Introduction to Programming in C

CSC 100 - Lyon College - Spring 2025

Marcus Birkenkrahe

January 6, 2025

Contents

1 What will you learn?

- **Topics**:
  - ☐ What is programming?
  - ☐ What is a program?
  - ☐ What is debugging?
  - ☐ Formal vs. natural languages
  - ☐ The first program
  - ☐ Online programming platform(s)
  - ☐ Making and fixing mistakes
  - ☐ Writing your first program
  - ☐ Compiling your first program
  - ☐ Running your first program
- **Reading**: Chapter 1 (**Think C**) - The way of the program, pp. 1-10.
- **Practice exercises** (in-class):
  1. Logging into a Linux VM on Google Cloud Shell
  2. Opening your Jupyter notebook on Google Colaboratory
  3. Writing, compiling and executing your first program.
  4. Upload a notebook file to Canvas.
- **Programming assignments (home)**:
  1. Write "Hello, world!" program `hello2.c` with detailed comments.
  2. Install Google Cloud Shell as an app on your PC.
  3. (Bonus) Be like Linus [Torvalds].
  4. (Bonus) Display `EXIT_SUCCESS` and `EXIT_FAILURE`.

2 What is programming? (Philosophy)

Figure 1: "The Allegory of the Cave" by Jan Saenredam (1604, inspired by Plato)

- What is your motivation to learn a programming language?
- What's shown in the painting and what does it mean?
- What does it take to learn programming?
- What is an abstraction? What is real?

3 What is a programming language?

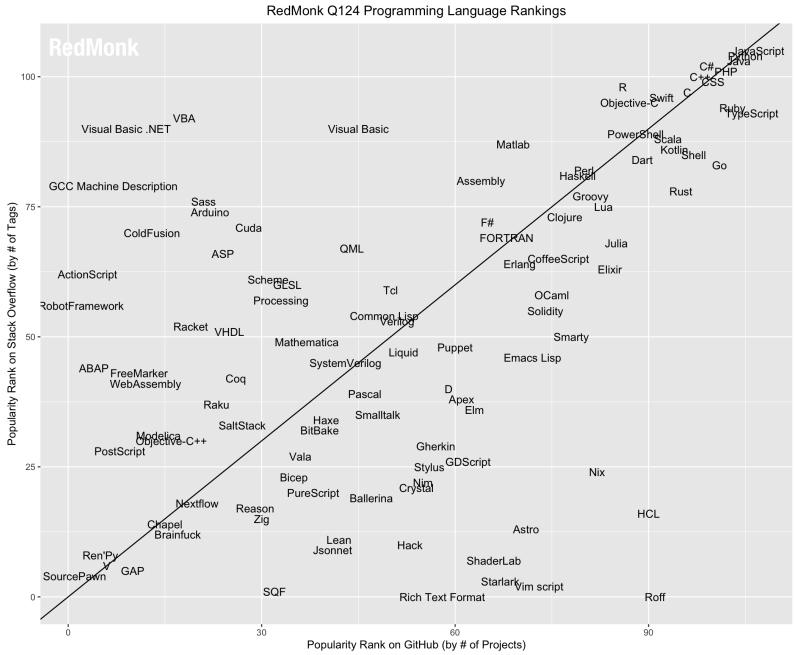
- Language = Syntax (rules) + semantics (meaning) + metadata (context)
- In the Biblical Tower of Babel story, which language component is the core of God's punishment for the people's blasphemy?
- *Learning a programming language doesn't make you a programmer.*

Figure 2: Tower of Babel by Pieter Bruegel the Elder (1563)

4 What is C? What is C++?

- C is a programming language created in the early 1970s.
- It grew out of the development of the UNIX operating system.
- In turn, UNIX grew out of a space travel game (Brock, 2019).
- C++ is an object-oriented extension of C ("C with classes"), created by Bjarne Stroustrup 1979-1985. I learnt it in 1990 as my first "real" or "professional" programming language.
- C/C++ consistently rank among the top programming languages:

Figure 3: Ken Thompson & Dennis Ritchie & DEC PDP-11, 1970. (Brock, 2019)



Position of language by # of Stackoverflow tags/GitHub projects.

5 Why is C important? Why is C++ important?

- C dominates real-world applications in every way:
  1. The Linux kernel (and therefore, Android)
  2. UNIX operating system (core of MacOS and iOS)
  3. Windows operating system (core of most PCs)
  4. Doom (early video game) and Wolfenstein 3D
  5. Git version control system (as in "GitHub")
  6. Compilers (Clang, GCC/MinGW for most other languages)
  7. Interpreted languages like Python, R,
  8. Any software that crosses platforms easily (portable)
  9. Software for the Curiosity Mars rover and most space apps
  10. Banking, telecommunication, and military software

- C++ dominates game software and other industrial applications where e.g. graphics are required. It accompanies many of C's implementations except for applications very close to the machine, i.e. with direct hardware access, low-level memory control, high portability and efficiency.

6 Why don't we just learn C++?

"Hello world" in C

```
#include <stdio.h>

int main(void) {
  puts("Hello, world");
  return 0;
}
```

"Hello world in C++

```
#include <iostream>

int main(void) {
  std::cout << "Hello, world!" << std::endl;
  return 0;
}
```

- Object-orientation is a difficult paradigm for beginners (C++).
- System programming is pure power (C).
- C is simpler, smaller, and faster.
- C has 35 keywords, C++ has 95.

7 Programming vs. Natural Languages

- Which has more syntax rules, programming languages (like C), or natural languages (like English)? And why?
  - **Natural Languages** (e.g., English) have far more syntax rules than programming languages. They are ambiguous and context-dependent, requiring complex syntax to clarify meaning, evolve continuously over time

(adding more rules), and are rich in variability. They are used (perhaps even "designed") for complex human communication.

- Programming Languages (e.g., C) are designed to be precise and unambiguous small, well-defined set of syntax rules, they are static and slow-changing, focus on clarity and functionality, and prioritize simplicity over expressiveness to ensure machines "get it".
- C for example has only 30 keywords, C++ has got 90, and English has 150 function words (like "articles", "pronouns" etc.) and 170,000 words.

8 Interpretation vs. Compilation

- Programming Languages are either interpreted or compiled to generate machine code from human-readable source code.
- Interpreted languages go straight from source to result:

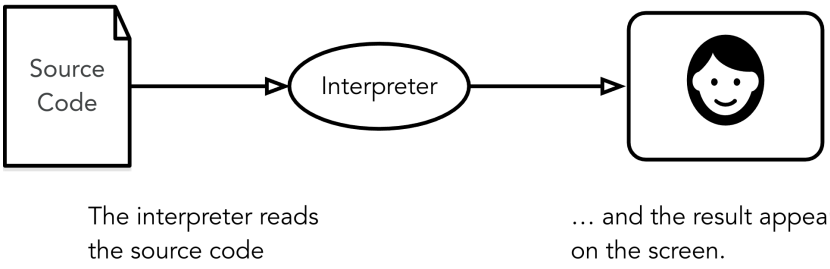


Figure 4: Scheffler, ThinkC (2019), p. 2

- Interpreted example (the first execution shows the console)
- Compiled languages require an intermediate "object code" step.
- Compiled example: The source code file (created with `echo`)) is compiled and executed on the shell:

```
echo '#include <stdio.h>' > hello  # redirect first line to file
echo 'main() { printf("Hello, world!\n");}' >> hello # append rest of code
gcc -x c hello # generate object code
./a.out # execute executable
```

```
>>> 10**3 / 0.31459
3178.740582981023
>>> print("Hello, World!")
Hello, World!
>>>
```

Figure 5: Python console / interpreter dialog fragment

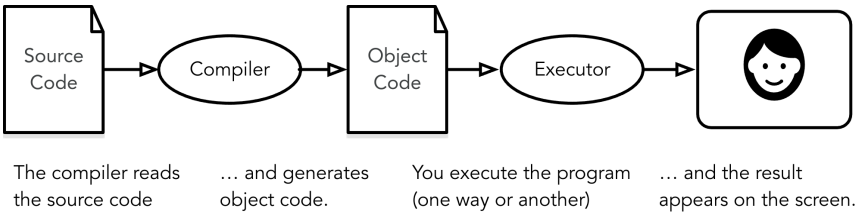


Figure 6: Scheffler, ThinkC (2019), p. 2

```
Hello, world!
```

- Check out this nice demo video (1983, shared via Chat)
- The actual translation journey in the machine from source to object code is more complicated and involves a number of intermediate files and programs (open in browser - tinyurl.com/compiler-driver):
- This diagram is a BPMN model that we'll use for pseudocode.

9 What does the machine see?

- The source program `hello.c` is a sequence of bits or memory cells, each with a value of 0 or 1, organized in 8-bit chunks called *bytes*.
- Each byte in machine memory has an integer value that corresponds to some character. For example, the `#` has the integer value 35.
- Here is the source file as you see it:

Figure 7: `https://youtu.be/_C5AHaS1mOA?si=RL3lOZftsldtObPV`

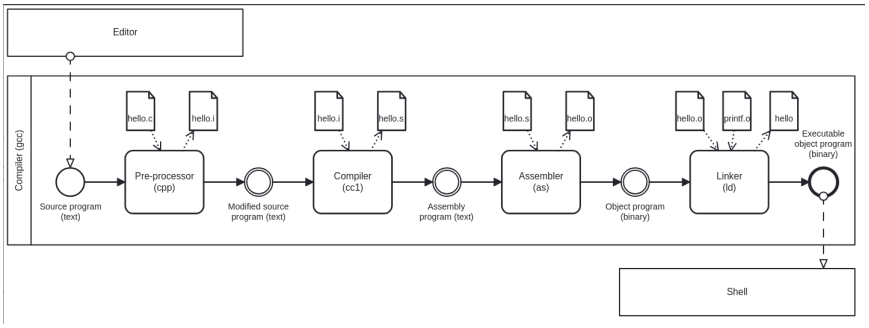


Figure 8: Source: Bryant/O'Halloran, Computer Systems (3e) 2015

```
#include <stdio.h>
int main()
{
printf("Hello, World!\n");
return 0;
}
```

- And here is the source file as the computer sees it:

```
  #   i   n   c   l   u   d   e  SP   <   s   t   d   i   o   .   h   >  \n
 35 105 110  99 108 117 100 101  32  60 115 116 100 105 111  46 104  62  10

  i   n   t  SP   m   a   i   n   (   )  \n   {  \n   p   r   i   n   t   f
105 110 116  32 109  97 105 110  40  41  10 123  10 112 114 105 110 116 102

  (   "   H   e   l   l   o   ,  SP   W   o   r   l   d   !   \   n   "   )   ;
 40  34  72 101 108 108 111  44  32  87 111 114 108 100  33  92 110  34  41  59

 \n   r   e   t   u   r   n  SP   0   ;  \n   }  \n
 10 114 101 116 117 114 110  32  48  59  10 125  10
```

- **Note:** The stand-alone *newline* character `\n` (10) is different from `"\n"` inside a string (92 + 10).

10 What is a (software) program?

- Is this a program or not?
- Answer:
  - The recipe is a set of instructions but not for computation by a computer, and not hampered by syntax rules (other than English grammar). It's a form of algorithm written in pseudocode. It is not written in a programming language.
  - It has some properties of a good program: A program header with title, author and date information, and other meta data (how many are served).
  - Meta data are equally important for programming but they are not standardized unless the program is sent

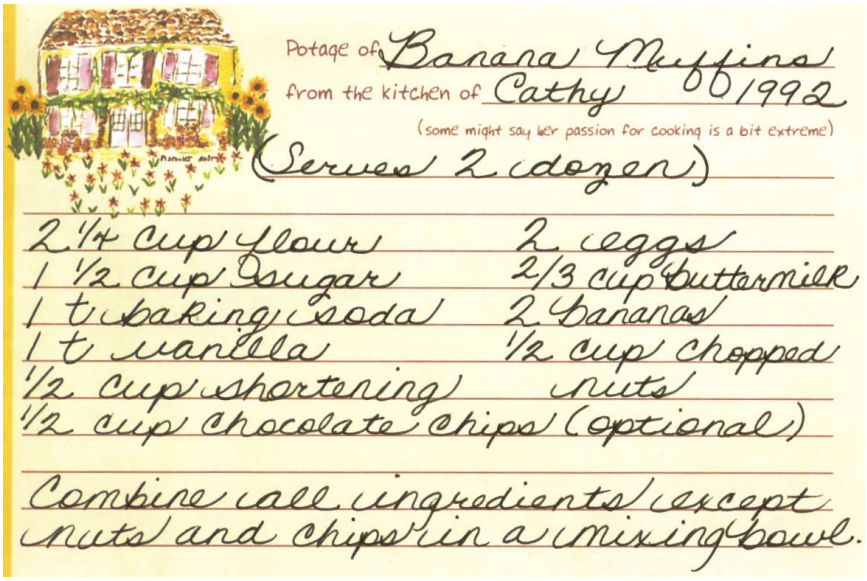


Figure 9: Banana muffin recipe

over a computer network (then the program header is used like an address on an envelope).

- What about this?
- The general template for a program:

  1. **Input** (fed to the program from the outside, or `stdin`)
  2. **Output** (generated by the program for the outside, or `stdout`)
  3. **Statements** (commands other than I/O).
  4. Baking recipe example:
     - Input = Baking ingredients
     - Output = Banana muffin
     - Statements = Baking instructions
  5. "Hello, world!" program:
     - Input = Character sequence: `"Hello, world"` (string)
     - Output = Screen message `Hello, world!` (stdin)
     - Statements = e.g. `printf("Hello, world!\n");` (function call)

```
section .data
  hello db "Hello, World!", 0xA  ; The string to print with a newline (0xA)
  len equ $ - hello              ; Calculate the length of the string

section .text
  global _start                  ; Define the entry point for the program

_start:
  ; Write syscall
  mov eax, 4                     ; syscall: sys_write (4)
  mov ebx, 1                     ; file descriptor: stdout (1)
  mov ecx, hello                 ; pointer to the string
  mov edx, len                   ; length of the string
  int 0x80                       ; call the kernel

  ; Exit syscall
  mov eax, 1                     ; syscall: sys_exit (1)
  xor ebx, ebx                   ; status: 0
  int 0x80                       ; call the kernel
```

Figure 10: "Hello World" in x86 Assembly using Linux system calls

11 What make programs work (hardware)?

(a) CPU + RAM + Non-Volatile Memory (NVM)

- Central Processing Unit ("brain"): very, very fast. General purpose (like Intel Core, AMD Ryzen or Apple M-series); embedded CPUs (on microcontrollers); server CPUs (Intel XEON, AMD's EPYC).
- GPUs are workhorses for parallel computing that usually run alongside a CPU (e.g. for fast scientific or graphics calculations). An example for AI is Google's TPU (Tensor Processing Unit) designed specifically for neural network machine learning.
- Secondary storage (non-volatile memory, NVM): very, very slow. Much too slow for the CPU. NVM can be a hard disk, or a Solid State Drive (SSD) - it doesn't disappear when the power goes off (by way of permanent magnetic fields).

b) Main memory (Random Access Memory): fast enough for the CPU. Organized as a "stack" of memory addresses. All programs must be loaded into memory before they can be executed. In C, you can access memory cells directly through the "pointer" data structure.

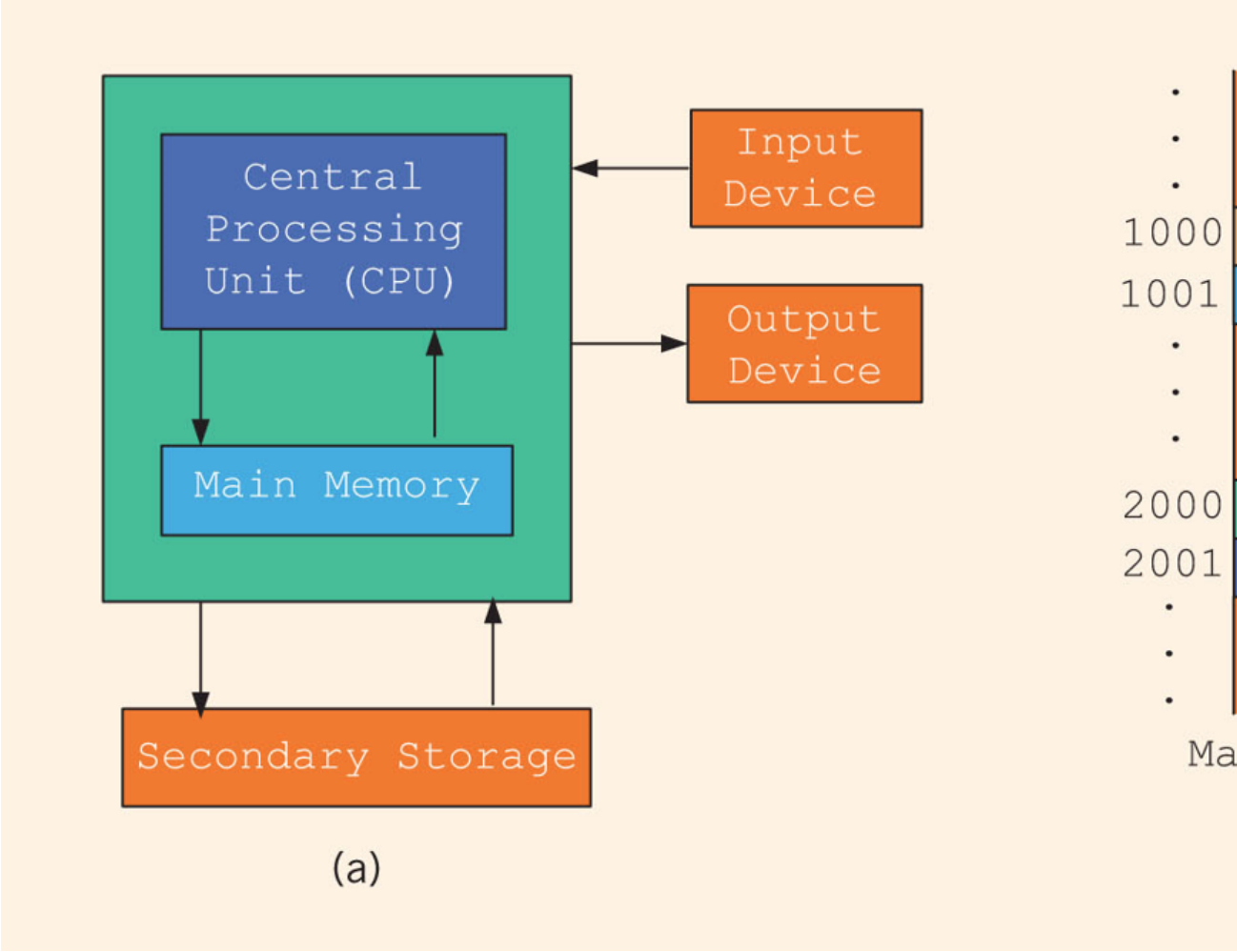


Figure 11: Computer architecture (simplified)

12 What is debugging?

- Programming is the process of creating programs that run and do what you want them to do (print, draw, compute, operate something).
- Debugging is when things don't go well and you get errors (or warnings), or unexpected results, or no results.
- Programming with debugging includes these steps:

  1. Understand the problem (if there is one).
  2. Plan a computational solution (if there is one).
  3. Open an editor
  4. Enter source code (statements + comments)
  5. Save source code to a file
  6. Close the editor
  7. Compile source code to an executable file (debug syntax)
  8. Run the executable file
  9. Check the results (debug logic)
  10. Perform a post-mortem

13 The first program (Demo)

Traditionally (since K&R, 1978), the first program in any language is "Hello, world!". It's very small but it packs a punch.

I will demonstrate the whole process using a cloud editor + shell now, and you will later do it in an integrated development environment (IDE).

1. Problem: Print the message `Hello, world!` to the screen.
2. Cognitive solution requires ability/decision to:
   (a) Generate a text message
   (b) Format it as a string that the machine can identify (quotation)
   (c) Know how the machine prints strings (function)
   (d) Know how to get the printing function (include header file)
   (e) Know how to wrap the solution in a program (header/main)

3. Open Open Google Cloud Shell, & enter `nano` at the prompt.
4. Write the code line by line using your keyboard.
5. Save to a file `hello.c` with `CTRL + X`.
6. Leave `nano`.
7. Compile the source code file with the GCC compiler `gcc`.
8. Run the executable output file at the prompt as `./a.out`.
9. Check the resulting printout.
10. Post-mortem (FAQ) & reflection:
    (a) Which of these steps transfer every time you code?
    (b) How does `printf` work? How can you find out more about it?
    (c) Are double and single quotes equally valid for strings?
    (d) How easy is it to use `nano`? Did you try something else?
    (e) In `nano`, did you try any of the other commands?
    (f) Is GCC the only compiler for the C programming language?
    (g) How can you find out more about the `gcc` command?
    (h) What is the command-line where you compile & run code?
    (i) Why is the output file called `a.out`?
    (j) Why do you run the file using `./a.out`?

14 Online programming platforms

- The cartoon by xkcd (Randall Munroe) contains a lot of information about editors:
    1. `nano` (what I've just used for the demonstration)
    2. `Emacs` (what I'm using at home and in class - *1985)
    3. `vim` (improved `vi` - *1976)
    4. `ed` (UNIX' original line editor, *1971)
    5. `cat` (GNU/Linux core utility viewing program)
    6. "Butterflies" (Chaos Theory)

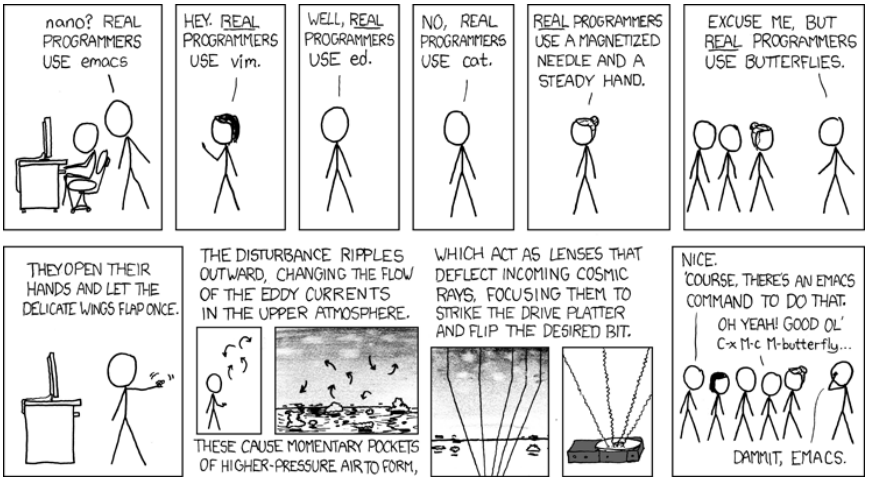


Figure 12: `https://xkcd.com/378`

7. *Jupyter* notebook (in Google Colaboratory)

- Many alternatives to Google Cloud Shell/Colab exist but they either have advertisements, require your credit card, want to sell you something, or are IDEs (Integrated Development Environments):

  1. onecompiler.com/c (IDE only)
  2. onlinegdb.com (with command-line)
  3. pythontutor.com (with visualization)
  4. programiz.com/c-programming/online-compiler/ (IDE)
  5. geeksforgeeks.org/ide/online-c-compiler (IDE)
  6. replit.com is an online REPL (Read-Eval-Print-Loop) application
  7. github.com/codespaces (free for students, with AI support)
  8. vscode.dev (IDE)

- In all scenarios, you need three software applications for C:

  1. An **editor** to write the source code.
  2. A **compiler** to translate source code into object code.
  3. A **shell** to execute the object code and see the results.

15 Making and fixing mistakes

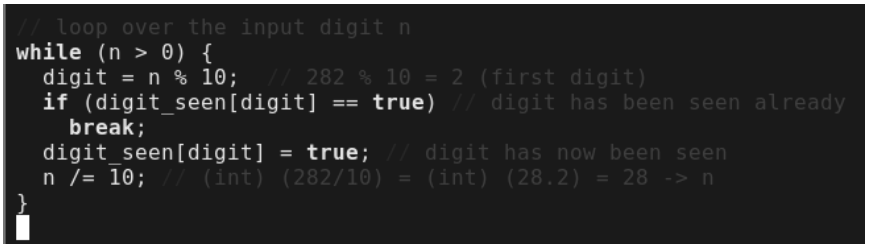
- The compiler tries to direct you to the source of the problem.
- Example error output for this innocent looking program:

```
#include <stdio.h>

int main() {
  printf("What's wrong with me?")
}
$ gcc error.c
error.c: In function ‘main’:
error.c:11:34: error: expected ‘;’ before ‘}’ token
   11 |   printf("What's wrong with me?")
      |                                  ^
      |                                  ;
   12 | }
      | ~
$
```

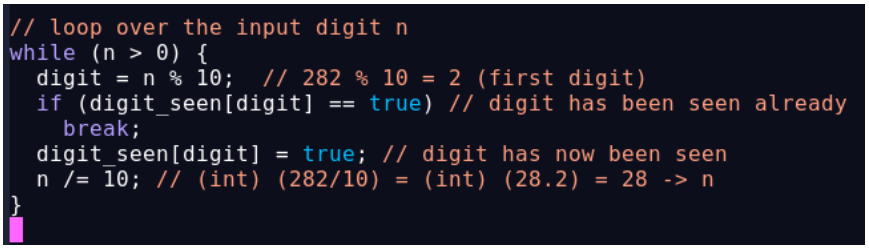
Figure 13: C program and compiler error message

- Alternatives are the use of a debugging tool (like `gdb`), or an online visualizer like `pythontutor.com`.
- Mistakes will occur in three scenarios:
  1. When you compile ("compile-time error" usually "syntax error")
  2. When you run the program ("run-time error")
  3. When you look at the results ("logical error")
- The more time you save preparing, the more you lose debugging.
- **Syntax highlighting** also helps greatly. Compare these two versions of the same program:
- One disadvantage of Google Colab compared to Google Cloud shell is the missing syntax highlighting.
- As you get better, you'll want to design your own coding environment that supports your ideal workflow.



```
// loop over the input digit n
while (n > 0) {
  digit = n % 10;  // 282 % 10 = 2 (first digit)
  if (digit_seen[digit] == true) // digit has been seen already
    break;
  digit_seen[digit] = true; // digit has now been seen
  n /= 10; // (int) (282/10) = (int) (28.2) = 28 -> n
}
```

Figure 14: Code block without syntax highlighting



```
// loop over the input digit n
while (n > 0) {
  digit = n % 10;  // 282 % 10 = 2 (first digit)
  if (digit_seen[digit] == true) // digit has been seen already
    break;
  digit_seen[digit] = true; // digit has now been seen
  n /= 10; // (int) (282/10) = (int) (28.2) = 28 -> n
}
```

Figure 15: Code block without syntax highlighting

16 PRACTICE C Programming with OneCompiler

Objective

The goal of this tutorial is to introduce students to writing, compiling, and running simple C programs using OneCompiler.com.

This is important so that you can complete your assignments & follow along with me in class while coding ("code along") if you wish.

Getting Started with OneCompiler

1. Open your web browser and go to `https://onecompiler.com/c`
2. Click on "New Program" and select "C" as the programming language.
3. You will see a default template with a simple C program.

Creating a new C file in OneCompiler

OneCompiler has two organisational levels: 1) code, 2) file, and we'll change them both.

1. Delete the existing code to start with your own program.

2. Click on the pen in the middle of the page to change the name of the code to `"Hello World"` and add a short description:

   > First program with OneCompiler. CSC 100 class practice: 01/17/2025.

3. Add the tag `helloworld`. Tags help greatly with search + find.

4. Check the `Visibility`, change it to `Unlisted (People with the Link)`, and click on `Save`.

5. Open the main menu (three horizontal lines, upper left side) and choose `My Account`.

6. Click on `CODES`. You should see the last code you edited. If you use dark mode (button at top of the page), you won't see the tag. In the settings (three dots) you can change the visibility or delete the code.

7. Click on the name of your code to get back to the editor view.

8. Hovering over the filename `NewFile1.c`, find the edit pen: In the popup, enter a new file name: `helloworld.c`.

9. To clean the slate, open the settings (three dots) and choose `Clear Output`. You can also download your file from here.

10. Open the setting again, and choose `Editor Settings`. In the popup, check `Disable Code Autocomplete/Suggestions`.

Basic Structure of a C Program

Every C program has a basic structure:

- Header information (*preprocessor*)
- `main` program ending with `return 0;`
- Program body enclosed in `{ }`
- Comments (optional) followed by `//` or enclosed in `/* */`

```
#include <stdio.h>

int main() {
  puts("Hello, World!");
```

```
  return 0;
}
```

Your task:

1. Type the program (with comments) into the editor.
2. At the end of each line, press `Enter`.
3. At the start of a new line, press `TAB` to indent
4. The file is automatically saved.
5. Click `RUN` (or `CTRL + ENTER`) and check the `Output` field.

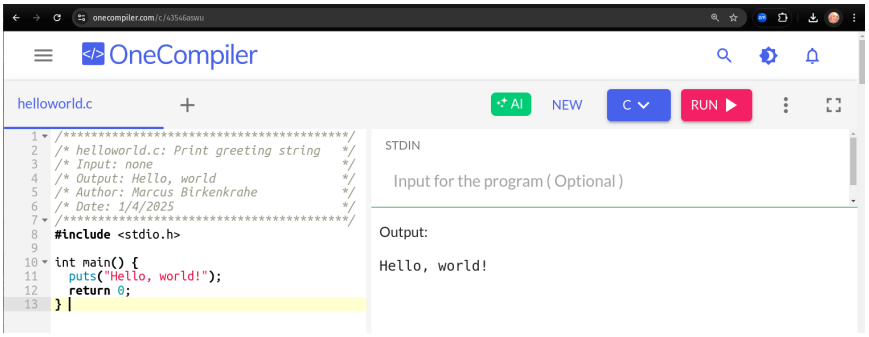


Figure 16: Final result in OneCompiler.com

Extensions

- Below the editor, there is extensive Syntax help for C programming, check it out.
- OneCompiler offers a nice free C tutorial if you want to work ahead. You find it in the top menu (three horizontal lines).
- There are programming challenges (some of which we'll be doing in and outside of class). You have to pick your language.
- There are cheatsheets, as a useful reference or a condensed overview of an advanced topic - check out `C++ Programming language`.
- Next time you want to get back straight to the C editor, go to `onecompiler.com/c`.

17 Tips and Extensions

1. It is advisable, especially at the start, to err on the side of over-commenting. Creating comments will be your first assignment.

2. Things to try when writing the program:

   - What happens if you compile with `printf();` ?
   - What happens if you leave out the `int` before `main`?
   - What happens if you remove the last line, `return 0;` ?
   - What happens if you remove everything but `main() {}`?

3. Things to try when compiling and running the program on a shell (you cannot do this in the OneCompiler IDE):

   - Run the program again with the command: `./a.out > hello`
   - Look at the output file with: `cat hello`
   - Compile again with: `gcc hello.c -o hello.out`
   - Run the program with the command: `./hello.out`
   - Run the program again with the command: `./hello.out > hello.txt`
   - Look at the output file with: `cat hello.txt`
   - List all files starting with `hello`, with: `ls hello*`

   Explanation:

   - The `printf` command requires a non-empty argument, at least `""`.
   - The `main` program will still run in its reduced form. The complete form is quite a bit more complicated.
   - `gcc [source] -o [target]` creates object code (executable) named `target`.
   - You can also redirect the output from the executable to a file using the *redirection* shell operator `>`.

For illustration, here is a complete `main` program: The argument is not `void` (missing) but instead contains the number of arguments `argc`, and an array of pointers `argv[]` to each argument passed to the program.

```
#include <stdio.h>

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

18 Assignments (Details in Canvas)

1. Hello world program `helloworld2.c` with comments.
2. Bonus: "Be like Linus." Print multiple lines.
3. Bonus: Display `EXIT_SUCCESS` and `EXIT_FAILURE` .
4. Read chapter 2 "Variables" in Think C: This chapter covers much of what we're going to talk about in the next lecture. It is the basis of the majority of the questions in the second test.

19 Glossary

Term	Definition
Programming	Process of creating programs that perform specific tasks.
Programming Language	Formal language with syntax, semantics, and metadata.
Syntax	Rules governing the structure and format of code.
Semantics	The meaning or behavior of valid program statements.
Metadata	Contextual information about data or a program.
Interpreted Language	Code is executed directly from source without compilation.
Compiled Language	Code is converted into machine-readable object code.
Bit	A memory cell of value 0 or 1
Byte	A chunk of 8 adjacent bits (stores 1 character)
ASCII	Encoding standard for 128 characters
Program	A structured set of instructions designed for computation.
Algorithm	A step-by-step procedure for solving a problem or task.
Debugging	The process of identifying/fixing errors (bugs).
Syntax Error	An error caused by code that violates syntax rules.
Compile-Time Error	An error detected during the compilation phase of a program.
Run-Time Error	An error that occurs while the program is running.
Logic Error	An error where the program runs but produces incorrect results.
Header File	A file containing definitions (like `printf`) for use in programs.
Input/Output (I/O)	Input: Data fed to the program; Output: Results produced.
Preprocessor	A program that processes source code before it is compiled.
Main Function	The entry point of a C program where execution begins.
Redirection	A shell feature for directing input/output to/from files.
Shell	A command-line interface for interacting w/the operating system.
a.out	Default output file name generated by the GCC compiler.
GCC	GNU Compiler Collection, a compiler for the C language.
Emacs	A powerful, extensible text editor first released in 1985.
nano	A simple, beginner-friendly terminal text editor.
vim	A highly configurable, improved version of the `vi` editor.
Google Cloud Shell	A web-based terminal environment for coding.
Google Colaboratory	An online interactive notebook using Jupyter
Chaos Theory	A theory in mathematics (butterfly effect).
Header Comment	Metadata block at the top of a program.
Exit Codes	Codes returned by a program to indicate success or failure.
Post-Mortem	Analyzing and reflecting on errors after debugging.
Compiler	A tool that translates source code into an executable file.
Shell Utilities	Tools like `ls`, `cat`, and `echo` for file operations on a command line.

20 Summary

The content explores foundational programming concepts and practices:

1. **What is Programming?** Programming is the process of creating instructions for a computer to solve problems.

2. **What is a Programming Language?** A programming language consists of syntax (rules), semantics (meaning), and metadata (context). The Biblical Tower of Babel metaphor highlights the importance of shared syntax.

3. **Programming vs. Natural Languages**: Programming languages have stricter and more formal syntax rules compared to natural languages like English.

4. **Interpretation vs. Compilation**: Interpreted languages execute code directly, while compiled languages translate code into machine-readable object code before execution.

5. **What is a Program?** A program is a structured set of instructions, with components like input, output, and statements. Examples include baking recipes (pseudocode) and assembly programs.

6. **What is Debugging?** Debugging is identifying and fixing syntax errors, run-time errors, and logic errors through planning, coding, and testing.

7. **First Program**: "Hello, World!" serves as the starting point for programming in any language, demonstrating key steps like input, compilation, and execution.

8. **Tools and Environments**: Editors like ‘nano‘, ‘Emacs‘, and ‘vim‘, along with tools like ‘gcc‘ and cloud platforms, support the programming process. Errors can occur at compile-time, run-time, or due to logic issues.

9. **Practice**: Practical exercises include writing, compiling, and running basic programs in Google Cloud Shell, focusing on understanding core programming workflows.