Lectures: C Programming Language

Programming Systems and Tools

State University of New York at Binghamton

January 25, 2024

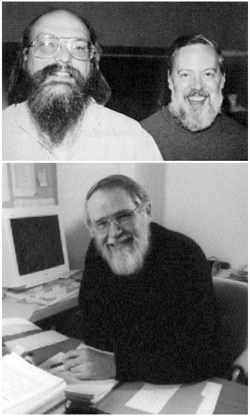
BINGHAMTON UNIVERSITY | THOMAS J. WATSON COLLEGE OF ENGINEERING AND APPLIED SCIENCE

A Tutorial Introduction

Outline

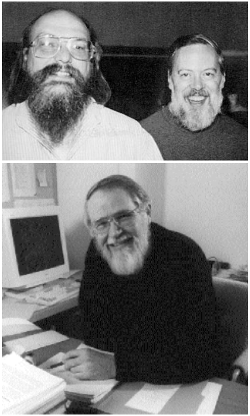
History of C

- 1969: Ken Thompson (left) develops the B programming language for the PDP-11 based on the BCPL language
- Early 1970s: Dennis Ritchie (right) joins Thompson at Bell Laboratories to develop Unix operating system
- 1972-1973: Ritchie develops C, a successor to B for writing Unix utilities. Thompson provides feedback to make it more useful as a systems programming language, particularly typing and structs
- 1973: The Unix kernel is rewritten in C, one of the first operating system kernels to not be written in assembly



History of C

- 1978: Brian Kernighan (bottom) and Ritchie write an informal specification of C ("C78"), which later becomes *The C Programming Language*
- 1989: C is standardized as ANSI X3J11 ("ANSI C"); Kernighan and Ritchie release a 2nd Edition.
- 1999: C99 is released with support for modern hardware. Most compilers today choose this as the default standard
- 2011: C11 is released to improved compatibility with C++
- 2018: C17 is released



Why C?

- Data model of C is based on bytes and words – tight integration with storage
- Syntax is based on assembly instruction set – easier to translate to machine code, but more human readable
- Type checking; C is *statically and weakly typed* – type enforcement reduces programmer errors when accessing memory
- Portability – C code can be targeted for any architecture provided your compiler has a code generator for that target

Compilation vs. Interpretation Models

Interpretation

Interpreter (runtime component) executes program statements/commands one at a time, updating internal state as it executes. Easy to debug and change. Examples: Ruby, Python, Javascript

Compilation

Translate statements into machine language, but does not execute it. This allows for optimization and reuse, but changes require recompilation. Examples: C and C++, Java, Rust

Compilation

- C is a compiled language, meaning we need a program that reads in a *source file* and outputs an *binary*, which can either be linked to other binaries, or executed.
  - Good: optimization is possible (output program is faster), many programmer errors are found at compile-time
  - Bad: requires static programming techniques
- We will use the GNU C Compiler toolchain (GCC), which has a driver program `gcc`.
- Why is it called a toolchain?

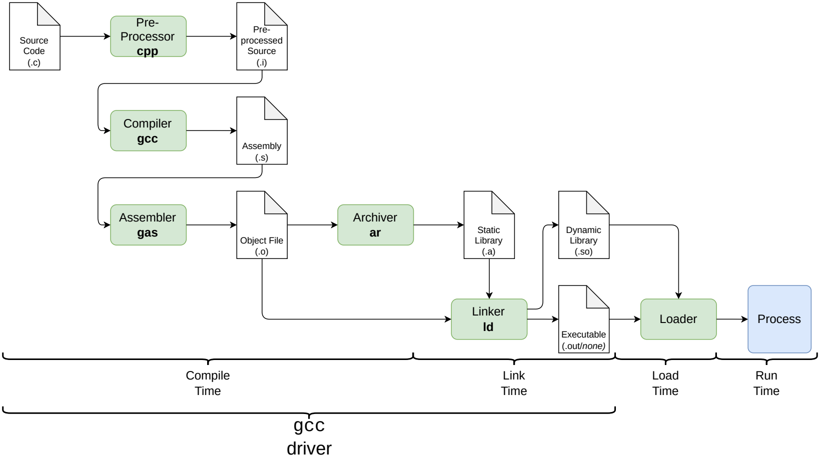
Compilation

- C is a compiled language, meaning we need a program that reads in a *source file* and outputs an *binary*, which can either be linked to other binaries, or executed.
  - Good: optimization is possible (output program is faster), many programmer errors are found at compile-time
  - Bad: requires static programming techniques
- We will use the GNU C Compiler toolchain (GCC), which has a driver program `gcc`.
- Why is it called a toolchain? GCC actually consists of parts other than the compiler, all of which are invoked by the driver program as needed ... more on this later!
- Examples:
  - `gcc source.c` →produces `a.out`, an executable binary
  - `gcc source.c -o source` →produces `source`, an executable binary
  - Note that Unix/Linux doesn't care about file extensions for determining executable permissions!

Libraries

- GCC can also output libraries from source code, eliminating some code repetition
- Libraries can also help abstract away complexity – for example, the C Standard Library handles memory management with the operating system, and provides a single function call to get heap memory for usage (`malloc()`)
- Still need to declare those functions (and potentially tell the compiler to have the linker link to the library): use *header files*

Lifecycle



Source Code (.c)
Pre-Processor **cpp**
Pre-processed Source (.i)
Compiler **gcc**
Assembly (.s)
Assembler **gas**
Object File (.o)
Archiver **ar**
Static Library (.a)
Dynamic Library (.so)
Linker **ld**
Executable (.out/*none*)
Loader
Process
Compile Time
Link Time
Load Time
Run Time
gcc driver

Hello World

Traditionally, the first program one writes in any new language is Hello World:

```
/* hello.c */        // This is a comment

#include <stdio.h>   // include information about standard library

void main(){         // main() is the "entry point" function
                     // our program begins here, and ends when main()
                     // returns. main() takes no arguments and
                     // returns no value ("void").

  puts("Hello World!\n"); // main() calls the library function puts()
                          // to print a sequence of characters.
                          // "\n" is the newline character.

}                    // statements in a function are enclosed in braces
```

Comments

- Comments explain what a program does
- Characters between `/*  */` or on a line after `//` are discarded by the preprocessor
- They may appear anywhere a whitespace character like `'\t'`,`'\n'`,␣ can appear

```c
// hello.c

...

/* print Fahrenheit-Celsius table
   for fahr in 0,20,...,300 */
```

Variables

- Variables create storage for data
- Variables must be *declared* to the compiler, which announces their properties
  - Includes the type (machine-dependent size) and identifier (name)
  - To set the value (stored data), use an *initializer* or *assignment* statement
  - If a variable is read before its value has been initialized or assigned, you get whatever data was previously stored at that address!

```
int fahr, celsius; //uninitialized variables
int lower = 0;     //initialized variables
int upper = 300;
```

Arithmetic Expressions

- C can perform arithmetic on any of the numeric types
- Warning! Unsigned arithmetic is performed mod2^n:
  `UINT_MAX + 1 == 0, 0 - 1 {}== UINT_MAX`
- Warning! Signed arithmetic overflows with undefined behavior according to the rules of the representation:
  `INT_MIN - 1 == INT_MAX` . . . it could even be optimized out!

Operator	Operator name	Example	Result
`+`	unary plus	`+a`	the value of **a** after promotions
`-`	unary minus	`-a`	the negative of **a**
`+`	addition	`a + b`	the addition of **a** and **b**
`-`	subtraction	`a - b`	the subtraction of **b** from **a**
`*`	product	`a * b`	the product of **a** and **b**
`/`	division	`a / b`	the division of **a** by **b**
`%`	remainder	`a % b`	the remainder of **a** divided by **b**
`~`	bitwise NOT	`~a`	the bitwise NOT of **a**
`&`	bitwise AND	`a & b`	the bitwise AND of **a** and **b**

Control Flow

Since we want to repeatedly perform the same task, stopping when a particular value is calculated, we can use *loops* to control flow of the program.

1. The *condition* in the parenthesis is tested
2. If the condition is true, the *body* is executed
3. If the condition is false, the loop ends, and the next statement after the loop body is evaluated

```c
while (fahr <= upper){
  // print a row in the table
}
```

Note the indentation: C is whitespace-agnostic. The tab is only for our readability!

Symbolic Constants

- To make our program more readable, we can replace "magic numbers" with symbolic constants using the **preprocessor**.
- Preprocessor definitions take the following format:

  #define NAME replacement-expr

```c
#define STEP 20
```

printf

`int printf(const char* message, ...)` is a general-purpose output formatting function.

- `printf()` takes a *format string* containing zero or more *format specifiers* beginning with %
- Each format specifier can be formatted using numeric and alphabetic modifers
- Each format specifier must be substituted with a variadic argument after the format string

```
// Print header of table
printf("Fahrenheit\tCelsius\n");
printf("----------\t-------\n")

// Print row of table, right align with
// its column and left-fill with spaces
printf("%10d\t%7d\n", fahr, celsius);
```

Printing

- We already encountered the `int puts(const char *)` function, which will print a string and end with a new line character – but it only works for string literals. What if we want to print formatted text?

Printing

- We already encountered the `int puts(const char *)` function, which will print a string and end with a new line character – but it only works for string literals. What if we want to print formatted text?
- `printf` is a standard library function in **stdio.h** that takes two parameters: a format string and a variable list and writes a formatted output to stdout.
  - Format strings are just like any other string literal, except they can be formatted using *specifiers*
  - Specifiers not only specify output formats, they can also specify precision and padding!

Printing

```c
// Prototype
int printf( const char *format, ... );

// Examples
// Print a string literal
printf("Hello world!\n");

// Print an decimal integer, a hexadecimal
// integer, and a float to 2 places
printf("x=%i, y=%x, z=%.2f\n", 2, 22, 2.5);

// Print an address
printf("address of foo: %p\n", &foo);
```

Note that we can print multiple values in the variables list (`printf` is a *variadic function*), but the number of values must equal the number of format specifiers!

Format Strings

Format Specifier	Meaning
%d	Decimal integer (base-10)
%o	Octal integer (base-8)
%x	Hexadecimal integer (base-16)
%c	byte as ASCII character
%s	ASCII string
%p	Pointer
%%	A percentage sign

Format Strings

Format String	Meaning
%d	print as decimal integer
%6d	print as decimal integer, at least 6 characters wide (space padded)
%f	print as floating point
%6.f	print as floating point, at least 6 characters wide (space padded), with no digits after the decimal place
%.2f	print as floating point, with 2 digits after the decimal place
%6.2f	print as floating point, at least 6 characters wide, with two digits after the decimal place

Arrays

- Arrays are an object whose elements are contiguous in memory
- All of the elements have to be the same type (unlike Python's containers such as lists)
- The size has to be known at compile time and cannot be resized (unlike C++/Java vectors or Python lists)

Declaring and Initializing Arrays

- Declare with type, name and number of elements:
  - `int c[10];`
  - `char hello[6];`
- We can initialize with scalar statement when declaring:
  - `int nums[5] = {1,2,3,4,5};`
  - If number of elements isn't declared, but an initializer scalar is available, the compiler can infer the number of elements for you. Compiler error otherwise!
  - Not enough initializers? C zero-initializes arrays.

    `int nums[5] = {1}; //{1,0,0,0,0}`
  - Too many initializers? Compiler error!
  - Cannot use scalar expression separately after declaration.

Accessing Arrays

- Arrays are zero-indexed
- Each element of the array has a subscript index, accessed using the member access operator: `arr[idx];`
- Member access operator is *syntactic sugar* for `*(&arr + sizeof(arr) * idx);`
  - We'll talk more about * and &, but in short they mean "give me the thing at address" and "what is the address of?"

```
int arr[] = {1,2,4,8,16}; // same as int arr[5] = {1,2,4,8,16};
int pow = arr[3]; // What is the value of pow?
arr[2] = 0; // What are the contents of arr?
```

Array Length and sizeof

- `sizeof` is a unary operator that returns the size in bytes of the operand
- Array size is the total size of the array in bytes
- Array length is the array's size divided by the elements' type's size

```c
int nums[] = {1,2,3,4,5};

// What three values print out?
printf("Size of nums:%i, Size of int:%i, Length of nums:%i\n",
  sizeof(nums),
  sizeof(int),
  sizeof(nums)/sizeof(int));
```

Character Arrays

- `char[]` is often referred to as a string. We can initialize them two ways:
  - `char hello[] = {'H','e','l','l','o'}; //scalar initializer`
  - `char hello[] = "Hello"; //syntactic sugar`
- Where's the end of a character array?
  - Strings must end in a null byte (\0), but character arrays might not!
  - Null byte is implicitly added when using string initializer:
    `char hello[] = "Hello" // {'H','e','l','l','o', \0};`

Character Arrays

- Characters are just 1-byte numbers, so strings are just arrays of numbers!
- We can perform arithmetic on character values. For example, `'A'` is 65, `'a'` is 97, therefore capitalization is subtracting 32 and alphabetization can be done by comparing the character values!

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Bounds Checking

- Bounds: the start and end of the array
- If size omitted, initializers determine the bounds:
  `int n[] = { 1, 2, 3, 4, 5 };` $\rightarrow$ 5 initializers, therefore 5 element array, therefore bounds are 0..4
- C arrays have no bounds checking $\rightarrow$ for array of length 5, arr[5] will return a value, but it's undefined behavior!
- Recall we said character arrays are null terminated. What happens if we try to write over that last zero in the array?

Lab: Arrays and Operators

In a file called **arrays.c**, add the code necessary to:

1. Create two arrays:
   - An array of five integers of your choice
   - A string of your choice
2. Print the following to stdout without using a string literal containing the answer:
   - The size of an int type
   - The size of your array
   - The length of your array
   - The last number in your array
   - The size of your string
   - The length of your string
   - The contents of 1 byte of memory (as a character) after your string
3. Compile your program into a executable called **arrays**
4. In the **notebook.md**, record and explain your findings
5. Demonstrate your working **arrays** executable.

Functions

- To avoid repeating ourselves, we can place subroutine code into reusable blocks with a name, called a function.
- Functions take the following format:

  `return_type identifier(arguments...){block-expr}`

```c
/* Conversion formula */
int ftoc(int f){
  return 5 * (f - 32) / 9;
}
```

Complete Tutorial Program

```c
#include <stdio.h>
#define STEP 20

/* Conversion formula */
int ftoc(int f){
  return 5 * (f - 32) / 9;
}

/* print Fahrenheit-Celsius table
   for fahr in 0,20,...,300 */
int main(){
  int fahr, celsius; //uninitialized variables
  int lower = 0;     //initialized variables
  int upper = 300;

  fahr = lower;
  printf("Fahrenheit\tCelsius\n");
  printf("----------\t-------\n")
  while (fahr <= upper){
    celsius = ftoc(fahr);
    printf("%10d\t%7d\n", fahr, celsius);
    fahr = fahr + STEP;
  }
  return 0;
}
```

Lab: Hello World

Traditionally, the first task in any new language is to print "Hello world!" to the console.

1. Create **hello.c** in a text editor:

```c
#include <stdio.h>

/* print "Hello world!" to the console */
void main(){
    puts("Hello world!");
}
```

2. Compile and run in the terminal:

```
mcole8@remote:~$ gcc hello.c -o hello
mcole8@remote:~$ ./hello
Hello world!
```

Lab: Hello World

3. Create **notebook.md**. In this file, using your own words, describe what compiler errors - if any - occur when you make each of these changes individually:
   - Omit the include statement (the entire line 1)
   - Omit the comment (the entire line 3)
   - Delete the block braces for the `main()` function on lines 4 and 7.
   - Change the signature of the `main()` function from `void main()` to `int main()`.
4. After changing the signature to `int main()`, add a statement `return 0;` after line 5. Run your executable again, then use the shell to observe **hello**'s return value.

```
mcole8@remote:~$ ./hello
Hello world!
mcole8@remote:~$ echo $?
0
```

Lab: Hello World

5. Try different return values including no return value at all: `return;` What happens when you call `echo $?` ?
6. Demonstrate your working **hello** executable to the Teaching Assistant or Lecturer and discuss your findings.