CM3 - C Language

Representations, programs, and builds

Louis Ledoux

2026-07-04

Table of contents I

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 printable formats, the same vertical detail slides become normal pages.

Today as a C course

- C programs are data plus representation choices.

Today as a C course

- C programs are data plus representation choices.
- Source files are translated, linked, and executed.

5 / 48

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.

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.

Listing 1 `read_byte.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.

Listing 2 `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

Listing 3 `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

What does this program print?

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

Signed overflow is different

Listing 4 `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`

Listing 5 `hello.c`

```
#include <stdio.h>                                  (1)

int main(int argc, char *argv[]) {                  (2)
    printf("hello, CM3\n");                         (3)
    return 0;                                       (4)
}
```

(1) Declaration of the standard I/O interface.
(2) Entry point; `argc` and `argv` describe command-line arguments.
(3) Formatted output.
(4) Exit status returned to the shell.

`argv` is just data from the shell

Listing 6 `session.sh`

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

Listing 7 `show_args.c`

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     for (int i = 0; i < argc; i++) {
5         printf("argv[%d] = %s\n", i, argv[i]);
6     }
7     return 0;
8 }
```

printf: integer first

Listing 8 printf.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

Listing 9 printf.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

Listing 10 `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

Listing 11 `build.sh`

```
$ gcc -E hello.c -o hello.i
$ gcc -S hello.i -o hello.s
$ gcc -c hello.s -o hello.o
$ gcc hello.o -o hello
```

- Preprocess: expand directives like `#include`.

The build pipeline

Listing 12 `build.sh`

```
$ gcc -E hello.c -o hello.i
$ gcc -S hello.i -o hello.s
$ gcc -c hello.s -o hello.o
$ gcc hello.o -o hello
```

- Preprocess: expand directives like `#include`.
- Compile: turn C into assembly.

The build pipeline

Listing 13 `build.sh`

```
$ gcc -E hello.c -o hello.i
$ gcc -S hello.i -o hello.s
$ gcc -c hello.s -o hello.o
$ gcc hello.o -o hello
```

- Preprocess: expand directives like `#include`.
- Compile: turn C into assembly.
- Assemble: turn assembly into an object file.

The build pipeline

Listing 14 `build.sh`

```
$ gcc -E hello.c -o hello.i
$ gcc -S hello.i -o hello.s
$ gcc -c hello.s -o hello.o
$ 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.

Listing 15 `session.sh`

```
$ 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

Listing 16 `counter.h`

```h
#ifndef COUNTER_H
#define COUNTER_H

extern int counter;
int next_value(void);

#endif
```

Listing 17 `counter.c`

```c
#include "counter.h"

int counter = 0;

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

Include declarations, not duplicate definitions

Listing 18 `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

In printable exports, prefer splitting this example into several shorter slides or keeping it as an appendix.

Listing 19 `counter.h`

```c
#ifndef COUNTER_H
#define COUNTER_H

extern int counter;
int next_value(void);

#endif
```

Listing 20 `counter.c`

```c
#include "counter.h"

int counter = 0;
```

Build Automation

Separate compilation

Listing 23 `build.sh`

```
$ 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

Listing 24 `Makefile`

```
CC = gcc
CFLAGS = -Wall -Wextra -g
OBJ = main.o counter.o report.o
EXEC = demo

all: $(EXEC)

$(EXEC): $(OBJ)
    $(CC) $(OBJ) -o $@

main.o: main.c counter.h
    $(CC) $(CFLAGS) -c $< -o $@

counter.o: counter.c counter.h
    $(CC) $(CFLAGS) -c $< -o $@
```


Full Makefile with tests

For printable exports, keep the long Makefile in a handout or repository file. The slide deck should show the dependency pattern, not every target.

The tab matters

Listing 25 `Makefile`

```
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

Listing 26 `session.sh`

```
$ 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

Listing 27 `ub.c`

```c
#include <stdio.h>

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

What should students answer?

- Not “0”.

Quiz: uninitialized local variable

Listing 28 `ub.c`

```c
#include <stdio.h>

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

What should students answer?

- Not "0".
- Not "a random number" as a language guarantee.

Quiz: uninitialized local variable

Listing 29 `ub.c`

```c
#include <stdio.h>

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

What should students answer?

- Not "0".
- Not "a random number" as a language guarantee.
- The correct C answer is: the program has undefined behavior.

Undefined behavior breaks reasoning

Undefined behavior can:

- appear stable on one machine;

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

Undefined behavior breaks reasoning

Undefined behavior can:

- appear stable on one machine;
- change after optimization;

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

Undefined behavior breaks reasoning

Undefined behavior can:

- appear stable on one machine;
- change after optimization;
- change when a debugger is attached;

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

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;

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

37 / 48

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

Listing 30 `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

Listing 31 `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

Listing 32 `session.sh`

```
$ 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

Printable exports should use a shorter diagnostic excerpt and discuss the full trace in the handout.

Debugger workflow

Listing 33 `gdb-session.sh`

```
$ 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

Listing 34 `valgrind-session.sh`

```
$ 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

Listing 35 `render.sh`

```
$ 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.

Listing 36 `handouts.sh`

```
$ 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 \
```


What to keep in the final course

Keep:

- code filenames and line highlights;

Avoid:

What to keep in the final course

Keep:

- code filenames and line highlights;
- vertical detail slides for optional explanations;

Avoid:

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;

Avoid:

48 / 48

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;

Avoid:

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:

48 / 48

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;

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;

48 / 48

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.