History of Linux and the command line

Zubin

February 1, 2026

Contents

1 Compiler vs interpreter

In the C programming language, we use a compiler. To understand the difference between an interpreter and a compiler, we can use an analogy of landing on a planet where inhabitants speak a strange language called "Gobbledygook". To get a mechanic to repair your spaceship, you need a translator.

1.1 The Interpreter

If you choose an interpreter:

- **Process:** The interpreter reads your first instruction, translates it immediately, and the mechanic executes it. Then it reads the second, translates, and executes, and so on.
- **Characteristics:** The interpreter stays with you, translating line by line.
- **Pros:** It allows you to correct mistakes as you go (interactive).
- **Cons:** It is a slow process because the mechanic waits for translation between steps.
- **Etymology:** "Inter" means between. The interpreter is always between your program and the computer.

1.2 The Compiler

If you choose a compiler:

- **Process:** The compiler takes your complete list of instructions and translates the whole lot at once. It then hands the translated list back to you and leaves.
- **Characteristics:** You hand the complete list to the mechanic, who executes them all in one go very quickly.
- **Pros:** Execution is very fast and efficient.
- **Cons:** Takes extra preparation time initially. If there is a mistake, it is too late to fix it during execution.
- **Etymology:** "Compile" means to pile together. It piles together your entire program and translates it all at once.

1.3 Summary

- **Interpreter:** Runs slowly, starts right away, allows you to see how things are going.
- **Compiler:** Takes preparation time, but runs very quickly and efficiently.

2 Memory representation: RAM, cells, word, byte, bit, memory address

How does the computer remember where it has stored the value for a certain variable? To answer this, we need to understand computer memory.

2.1 Types of Memory

One role of the operating system is to manage the computer's memory.

- **RAM (Random Access Memory):** Temporary, volatile memory used to execute programs. It is quick to access.
- **Non-volatile Memory:** Permanent storage, such as the hard drive, used for storing files.

2.2 Structure of RAM

Most programs use RAM during execution.

- **Bit:** A single binary memory cell (0 or 1).
- **Word:** A group of bits forming the fundamental unit of data moved between RAM and the processor.
- **Word Size:** The number of bits in a word (e.g., 8, 16, 32, 64 bits).
- **Byte:** A unit consisting of 8 bits.

2.3 Memory Addressing

To locate data, memory cells are grouped into words, and each word is assigned an address.

- **Memory Address:** A whole number describing the location of a word in memory (similar to house addresses on a street).
- **Example:** If a computer has 8-bit words:
  - Address 0 points to the first word (first 8 bits).
  - Address 1 points to the second word (next 8 bits).

2.4 Memory and C Programming

In C, it is possible to access these memory addresses directly.

- **Access:** You can obtain the address where a variable's value is stored.
- **Optimization:** This allows for low-level memory management and optimization of execution speed.

---

3 Manage the memory with the command line: free, top, htop

To manage memory, we first need to see how much memory is used by the programs running on the Linux system.

3.1 The free Command

The `free` command displays the amount of free and used system memory.

- **Options:** Use `-b` (bytes), `-k` (kilobytes), `-m` (megabytes), or `-g` (gigabytes) to set the unit.
- **Example:** `free -m` displays the table in megabytes.
- **Output:** The `Mem` line shows the total, used, and free memory.

3.2 The top Command

The `top` command shows memory usage per process.

- **Statistics:** The header shows total used and free memory.
- **VSZ (Virtual Size):** Represents the virtual memory for each program (sometimes labeled **VIRT**).
- **Sorting:** Press the **M** key to sort processes by memory usage.
- **Visualization:** Pressing the **S** key (in this specific version) switches to a view showing only memory usage.

3.3 The htop Command

`htop` provides a visual representation of system resources.

- **Visuals:** Bars indicate used and free memory.
- **Sorting:**
  1. Press **F6** to select "Sort by".
  2. Select **M_SIZE** (Memory Size) and press **Enter**.

3. This sorts the list by the **VIRT** column.

Monitoring memory helps identify programs putting too much pressure on the system, which can then be killed if necessary.

---

4 Memory consumption of a program using htop, virtual memory

We can analyze how much memory a C program uses by running it and monitoring it with `htop`. To do this effectively, the program must pause (e.g., waiting for user input via `scanf`) so it stays active in memory.

4.1 Memory Usage by Data Type

By declaring arrays of different types with one million elements, we can observe how memory consumption changes in the `VIRT` (Virtual Memory) column of `htop`:

- **Basic Program:** A simple program might use $\approx$ 952 KB.
- **Char Array (1,000,000 chars):** Usage increases to $\approx$ 1,800 KB.
- **Int Array (1,000,000 ints):** Usage increases significantly to $\approx$ 4,728 KB (approx. 4 MB).
- **Double Array (1,000,000 doubles):** Usage doubles compared to integers, reaching $\approx$ 8,632 KB (approx. 8 MB).

4.2 Virtual vs. Physical Memory

An important observation is that even when a program reserves a large amount of memory (high `VIRT`), the system's physical memory usage (the bar at the top of `htop`) may not increase correspondingly.

- **Virtual Memory:** The amount of memory the program has asked for or reserved.
- **Physical Memory:** The actual RAM hardware used.
- **Lazy Allocation:** Linux is smart; if a program reserves memory (like a large array) but never writes to or reads from it, the system does not immediately allocate physical RAM. Physical memory is consumed only when the virtual memory is actually used.

4.3 Memory Management Tips

- **Optimization:** Ensure that large arrays or variables are used efficiently.

- **Dynamic Allocation:** If you allocate memory dynamically in C, always remember to **free** it to return it to the system for other programs.

---

5 Interactive programs in C using scanf, fflush

This section explains how to create interactive C programs and handle a common issue related to terminal output buffering.

5.1 Basic Interactive Program with scanf

An interactive program prompts the user for input and reads it from the terminal.

- **Prompting:** Use `printf` to display a question to the user.
- **Reading Input:** Use `scanf` to wait for and read the user's input.
- **Example:** A program can ask for a family name, first name, and age.
  - `scanf("%s", familyName)` reads a string into a character array.
  - `scanf("%d", &age)` reads an integer into an integer variable.
- Typically, each prompt with `printf` ends with a newline character (`\n`) to move the cursor to the next line for the user's input.

5.2 The Problem of Buffered Output

The `printf` function in C uses a buffer, which is a temporary memory area.

- **Buffering:** Output is not always sent directly to the screen. It is first stored in a buffer. The buffer is "flushed" (its contents are sent to the terminal) when certain conditions are met.
- **Newline Trigger:** A common trigger for flushing the buffer is encountering a newline character (`\n`).
- **The Issue:** If you write a `printf` statement for a prompt without a trailing `\n`, the text may not appear on the screen immediately. The program might wait for the user's input with `scanf` before the prompt is even visible. All the buffered prompts might appear in a single line only after the program finishes or the buffer is flushed for another reason.

5.3 The Solution: Flushing the Buffer with fflush

To ensure prompts appear immediately without a newline, you can manually flush the output buffer.

- **The Command:** The `fflush(stdout)` function forces the standard output buffer (`stdout`) to be written to the screen.
- **Implementation:** Place `fflush(stdout);` immediately after each `printf` statement that you want to appear on the screen right away.
- **Example:**

```
printf("What is your first name?  ");
fflush(stdout);
scanf("%s", firstName);
```

- This ensures the question is visible before the program waits for the `scanf` input, even without a `\n`. The user's answer will then appear on the same line as the question.

5.4 Important Considerations

- **System Variability:** This buffering behavior is not consistent across all Linux systems. Some systems or C libraries might flush output more frequently, even without a newline.
- **Best Practice:** To write portable and predictable interactive programs, always ensure the output buffer is flushed. The two main ways are:
  1. End your prompt strings with `\n`.
  2. Call `fflush(stdout);` after the prompt.

---

6 Use scanf and file redirection to simulate an input

We can automate or simulate user input for interactive programs using pipes and file redirection. This is useful for testing or when running programs in scripts.

6.1 Creating an Input File

First, create a file containing the inputs expected by the program, separated by newlines.

- **Example:** Create a file named `answers` with the following content:

```
sharrock
remi
18
```

- You can create this using `nano` or `cat > answers`.

6.2 Method 1: Using Pipes

You can pipe the output of a command (like `cat`) into your program.

- **Command:** `cat answers | ./program`
- **Explanation:** The shell takes the output of `cat answers` and feeds it as standard input to `./program`. The `scanf` functions in the program read from this stream just as if it were typed on the keyboard.

6.3 Method 2: Input Redirection

You can redirect a file directly into the program's standard input using the less-than sign (<).

- **Command:** `./program < answers`
- **Explanation:** This tells the shell to open the file `answers` and connect it to the input of `./program`.
- **Note:** This is the reverse of output redirection (>), which sends output to a file. Here, we send a file to the input.

---

7 Don't use scanf, use fgets getline or readline

Using `scanf` for string input can be dangerous and lead to unexpected behavior for both programmers and users.

7.1 The Problem with scanf

`scanf` splits input based on whitespace (spaces, tabs, newlines).

- **Scenario:** A program asks for Family Name, First Name, and Age sequentially.
- **Input:** If the user types "Sharrock Remi 18" at the first prompt and hits Enter.
- **Result:**
  - Family Name becomes "Sharrock".
  - First Name automatically becomes "Remi" (consuming the next part of the buffer).
  - Age automatically becomes "18".
- The program skips waiting for the subsequent inputs because `scanf` consumed the space-separated values from the first line. This behavior makes `scanf` risky for robust user interaction.

7.2 Better Alternatives

To handle string input more safely, consider these functions:

- **fgets:** Reads the entire line, including spaces. Note that it also captures the newline character (`\n`) at the end, which you may need to remove.
- **getline:** A POSIX-standard function (available on most Unix-like systems) that reads an entire line.
- **GNU readline:** A powerful function specific to Linux/GNU systems.

For robust C programming, it is recommended to research these functions or take a dedicated C programming course.

---

8 Use the math library

8.1 Using the pow function

We can use the math library to perform calculations, such as finding the square of a number, instead of writing our own function. The `pow` function calculates a number raised to the power of another.

- **Prototype:** Also known as the declaration, header, or signature. It defines the return type, function name, and parameter types.
- **Example:** `double pow(double x, double y);`
- **Usage:** `pow(a, 2)` calculates a^2.

To use this function, we need to link our program to the binary file containing its definition.

8.2 Locating the Library Binary

Compiled library files are typically stored in `/usr/lib`.

- The math library is often named `libm.a` (archive) or `libm.so` (shared object).
- **Linking with Absolute Path:** You can compile by specifying the full path to the library:

```
gcc program.c /usr/lib/libm.a -o program
```

8.3 Header Files

Header files, which contain function prototypes, are located in `/usr/include`.

- The header for the math library is `math.h`.

- **Including the Header:** Instead of typing the prototype manually, use the `#include` directive at the top of your C file. This copies the content of the header file into your program.

  ```
  #include <math.h>
  ```

8.4 Linking with the -l Option

A more common and convenient way to link libraries is using the `-l` flag.

- **Naming Convention:** To derive the flag, take the library name (e.g., `libm.a`), remove the `lib` prefix and the file extension.
- **Example:** `libm.a` becomes `m`.
- **Command:**

  ```
  gcc program.c -lm -o program
  ```

---

9 Use multiple libraries in C

We will now look at a program that uses several external libraries. This requires including header files (definitions and prototypes) and linking object files (compiled code).

9.1 Locating Headers and Libraries

To find the necessary files on the system:

- **Header Files:** Located in `/usr/include`. We can list them to a file: `ls /usr/include > headers.txt`.
  - `curses.h`: Allows cursor manipulation and colors.
  - `menu.h`: Allows creation of menus.
- **Object Files:** Located in `/usr/lib`. We can list them: `ls /usr/lib > libraries.txt`.
  - `libcurses.so`: The compiled code for curses.
  - `libmenu.so`: The compiled code for menus.

9.2 Including Headers

In the C program, we must include the headers for the functions we are using:

```
#include <curses.h>
#include <menu.h>
#include <stdlib.h>
```

Note: `stdlib.h` is included because the program uses `calloc` for memory allocation.

9.3 Linking Libraries

To include the actual code for the functions, we must link the libraries during the build process.

9.3.1 Method 1: Absolute Path

Add the exact path to the library on the build line:

```
gcc program.c /usr/lib/libcurses.so /usr/lib/libmenu.so -o program
```

The standard library (`stdlib`) is automatically linked, so it does not need to be added here.

9.3.2 Method 2: Using the -l Flag

A more convenient way is to use the `-l` flag followed by the library name (without "lib" or extension):

```
gcc program.c -lcurses -lmenu -o program
```

---

10 GCC details

In this section, we explain the details of the build command and the steps performed by the compiler (GCC) to turn C source code into an executable program.

10.1 The Build Command and Flags

When you press the "Run It" button in WebLinux, or build manually, a command like this is executed:

```
gcc -std=C11 -Wall -fmax-errors=10 -Wextra -o program program.c
```

Here is a breakdown of the flags used:

- **-std=C11**: Specifies the C standard to use. C11 is a modern standard.

- **-Wall**: Displays all warnings. It is important to look at these and fix them, as they often indicate errors.
- **-fmax-errors=10**: Tells the compiler to stop after 10 errors so the user is not overwhelmed.
- **-Wextra**: Enables extra warnings that are not included in `-Wall`.
- **-o program**: Specifies the name of the output file (the executable). If this flag is omitted, GCC creates an executable named `a.out` by default.

10.2 Compilation Stages

When you run GCC, it performs three main steps: preprocessing, compilation, and linking. We can invoke these steps separately.

10.2.1 1. The Preprocessor

The preprocessor performs simple textual replacement. For example, it finds the contents of header files (like `stdio.h`) and copies them verbatim into the source code.

10.2.2 2. Compilation (Translation)

This step translates the source code into machine language, creating a so-called **object file**.

- To perform only this step, use the `-c` flag.
- **Command:** `gcc -c program.c -o program.o`
- This creates `program.o`. It is the binary, computer-readable version of the code, but it cannot be executed directly (it lacks system libraries).

10.2.3 3. The Linker

The linker bundles the object file with necessary system libraries to produce the final executable.

- **Command:** `gcc program.o -o program`
- This takes the object file `program.o` and creates the executable `program`, which can be run with `./program`.

Understanding these steps is essential for separate compilation, where source code is split across multiple files.

---

11 Object files

In this section, we demonstrate **separate compilation**, which means spreading source code over several files and assembling them into one final executable. This is essential for organizing large programs by topic.

11.1 Creating Separate Files

We separate the `average_temperature` function from the main program into its own file.

- **weatherstats.c**: Contains the source code for the `average_temperature` function.
- **weatherstats.h**: A header file containing the function prototype.
- **program.c**: Contains the `main` function.

To use the function in `program.c`, we must include the header.

```
#include "weatherstats.h"
```

Note: We use quotation marks (`""`) instead of angle brackets (`<>`). Angle brackets tell the compiler to look in standard system paths (like `/usr/include`), whereas quotation marks tell it to look in the current directory.

11.2 Compiling Object Files

We compile each source file into an object file (`.o`) individually.

11.2.1 Compiling the Module

```
gcc -c weatherstats.c -o weatherstats.o
```

We must use the `-c` flag. If omitted, GCC tries to build an executable and fails because `weatherstats.c` has no `main` function.

11.2.2 Compiling the Main Program

```
gcc -Wall -c program.c -o program.o
```

Using `-Wall` ensures we see warnings, such as "implicitly declared function" if we forget to include the header file.

11.3 Linking

Finally, we invoke the linker to bundle the object files into an executable.

```
gcc -o program program.o weatherstats.o
```

This creates the final executable `program` by combining the machine language from both object files.

---

12 Modify object files

To reinforce what we have learned so far about spreading source code over multiple files, compiling, and linking, we will walk through the steps required to modify this code. You will see that these steps can become tedious, which motivates the use of **Makefiles** (covered in the next topic) to automate the process.

Suppose we want to add a maximum temperature feature to our `weatherstats` source file and call it from the main program.

12.1 Modifying the Source Code

12.1.1 Updating the Main Program

In `program.c`, we add a call to the new function and print the result.

```
double max = maxTemp(temperatures, 7);
printf("Highest temp:  %.2lf\n", max);
```

12.1.2 Updating the Module

We add the `maxTemp` function to `weatherstats.c`.

```
double maxTemp(double *temperatures, int size) {
double max = temperatures[0];
int i;
for (i = 1; i < size; i++) {
if (max < temperatures[i]) {
max = temperatures[i];
}
}
return max;
}
```

12.1.3 Updating the Header

We must add the function prototype to `weatherstats.h` so the compiler knows about it when compiling `program.c`.

```
double maxTemp(double *temperatures, int size);
```

12.2 Compiling and Linking

Now we must translate the code into machine language.

1. **Compile program.c:**

   ```
   gcc -Wall -c program.c -o program.o
   ```

2. **Compile weatherstats.c:**

   ```
   gcc -Wall -c weatherstats.c -o weatherstats.o
   ```

3. **Link them together:**

   ```
   gcc -o program program.o weatherstats.o
   ```

Running `./program` will now display both the average and the maximum temperature.

12.3 Partial Recompilation

If we modify only one file, we do not need to recompile everything. For example, if we change a temperature value (e.g., changing 5.3 to 15.3) in `program.c`:

1. **Recompile only the changed file:**

   ```
   gcc -Wall -c program.c -o program.o
   ```

   We do not need to recompile `weatherstats.c` since it hasn't changed.

2. **Relink:**

   ```
   gcc -o program program.o weatherstats.o
   ```

When we run the program now, we get the updated result. As you can see, making modifications to source code spread out over multiple files can be rather tedious, which is why we will look at Makefiles next.

---

13 Make file

We will now see how to automate all of the necessary steps to build the final executable program from source code spread out over multiple files. We will use the `make` program, which reads instructions from a file named `Makefile`.

13.1 The Makefile Structure

To use `make`, we create a file named `Makefile`. The structure of the rules in this file is specific:

```
target:  dependencies
      command
```

Note: The indentation before the command must be a **tab** character, not spaces.

- **Target:** What is to be produced.
- **Dependencies:** What is needed to produce the target.
- **Command:** How to produce the target.

13.2 Constructing the Makefile

We define the targets for our weather statistics program.

13.2.1 The Executable

The final target is the executable `program`. It depends on the object files.

```
program:  program.o weatherstats.o
      gcc -std=C11 -Wall -fmax-errors=10 -Wextra -o program program.o
weatherstats.o
```

13.2.2 The Object Files

We also need to tell `make` how to build the object files.

- **program.o:** Depends on `program.c` and `weatherstats.h`.

  ```
  program.o:  program.c weatherstats.h
        gcc -std=C11 -Wall -fmax-errors=10 -Wextra -c program.c -o
  program.o
  ```

- **weatherstats.o:** Depends on `weatherstats.c`.

```
weatherstats.o:  weatherstats.c
      gcc -std=C11 -Wall -fmax-errors=10 -Wextra -c weatherstats.c
-o weatherstats.o
```

13.3 Running Make

To build the program, simply type `make` in the terminal.

```
$ make
```

This will execute the necessary commands to build the targets.

13.4 Efficiency

The `make` program is smart and only recompiles what is necessary.

- If you modify `program.c` but not `weatherstats.c`, running `make` will only recompile `program.o` and then link the program. It will not recompile `weatherstats.o`.
- If you run `make` when everything is up to date, it will do nothing.

This automation simplifies the build process significantly.

---

14 More elaborate Makefile

I want to quickly show you an alternative `Makefile`, one that is more versatile towards making changes and can be more broadly used for other purposes. This is an optional extra topic, but if you look on the internet for Makefiles, you will most likely see something that is more like the one shown here.

14.1 Variables and Comments

In this Makefile, we use variables to define settings at the top of the file. This makes it easy to change compilers, flags, or file names without modifying the logic below.

- **Comments:** Anything starting with a pound sign (`#`) is a comment.
- **CC:** Defines the compiler (e.g., `gcc`).
- **CFLAGS:** Defines the compiler flags.
- **OBJ:** Lists the object files needed for the project.
- **MAIN:** Defines the name of the final executable.

14.2 Generic Rules

The targets use these variables.

- **all:** The default target, which depends on the executable variable `$(MAIN)`.
- **$(MAIN):** Depends on the object files `$(OBJ)`. It uses the variables to construct the link command.
- **%.o: %.c:** A generic rule to compile any `.c` file into a `.o` file.
  - `$<`: Refers to the source file (the dependency).
  - `$@`: Refers to the output file (the target).

14.3 The Clean Target

A `clean` target is added to remove object files and the executable, allowing for a fresh build.

14.4 Example Code

Here is the complete elaborate `Makefile`:

```
# Define the compiler
CC = gcc
# Define compiler flags
CFLAGS = -std=c11 -Wall -fmax-errors=10 -Wextra

# Define object files
OBJ = program.o weatherstats.o
# Define the executable name
MAIN = program

all:  $(MAIN)

$(MAIN): $(OBJ)
	$(CC) $(CFLAGS) -o $(MAIN) $(OBJ)

# Generic rule for compiling .c to .o
%.o:  %.c
	$(CC) $(CFLAGS) -c $< -o $@

clean:
	rm *.o $(MAIN)
```

Running `make` will execute the steps defined by these generic rules. Running `make clean` will remove the generated files.

15 Create your library

We are finally ready to create our own libraries. Libraries typically consist of a number of object files bundled together into an archive.

15.1 Static Libraries

A static library is an archive of object files, typically with the extension `.a` on Linux. The linker extracts only the necessary functions from the archive and copies them into the executable.

15.1.1 Creating a Static Library

1. **Compile the module:** Compile the source files into object files.

```
gcc -c weatherstats.c -o weatherstats.o
```

2. **Create the archive:** Use the `ar` command to bundle the object files.

```
ar rcs libweather.a weatherstats.o
```

   - **r (replace):** Replaces existing files in the archive.
   - **c (create):** Creates the archive if it does not exist.
   - **s (index):** Creates an index for faster access.
   - **Naming Convention:** Libraries typically start with `lib` and end with `.a` (e.g., `libweather.a`).

15.1.2 Linking a Static Library

There are two ways to link the library to your main program.

Method 1: Direct File Path

```
gcc -o program program.o libweather.a
```

Method 2: Using -l and -L flags

```
gcc -o program program.o -L. -lweather
```

- **-lweather:** Links the library named `weather`. GCC automatically expands this to `libweather.a`.

- **-L.:** Adds the current directory (`.`) to the library search path. Without this, GCC only looks in standard system directories.

15.2 Dynamic (Shared) Libraries

Dynamic libraries (Shared Objects) have the extension `.so` on Linux (or `.dll` on Windows).

15.2.1 Static vs. Dynamic Libraries difference Table

Static Versus Dynamic Libraries in C	
Static	**Dynamic**
Linker finds all used library functions (such as `printf()`, `sqrt()`, etc.) and copies them into your executable file.	Linked dynamically at run-time by the OS; every program that accesses the library uses the same copy.
Libraries have the extension `.a` (Linux) or `.lib` (Windows).	Libraries have the extension `.so` (Linux) or `.dll` (Windows).
Executable is a larger file, needing more disk space and main memory.	Executable only contains the name of (a link to) the library.
If the library changes, the executable does not automatically update and needs to be re-linked.	If the library changes, the executable will automatically use the new library code.
If the library becomes incompatible with your code, your executable will still run as long as you do not re-link.	If the library becomes incompatible with your code, your executable will no longer run.
Library access is faster.	Dynamic querying of symbols takes time.

Table 1: Comparison of Static and Dynamic Libraries in C

15.2.2 Static vs. Dynamic Libraries

- **Static (.a):**
  - **Linking:** Code is copied into the executable.
  - **Size:** Executable is larger.
  - **Memory:** Uses more memory if multiple programs use the same library (each has its own copy).
  - **Updates:** Executable must be recompiled to use updated library code.
  - **Speed:** Faster execution.

- **Dynamic (.so):**
  - **Linking:** Code is linked at runtime. The executable contains only references.
  - **Size:** Executable is smaller.
  - **Memory:** Efficient; one copy of the library in memory is shared by multiple programs.
  - **Updates:** Executable automatically uses the updated library (unless incompatible).
  - **Speed:** Slightly slower due to dynamic symbol lookup.

15.2.3 Creating a Dynamic Library

1. **Compile with PIC:** Use the `-fpic` (Position Independent Code) flag.

   ```
   gcc -c -fpic weatherstats.c -o weatherstats.o
   ```

2. **Create the Shared Object:** Use `gcc` with the `-shared` flag.

   ```
   gcc -shared -o libweather.so weatherstats.o
   ```

15.2.4 Linking and Running

Linking uses the same command as static libraries:

```
gcc -o program program.o -L. -lweather
```

However, to run the program, the operating system must be able to find the shared library. You may need to update the `LD_LIBRARY_PATH` environment variable:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PWD
./program
```

---

16 Modify your library

In this section, we will look at how to make modifications to a static library and to a program that uses it. We assume we have `program.c` and `weatherstats.c`, but we haven't created the library `libweather.a` yet.

Static Versus Dynamic Libraries in C (Linux OS)		
Library	**Static**	**Dynamic**

Compile library files	`gcc -c part1.c -o part1.o`
	`gcc -c part2.c -o part2.o`
	`...`
		`gcc -c -fpic part1.c -o part1.o`
		`gcc -c -fpic part2.c -o part2.o`
		`...`

Create library	`ar rcs libmylib.a part1.o part2.o ...`	`gcc -shared -o libmylib.so part1.o part2.o ...`
Compile main program	`gcc -c program.c -o program.o`	`gcc -c program.c -o program.o`

Link library to main program	`gcc -o program program.o -L. -lmylib`	`gcc -o program program.o -L. -lmylib`
		Add library path to environment variable:
		`export: LD_LIBRARY_PATH=$PWD :$LD_LIBRARY_PATH`

Run program	`./program`	`./program`

Table 2: Steps to create and use static and dynamic libraries in C on Linux

16.1 Creating the Initial Library

First, we compile the existing module `weatherstats.c` into an object file.

```
gcc -c weatherstats.c -o weatherstats.o
```

Then, we create the static library archive `libweather.a`.

```
ar rcs libweather.a weatherstats.o
```

16.2 Adding a New Module

We want to add a function that prints out temperatures. We create a new source file named `weatherio.c`.

16.2.1 The Source Code (weatherio.c)

```
#include <stdio.h>

void printTemps(double *temperatures, int size) {
printf("Over the past %d days the temperatures were:\n", size);
int i;
for (i = 0; i < size; i++) {
printf("On day %d the temperature was %.2lf\n", i + 1,
temperatures[i]);
}
}
```

16.2.2 The Header File (weatherio.h)

We also create the header file with the function prototype.

```
void printTemps(double *temperatures, int size);
```

16.2.3 Compiling the New Module

We compile this new file into an object file.

```
gcc -c weatherio.c -o weatherio.o
```

16.3 Updating the Library

Now we add the new object file to our existing library. We can simply run the `ar` command again with all the object files we want in the library.

```
ar rcs libweather.a weatherstats.o weatherio.o
```

The library `libweather.a` now contains both `weatherstats.o` and `weatherio.o`.

16.4 Using the Updated Library

To use the new function in our main program, we need to update `program.c` and manage our headers.

16.4.1 Creating a Master Header (weather.h)

Instead of including multiple header files in `program.c`, it is cleaner to create a single header file for the library that includes the others.

```
#include "weatherstats.h"
#include "weatherio.h"
```

16.4.2 Updating the Main Program

In `program.c`, we include the new master header and call the `printTemps` function.

```
#include "weather.h"
...
printTemps(temperatures, 7);
```

16.4.3 Compiling and Linking

We can compile the program and link it with the library in a single command. We use `-L.` to tell GCC to look in the current directory for libraries, and `-lweather` to link against `libweather.a`.

```
gcc program.c -L. -lweather -o program
```

Running `./program` will now display the temperatures using the new function from the updated library.

---

17 Ultimate makefile

Let's finish this section on libraries by updating our previous Makefiles. We assume that our library `libweather.a` exists and is unchanged. Therefore, our Makefile will only be concerned with creating the executable program.

17.1 Updating the Simple Makefile

The program no longer depends on `weatherstats.o` because we are using the library. We update the build command to include the library path (`-L.`) and the library itself (`-lweather`).

```
program:  program.o
      gcc -std=C11 -Wall -fmax-errors=10 -Wextra -o program program.o
-L. -lweather

program.o:  program.c
      gcc -std=C11 -Wall -fmax-errors=10 -Wextra -c program.c -o
program.o

launch:  program
      ./program
```

When we run `make launch`, `make` will check dependencies: it builds `program.o` from `program.c`, then builds `program` by linking `program.o` with the library, and finally executes it.

17.2 Updating the Elaborate Makefile

We can also adjust the more general Makefile to accommodate the library usage. We introduce variables for the library flags and names.

```
CC = gcc
CFLAGS = -std=c11 -Wall -fmax-errors=10 -Wextra
LFLAGS = -L.
LIBS = -lweather

OBJ = program.o
MAIN = program

all:  $(MAIN)

$(MAIN): $(OBJ)
      $(CC) $(CFLAGS) -o $(MAIN) $(OBJ) $(LFLAGS) $(LIBS)

%.o:  %.c
      $(CC) $(CFLAGS) -c $< -o $@

launch:  $(MAIN)
      ./$(MAIN)

clean:
      rm *.o $(MAIN)
```

This Makefile is versatile. You can use it for other projects by simply updating the variables at the top. The `clean` target allows you to start from a clean slate by removing object files and the executable.