COSC 50: Software Design and Implementation
Winter 2026

Lecturer: Professor Katherine Salesin

Notes by: Farhan Sadeek

Last Updated: April 4, 2026

Contents

1 January 5, 2026 **1**

2 January 7, 2026 **2**

3 January 12, 2026 **2**

1 Git concepts and commands . . . 2
2 Introduction to C . . . 6
3 The C Compilation Process . . . 8
4 Naming Constants, Variables, and Functions in C . . . 10
5 C Header Files . . . 11
6 The Math Library (`math.h`) . . . 12

4 January 14, 2026 **13**

1 C Types . . . 13
2 C - Global and Local Scopes . . . 15
3 Command-Line Arguments in C . . . 17
4 Files: Opening, Reading, and Writing . . . 18
5 C – Strings . . . 20
6 Memory, pointers, and malloc . . . 20
7 Dynamic memory allocation . . . 21
8 When do we need to `malloc` a string? . . . 22

1 January 5, 2026

Today was the first day of classes, and as usual we came in late but we started off discussing the syllabus and then professors showed some of the most common Unix commands and then we went over the worksheet for

the first week. Today we had the first lab that was due on the same day in class. **Common Unix Commands**

Action	Example Command(s)
Show current directory	`pwd`
List files in a directory	`ls`, `ls -l`, `ls -a`
Create a file	`touch filename.txt`
Make a new directory	`mkdir my_folder`
Remove a file	`rm myfile.txt`
Remove a directory (and files)	`rm -r my_folder`
Change to another directory	`cd /path/to/dir`
Go up one directory	`cd ..`
Copy a file	`cp source.txt dest.txt`
Move or rename a file	`mv oldname.txt newname.txt`
Show contents of a file	`cat file.txt`, `less file.txt`
Edit a file (with nano)	`nano file.txt`
Print text to terminal	`echo "Hello, World!"`
Compare two files	`diff file1.txt file2.txt`

2 January 7, 2026

3 January 12, 2026

This week was mostly about getting used to Git and the C Programming language.

1 Git concepts and commands

In this unit, we dive into Git to better understand its core concepts and commands. We cover:

- The concept of version control
- The Git workflow and the Three State Model
- Git setup and initializing a repository
- Basic Git commands
- The concept of a commit history
- Branches in Git
- Remotes in Git
- Git commands for committing, branching, merging, pushing, and pulling

Of course, it's impossible to describe everything about Git in these notes. For further learning, we can consult additional Git resources.

Version Control

We have likely used version control without even realizing it! For example, Google Docs maintains a history of our edits to a document—we can view previous versions, or revert to an earlier state. Version-control systems are tools that help us keep track of changes in files (or sets of files) over time.

In software development, such systems are commonly referred to as source-code control systems (`sccs`). Today, Git is the most common such tool, inheriting ideas from predecessors such as `svn`, `cvs`, `rcs`, and even a system called `sccs`, all of which are still in use in some places.

Git (and similar systems) maintains an organized historical record of a software project, potentially across hundreds of files, directories, and collaborators. A key part of using such a system is *committing* changes: at specific points, we explicitly record an updated version of our files to the historical record. The result is a **commit history**: a sequence of snapshots reflecting the evolution of the project.

The power of Git goes further: we can compare the current state of files to any earlier commit, or revert the project to an earlier snapshot. Though Git includes many advanced features (and can seem confusing), the basic workflow is learnable by anyone with a bit of practice.

Repositories and Clones

A Git **repository** (or **repo**) is a directory of files and subdirectories managed together. We can have multiple copies ("clones") of a repository. As individuals, we usually work with one clone on our own computer. In teams, every member has their own clone, each holding the entire commit history.

Each person makes changes and commits to their own local clone—even offline! Synchronization of changes among team members is handled via *remotes* (more on this later).

The Three-State Model and Git Workflow

Whenever we edit files in a Git repository, Git considers three different regions:

1. The **working copy**—the files as they currently appear in our editor or directory.
2. The **staging area**—where files are prepared ("staged") to be included in the next commit.
3. The **local repository**—where commits (snapshots) have been recorded in the past.

   Example workflow:

```
$ mkdir myproject
$ cd myproject
$ git init
$ git branch -m main       # rename initial branch to 'main'
$ echo "# My Project" > README.md
$ git status               # shows README.md as untracked
$ git add README.md        # adds the file to the staging area
$ git status
$ git commit -m "Add README.md"  # commits staged changes
```

Each step corresponds to moving our work through the three states, ending with a snapshot preserved in the local repository.

Viewing History: git log

We can use `git log` to see a detailed history of commits, including author, date, message, and a unique hash for each commit. If there are many commits, the log uses a pager (`less`) so we can scroll (space for next page, `q` to quit).

Comparing Changes: git diff

To see what has changed since the last commit:

```
$ git diff
```

To compare our state with an earlier commit, use its hash (from `git log`):

```
$ git diff 56fd541
```

Branches

A Git repo's history can branch: we can work on different features in parallel, isolated from each other.

For example:

```
$ git branch fribble    # create a new branch called 'fribble'
$ git switch fribble   # switch to the 'fribble' branch
# ... work and commit as usual ...
$ git commit -a -m "Finish fribble feature"
$ git switch main      # back to the main branch
```

We can have multiple branches for different features. To list branches:

```
$ git branch
```

The branch with an asterisk is the one we're currently on.

Merging Branches

To bring changes from one branch (e.g., `fribble`) into another (e.g., `main`):

```
$ git switch main
$ git merge fribble
```

This attempts to combine the work; if there are conflicting changes to a file, we'll need to resolve those conflicts manually.

Merging creates a new "merge commit" in the history.

Remotes, Pushing, and Pulling

Git was designed for distributed development. Any clone of a repository can be connected to one or more **remotes**—copies of the repo on another server, such as GitHub.

To add a remote named `origin` pointing to our repo on GitHub:

```
$ git remote add origin git@github.com:username/reponame.git
```

Pushing our commits to the remote:

```
$ git push -u origin main
```

After the first push, we can just use `git push`.

To get commits from the remote repo:

```
$ git pull
```

or, to be explicit,

```
$ git pull origin main
```

List our git remotes:

```
$ git remote -v
```

Naming: Branch 'master' vs. 'main'

Historically, the default branch in Git was called `master`. Today, most new repos use `main` as the default. In CS50 and these notes, we refer to `main`.

To ensure the main branch exists and is named accordingly:

```
$ git init
$ git branch -M main
```

.gitignore Files

By default, Git will let us add any file to the repository—including object files, backups, binaries, etc. It's good practice to only commit *source files* and exclude derived files. To specify files Git should ignore, we create a `.gitignore` file.

Example tree structure with ignores in place:

```
.
|-- .gitignore
|-- README.md
|-- client
|   |-- .gitignore
|   |-- README.md
|   `-- client.c
`-- server
    |-- .gitignore
    |-- README.md
    `-- server.c
```

Best Practices

- Never use `git add .` blindly—it adds all unignored files, including unintended ones.
- Make all our edits locally; do **not** edit files in the web browser on GitHub.
- Always write meaningful commit messages.
- Use branches to manage separate features or bug fixes.
- Use .gitignore to keep our repository clean.

2 Introduction to C

The first few units have been a crash course in the shell and shell programming. Now we move to the **C language**. We now begin developing our C programming skill set by first understanding the basics of the language and then (through examples) studying good code and writing our own. This first unit serves as an introduction to the C language:

Topics:

- Structure of a C program
- Compiling and running a C program
- Input and output with `stdin`, `stdout`
- Random numbers
- Functions

C: General Overview

C can be described as a successful, general-purpose programming language. Like Java and C++, it is widely used for a broad range of applications. C is a *procedural* programming language, not an object-oriented language like Java or C++ and not a functional language like Haskell or Clojure. "Good" C programs

are written clearly, make use of high-level programming practices, and are well-documented with sufficient comments and meaningful variable names—traits of good programming in any high-level language.

C provides a full set of high-level programming features typical of procedural languages: strongly typed variables, constants, structured and enumerated types, mechanisms for defining our own types, aggregate structures, control structures, recursion, and modularization.

C does *not* provide: sets of data, Java's concept of a class or objects, nested functions, subrange types, or the use of subrange types as array indices. It has only recently included a Boolean datatype.

C **does** provide: separate compilation, conditional compilation, bitwise operators, pointer arithmetic, and language-independent input/output.

Why C?

C is the programming language of choice for much systems-level, engineering, and scientific programming. The world's popular operating systems—Linux, Windows, and macOS—are written in C. The internet's core infrastructure, networking protocols, web servers, and email systems are largely written in C. Many graphical interface libraries, numerical/statistical/encryption/compression algorithms, and vast quantities of software for embedded devices (from routers to appliances to game consoles) are written in C. Modern languages such as Objective C, C#, and Swift are also descended from or inspired by C.

C's philosophy: *"Get out of the programmer's way."* It gives a lot of power—and responsibility—to the programmer.

Compilation vs. Interpretation

The shell scripts we wrote previously are *interpreted*: the source code is read and immediately executed by an interpreter. A C program must be *compiled*: the code is first checked for errors, then translated to assembly language, and finally assembled into machine (binary) code.

Building Up a Simple C Program: The Guessing Game

In class, we build a C version of a simple guessing game, similar to the bash script `guess1a.sh`, step by step, improving its structure as we go.

If we have problems with the `mygcc` command on `plank`, make sure we've run:

```
echo source ~/cs50-dev/dotfiles/bashrc.cs50 >> ~/.bashrc
echo source ~/cs50-dev/dotfiles/profile.cs50 >> ~/.profile
```

After this, log out and back in.

Advice: Watch the video demo first, then refer to the code examples. We can use `diff` to compare versions and see changes. Example:

```
$ diff guess[23].c
```

Highlights from Iterative Examples:

- **guess1.c:** Minimal C version of the guessing game. Shows use of #include, `main()`, variable declaration, `printf` (output), `scanf` (input), and constants.
- **guess2.c:** Adds user guidance with `if/else`.
- **guess3.c:** Extracts input logic to a function. Shows function declaration/definition.
- **guess4.c:** Moves the function to after `main()`, requiring a *function prototype*.
- **guess5.c:** Uses command-line arguments (`argc`/`argv`), parses arguments, sets a random answer using `rand()`, shows need to seed the random number generator (`time()`).
- **guess6.c:** Reads user input as a string, checks input validity, leverages `readLine()` helper, covers careful type conversion.

Example: Converting a String to an Integer In modern C (with `<stdbool.h>`), we can write a robust `str2int` as:

```
/* Convert a string to an integer, returning true on success */
bool str2int(const char string[], int* number) {
    char nextchar;
    return (sscanf(string, "%d%c", number, &nextchar) == 1);
}
```

This ensures the input contains only an integer and nothing extra.

Summary of Key Points

- C is a powerful, widely-used procedural programming language.
- C programs must be compiled, not interpreted.
- Good C code emphasizes clarity, structure, and documentation.
- We will develop our skills by iteratively improving a simple guessing game, learning practical C syntax and best practices along the way.
- Understanding how to rigorously handle input/output and validation is essential in C.

3 The C Compilation Process

As we begin C programming, we'll be asked to compile every program using specific command-line flags that tell the C compiler (`gcc`) to be as careful as possible. Specifically, we should compile our programs like this:

```
gcc -Wall -pedantic -std=c11 -ggdb program.c -o program
```

Here's what these flags mean:

- `-Wall` enables all recommended compiler warnings to help point out possible mistakes in our code.
- `-pedantic` asks the compiler to be extra picky about syntax, helping us conform to the language standard.
- `-std=c11` ensures our code is compiled under the C11 standard.
- `-ggdb` includes debugging information needed for debugging tools like `gdb`.

To make things easier, the class environment sets up a Bash `alias`:

```
alias mygcc='gcc -Wall -pedantic -std=c11 -ggdb'
```

which allows us to just type:

```
mygcc program.c -o program
```

or, more typically, for our source file `hello.c`:

```
mygcc hello.c -o hello
```

This command tells the compiler to use our source code file (`hello.c`), and create an executable named `hello` after compilation.

Important Warning About `-o` Order

Whether we use `gcc` or `mygcc`, **be careful to get the order of arguments correct when using the `-o` switch!** The flag `-o` tells the compiler that the file immediately *following* it will be the name of the output executable.

Correct usage:

```
mygcc -o hello hello.c
```

produces the executable `hello` from the source `hello.c`.

Disastrous (wrong) usage:

```
mygcc -o hello.c hello
```

This tells the compiler to overwrite `hello.c` with the output, and tries to use `hello` (our executable!) as the source code. As a result, we may lose our actual C source file! The compiler first erases the file named after the output argument just before compiling.

About `mygcc` and Bash

Remember, `mygcc` is only a Bash alias. It won't work if we try to invoke it from inside some text editors (like emacs) or environments where Bash aliases aren't available—we'll have to type the full `gcc` command with all the flags.

What Actually Happens When We Compile C Code?

When we run a compile command like:

```
mygcc names.c readlinep.c -o names
```

the compiler is actually performing several steps behind the scenes for each source file:

1. **Preprocessing:** Runs the C preprocessor (`cpp`) on `names.c`, resulting in `names.i` (still C code, but with comments removed and all `#include` files expanded).
2. **Compilation:** Compiles `names.i` to assembly code (`names.s`).
3. **Assembly:** Assembles `names.s` into an object file (`names.o`), which contains machine code for our CPU.
4. **Repeat:** Performs the above steps for `readlinep.c`, resulting in `readlinep.o`.
5. **Linking:** The linker (`ld`) combines all object files (`names.o`, `readlinep.o`), along with standard library code (like `stdio.a`), to produce the final executable program (`names`).

The process is the same (even if less visible) when we compile a single source-file program: the compiler goes through these four stages, finally turning our code into an executable that we can run.

Note: The diagram above applies whether we are compiling a single source file or multiple files. Each `.c` file is compiled and assembled into a `.o` file, and all of the `.o` files are linked together at the end.

Summary

- Always use the recommended compiler flags to help catch mistakes early.
- Be careful with the order of arguments when using `-o`, or we might overwrite our own source code!
- The compilation process involves preprocessing, compilation, assembly, and linking—even though we only type a single command.
- `mygcc` is a helpful alias for bash, but not available everywhere.

4 Naming Constants, Variables, and Functions in C

In C, the names of symbols (such as variables, functions, and constants) must begin with a letter (A–Z, a–z) or an underscore (_), followed by any combination of letters, digits (0–9), or underscores. For example, all of these are valid names:

```
counter
MAX_SIZE
_tempVar3
```

Names are **case-sensitive**: `count`, `Count`, and `COUNT` refer to different variables.

Length: Modern compilers (such as gcc) allow names up to 256 characters, but older C compilers may only recognize the first 8 characters. For portability, keeping names reasonably short can be helpful.

Naming Conventions:

- **Constants** are usually written in ALL_CAPS with underscores, e.g., `MAX_SIZE`, `BUFSIZE`.
- **Variables and functions** are generally written in lowercase, sometimes with underscores to separate words, e.g., `total_count`, `read_line()`.
- Compound word names may be written in several styles:
  - **snake_case**: words separated by underscores (`line_length`)
  - **camelCase**: words joined, capitalizing each but the first (`lineLength`)
  - **PascalCase**: words joined, capitalizing every word (`LineLength`)
- **CS50 Coding Style:** For this class, use lower_case_with_underscores for variables/functions, and ALL_CAPS_WITH_UNDERSCORES for constants.

It is critical to pick a naming style and stick with it throughout our project for readability and maintainability. For more details on naming conventions, refer to this Devopedia article.

5 C Header Files

Header files in C (`.h` files) allow us to share declarations between source files so that they agree on function signatures and data structures. They are included at the top of our source files using `#include`.

How to Include Header Files

Standard system header files (like `stdio.h`, `stdlib.h`) use angle brackets:

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

This tells the compiler to look in the system's standard include directories.
User-defined header files (like `readlinep.h`) use double quotes:

```
#include "readlinep.h"
```

This tells the compiler to look in the current directory (or specified include search paths).
We can also specify a path, such as:

```
#include "includes/readlinep.h"
```

Why Use Header Files?

Suppose we have a function `readLinep` used in several source files. We declare its prototype in `readlinep.h`, so both `readlinep.c` (its implementation) and any other file (say, `sorter.c`) that calls it can `#include`

`"readlinep.h"`. This allows the compiler to warn us if declarations and definitions don't match.

Header Protection

To avoid accidental multiple inclusion (which can cause errors), wrap all our header files with a guard:

```
#ifndef __READLINEP_H__
#define __READLINEP_H__
// declarations go here
#endif // __READLINEP_H__
```

This ensures the file's contents are included only once, even if we `#include` it multiple times.

Testing Header Files

We can check whether our header file is self-sufficient by compiling it directly:

```
mygcc -c foo.h
```

If there are missing dependencies or other errors, the compiler will tell us.
When we're done, we can safely delete the resulting `foo.h.gch`.

6 The Math Library (`math.h`)

C provides standard math functions (such as `sqrt()`, `pow()`, `cos()`) in a separate math library. To use these functions, we must:

1. `#include <math.h>` at the top of our source file.
2. Link with the math library using `-lm`:

```
mygcc -o sqrt sqrt.c -lm
```

Example: Square Root

```c
// sqrt.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char *argv[]) {
    for (int i = 1; i < argc; i++) {
        char *endptr;
        double num = strtod(argv[i], &endptr);

        if (*endptr == '\0') {
            printf("sqrt(%f) = %f\n", num, sqrt(num));
        } else {
            printf("%s: invalid number\n", argv[i]);
        }
    }
    return 0;
}
```

`strtod` is used to safely convert command-line arguments to numbers, and `sqrt()` computes their square roots. Be sure to compile with `-lm` to link the math library.

We should study the `man` pages in section 3 for details on math functions, e.g., run `man 3 sqrt` or `man 3 pow`.

4 January 14, 2026

1 C Types

In C, every variable, constant, or function is declared with a specific type. These types can be either C's built-in set or user-defined.

Basic Types

C's most common base types:

Type	Description
`void`	No data; used as a return type or for generic pointers
`char`	Character data (typically 1 byte)
`short`	Short integer (may be less than `int`)
`int`	Standard integer
`long`	Long integer (may be more than `int`)
`bool`	Boolean (true/false; `#include <stdbool.h>`)
`float`	Single precision floating-point
`double`	Double precision floating-point
`long double`	Extended precision float

Type Sizes

Unlike Java, C does not specify exact type sizes; it only guarantees:

$$\texttt{sizeof(char)} \leq \texttt{sizeof(short)} \leq \texttt{sizeof(int)} \leq \texttt{sizeof(long)}$$

We can use `sizeof(type)` to determine a type's size (in bytes) on our machine.

Example:

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

int main() {
    char c = 'a';
    int i = 2;
    short s = 3;
    long l = 4;
    bool b = true;
    float f = 1000.25;
    double d = 1.1E24;
    long double ld = 1.0E31;

    printf("char: %c \t sizeof %zu bytes\n",  c, sizeof(c));
    printf("int: %d \t sizeof %zu bytes\n",   i, sizeof(i));
    printf("short: %d \t sizeof %zu bytes\n", s, sizeof(s));
    printf("long: %ld \t sizeof %zu bytes\n",  l, sizeof(l));
    printf("bool: %d \t sizeof %zu bytes\n",  b, sizeof(b));
    printf("float: %f \t sizeof %zu bytes\n", f, sizeof(f));
    printf("double: %e \t sizeof %zu bytes\n", d, sizeof(d));
    printf("long double: %Le \t sizeof %zu bytes\n", ld, sizeof(ld));
    return 0;
}
```

void Type

`void` is used for:

- Indicating a function returns no value (`void foo();`).
- Declaring a pointer without specifying type (`void *ptr;`).

We cannot declare variables of type `void`.

Type Coercion and Casting

C automatically converts between compatible types in assignments or expressions, but explicit casts can be used (and are sometimes required).

```
int a = 4;
float b = 0.0;

b = a;          // automatic: int to float
a = b;          // automatic: float to int (may lose data)
a = (int) b;    // explicit cast
```

Unsigned Types

Adding `unsigned` before a type makes it non-negative and doubles the maximum value for that size (e.g., `unsigned int`, `unsigned short`).

Constant Variables

Prefix with `const` to make a variable read-only:

```
const float pi = 3.14159;
const int maxNameLength = 50;
```

We can use `const` for function parameters to indicate they won't be modified.

User-defined Types (`typedef`)

`typedef` creates an alias for an existing type; improves code clarity.

```
typedef short boolean;     // 'boolean' is now another name for short
const boolean TRUE = 1;
const boolean FALSE = 0;
```

Used for readability or to abstract implementation details.

Storage Class Modifiers: extern and static

- `extern`: Declares a variable/function defined elsewhere (often another source file). Used in headers.

```
extern bool readLine(char* buf, const int len);
```

- `static`:
  - For global variables or functions: limits visibility to the current file (like "private" in Java).
  - For local variables in functions: preserves variable value across function calls.

2 C - Global and Local Scopes

Every variable and function in C has a **scope**: the region of source code in which its name is visible and can be used. C's main types of scope are **global** and **local**, though every block inside `{}` defines a new nested scope.

- **Global scope**: Names (usually functions, constants, and types in good style) defined outside any function are visible from their point of definition to the end of the file.
- **Local scope**: Names defined inside a function or block are visible from their point of definition to the end of that block only.
- **Loop/block scope**: Variables declared at the top of a block (for example, in a `for`-loop) are only usable inside that block.

Initialization

- **Global variables**: Automatically initialized to zero.
- **Local variables**: *Not* automatically initialized—we must always initialize them ourselves!

Examples of Scope

```c
// Global function prototypes (visible anywhere after this point)
int square(int n);
void printSquares(int count);

// Global constant
const int globalLimit = 10;

int main(void) {
    int localSum = 0; // Local to main

    for (int i = 1; i <= globalLimit; i++) { // i: loop variable, only in loop
        int squared = square(i);  // squared: local to the block of for loop
        localSum += squared;
        printSquares(squared);    // function call
    }

    return 0;
}

int square(int n) { // Global function, visible to all
    return n*n;
}

void printSquares(int count) {
    // count: local parameter
}
```

static vs extern

- `static` at file scope: restricts access to within that file.
- `extern`: declaration says "this is defined elsewhere" (used to share across files).

Shadowing

A local variable can *shadow* a global one:

```c
int x = 3; // global

void foo() {
    int x = 5; // shadows global x in this function
}
```

3 Command-Line Arguments in C

C programs can accept command-line arguments. When the OS executes a C program, it usually calls:

```
int main(int argc, char* argv[])
```

- `argc` is the argument **count**, including the program name.
- `argv` is an **array of strings** (`char*`), each one holding an argument from the command line.
- `argv[0]` is the program name; `argv[1]`...`argv[argc-1]` are the actual arguments.

A typical example:

```
// arguments.c - prints out command-line arguments
#include <stdio.h>

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

Sample run:

```
$ ./args a b c "d e"
5 arguments:
0: ./args
1: a
2: b
3: c
4: d e
```

Parsing Command-Line Flags (Switches)

It's common to handle flags (starting with `-`) and filenames:

```c
// nosort.c - demonstrates command-line switches
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
    int reverse = 0, unique = 0, numeric = 0;

    while (argc > 1 && argv[1][0] == '-') {
        if (strcmp(argv[1], "-r") == 0) reverse = 1;
        else if (strcmp(argv[1], "-u") == 0) unique = 1;
        else if (strcmp(argv[1], "-n") == 0) numeric = 1;
        else {
            printf("Error: bad option '%s'\n", argv[1]);
            printf("Usage: %s [-r] [-u] [-n] filename...\n", argv[0]);
            return 1;
        }
        argc--; argv++;
    }
    if (reverse) printf("reverse sort\n");
    if (unique) printf("unique sort\n");
    if (numeric) printf("numeric sort\n");
    for (int i = 1; i < argc; i++)
        printf("next argument '%s'\n", argv[i]);
    return 0;
}
```

Only simple one-letter switches are accepted, and flags must each be given separately like `-r -u -n`, not combined (`-run`).

4 Files: Opening, Reading, and Writing

Unlike languages like Java and Python, C handles file input/output (I/O) through its standard library, not as part of the language itself. We need to include `<stdio.h>` for file operations.

A file in C is accessed through a **file pointer** of type `FILE*`. Files must be explicitly opened and closed. Use `fopen()` to open a file and `fclose()` when we're done.

```c
// Example: opening and closing a file
#include <stdio.h>

int main(void) {
    FILE* fp = fopen("hello.txt", "w");  // open for writing
    if (fp == NULL) {
        fprintf(stderr, "Could not open file!\n");
        return 1;
    }
    fprintf(fp, "Hello, COSC 50!\n");   // write to the file
    fclose(fp);                         // close file
    return 0;
}
```

Check that `fopen()` succeeds by testing `fp` $\neq$ `NULL`. Always `fclose()` files when done.

Reading from a file:

```c
FILE* fp = fopen("data.txt", "r");
if (fp == NULL) { /* error */ }

char word[100];
while (fscanf(fp, "%s", word) == 1) {
    printf("Word: %s\n", word);
}
fclose(fp);
```

To read a file character-by-character:

```c
int c;
while ((c = fgetc(fp)) != EOF) {
    putchar(c);
}
```

Standard I/O streams: C programs have three standard streams already open:

- `stdin` — standard input (keyboard)
- `stdout` — standard output (screen)
- `stderr` — error output

`printf( ... );` is equivalent to `fprintf(stdout, ... );`.

Format strings: C I/O functions use format strings (like `"%d"`, `"%s"`, etc.) to control the output/input. See `man 3 printf` for details.

Always check for errors: File functions often return `NULL` or special values on failure; we should always check their return values.

For more info: `man fopen`, `man fclose`, `man 3 printf`.

5 C – Strings

C does not have a true `string` type; instead, strings are represented as arrays of characters ending with a null character (`'\0'`).

Defining a string constant:

```
char* CS = "Computer Science"; // points to first char of constant string
```

String in memory:

C	o	m	p	u	t	e	r		S	c	i	e	n	c	e	\0

The last byte is always `'\0'`.

Two ways to declare string variables:

1. `char dept[30];` // array with room for 29 chars + null
2. `char* department = NULL;` // pointer (good practice to initialize)

Copying strings:

```
strcpy(dept, "Computer Science"); // copies string into dept (needs <string.h>)
department = CS;        // copies pointer (not the string itself)
```

Note on syntax:

```
// The following are equivalent
char   firstName[];
char*  firstName;
char * firstName;
char*  firstName;
```

When used as function parameters, these all mean "pointer to char" (string).

NULL vs \0: `NULL` is a null pointer (address 0), `'\0'` is the null character (value 0).

String Library: For string operations, include:

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

See `man string` for standard functions such as `strcpy`, `strcmp`, `strlen`, etc.

6 Memory, pointers, and malloc

To truly understand C, especially pointers, we must understand how C manages memory. Every value in our program—variables, data, code—lives somewhere in the computer's memory. Each byte of memory has a unique address.

Memory addresses and the & (address-of) and * (dereference) operators:

```
int x = 42;        // x is an integer
int* xp = &x;      // xp is a pointer to x
int y = *xp;       // y gets the value pointed to by xp (42)
```

`xp` stores the address of `x`. The expression `*xp` gives the value at the address stored in `xp`. This is called *dereferencing*.

The NULL pointer: `NULL` simply means address 0. Attempting to dereference a NULL pointer (e.g., `*p` where `p = NULL`) will cause a segmentation fault—a crash.

C program memory regions:

- **Code (text) segment:** Our compiled instructions (functions).
- **Global (static) segment:** Global variables and constants defined outside functions.
- **Stack:** Holds local variables and tracks function calls. Each function call "pushes" a stack frame; returning "pops" it.
- **Heap:** Dynamically allocated memory (using `malloc` and `free`).

Global variables have global scope (visible to all functions below their definition) and *static storage* (allocated before main; address never changes).

```
const float pi = 3.14159;
int count = 0;
```

Stack variables are created when a function is called and destroyed when it returns.

The heap is for dynamic memory allocation:

```
int* arr = malloc(10 * sizeof(int)); // allocates space for 10 ints
// ... use arr ...
free(arr); // release memory when done
```

We must always `free()` any memory we allocate.

Segmentation faults: This is a common crash if we read/write memory we shouldn't. For example:

```
int* x = NULL;
*x = 42; // Crashes: dereferencing NULL
```

We can use `gdb` (the GNU Debugger) to inspect crashes and variables at the moment of failure.

7 Dynamic memory allocation

C lets us dynamically get memory on the heap using `malloc()` and later release it with `free()`. When we call `malloc(size)`, it gives us a pointer to `size` bytes of uninitialized memory; it's up to us to assign it to the correct pointer type and initialize as needed.

Example:

```
int* arr = malloc(10 * sizeof(int)); // space for 10 integers
// ... use arr ...
free(arr); // always free heap memory when done
```

C also provides:

- `calloc(count, size)` – like malloc, but zeroes out the memory.
- `realloc(p, newsize)` – resize a previously-allocated block with possible move.

If we forget to free memory, we cause a **memory leak**. If we free something twice or free memory not created by malloc/calloc/realloc, we may cause a crash.

Tips:

- Write the matching `free()` right after `malloc()` (or document whose job it is).
- Set pointers to `NULL` after we free them to avoid using dangling pointers.

For dynamic data structures, like linked lists or arrays of strings, we must explicitly allocate and free every heap block we use.

Valgrind (covered later) is a tool that can help us catch leaks and misuse of heap memory.

8 When do we need to `malloc` a string?

We only need to use `malloc` for a string (or any array) if:

- We do **not** know the size at compile-time and need to determine it at run-time, **or**
- We need the array (pointer) to last beyond the function where it's created (e.g., return it from a function or store it for later use).

Otherwise, declare a local array: `char s[N]` for fixed-size strings limited to the function or block.

Examples:

- `char str[20];`
- `char *str = malloc(len+1);`

Scope: Local arrays disappear when their scope ends. Do *not* return a pointer to a local array.

Pattern for any type:

```
TYPE* arr = malloc(NUM * sizeof(TYPE));    // for dynamic arrays
```

Remember to `free()` anything we `malloc()`!