C Programming Introduction

COMPSCI 1XC3 - Computer Science Practice and Experience: Development Basics

Sharmin Ahmed
Winter 2022

C

- Created in 1972 by Dennis Ritchie
- Famous computer scientist also involved in Unix creation
- Languages like Fortran, C and others were some of the first "high-level programming languages"
- Prior to C and these other languages, programs were written using much more tedious to write **assembly code** and **machine code**

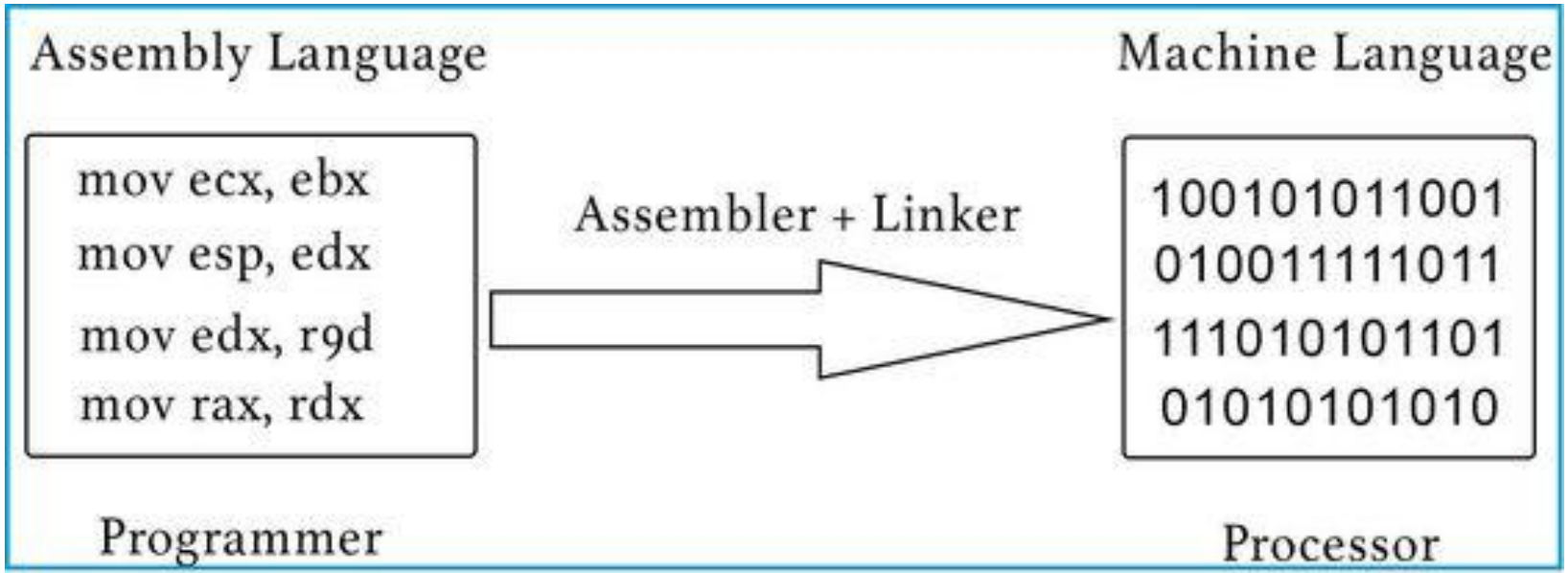
Machine Language

- Machine languages are the native languages that the CPU processes.
- Each manufacturer of a CPU provides the instruction set for its CPU.
- Instructions are very simple at the machine code level
  - Add register1 and register2 together and store the result in register 3
  - Fetch this address of memory and store it in register2
  - Store the content of register2 in this address of memory
- Machine code is represented with binary (1s and 0s) with numbers for each instruction, register, etc.

Assembly code

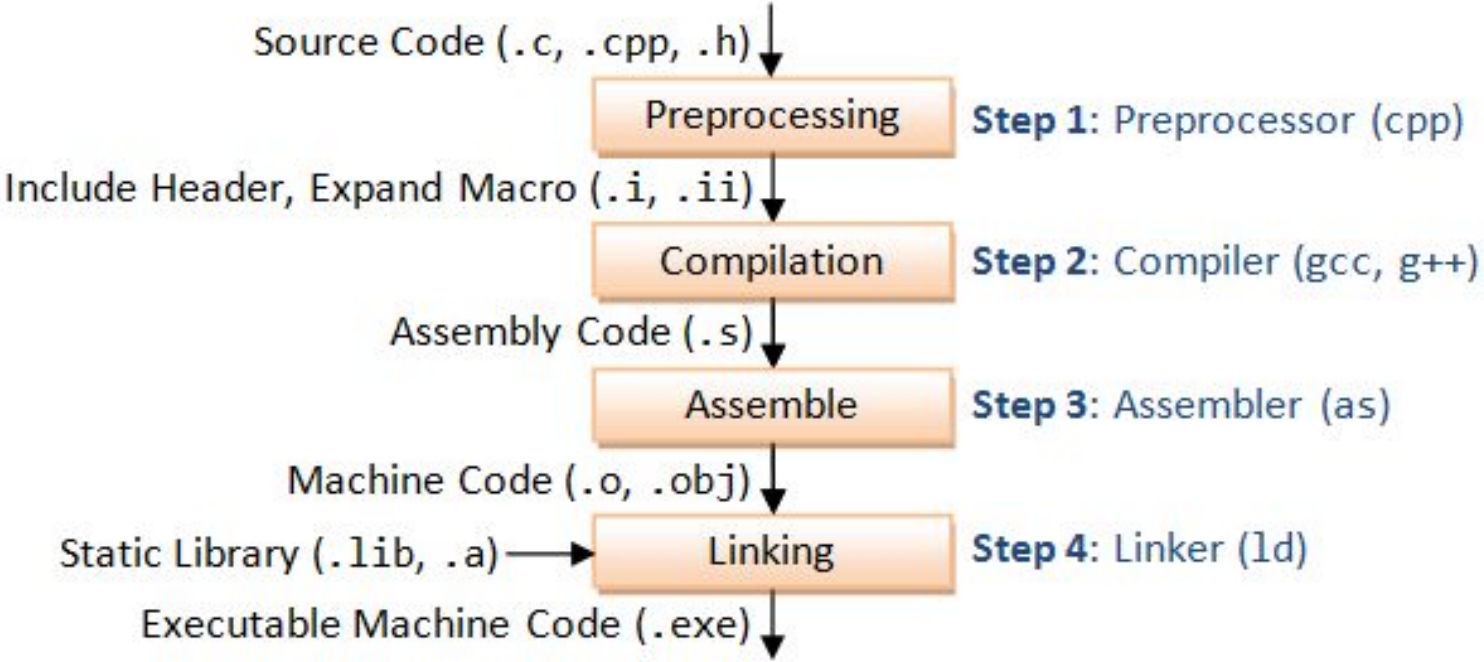
- Assembly code are readable versions of the native machine languages. And simplify coding considerably.
- Instead of numbers, operations like "add" and symbols like "r1" are used to represent instructions, registers, etc.

Machine code	Assembly code	Description
001 1 000010	LOAD #2	Load the value 2 into the Accumulator
010 0 001101	STORE 13	Store the value of the Accumulator in memory location 13
001 1 000101	LOAD #5	Load the value 5 into the Accumulator
010 0 001110	STORE 14	Store the value of the Accumulator in memory location 14
001 0 001101	LOAD 13	Load the value of memory location 13 into the Accumulator
011 0 001110	ADD 14	Add the value of memory location 14 to the Accumulator
010 0 001111	STORE 15	Store the value of the Accumulator in memory location 15
111 0 000000	HALT	Stop execution


Assembly Language
Machine Language
mov ecx, ebx
mov esp, edx
mov edx, r9d
mov rax, rdx
Assembler + Linker
100101011001
010011111011
111010101101
01010101010
Programmer
Processor

C

- C allows us to write programs using higher-level constructs you should be familiar with from Python e.g. If-statements, loops, variables
- A C *compiler* is an operating system program that converts C language statements into machine language equivalents.
- A compiler takes a set of program instructions as input and outputs a set of machine language instructions.
- We call the input to the compiler our *source code* and the output from the compiler the *binary code*.
- Once we compile our program, we do not need to re-compile it, unless we have changed the source code in the interim.

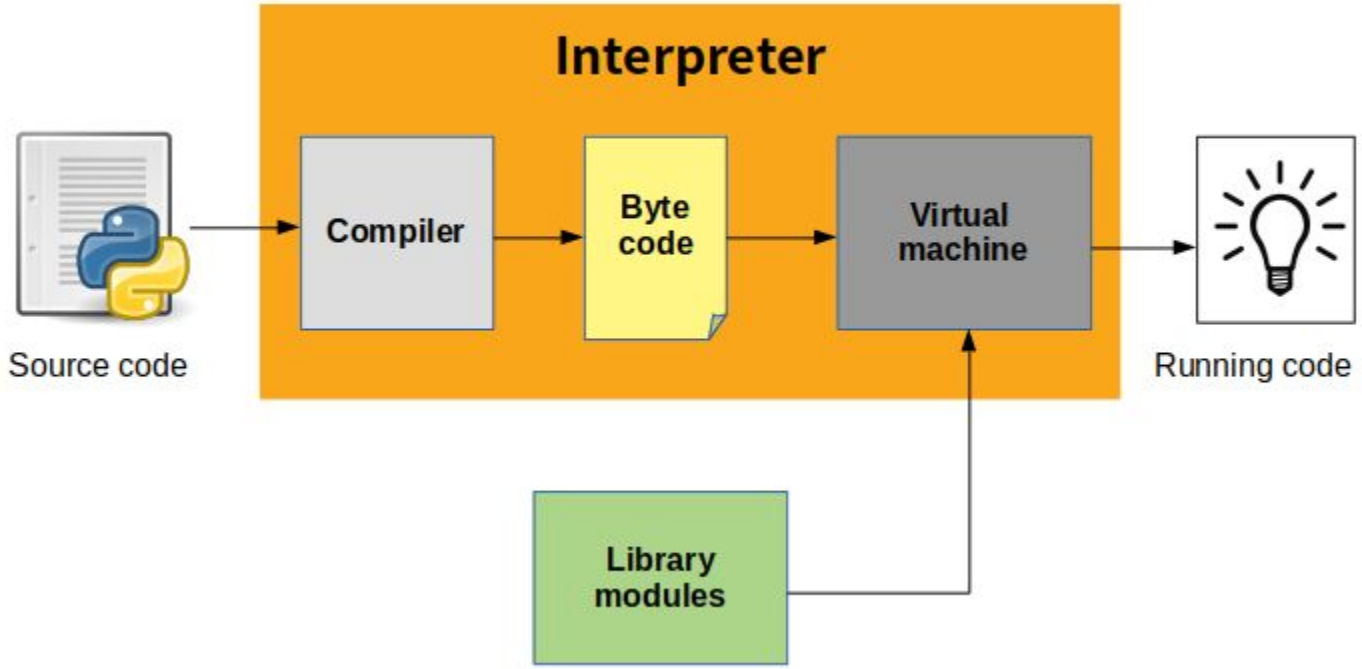

Source Code (.c, .cpp, .h)
Preprocessing
Step 1: Preprocessor (cpp)
Include Header, Expand Macro (.i, .ii)
Compilation
Step 2: Compiler (gcc, g++)
Assembly Code (.s)
Assemble
Step 3: Assembler (as)
Machine Code (.o, .obj)
Static Library (.lib, .a)
Linking
Step 4: Linker (ld)
Executable Machine Code (.exe)

C

- One of the **really** important properties of high-level languages like C is that they allow us (in theory) to write our code once and compile it for different machines
  - Before the program was very tied to the machine itself
- Algorithm representation is up a layer of abstraction from the machine instructions
  - Can be compiled for different machines!
- This means we can write a program once, and run it on different machines
  - At the time, this was a very, very big deal

How is it different from Python?

- Python is considered an **interpreted language**
- When you run a .py file or run a cell in a Jupyter notebook, Python code gets compiled to **bytecode**
- **Bytecode** is a language similar to machine code in spirit (basic instructions like add, subtract, etc)
- A special virtual machine program called an **interpreter** runs the bytecode, it does not run directly on the CPU
- In other words…
- After compilation, C programs run directly on the CPU
  Python programs run on an interpreter which is itself running on the CPU (another layer of abstraction)


Interpreter
Compiler
Byte code
Virtual machine
Source code
Running code
Library modules

Compilation vs Interpretation

- C programs run on the CPU
  - After they've been compiled…
- Python programs run on a program (virtual machine) that runs on the CPU
- This extra layer of abstraction slows things down
  - The Python code is compiled into bytecode, which is **then** run on the machine itself
  - There is an extra step in the process

Compilation vs Interpretation

- If we compile a C program, it gets turned into machine code ***for a specific type of machine***
  - If I want to run the program on different machines, I need to compile it for those machines too
  - I can't just take the one executable and run it anywhere
- A Python script however can be run on any machine that has a Python interpreter
  - I can take a Python program and run it anywhere (that has a Python interpreter)
- Interpretation might result in **slower** programs, but we gain much greater **portability** as a result

Compilation vs Interpretation

- Modern computers are now so fast that the cost of interpretation almost never matters anymore
- As a result, interpreted languages have taken over areas formerly dominated by compiled languages
  - e.g. JavaScript is an interpreted language, run by the web browser
  - Office 365 and other web applications are now standard ways of conducting work

Compilation vs Interpretation

- Another advantage of C code and compilation is that because the code runs on the hardware directly, it has **access** to the hardware directly
  - This is why despite C being a "high level programming language" we also say that it is a "lower level language" relative to Python
- So beyond applications which require high performance, we also see C being used in applications that require access to low-level features (such as accessing memory directly)
  - Operating systems
  - Embedded capabilities
  - Device drivers

Hello world in C

```
/* My first program          // comments introducing the source file
   hello.c          */

#include <stdio.h>          // information about the printf identifier

int main(void)              // the starting point of the program
{
        printf("This is C"); // send output to the screen

        return 0;           // return control to the operating system
}
```

main function

- Every C program includes a clause like **int main(void)**.
- Program execution starts at this line.
- We call it the program's entry point.
- The pair of braces follows this clause encompasses the program instructions.
- When we or the users load the executable code into RAM (**hello.out** or **hello.exe**), the operating system transfers control to this entry point.
- The last statement (**return 0;**) before the closing brace transfers control back to the operating system.

```
int main(void)          // program startup
{
        return 0;       // return to operating system
}
```

main function

- All C progams begin their execution at the top of the **main function**
- int main(void) { ... }
  - int means that the main function will return an integer values... C is much, ***much*** more specific about **types** than Python
  - void means that the function accepts no arguments
  - { ... } curly braces define a block of code that executes together... in this case the main function body
    - We'll use curly braces for if-statement, loop bodies, etc.
    - In Python, blocks were defined with ***indenting***

return 0;

- **return 0;**
  - The return means that the function returns 0
    In practice, this doesn't really matter much ever... the shell that runs the program receives this value as an "exit code" and it's standard to return 0
  - We could technically return other values to signal a problem has occurred, but again that's rare in practice

- What's with the ; characters at the end of lines?
  - ; is how we end a statement in C, i.e. an individual line of code that does something
  - This is another thing that Python handled with spacing!

“Hello, world” explained

- **#include <stdio.h>**
  - This includes the standard input output library
  - A library of functions like printf that allow us to do input and output
  - We'll include libraries like this to extend the functionality of C
  - Notice how ***we can't even print to the screen*** in the normal way without a library?
- **printf(...)**
  - Function that accepts a string argument and prints it to the screen (has some powerful features we'll discuss)
  - The "\n" stands for newline, and it creates a newline on the terminal when the string is printed

Pattern matching

gcc -o hello_world hello_world.c

- gcc is a C compiler available on Unix-like systems
- https://gcc.gnu.org /
- There are many other C compilers and IDEs
- If I save my file as hello.c, then I can compile it with the above command
- You do **need** to save your C files with a **.c extension**
- **-o hello** is specifying the name of the executable file that will result
- I can then run the executable file with ./hello

Compiler error messages

- If you have an error in your code..
- The compiler will tell us about the error
- The compiler tells us where the error is
- The compiler also tries to tell us what it thinks is wrong
- Often times the error message is cryptic and hard to understand... if this is the case, there is no shame in using Google / Stackoverflow at all

Types

- One big thing that distinguishes programming language is how they handle **types** for things like variables, function return value, etc.
- Python is a **dynamically** typed language
  - This means that variables don't have a type associated with them at compile-time (before the code is run)
  - So we can say:

    X=5
  - And the Python interpreter figures out that X is storing an integer... but we didn't have to say (as the programmer) that "X is for storing integers

Types

- C is a **statically typed** language
- When we create variables, we actually ***do*** specify what type of value the variable is intended to store
- There are more, but some standard C types include:
  - int: smaller integers
  - long: longer integers
  - float: decimal numbers (lower precision, less digits)
  - double: decimal numbers (higher precision, more digits)
  - char: characters (e.g. 'a', 'b')

C Basic Data Types	32-bit CPU		64-bit CPU	
	Size (bytes)	Range	Size (bytes)	Range
char	1	-128 to 127	1	-128 to 127
short	2	-32,768 to 32,767	2	-32,768 to 32,767
int	4	-2,147,483,648 to 2,147,483,647	4	-2,147,483,648 to 2,147,483,647
long	4	-2,147,483,648 to 2,147,483,647	8	- 9,223,372,036,854,775,808- 9,223,372,036,854,775,807
long long	8	9,223,372,036,854,775,808- 9,223,372,036,854,775,807	8	9,223,372,036,854,775,808- 9,223,372,036,854,775,807
float	4	3.4E +/- 38	4	3.4E +/- 38
double	8	1.7E +/- 308	8	1.7E +/- 308

Some C variable types

```
#include <stdio.h>
 int main(void)
 {
        int x = 4;
        long y = 1234567890;
        float a = 4.45;
        double b =  1234567890.0123456789;
        char c = 'd';
        return 0;
 }
```

```
#include <stdio.h>

int main(void)
{
        int x = 4;
        char a = 'a';
        int y;
        y=x+a;
        printf("y: %d\n",y);
        return 0;
}
```

I get an error if I try to do the same thing in Python...

Types

- Python is a more **strongly-typed** language
  - Even though variables don't have specific types…
  - ...Python is more strict about allowing you to use different types together in operations or in situations where another type is expected
- C is a more **weakly-typed** language
  - Even though variables have specific types…
  - ...C is less strict about allowing you to use types in situations where that type is not expected
  - C is often compared to using a scalpel... total control, but if you're not careful you can "cut yourself" easily

What actually gets stored in y?

Why does 4 + 'a' result in 101?!

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Types

- C treats the programmer like they know what they're doing
  - Result: more power, but waaaay easier to make mistakes
- Python prevents the programmer from making mistakes
  - Result: less power, but helps you to catch your own mistakes
- **Strong vs weakly typing** is not a black and white idea... sometimes C is more strict about types too
  - We'll run into these as we write our programs

Types

- printf("y: %d\n", y);
- %d is a **placeholder** that tells the printf function "you will be given an int to output at this place"
- We then supply the value (y) as an additional argument to the printf function
- We'll talk more about placeholders soon, but generally we need a reference to help us:
- https://en.wikipedia.org/wiki/Printf_format_string

C compilers

- We'll be using **gcc** officially in this course (it's there in pascal)
- Eventually we'll be using it to compile more complicated C programs contained across several files
- But there are plenty of other compilers, even some in- browser ones if you just want to try things:
- https://www.onlinegdb.com/online_c_compiler
- https://www.programiz.com/c-programming/online- compiler/
- https://www.tutorialspoint.com/compile_c_online.php
- **https://replit.com/languages/c**

Next we'll learn how concepts like control structures work in C...

```
#include <stdio.h>

 int main(void)
 {
        int number = 0;
        while(number !=-1)
        {
            printf("Please enter a number (-1 to quit)");
            scanf("%d",&number);
            if(number == -1)
               printf("Bye Bye!\n");
            else if(number>50)
               printf("Too high!\n");
            else
               printf("You got it!\n");
        }
        return 0;
 }
```

Python vs C

You'll find with control structures that though they might have different **syntax**, **semantically** they are very, very similar

Syntax is the actual text used to express things in a language

Semantics is the meaning and behaviour of those expressions