $(TEST_SRC:%.c=$(BUILD_DIR)/%.o)

.PHONY: all test clean distclean run

all: $(APP)

run: $(APP)
	./$(APP)

test: $(TEST_APP)
	./$(TEST_APP)

$(APP): $(APP_OBJ)
	$(CC) $(APP_OBJ) $(LDFLAGS) -o $@

$(TEST_APP): $(TEST_OBJ)
	$(CC) $(TEST_OBJ) $(LDFLAGS) -o $@

$(BUILD_DIR)/%.o: %.c include/counter.h
	mkdir -p $(dir $@)
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@

clean:
```

```
    rm -rf $(BUILD_DIR)/**/*.o

distclean:
    rm -rf $(BUILD_DIR)
```

# The tab matters

**Makefile** MAKE

```
clean:
	rm -f *.o demo
```

The command line under a target starts with a real tab character.

> **Tip**
>
> For class, make the invisible tab visible once, then stop talking about it and let `make` enforce the rule.

# What `make` changes

session.sh — SHELL

```
$ make
gcc -Wall -Wextra -g -c main.c -o main.o
gcc -Wall -Wextra -g -c counter.c -o counter.o
gcc -Wall -Wextra -g -c report.c -o report.o
gcc main.o counter.o report.o -o demo

$ touch report.c
$ make
gcc -Wall -Wextra -g -c report.c -o report.o
gcc main.o counter.o report.o -o demo
```

Only the affected object and the final executable are rebuilt.

# Undefined Behavior

# Quiz: uninitialized local variable

ub.c

```c
#include <stdio.h>

int main(void) {
    int x;
    printf("%d\n", x);
    return 0;
}
```

What should students answer?

# Undefined behavior breaks reasoning

Undefined behavior can:

- appear stable on one machine;
- change after optimization;
- change when a debugger is attached;
- disappear when a print is added;
- survive tests and fail later.

The teaching rule: compile with warnings, remove undefined behavior first, then debug logic.

# Use braces

braces.c

```c
if (ready)
    start();

if (ready) {
    start();
}
cleanup();
```

The second form makes scope visible and prevents future edits from changing control flow by accident.

# C truthiness

truth.c

```c
if (0) {
    puts("false");
}

if (42) {
    puts("true");
}

if ("hello") {
    puts("also true");
}
```

In C17, include `<stdbool.h>` if you want `bool`, `true`, and `false`.

# Tools

# Warnings are part of the course

session.sh SHELL

```
$ gcc -Wall -Wextra -Wpedantic -g ub.c -o ub
ub.c: In function 'main':
ub.c:5:5: warning: 'x' is used uninitialized [-Wuninitialized]
```

Do not treat warnings as decoration. In this course, students should read them from the top.

# Scrollable diagnostics trace

This slide is intentionally too long for a fixed slide. The point is to practice reading from the first meaningful diagnostic.

diagnostics.sh SHELL

```
$ gcc -std=c17 -Wall -Wextra -Wpedantic -g src/main.c src/counter.c -o demo
src/main.c: In function 'main':
src/main.c:8:5: warning: implicit declaration of function 'report_value' [-
        Wimplicit-function-declaration]
    8 |     report_value("initial");
      |     ^~~~~~~~~~~~
src/main.c:10:17: warning: format '%d' expects argument of type 'int', but
        argument 2 has type 'double' [-Wformat=]
   10 |     printf("%d\n", 3.14);
      |             ~^     ~~~~
      |              |     |
      |              int   double
      |             %f
src/main.c:12:9: warning: unused variable 'unused' [-Wunused-variable]
   12 |     int unused = 0;
```

```
      |          ^~~~~
src/counter.c: In function 'next_value':
src/counter.c:7:1: warning: control reaches end of non-void function [-
        Wreturn-type]
    7 | }
      | ^
/usr/bin/ld: /tmp/ccxT4y0F.o: in function 'main':
src/main.c:8: undefined reference to 'report_value'
/usr/bin/ld: src/main.c:9: undefined reference to 'report_value'
collect2: error: ld returned 1 exit status
```

Start at the first warning that explains a real source problem, then rebuild after each fix.

# Debugger workflow

gdb-session.sh SHELL

```
$ gcc -Wall -Wextra -g main.c counter.c report.c -o demo
$ gdb ./demo
(gdb) break main
(gdb) run
(gdb) next
(gdb) print counter
(gdb) backtrace
```

Good debugger slides are usually command traces plus a small code fragment, not screenshots.

# Memory checker workflow

valgrind-session.sh SHELL

```
$ valgrind --tool=memcheck ./demo
==12345== HEAP SUMMARY:
==12345==     in use at exit: 16 bytes in 1 blocks
==12345==   total heap usage: 1 allocs, 0 frees
==12345==
==12345== LEAK SUMMARY:
==12345==    definitely lost: 16 bytes in 1 blocks
```

The tool is useful after the program compiles and the failure is runtime behavior.

# Export Strategy

# One source, three outputs

render.sh SHELL

```
$ PATH="$PWD/venv/bin:$PATH" quarto render cm3.qmd --to revealjs
$ PATH="$PWD/venv/bin:$PATH" quarto render cm3.qmd --to beamer
$ PATH="$PWD/venv/bin:$PATH" quarto render cm3.qmd --to typst
```

| Output | File | Use |
|---|---|---|
| RevealJS | `cm3.html` | live lecture with navigation, fragments, scroll, animation |
| Beamer | `cm3-beamer.pdf` | slide PDF close to the classroom deck |
| Typst | `cm3-handout.pdf` | printable reading handout from the same source |

# Print handouts

RevealJS can still be printed from the browser using `?print-pdf`, but the student n-up PDFs use the Beamer export as their source. That keeps the printed versions clean, centered, and free of the web background.

**handouts.sh** SHELL

```
$ chromium --headless --no-sandbox \
    --print-to-pdf=cm3-reveal-print.pdf \
    "file://$PWD/cm3.html?print-pdf"

$ pdfjam cm3-beamer.pdf --nup 1x2 --paper a4paper \
    --frame true --delta "0cm 0.25cm" --scale 0.96 \
    --outfile cm3-2up.pdf
$ pdfjam cm3-beamer.pdf --nup 2x2 --landscape --paper a4paper \
    --frame true --delta "0.20cm 0.20cm" --scale 0.96 \
    --outfile cm3-4up.pdf
$ pdfjam cm3-beamer.pdf --nup 2x3 --paper a4paper \
```

```
    --frame true --delta "0.18cm 0.18cm" --scale 0.96 \
    --outfile cm3-6up.pdf
$ pdfjam cm3-beamer.pdf --nup 3x3 --landscape --paper a4paper \
    --frame true --delta "0.12cm 0.12cm" --scale 0.94 \
    --outfile cm3-9up.pdf
```

These PDFs are for students who print. They should not depend on scrolling.

# What to keep in the final course

Keep:

- code filenames and line highlights;
- vertical detail slides for optional explanations;
- one executed example per topic when it proves something;
- short command traces for terminal stories;
- Typst handouts for printable reading.

Avoid:

- long scrollable slides as the only source of information;
- terminal styling that competes with syntax highlighting;
- animations that carry essential meaning with no PDF fallback.