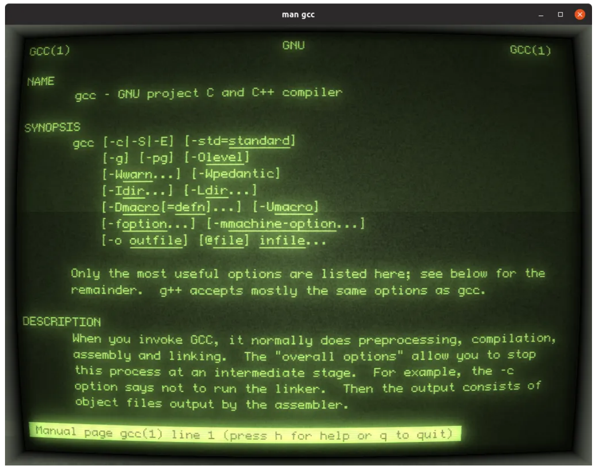
Coding your own Shell in C

Santiago Follow 13 min read · Nov 16, 2022

160 3



```
man gcc

GCC(1)                          GNU                          GCC(1)

NAME
       gcc - GNU project C and C++ compiler

SYNOPSIS
       gcc [-c|-S|-E] [-std=standard]
           [-g] [-pg] [-Olevel]
           [-Wwarn...] [-Wpedantic]
           [-Idir...] [-Ldir...]
           [-Dmacro[=defn]...] [-Umacro]
           [-foption...] [-mmachine-option...]
           [-o outfile] [@file] infile...

       Only the most useful options are listed here; see below for the
       remainder.  g++ accepts mostly the same options as gcc.

DESCRIPTION
       When you invoke GCC, it normally does preprocessing, compilation,
       assembly and linking.  The "overall options" allow you to stop
       this process at an intermediate stage.  For example, the -c
       option says not to run the linker.  Then the output consists of
       object files output by the assembler.

 Manual page gcc(1) line 1 (press h for help or q to quit)
```

img source: https://github.com/Swordfish90/cool-retro-term

In the movies, we see how programmers work in front of a dark screen with small green letters, which in most cases does not contain actual code or commands, except for some cases, such as the fantastic series of Mr. Robot. Still, the rest of the series or movies do not reflect how the developers really work.

Actually, most of the developers I know barely use the terminal (that dark screen with green letters you see in movies), they only use it when strictly necessary, to install a library for example. Something really sad, the more you go into it, the more you learn.

I forced myself to use Vim and the terminal as my only text editor. It wasn't until a long time ago that I started using VS Code with a Vim extension since I stuck to the good habit of using only the keyboard whenever possible, I even have a Vim extension to navigate the web using the keyboard.

The point is that I learned a lot more things compared to my colleagues who have used a text editor from the beginning. Well, the fact that I had to use only Vim to write a considerable amount of code, sooner or later I had to customize Vim (to be more exact, go from Vim to NeoVim and the world behind it) to be more efficient at the time to write code. Thanks to this, I learned about the configuration files of my operating system, soft and hard links, aliases, and even Bash scripts to automate my work environment. I probably wouldn't have learned as much about Linux (here's a good resource for learning the essentials) if I'd jumped right into VS Code.

And this not only applies to the code editor, but also to programming languages, and really to anything in life. Python is a high-level language, but do you really understand why? My first lines of code were in python, however, it wasn't until I learned C that I realized the level of abstraction that python handles and the many things that I take for granted when developing in this language. C allowed me to see further, C gave me the foundation to understand extremely important concepts like pointers, dynamic memory allocation, or compiled vs interpreted languages, which I probably would never have learned with just python. Now, every time I write code in Javascript or Python, I value the work behind it much more.

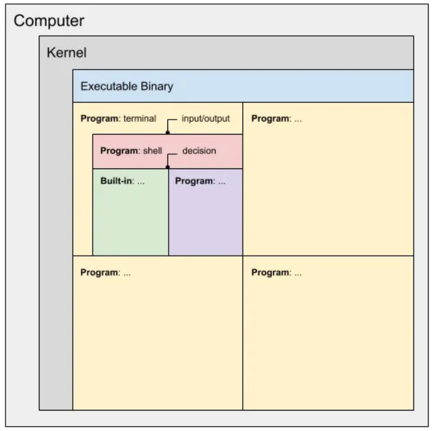
With this said, we are going to develop a simple shell using C. I promise you that it will be very useful even if you do not use this language in your work because I am sure that it will help you understand and realize things that you were not even aware of before.

What is a Shell?

Before diving into the process, we should understand what is a shell. To answer this question, a Shell is a program that takes the command inputs written from the user's keyboard and passes them to the machine to execute them through the kernel. It also verifies if the command input from the user is correct.

Terminals, kernels, and shells...

It is worth clarifying that the kernel, the terminal, and the shell are different things. Here is an overview to understand the differences:



One thing I want to clarify in the above image is that each box is within the parent box. So for example, The "Program: terminal" is interacting with a contained shell, while the shell then spawns either a "Built-in" or "Program". source: https://www.integralist.co.uk/posts/terminal-shell/

Differences and important concepts:

- **Kernel:** a computer has a kernel, the kernel is responsible for managing the computer's system, the kernel has no user interface, to interact with the kernel you use an intermediary program.
- **Program:** a program is a structured collection of instructions (machine code) that a computer can execute. Your computer has many programs. One such example would be the "terminal emulator" program.
- **Executables:** executables (or executable binaries) are programs. More specifically, an executable is a file that *contains* a program, generally the *result* of a program being turned into something that can be executed by the computer.
- **Terminal:** a terminal is an input/output device. Traditionally they would have been a ***real*** hardware device that you used to interact with a computer. e.g. the computer would be a large box in a server room, and the terminal would be a monitor/keyboard connected to the computer. In modern computing, we have a "terminal emulator". If you don't want to use a GUI (graphical user interface) to interact with your computer, you can use a terminal emulator.
- **Shell:** a shell is a program that is accessed *via* a terminal emulator. The terminal accepts input, passes it to the shell, and the shell's output is sent back to the terminal to be displayed. The shell accepts input as a set of commands.

Follow this link for a deeper look or visit my previous article if you want it in plain English.

Let's code the Shell…

Now we know what is a Shell, let's open the code editor… But before let's take a look at how will be the lifetime of our shell:

- **Initialize:** in this step, a typical shell would read and execute its configuration files. These change aspects of the shell's behavior.
- **Interpret:** next, the shell reads commands from stdin (which could be interactive, or a file) and executes them.
- **Terminate:** after its commands are executed, the shell executes any shutdown commands, frees ups any memory, and terminates.

Our shell will be simple, so the shell initialization and configuration stage will not be necessary. A Shell can work interactively or non-interactively, so the main function is responsible for determining which of the two ways our shell will work:

```
//main.c
#include "shell.h"

/**
 * main - function that checks if our shell is called
 *
 * Return: 0 on success
 */
int main(void)
{
 // determines if file descriptor is associated with a terminal
 if (isatty(STDIN_FILENO) == 1)
 {
  shell_interactive();
 }
 else
 {
  shell_no_interactive();
 }
 return (0);
}
```

`issaty` is a function that determines if a file descriptor (fd) is associated with a terminal. It take as argument a fd and returns 1 fi fd is referring to a terminal, otherwise 0 is returned. All the functions used will be listed at the end of the document, so if you have any questions while writing the code, I encourage you to visit the man pages or search on the web. It's very important that you really understand how the functions work.

Don't forget to create the header file which contains C function declarations and macro definitions to be shared between several source files.

```
//shell.h
#ifndef SHELL_H
#define SHELL_H

/*---LIBRARIES---*/
#include <stdio.h>
#include <unistd.h>
```

```
/*---PROTOTYPES---*/
/* main.c */
void shell_interactive(void);
void shell_no_interactive(void);

#endif
```

For now it is something simple, however, this file will grow along with our code.

Shell: a RPEL process

In a very simple way, the flow of the Shell can be considered to be of type RPEL. The first action that a Shell must complete is read a line from the stdin, R is the stand for this action (read). Once the line has been read, we should split it, since we know that the first string refers to the command. Separated by a space, we found the flags and arguments of the command, P is for parse. E refers to execute, of course, we execute the command along with its arguments. Finally, our shell should iterate (L — loop) this process until the user exit the process or the shell reaches the end of the file (EOF).

This applies to both, the interactive and no-interactive Shell.

Shell Interactive

This is the typical case where you type into the terminal and the Shell process and execute these commands.

```
//shell_interactive.c
#include "shell.h"

/**
 * shell_interactive - UNIX command line interpreter
 *
 * Return: void
 */
void shell_interactive(void)
{
 char *line;
 char **args;
 int status = -1;

 do {
  printf("simple_prompt$ "); /* print prompt symbol */
  line = read_line(); /* read line from stdin */
  args = split_line(line); /* tokenize line */
  status = execute_args(args);
  /* avoid memory leaks */
  free(line);
  free(args);
  /* exit with status */
  if (status >= 0)
  {
   exit(status);
  }
 } while (status == -1);
}
```

As you can see, this function contains the, read (`read_line`), parse (`split_line`), execute (`execute_arg`), and loop (the mentioned functions will iterate until the status gives to us a signal to end the process). In a moment you will understand why the status is equal to -1. Now it's time to build out function.

Top highlight

Read Line

In the shell_interactive.c file, We declare a char pointer `line` to allocate the string returned by `read_line` function. `read_line` function reads and allocates a string containing the line read from the stream and returns it. If it is not possible to read the line from the stream, the cause is checked. If the EOF is reached, the function is terminated successfully, otherwise, the function will exit with failure.

```
//read_line.c
#include "shell.h"

/**
 * read_line - read a line from stdin
 *
 * Return: pointer that points to a str with the line content
 */
char *read_line(void)
{
 char *line = NULL;
 size_t bufsize = 0;

 if (getline(&line, &bufsize, stdin) == -1) /* if getline fails */
 {
  if (feof(stdin)) /* test for the eof */
  {
   free(line); /* avoid memory leaks when ctrl + d */
   exit(EXIT_SUCCESS); /* we recieved an eof */
  }
  else
  {
   free(line); /* avoid memory leaks when getline fails */
   perror("error while reading the line from stdin");
   exit(EXIT_FAILURE);
  }
 }
 return (line);
}
```

Moving back to the `shell_interactive` function, once `read_line` is executed, will have the string read from the stdin allocated in the `line` variable. Now we must split the entire string to get the command and its arguments.

Parse String

The purpose of split_line is to obtain an array and thus separate each element from the original text string that was obtained with the read_line function. When we split a string into multiple elements, we call this process "tokenize" in C programming.

```
//split_line.c
#include "shell.h"

/**
 * split_line - split a string into multiple strings
 * @line: string to be splited
 *
 * Return: pointer that points to the new array
 */
char **split_line(char *line)
{
 int bufsize = 64;
 int i = 0;
 char **tokens = malloc(bufsize * sizeof(char *));
 char *token;

 if (!tokens)
 {
  fprintf(stderr, "allocation error in split_line: tokens\n");
  exit(EXIT_FAILURE);
 }
 token = strtok(line, TOK_DELIM);
 while (token != NULL)
 {
  /* handle comments */
  if (token[0] == '#')
  {
   break;
  }
  tokens[i] = token;
  i++;
  if (i >= bufsize)
  {
   bufsize += bufsize;
   tokens = realloc(tokens, bufsize * sizeof(char *));
   if (!tokens)
   {
    fprintf(stderr, "reallocation error in split_line: tokens");
    exit(EXIT_FAILURE);
   }
  }
  token = strtok(NULL, TOK_DELIM);
 }
 tokens[i] = NULL;
 return (tokens);
}
```

Basically, what `split_line` does is call `strtok` passing as arguments the string to split and the delimiters to know when to break and add the extracted element to an array. The code can be seen as tricky but if you take a look at `strtok man page` you will find that you need to give it the string argument only for the first call, each time you call the function again you don't need to, `strtok` will return the next token until the end of the initial string is reached.

This is the reason why we store in the token variable the returned string by `strtok`. Then we start to iterate, in each iteration, we call `strtok` again but with a NULL string argument, then we allocate the returned token in an array of string (`char **tokens`) until `strtok` finds no more tokens. Inside the loop, we check if the size of tokens needs to increase to store more elements. Finally, `split_line` returns an array with multiple tokens.

Execute Arguments

We know that the first element of the tokens array refers to the command to be executed and the following ones are its flags. Most of the commands that a shell executes are programs, but not all of them, some commands live in the shell itself, they are called built-ins.

For example, most shells are configured by the rules specified in the configurations scripts, like "~/.zshrc". Those scripts use commands that change the operation of the shell, is possible because they were implemented within the shell code. So, the commands that are in ".zshrc" file are built-in commands.

We should create a function that checks if the entered command matches a builtin command and, if so, executes it. If there is no match, the program should start a new process that finds and runs the file containing the desired command. In our shell, `execute_args` will take care of that.

```
//execute_args.c
#include "shell.h"

/**
 * execute_args - map if command is a builtin or a process
 * @args: command and its flags
 *
 * Return: 1 on sucess, 0 otherwise
 */
int execute_args(char **args)
{
 char *builtin_func_list[] = {
  "cd",
  "env",
  "help",
  "exit"
 };
 int (*builtin_func[])(char **) = {
  &own_cd,
  &own_env,
  &own_help,
  &own_exit
 };
 int i = 0;

 if (args[0] == NULL)
 {
  /* empty command was entered */
  return (-1);
 }
 /* find if the command is a builtin */
 for (; i < sizeof(builtin_func_list) / sizeof(char *); i++)
 {
  /* if there is a match execute the builtin command */
  if (strcmp(args[0], builtin_func_list[i]) == 0)
  {
   return ((*builtin_func[i])(args));
  }
 }
 /* create a new process */
 return (new_process(args));
}
```

We use `strcmp` to compare the command and built-in functions, if there is a match, we will use a pointer to a function to call the target function passing the command and its flags. `execute_args` function will return the returned value of the target built-in function. Otherwise, `execute_args` will launch a new process by calling `new_process` function and return the returned value of `new_process`.

Execute a non-built-in command

When our Shell is called, a new process is created and it is active (running) until we end, kill or exit the program. But what is a program? In a nutshell, a program is an executable file held in storage on your machine. Anytime you run a program, you have created a process. Since we are running our shell, it has a unique process ID and lives with some specific environment variables. As we discussed before, a non-built-in function is a program (executable file) that lives somewhere in our memory.

Let's code a function that creates a new process and execute the file that contains the command that we want to perform.

```
//new_process.c
#include "shell.h"

/**
 * new_process - create a new process
 * @args: array of strings that contains the command and its flags
 *
 * Return: 1 if success, 0 otherwise.
 */
int new_process(char **args)
{
 pid_t pid;
 int status;

 pid = fork();
 if (pid ==  0)
 {
  /* child process */
  if (execvp(args[0], args) == -1)
  {
   perror("error in new_process: child process");
  }
  exit(EXIT_FAILURE);
 }
 else if (pid < 0)
 {
  /* error forking */
  perror("error in new_process: forking");
 }
 else
 {
  /* parent process */
  do {
   waitpid(pid, &status, WUNTRACED);
  } while (!WIFEXITED(status) && !WIFSIGNALED(status));
 }
 return (-1);
}
```

`new_process` function takes as argument the array of strings that `split_line` function returns. Using `fork` function, we create a new process that has the same environment variables as the parent process, we allocate these data in a pid_t structure. Now we have two processes running concurrently.

In the child process, we want to run the command given by the user. We're going to use `execvp` function that takes the name of the executable (the command for our case, that would be in the first position of the array, args[0]) and an array (vector) of string arguments (like the argument vector) which contains the flags. The 'p' means that instead of providing the full file path of the program to run, we're going to give its name and let the operating system search for the program path.

The parent process needs to wait for the command (child process) to finish.

We use `waitpid` function to wait for the process's state to change. We use the macros provided with `waitpid` to wait until either the processes are exited (WIFEXITED, evaluates if the child process terminates normally) or killed (WIFSIGNALED, child process was finished by a signal). Then, the function finally returns 1 as a signal to the calling function that we should prompt for input again.

So far, we have seen how the RPEL process works for our shell and the life cycle will be completed when the user decides to exit our EOF will be reached.

Shell no-interactive

We have coded a simple shell that can perform basic tasks in an interactive way, which means that the shell is waiting for commands that the user will give through the terminal, and will display the output to the user.

But what if we want to perform some actions from a bash script or run commands through some automated process? This is when the no-interactive shell hits the floors.

```
//shell_no_interactive.c
#include "shell.h"

/**
 * shell_no_interactive - unix command line interpreter
 *
 * Return: void
 */
void shell_no_interactive(void)
{
 char *line;
 char **args;
 int status = -1;

 do {
  line = read_stream();
  args = split_line(line); /* tokenize line */
  status = execute_args(args);
  /* avoid memory leaks */
  free(line);
  free(args);
  /* exit with status */
  if (status >= 0)
  {
   exit(status);
  }
 } while (status == -1);
}
```

It is no coincidence that `shell_no_interactive` function is slightly different from `shell_interactive` function. In fact, the only difference occurred when we assign a value to a `line` variable, because we get the string that contains the command in a different way. The change is so small that there was no need to create a new function and implement it from scratch. However, I decided to split it to make the code easier.

Read Line from the stdin stream

```c
//read_stream.c
#include "shell.h"

/**
 * read_stream - read a line from the stream
 *
 * Return: pointer that points the the read line
 */
char *read_stream(void)
{
 int bufsize = 1024;
 int i = 0;
 char *line = malloc(sizeof(char) * bufsize);
 int character;

 if (line == NULL)
 {
  fprintf(stderr, "allocation error in read_stream");
  exit(EXIT_FAILURE);
 }
 while (1)
 {
  character = getchar(); /* read first char from stream */
  if (character == EOF)
  {
   free(line);
   exit(EXIT_SUCCESS);
  }
  else if (character == '\n')
  {
   line[i] = '\0';
   return (line);
  }
  else
  {
   line[i] = character;
  }
  i++;
  if (i >= bufsize)
  {
   bufsize += bufsize;
   line = realloc(line, bufsize);
   if (line == NULL)
   {
    fprintf(stderr, "reallocation error in read_stream");
    exit(EXIT_FAILURE);
   }
  }
 }
}
```

The `read_stream` function enters an infinite loop and reads the first character from the stdin stream using the `getchar` function, after we allocate the first byte in the `character` variable, we check if the byte corresponds to the EOF, if it is the case, we exit with program with success status. If the above condition is not met, we check if there is a line break, which tells us that the line has end, we allocate a null byte at the end of `line` and return it. Otherwise, we allocate the byte in the corresponding position of `line`. Finally, we check if `line` needs more space to store the following bytes.

Finally, the header file should look like:

```c
#ifndef SHELL_H
#define SHELL_H

/*---LIBRARIES---*/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>

/*---Macros---*/
#define TOK_DELIM " \t\r\n\a\""
extern char **environ;

/*---PROTOTYPES---*/
/* main.c */
void shell_interactive(void);
void shell_no_interactive(void);

/* shell_interactive.c */
char *read_line(void);
char **split_line(char *line);
int execute_args(char **args);

/* execute_args */
int new_process(char **args);

/* shell_no_interactive */
char *read_stream(void);

/*---Builtin func---*/
int own_cd(char **args);
int own_exit(char **args);
int own_env(char **args);
int own_help(char **args);

#endif
```

Here you can find the Github repository of the project.

Find me on:

Linkedin: https://www.linkedin.com/in/santiagobedoa/

Github: https://github.com/santiagobedoa

Resources

main.c

- `isatty`
- `fileno`

read_line.c

- `getline`
- `feof`
- `perror`

split_line.c

- `strtok`

execute_args.c

- `strcmp`

new_process.c

- `fork`