CS265
Advanced Programming Techniques

Intro to C

C, 1972

Dennis Richie

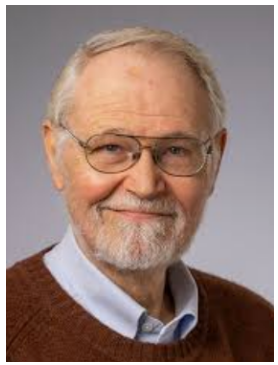
AT&T Labs

K&R C, 1978

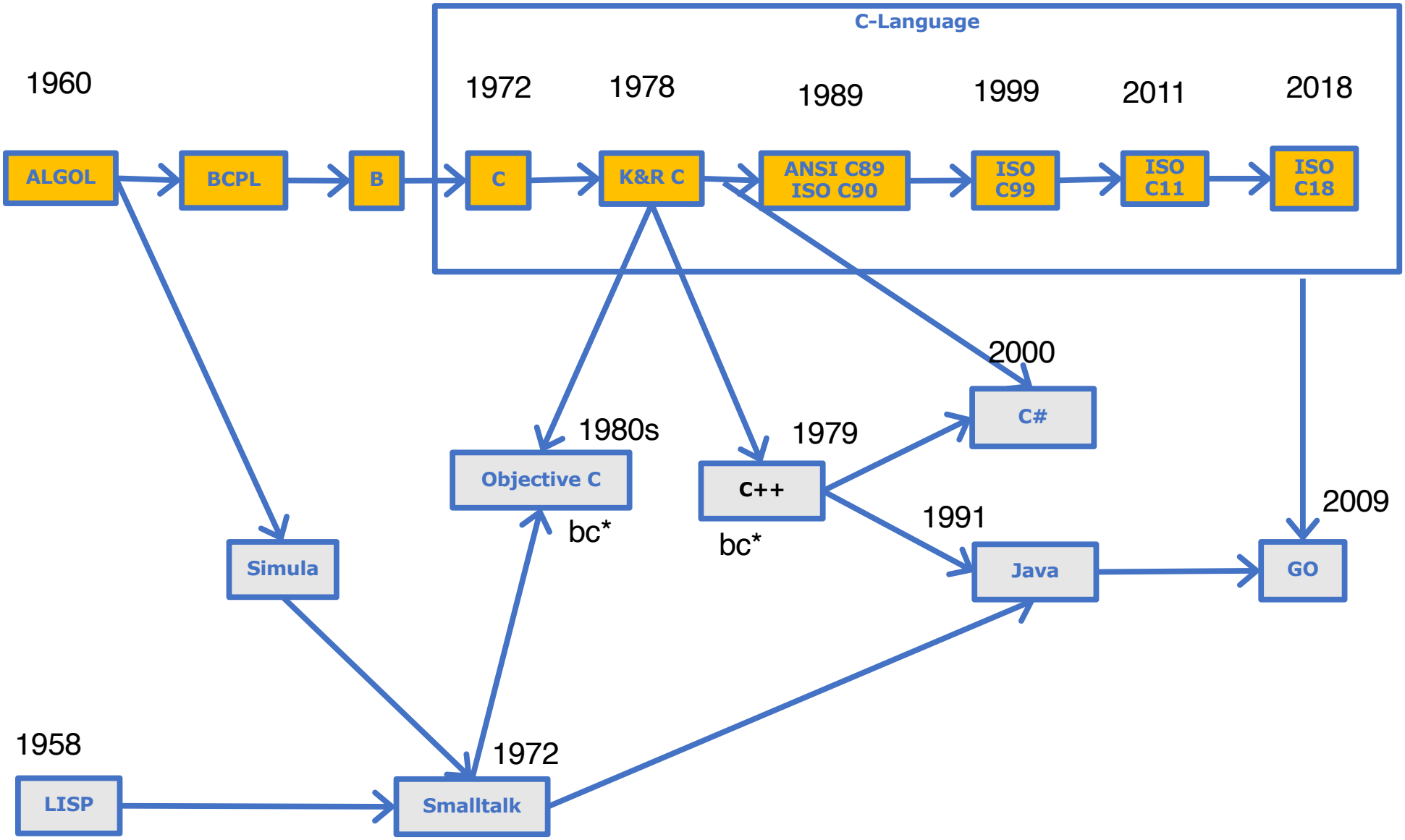
Dennis Richie

Brian Kerninghan

AT&T Labs



History of C



bc = backwards compatible with C

C-family programming languages

- C, 1972
- C++, 1979
  - Directly derived from C
  - C with classes
- Objective C, 1980s
  - Influenced by C & Smalltalk
  - Heavily used at NeXT and then Apple (until recently displaced)
- C#, 2000
  - Directly derived from C
  - OO language, very similar to Java
  - Microsoft's answer to Java's popularity in the early 2000s
- Go, 2009
  - Out of Google's 20% project
  - C for the 20th century
  - Java like but without inheritance
  - Blends object-oriented programming with functional programming
  - Parallelism and modern data types (gifs, slices, maps, JSON, HTML,..)

The players

Ken Thompson
ATT Labs
Google

Creates UNIX
Creates the B Programming Language

Dennis Richie
ATT Labs

Creates the C Programming Language

The players

Ken Thompson
ATT Labs
Google

Dennis Richie
ATT Labs

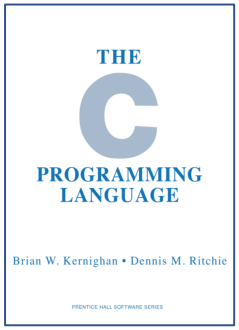
Co-create UTF-8 character encoding

Later, co-create the GO programming language (with Robert Griesemer)

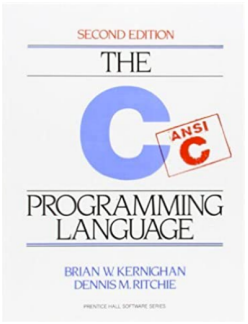
Rob Pike
Google

The books

Ken Thompson
ATT Labs
Google

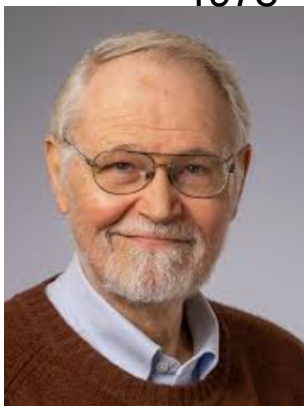


1978

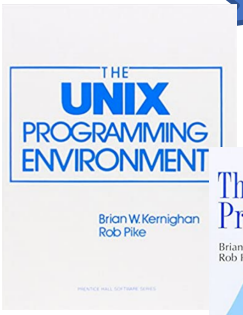


1988

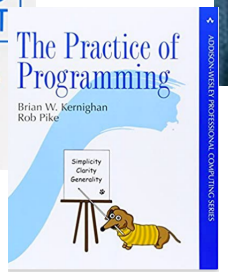
Dennis Richie
ATT Labs



Brian Kernighan
ATT Labs
Princeton U



1983

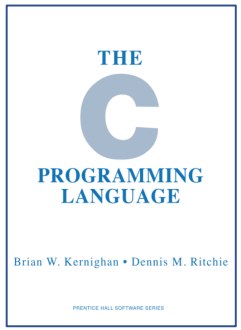


1999

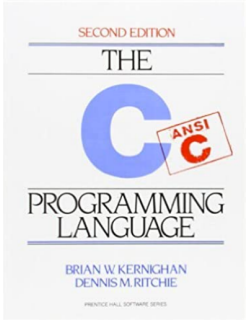
Rob Pike
Google

The books

Ken Thompson
ATT Labs
Google

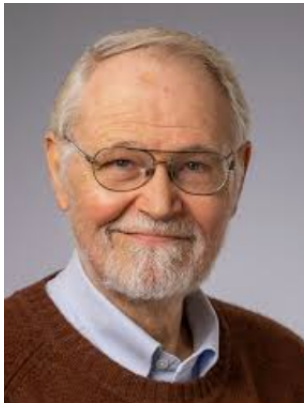


1978



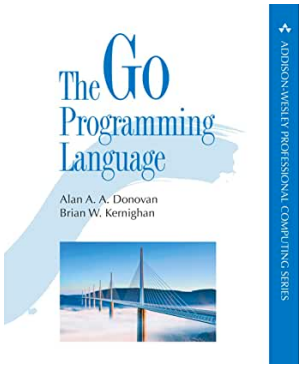
1988

Dennis Richie
ATT Labs

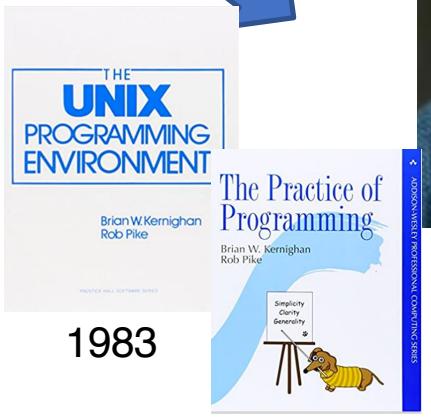


Brian Kernighan
ATT Labs
Princeton U

Alan Donovan
Google



2015



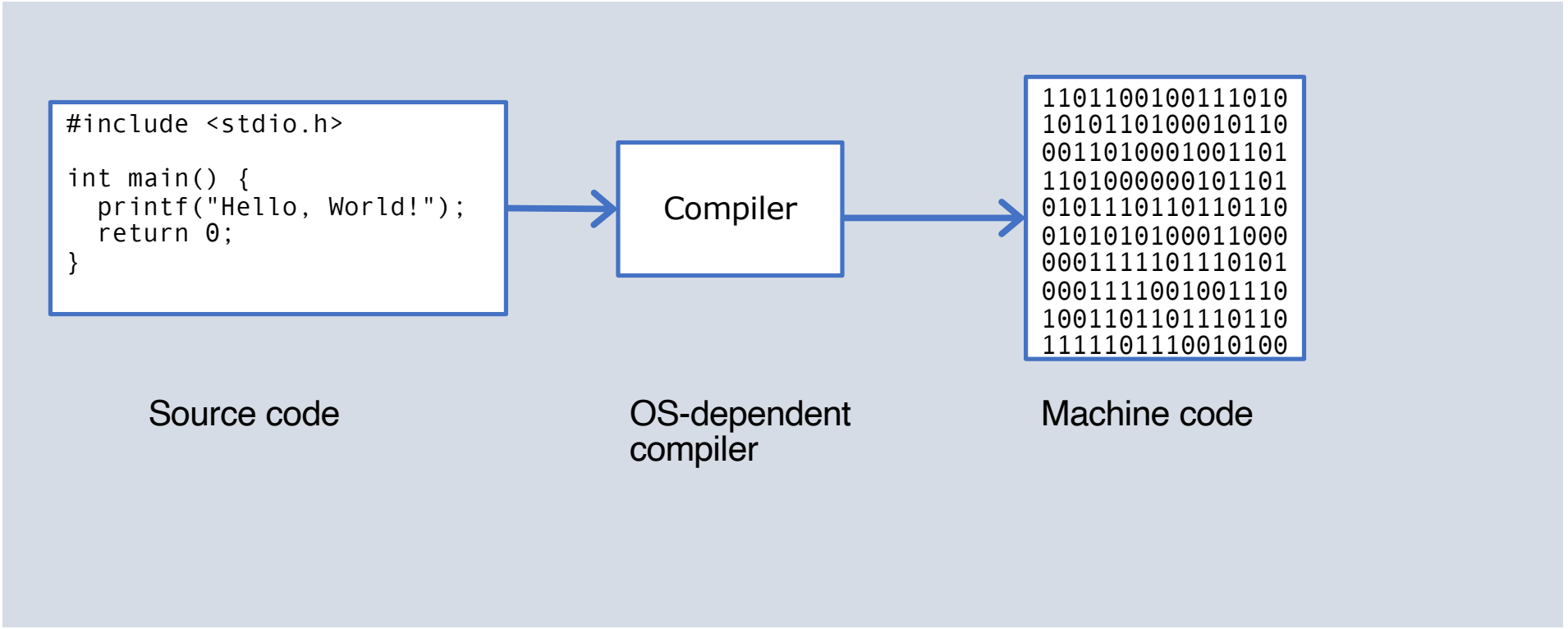
1983

Rob Pike
Google

C vs other programming languages

- C is a compiled language that compiles to machine code
  - Unlike Java that compiles to bytecode and then it gets interpreted
  - Unlike Python which is an interpreted language
  - Unlike bash and AWK which are scripting languages
- C is a procedural language
  - Unlike Python and Java that are object-oriented
- C is a low level programming language
  - Closer to the HWD and the OS
  - Unlike Python and Java that are both high-level (you can't access the OS directly from Java or Python)
- C is a systems language
  - Not a lot of bells and whistles
  - Unlike Java and Python that are more for applications programming and data science
- C is lean and efficient

C is a Compiled Language
Compiled languages - compile to machine code



Examples

- C, C++, Go, Haskell

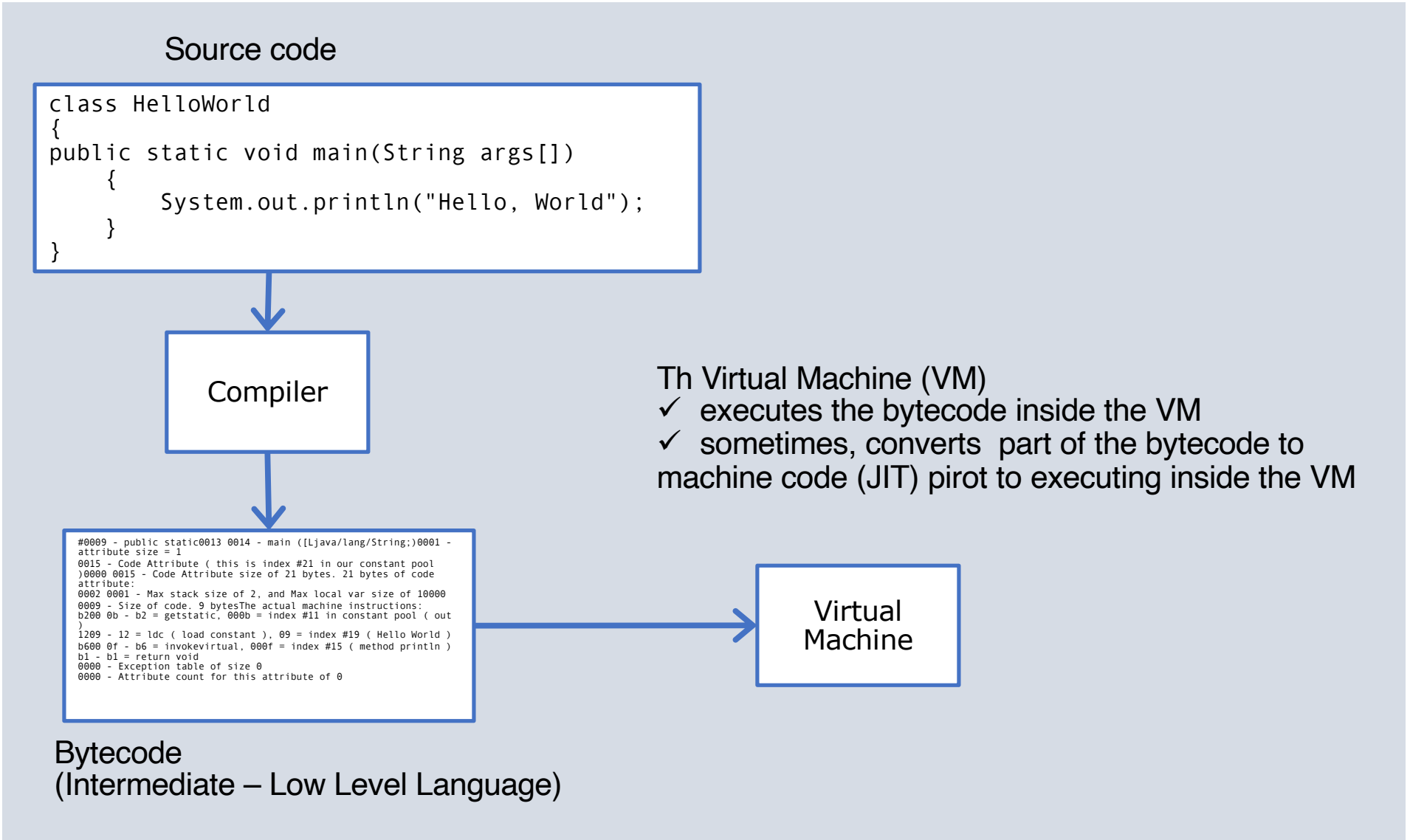
Interpreted languages & scripting languages do not translate to machine code

Source code

The interpreter is
a special run-time environment that
executes the commands

- Examples:
  - Javascript, python, bash, awk, pearl, ruby, …

Some interpreted languages compile first to bytecode

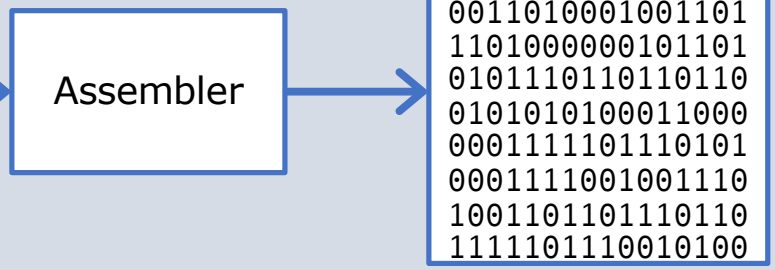


Examples

- Java, C#, Python, …

Many C compilers compile first to assembly language and then to machine code (but this happens automatically, under the hood)

```
.model small
.data
 opr1 dw 1234h
 opr2 dw 0002h
 result dw 01 dup(?),\'$\'
code
 mov ax,@data
 mov ds,ax
 mov ax,opr1
 mov bx,opr2
 clc
 add ax,bx
 mov di,offset result
 mov [di], ax
 mov ah,09h
 mov dx,offset result
 int 21h
 mov ah,4ch
 int 21h
end
```




```
1101100100111010
1010110100010110
0011010001001101
1101000000101101
0101110110110110
0101010100011000
0001111101110101
0001111001001110
1001101101110110
1111101110010100
```

Assembly Language

- Low Level Programming Language
- Processor specific

Machine Code

- Binary code (0s & 1s)
- Understood by computers
- Processor specific

Unix and C

- The first version of Unix was written in assembly language
- C was born out of the desire to write Unix in a high-level language
- C translates to assembly language as part of its compilation process
  - not apparent to the user of the compiler

Typical C Program

```
#include <stdio.h>

/* Brian Kernighan wrote the  */
/* first Hello World program  */
/* main returns 0 for success */


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

hello_world.c

C programs contains a `main` global function that has one of the following two forms

```
int main() { body }

int main(int argc, char *argv[]) { body }
```

`stdio.h` is not a library, but a header file containing C function declarations

C Program – How to Compile

```
#include <stdio.h>

/* Brian Kernighan wrote the  */
/* first Hello World program  */
/* main returns 0 for success */


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

hello_world.c

Two options

`gcc hello_world.c` generates default executable `a.out`

`gcc hello_world.c -o hello_world` generates executable `hello_world`

C Program – How to run

```
#include <stdio.h>

/* Brian Kernighan wrote the  */
/* first Hello World program  */
/* main returns 0 for success */


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

hello_world.c

Depending on the name of the executable created by the compilation

`./a.out` if default executable `a.out`

`./hello_world` if named executable `hello_world`

a.out originally stood for assembler output

C Compiler - Under the hood

There are four steps

- Pre-processing
- Compiling
- Assembling
- Linking

C Program – Step 1 – Preprocessing

hello_world.c

```
#include <stdio.h>

/* Brian Kernighan wrote the  */
/* first Hello World program  */
/* main returns 0 for success */


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

output of preprocessor

```
…


extern int printf (const char
*__restrict __format, ...);
…


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

- Preprocessor
  - Removes comments
  - Locates the `include` file `<stdio.h>` in `/usr/include/stdio.h`
  - Replaces the include line with the contents of the file
  - Performs many other additional pre-processing directives!
- No need for programmers to invoke the preprocessor explicitly.
- Although you can, using the `-E` option

```
gcc -E hello_world.c
```

C Program – Step 2 – Compiling

output of preprocessor

```
…


extern int printf (const char
*__restrict __format, ...);
…


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

hello_world.s

```
    .file "hello_world.c"
    .text
    .section .rodata
.LC0:
    .string  "Hello World"
    .text
    .globl   main
    .type main, @function
main:
.LFB0:
    .cfi_startproc
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq  %rsp, %rbp
```

- Compiles into assembly code
- The output contains the signature of `printf` (definition), not the body
- No need for programmers to invoke this step explicitly.
- Although you can, using the `-S` option

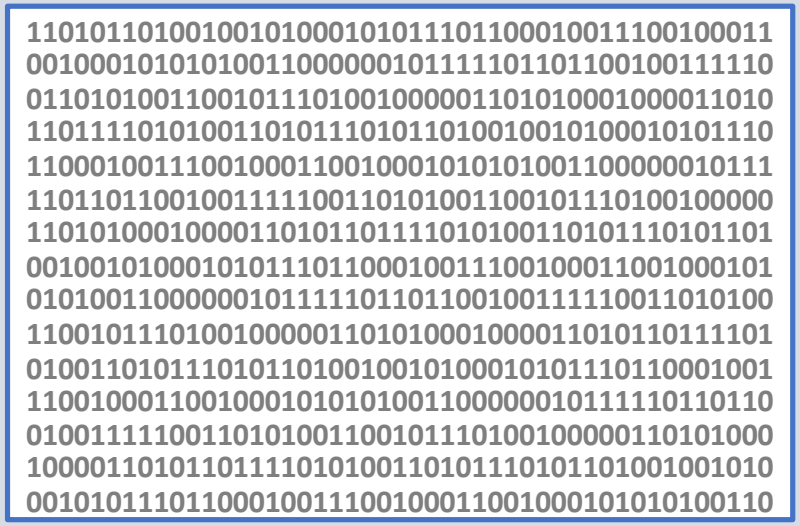
```
gcc -S hello_world.c
```

C Program – Step 3 – Assembling

Hello_world.s

```
    .file "hello_world.c"
    .text
    .section .rodata
.LC0:
    .string  "Hello World"
    .text
    .globl   main
    .type main, @function
main:
.LFB0:
    .cfi_startproc
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq  %rsp, %rbp
```

hello_world.o

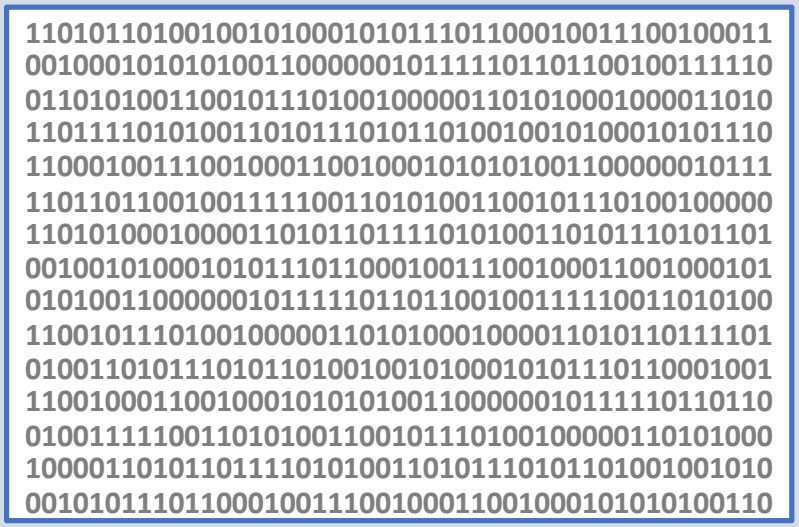


- Translates the assembly code to machine code
- Note that there is no translation at this point of the code for `printf`
- No need for programmers to invoke this step explicitly.
- Although you can, using the `-c` option

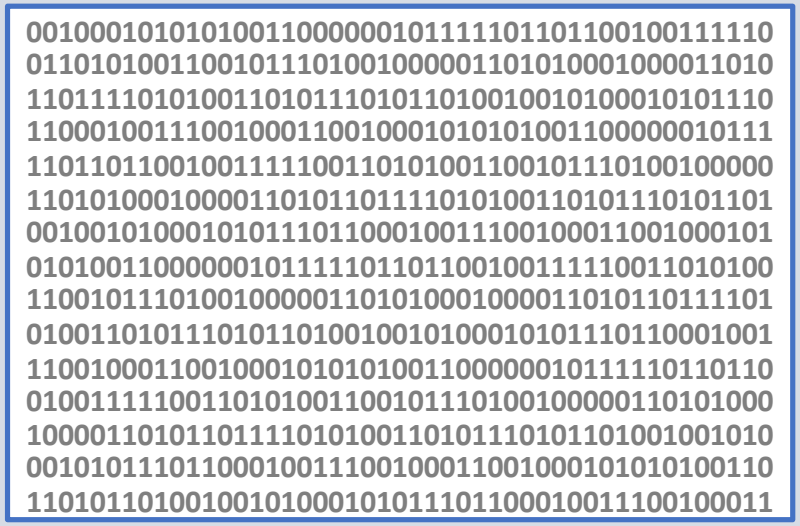
```
gcc -c hello_world.s
```

C Program – Step 4 – Linking

hello_world.o



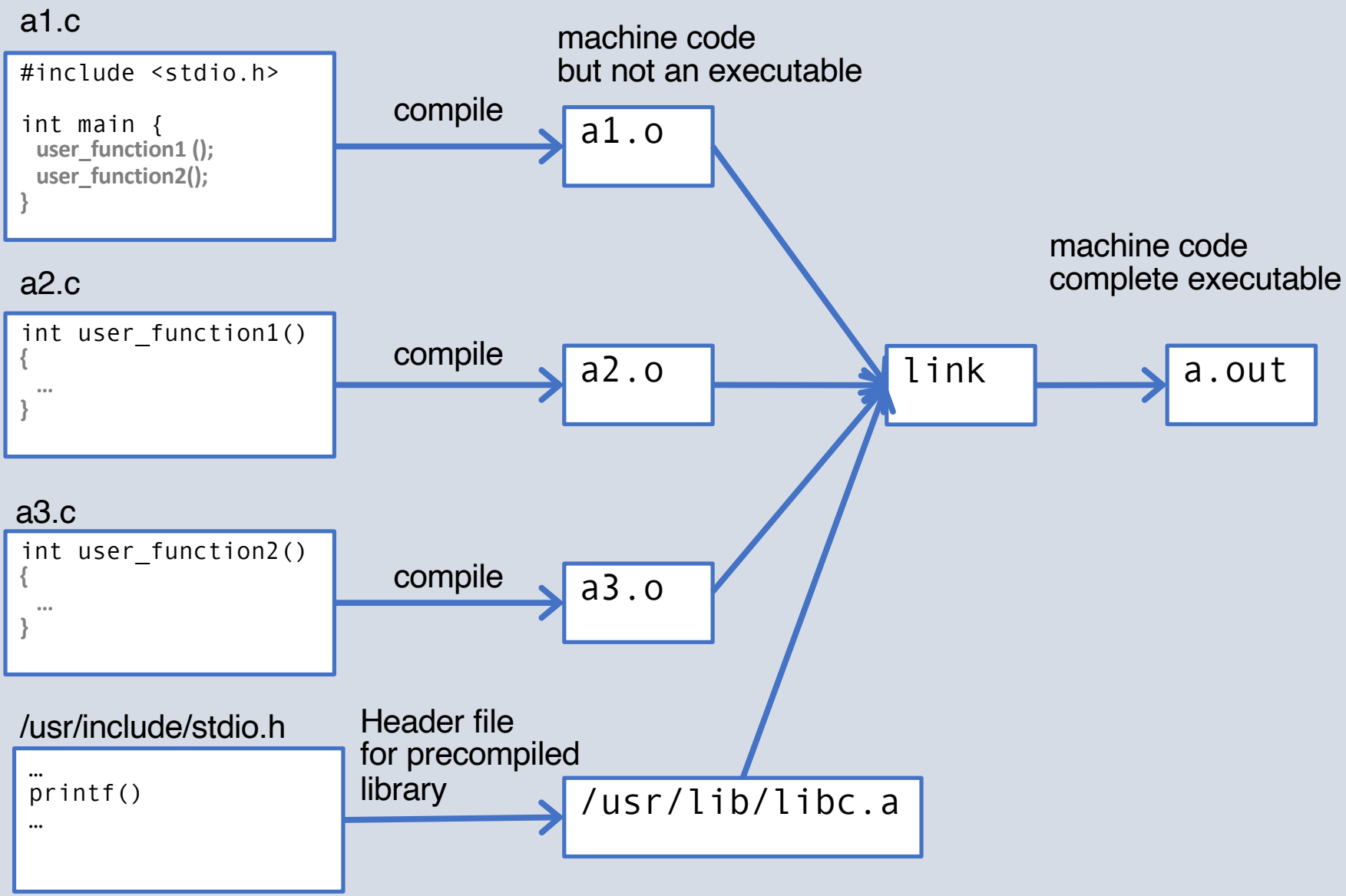
hello_world



- Linker
  - Fetches machine language code from the standard C libraries to make the program complete
  - This step is needed both for external build-in functions and external user-defined functions
- The compiler's `-o` option invokes the linker

```
gcc hello_world.o -o hello_world
```

Compiling vs Linking



C Example from K&R Book – Character copying

```
#include <stdio.h>

/* Program from K&R book */
/* Copy characters from input to output  */

main()
{
    int c;

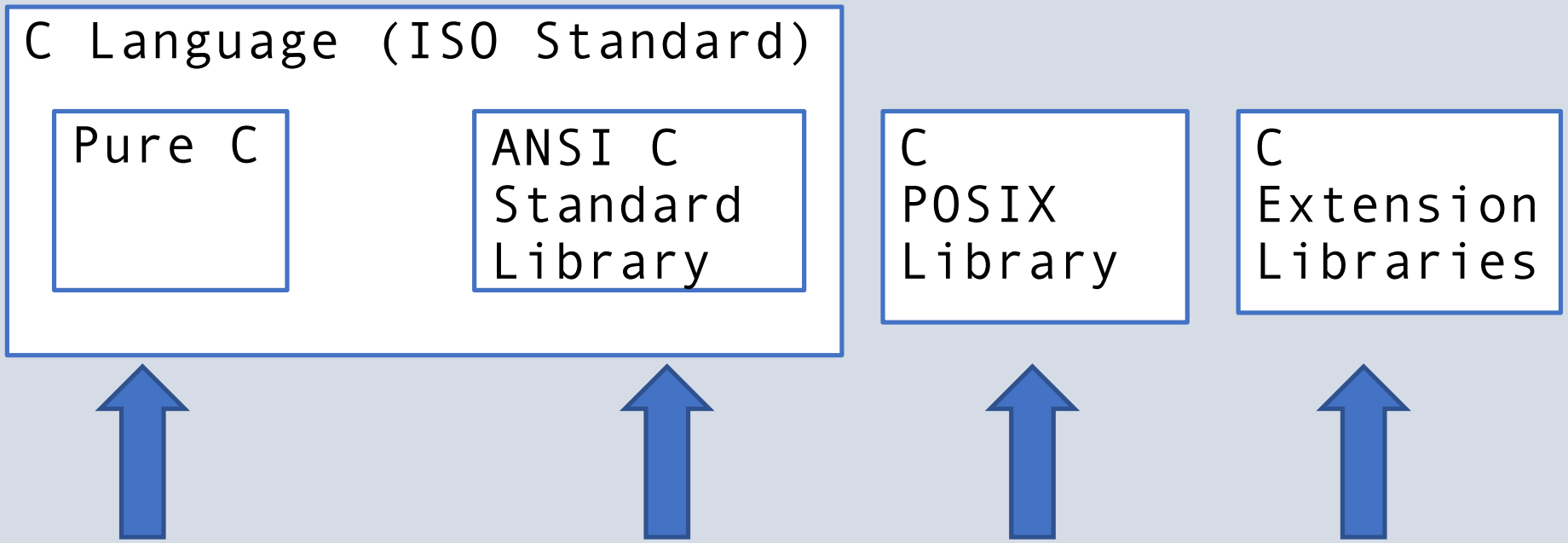
    while ((c = getchar()) != EOF)
        putchar(c);
}
```

This little program can teach us a lot about C:

- `stdio.h` part of the standard library, a header file

- `main()` can be defined without any arguments
- all statements must end with a `;` (not optional)

C Standard Library and other C Libraries



C Language Proper

- Defined in the Standard for C
- Includes standard I/O, etc.

`libc.a`

POSIX Spec

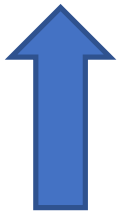
e.g. regular expressions, sockets, etc.

- Features not in the standard
- E.g. variable length arrays , vectors, etc.

What's in a library?

The C Standard Library (`libc.a`) defines functions, type definitions and macros

`<stdio.h>`	input/output processing functions	
`<stdlib.h>`	standard library (memory mgmt, process control, etc.)	
`<string.h>`	string handling functions	
`<math.h>`	common mathematical functions	
`<assert.h>`	assertions for detecting errors	
`<stdbool.h>`	Boolean functions	(since C99)
`<threads.h>`	Threads functions	(since C11)
`<uchar.h>`	Unicode char functions	(since C11)



There are many others

✓ **Fun fact: C Standard Library comes with UNIX, not C!**

header files or C headers

Back to the K&R Book Example – Character Copying

file_copy.c

```
#include <stdio.h>

/* Program from K&R book */
/* Copy characters from input to output  */

main()
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(c);
}
```

More things this little program can teach us about C

- `getchar()` and `putchar()` – simple I/O functions that read/ write one character at a time
- `c` is defined using an `int` type, not a `char` type – why?
- Because `c` will get the value of `EOF` eventually
- `EOF` is a predefined constant, it is `-1` (which is not a character), so we can't use a `char` to hold it

`char` vs `int` Data Types in C

Type	Storage size
char	1 byte
unsigned char	1 byte
signed char	1 byte
int	2 or 4 bytes
unsigned int	2 or 4 bytes
short	2 bytes
unsigned short	2 bytes
long	8 bytes
unsigned long	8 bytes

designed to hold a single ASCII char, not a Unicode
a char can hold 256 (2^8) different numbers

(there are some variations based on OS and compiler)

Character Sets and Character Encodings

Character Sets describe the "valid" characters

- ASCII
- Unicode

Character Encodings are used to represent characters in binary form

- ASCII – encodes ASCII characters in 7 bits
- ANSI – encodes ASCII characters in 8 bits
- EBSIDIC – alternate encoding for ASCII
- UTF-8 : encoding for Unicode; uses 1 byte to represent ASCII chars, 2 bytes for more alphabetic blocks and 3 or 4 for the rest of characters, e.g., Letter D is 0x44

ASCII Characters & ASCII ANSI Encoding

e.g., the character 'A' is encoded with the decimal number 65

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: http://www.asciitable.com/

Unicode Characters (example)

Unicode contains over 143,000 characters covering 154 modern and historical scripts

Code	Glyph	Decimal	HTML	Description	#
U+0100	Ā	256	Ā	Latin Capital Letter A with macron	0192
U+0101	ā	257	ā	Latin Small Letter A with macron	0193
U+0102	Ă	258	Ă	Latin Capital Letter A with breve	0194
U+0103	ă	259	ă	Latin Small Letter A with breve	0195
U+0104	Ą	260	Ą	Latin Capital Letter A with ogonek	0196
U+0105	ą	261	ą	Latin Small Letter A with ogonek	0197
U+0106	Ć	262	Ć	Latin Capital Letter C with acute	0198
U+0107	ć	263	ć	Latin Small Letter C with acute	0199
U+0108	Ĉ	264	Ĉ	Latin Capital Letter C with circumflex	0200
U+0109	ĉ	265	ĉ	Latin Small Letter C with circumflex	0201
U+010A	Ċ	266	Ċ	Latin Capital Letter C with dot above	0202
U+010B	ċ	267	ċ	Latin Small Letter C with dot above	0203
U+010C	Č	268	Č	Latin Capital Letter C with caron	0204
U+010D	č	269	č	Latin Small Letter C with caron	0205
U+010E	Ď	270	Ď	Latin Capital Letter D with caron	0206

source: https://en.wikipedia.org/wiki/List_of_Unicode_characters

Notable Characters in C Language

Character	ASCII	UTF-8	C
null (nothing)	0	0xC0 (0)	‘\0’
backspace	8	0x08 (08)	‘\b’
horizontal tab	9	U+2B7E	‘\t’
newline	10	0x0A (0a)	‘\n’
carriage return	13	0x0D (0d)	‘\r’
space	32	0x20 (20)	‘ ’

The `char` data type

- Is designed to hold ASCII characters
- A single character should be in single quotes

```
x='c'
```

- Do not use double quotes. This is not a character

```
x="c"
```

The `stdio.h` library – reading data

FGETC(3) Linux Programmer's Manual FGETC(3)

NAME

fgetc, fgets, getc, getchar, ungetc – input of characters and strings

SYNOPSIS

```
#include <stdio.h>

int fgetc(FILE *stream);

char *fgets(char *s, int size, FILE *stream);

int getc(FILE *stream);

int getchar(void);

int ungetc(int c, FILE *stream);
```

DESCRIPTION

fgetc() reads the next character from stream and returns it as an unsigned char cast to an int, or **EOF** on end of file or error.

getc() is equivalent to **fgetc**() except that it may be implemented as a macro which evaluates stream more than once.

getchar() is equivalent to **getc**(stdin).

fgets() reads in at most one less than size characters from stream and stores them into the buffer pointed to by s. Reading stops after an **EOF** or a newline. If a newline is read, it is stored into the buffer. A terminating null byte ('\0') is stored after the last character in the buffer.

ungetc() pushes c back to stream, cast to unsigned char, where it is available for subsequent read operations. Pushed-back characters will be returned in reverse order; only one pushback is guaranteed.

The `stdio.h` library – writing data

```
PUTS(3)                    Linux Programmer's Manual                    PUTS(3)

NAME
       fputc, fputs, putc, putchar, puts - output of characters and strings

SYNOPSIS
       #include <stdio.h>

       int fputc(int c, FILE *stream);

       int fputs(const char *s, FILE *stream);

       int putc(int c, FILE *stream);

       int putchar(int c);

       int puts(const char *s);

DESCRIPTION
       fputc() writes the character c, cast to an unsigned char, to stream.

       fputs()  writes  the  string  s to stream, without its terminating null
       byte ('\0').

       putc() is equivalent to fputc() except that it may be implemented as  a
       macro which evaluates stream more than once.

       putchar(c) is equivalent to putc(c, stdout).

       puts() writes the string s and a trailing newline to stdout.
```

Ok. Back to the K&R Book Example – Character Copying

```
#include <stdio.h>

/* Program from K&R book */
/* Copy characters from input to output  */

main()
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(c);
}
```

file_copy.c

This little program can teach us a lot about C (continued)

- `getchar()` is a function that returns an integer
- `(c=getchar())` is both an assignment and an expression that evaluates to true or false
- Unlike Python or Java

Assignment

An assignment is both a statement and an operator that evaluates to the value of the assignment

- This assigns `3` to `a` and, in addition, it evaluates to `3`

```
a = 3
```

- This assignment sets all variables to 0 and evaluates to 0

```
a = b  = c = 0
```

- It evaluates from right to left. It is equivalent to

```
a = (b  = (c = 0))
```

- This is valid C

```
if (a=b=c=0) {..}
```

- The semantics of the assignment are different from Java / Python

Assignment vs Equality Comparison

- For assignment use =

```
c=getchar();
```

- For comparison use ==

```
if (c == 'A') {….}
```

- Don't confuse this for an equality test. The use of = here is an assignment, it is not an equality test

```
if ((c=getchar()) != EOF)
```

C is a weakly-typed, statically typed language

	Statically Typed	Dynamically Typed	
Strongly Typed	Java, Go	Python	Variables are bound to specific types; type errors occur if types do not match
Weakly Typed	C	Javascript Bash	Variables are not bound to specific data types; they still have a type but type safety constraints are lower
	Type determined at compile time	Type determined at runtime	

The type system is different between C and Java / Python

weakly typed and statically typed – What does this mean? Let's see another example from K&R

```
main()
{
    int c, i, nwhite, nother;
    int ndigit[10];

    nwhite = nother = 0;

    while ((c = getchar()) != EOF) {
        if (c >= '0' && c <= '9')
            ++ndigit[c-'0'];
        else if (c == ' ' || c == '\n' || c == '\t')
            ++nwhite;
        else
            ++nother;
    }
```

counts digits, white space, others

Statically typed

- You must declare variables before using them (unlike Python)
- You must declare a datatype for them (unlike Python)

Weakly typed

- C is not too particular about comparing the values of different data types (unlike Java/Python)

```
(c >= '0' && c <= '9')
```

Arrays

- Define an array of 10 integers

```
int ndigit[10];
```

- Can be initialized at declaration

```
int a[] = { 121, 17, 2, 88, -273 } ;
```

- Access the array using indexes 0,… 9

```
mdigit[0], ndigit[1], … ndigit[9]
```


- Arrays are similar between C and Java/Python

String Arrays

- Strings are arrays of characters ending with `'\0'` (`null`)
- Define a string as an array using characters

```
name= ['d', 'i', 'm', 'i', 't', 'r', 'a', '\0'];
```

- Or, define a string array using double quotes ""

```
name = "Dimitra";
```

- Access the array elements using indexes 0,… 9, e.g.,

```
name[0], name[1], … name[6]
```

- Note, `'c'` is a character while `"c"` is not

```
"c" = ['c', '\0']
```

Strings and =

- Use = for initialization only

```
s="Dimitra"
```

- Do not use = for assignment
- Use `strcpy()` for assignment

```
strcpy(s, "Dimitra")
```

- Better yet, use this (avoids buffer overflows)

```
strncpy(s, "Dimitra", strlen("Dimitra"))
```

strlen VS sizeof

- `sizeof` operator is a compile time unary operator which can be used to compute the size of its operand
  - it returns the actual memory allocated by the operand
  - depends on data type
- `strlen` searches for the NULL character and counts the number of memory addresses passed, so it actually counts the number of elements present in the string before the NULL character

```
#include <stdio.h>
#include <string.h>


int main()
{
    int i = 2;
    char c = 'A';
    char str[] = "Dimitra";


    printf("Size of char is %lu\n", sizeof(c));
    printf("Size of int is %lu\n", sizeof(i));
    printf("Size of String is %lu\n", sizeof(str));

    printf("Length of String is %lu\n", strlen(str));
}
```

Output

```
Size of char is 1
Size of int is 4
Size of String is 8
Length of String is 7
```

Functions

- So far we've seen `printf()`, `getchar()`, and `putchar()`
- Functions are defined as

```
return-type function-name(parameter declarations, if any)
{
        declarations
        statements
}
```

- In C, there are two ways to pass parameters
- Return types can be any data type or `void` (no return value)
- Call by value
  - Any changes made to a parameter inside a function are gone when the function completes
- Call by reference
  - Any changes made to a parameter inside a function persist even after the function completes

Functions

- Various parts of a function declaration may be absent

- This is a valid function in C

```
dummy () {}
```

- When return type is missing `int` is assumed

- Useful when stubbing code for development

Functions and C Header Files

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include “calc.h”

main() {
 --
}
```

calc.h

```
#define NUMBER ‘0’

void push(double);

double pop(void);

int getOp(char[]);
```

getop.c

```
include <stdio.h>
include <ctype.h>
include “calc.h”

getop() {
 …
}
```

stack.c

```
include <stdio.h>
include “calc.h”

#define MAX_VAL 100

int sp=0;
double val[MAX_VAL];

void push (double ){
…
}

double pop(void) {
 …
}
```

- A large program can be broken into many files

  `main.c getop.c stack.c`

- Each `.c` file contains the variables and functions that “go together”
- Each `.c` file can have its own `.h` header file or they can share

  `calc.h`

- Note the difference between `#include <stdio.h>` vs `#include “calc.h”`
- Quotes mean the header file is in the current directory

Function – Call by value

Any changes made to a parameter inside a function are gone when the function completes

```
int add_one(int a)
{
        a = a + 1;

        return (a);

}
```

The following program

```
a=10;

b=add_one(a);

printf("Current value a=%d\n", a);
```

Returns

```
Current value a=10
```

Function – Call by reference

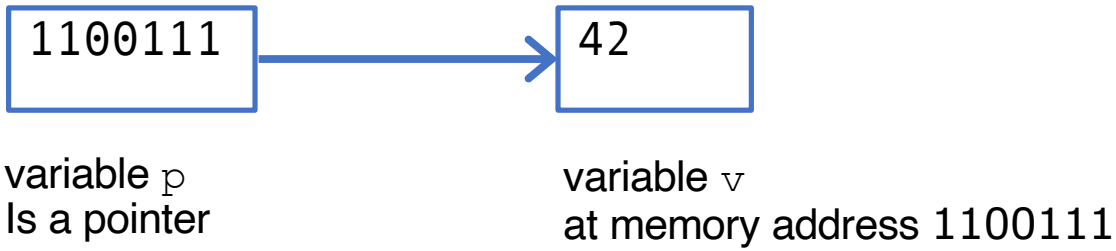
- To understand how call by reference works, we first need to understand

Pointers!!!

To understand C functions and call by reference
we need to understand pointers

Pointers

- Pointers in C are variables that store the address of another variable




```
p= 1100111
v=42
*p=42          unary operator * is what the pointer p points to
&v= 1100111    unary operator & is the address of variable v
```

- Therefore

```
if p==&v, then *p=v
```

C is a low level programming language

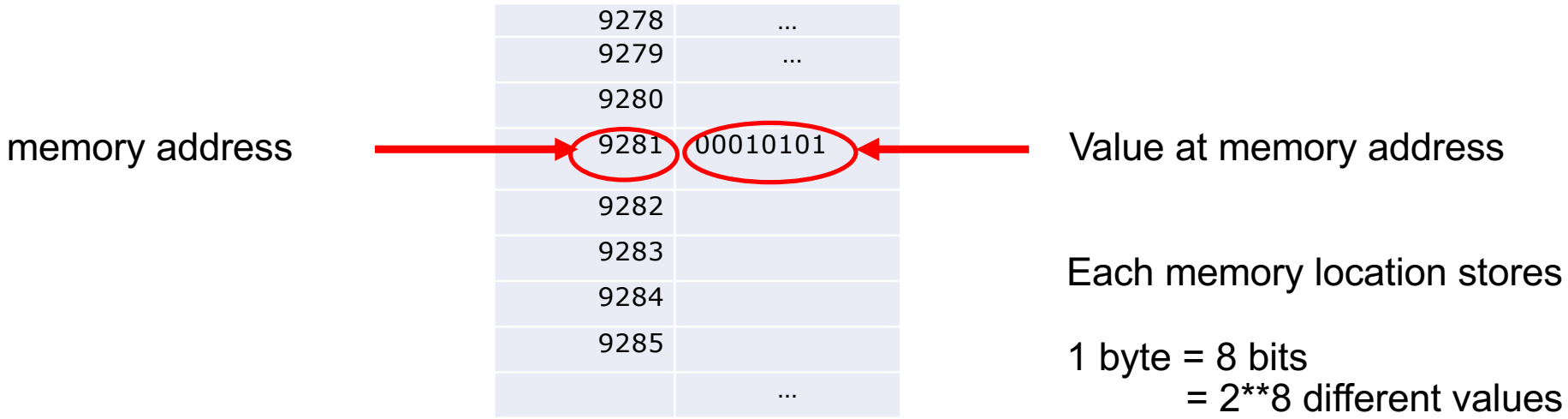
- C is close to the OS and its resources
- Pointers exemplify this
- Pointers allow the C program to manipulate the memory of the computer

So let's take another short detour to talk about how memory works…

To understand C functions and call by reference
we need to understand pointers
we need to understand how memory works

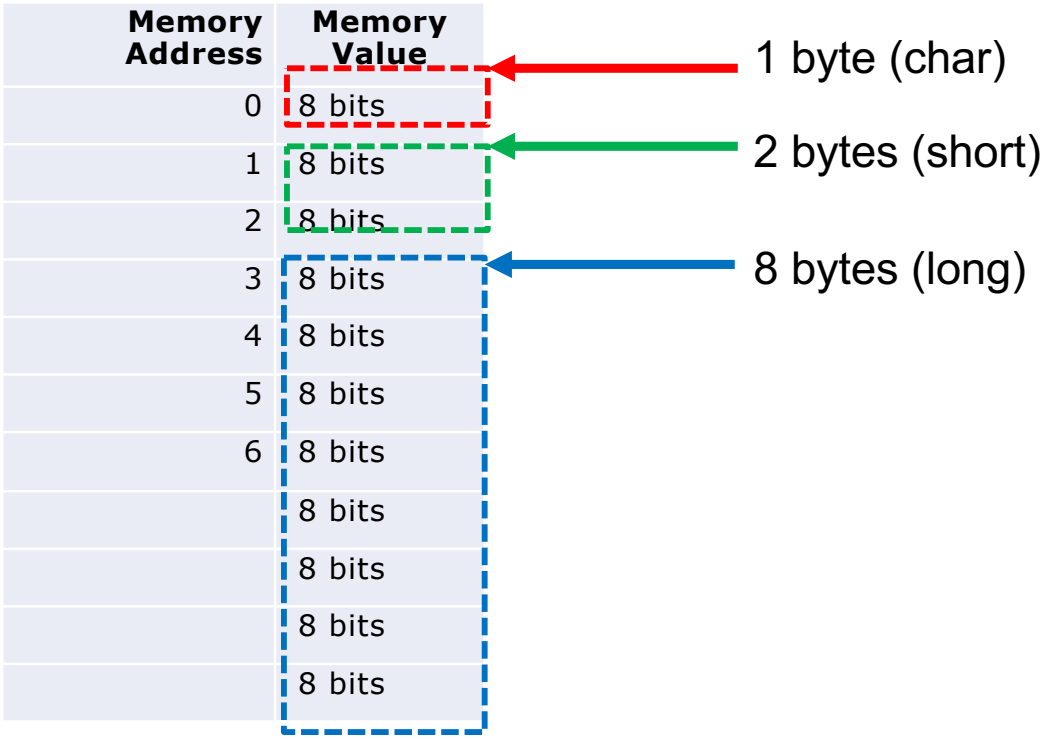
Main Memory

- Main memory is made up of small, consecutive memory units
- The smallest memory unit is a byte
- Associated with each memory unit is a unique number called an *address*
- The CPU registers stores memory addresses, which is how the processor accesses data from RAM.



Memory is byte-addressable

What if the data type is larger than a byte?



- It is the data type of the variable that tells us how many bytes of memory we need
  - `char` = 1 byte
  - `short` = 2 bytes
  - `long` = 8 bytes

Memory Size - 32-bit computers

- How many memory locations (bytes) are there?
- It depends!!!
- In a 32-bit architecture every CPU register has 32 bits (4 bytes) which can reference up to 2^{32} different locations

Memory Address	Memory Value
0	8 bits
1	8 bits
2	8 bits
3	8 bits
4,294,967,295	8 bits

I have 32 bits to reference a memory location
2^{32} = 4,294,967,296 = 4GB

32-bit architecture

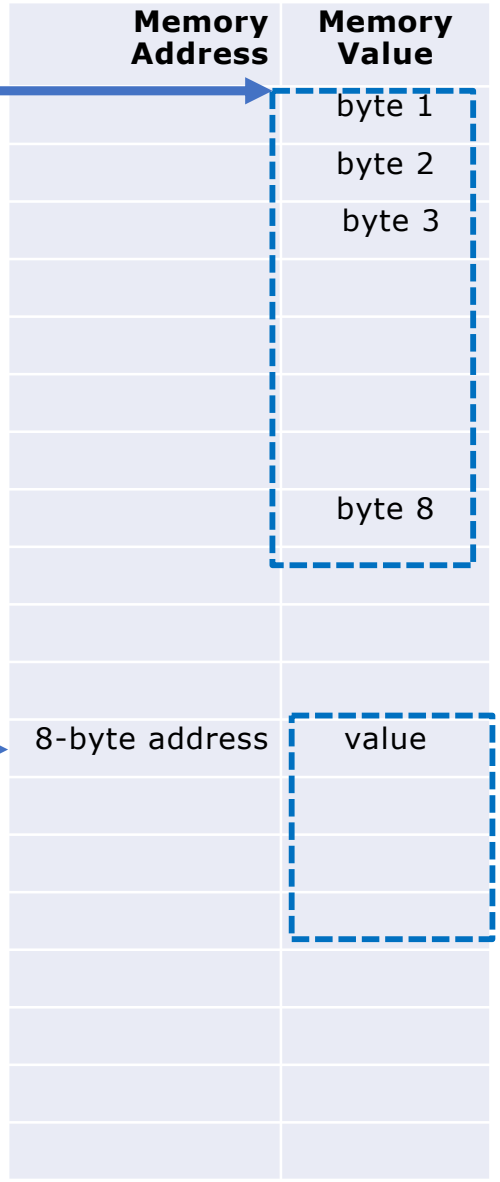
- Most computers made in the 1990s and early 2000s were 32-bit machines
- 32-bit computers can address a max of 4GB of memory
- Each CPU register in a 32-bit architectures is 4 bytes

64-bit architecture

- Most computers made in the 1990s and early 2000s were 32-bit machines
- 32-bit computers can address a max of 4GB of memory
- Each CPU register in a 32-bit architectures is 4 bytes
- Each CPU register in a 64-bit architecture is 8 bytes
- A 64-bit register can theoretically reference 18,446,744,073,709,551,616 bytes or 17,179,869,184 GB (16 exabytes) of memory (To put this in perspective in 2014 the size of the entire internet was 1M exabytes)
- That's several million more than an average computer would need
- If a computer has 8 GB of RAM, it better have a 64-bit processor. Otherwise, at least 4 GB of the memory will be inaccessible by the CPU.

Pointers in a 64 bit architecture are 8 bytes

A pointer p
is a variable that
stores a memory address



A pointer in a 64 bit architecture is 8 bytes

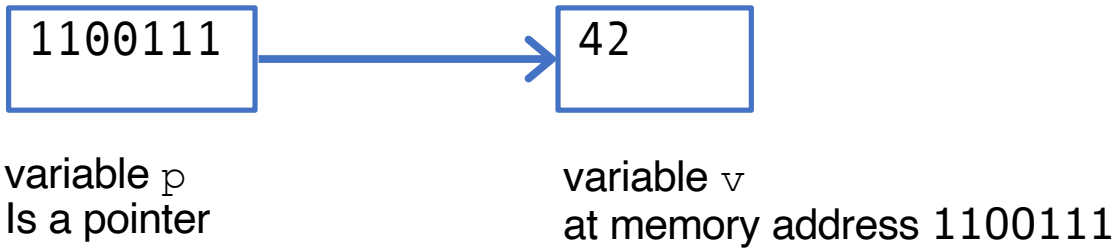
the address
where p points to

where a pointer points to depends
on the data type of the pointer

Back to C pointers

Pointers

- Pointers in C are variables that store the address of another variable




```
p= 1100111
v=42
*p=42          unary operator * is what the pointer p points to
&v= 1100111    unary operator & is the address of variable v
```

- Therefore

```
if p==&v, then *p=v
```

Pointers – how to declare

Variables

`int *p`	a pointer to a memory location for an integer
`char *p`	a pointer to a memory location for a char
`void *p`	a generic pointer
	i.e., a pointer to any location
	i.e., a pointer to an undefined data type

Note

`int *p`	as a declaration it is a mnemonic for pointer
	it says that `*p` (where p points to) is an `int`
`*p`	anywhere else it is the value of where p points to

Using pointers in expressions

if `p` is a pointer to `x`, then `*p` can appear where `x` can appear

`x = x + 10` is the same as `*p = *p + 10`

`x++` is the same as `(*p)++`

Pointer arithmetic

The value of pointers are memory addresses, thus numbers, so we can do arithmetic. There are four arithmetic operators that can be used in pointers

++

--

+

-

- Move the pointer to the next object (move one byte, if `char *p`, two bytes if `short *p`)

```
p = p + 1;
```

- Same

```
p++;
```

- Advance the pointer by 4 “objects”

```
p = p + 4;
```

Back to functions

Functions – Call by value

swap

```
void swap(int x, int y)  /* WRONG */
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

Before calling `swap(x,y)` the values are `x=1 y=2`

After calling `swap(x,y)` the values are `x=1, y=2`

Functions – call by reference

swap

```
void swap(int *px, int *py)  /* interchange *px and *py */
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```

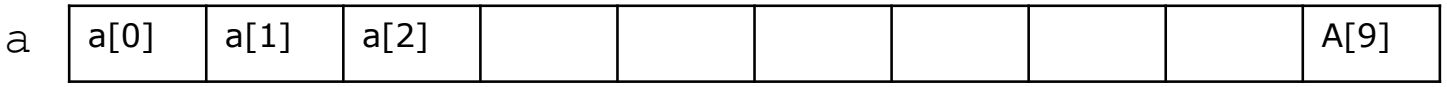
Before calling `swap(&x,&y)` the values are `x=1 y=2`

After calling `swap(&x,&y)` the values are `x=2 y=1`

Is python pass by reference or pass by value?

Pointers and Arrays

- In C there is a strong relationship between pointers and arrays
- Any operation that involves array subscripting can use pointers
- The declaration `int a[10]` defines and array

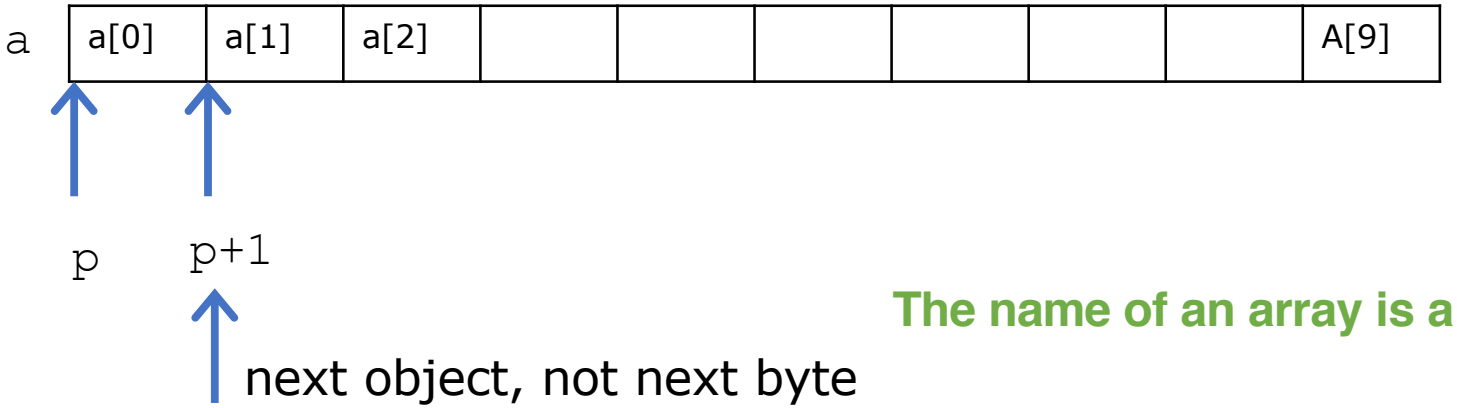


- If we declare a pointer that points to the beginning of the array

```
int *p;

p = &a[0];
```

- Then `a[i]` is the same as `*(p+i)` and `&a[i]` is the same as `p+i`



The name of an array is a pointer!

Pointers – in Summary

- Array of Pointers
  - You can define arrays to hold a number of pointers.

```
int *a[10];
```

- Pointer to pointer
  - C allows you to have pointer on a pointer and so on

```
int  **p;
```

- Passing pointers to functions
  - Passing an argument by reference or by address enable the passed argument to be changed in the calling function by the called function.

    In main (int argc, int **argv) { }

    Int main (int argc, int *argv[]) { }

- Return pointer from functions
  - C allows a function to return a pointer to the local variable, static variable, and dynamically allocated memory as well.

Arrays, Pointer and Functions

- We can pass pointers as values into functions (pass by reference)

```
void add3( int *p )
{ *p += 3 ; }
```

- Array names are pointers. The following 2 prototypes are equivalent

```
void foo( float a[], int n ) ;
void foo( float *a, int n ) ;
```

- You can have a pointer of pointers

```
int * argv[];  /* an array of pointers to integers */
int ** argv;   /* a pointer to a pointer to an integers */
```

NULL Pointer

- It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a null pointer.
- The NULL pointer is a constant with a value of zero defined in several standard libraries.

```
#include <stdio.h>

int main () {

        int *ptr = NULL;

        …

}
```

Arguments to main

```
int main( int argc, char *argv[] )
{
int i;

for( i=0; i<argc; ++i )
    printf( "%3d %s\n", i, argv[i] )
return(0) ;
}
```

```
int main( int argc, char **argv)
{
int i;

for( i=0; i<argc; ++i )
    printf( "%3d %s\n", i, argv[i] )
return(0) ;
}
```

- The actual arguments are `argv` , an array of character strings

  `char * argv[]`

  `char ** argv`

- The number of arguments passed is `argc`

  `argv[0]` is the program name (invocation)

  `argv[1]` is the first argument

  and so on

Pointers – Final points

- Array of Pointers
  - You can define arrays to hold a number of pointers.

```
int *a[10];
```

- Pointer to pointer
  - You can have pointer on a pointer and so on

```
int  **p;
```

struct

- A struct (structure) is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.
- A struct is an aggregate data type, allowing related data elements to be accessed and assigned as a unit.
- Example

```
struct point {
    int x;
    int y;
};
```

- Define

```
struct point temp;
```

- Use

```
temp.x
```

- Pointers to structures are so frequently used that an alternative notation is provided as a shorthand. If p is a pointer to a structure, then

```
p->x
```

enum

- Enumerated data types are possible with the `enum` keyword. They are freely interconvertible with integers.

```
#include <stdio.h>

enum day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY};


int main()

{

    enum day d = TUESDAY;

    printf("The day number stored in d is %d", d);

    return 0;

}
```

- Output:

  The day number stored in d is 2

cast

- Cast allows us to change a type into another type

```
mean = (double) sum / count;
```

`malloc` and `free`

- Memory allocation using `malloc` is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form.

```
ptr = (cast-type*) malloc(byte-size)
```

- Examples

```
p = (String) malloc(100);

ptr = (int*) malloc(100 * sizeof(int));

return (Treeptr) malloc(sizeof(Treenode));
```

- Free memory previouly allocated with `malloc()` using `free()`

```
buffer = malloc(num_items*sizeof(double));

free(buffer);
```

C Language Overview

Data types	char, int, float, double, bool
Integer types	(unsigned, signed) char (unsigned, signed) short (unsigned, signed) int (unsigned, signed) long
Floating point	float double long double
Constants	#define MAX 1000 const int MAX = 1000;
Type conversions / cast	(type-name) expression
Binary Arithmetic Operators	+, -, *, /, %.
Logical Operations	&&, ||, !
Comparisons	=, !=, <, >, <=, >=
Bitwise logic	<<, >>, &, ^, |, ~
Assignment	=
Assignment (extended)	=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
Increment / decrement	i++ i-- ++I --i

C Language Overview

Data types	char, int, float, double, bool
Conditional evaluation	? :
Equality Testing	== !==
Calling functions	()
Increment and decrement	++ --
Object size	sizeof
Member selection	-> .
Reference and dereference (pointers, arrays)	& * [ ]
Type conversion	(new type)

Precedence and Associativity of Operators

OPERATORS	ASSOCIATIVITY
`( ) [ ] -> .`	left to right
`! ~ ++ -- + - * & (type) sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= ^= |= <<= >>=`	right to left
`,`	left to right

Unary +, –, and * have higher precedence than the binary forms.

`If` statement

- Typical form

```
if (expression) {
        statements1
}

else {
        statements2

}
```

- *Ternary form*

$expr_1$ `?` $expr_2$ `:` $expr_3$

Instead of

```
if (a < b)
    { c = a; }
else
    { c = b; }
```

Use

```
c = (a < b) ? a : b;
```

Switch statement

```
switch (expression) {
          case const-expr :

             statements;

             break; /* optional */
          case const-expr :

             statements;

             break; /* optional */
          default:

             statements;
}
```

For loop

- ```
  for (expr1;  expr2;  expr3)
            statement
  ```
- This is valid C

```
int i;
for (i=0; i< 10; i++) { …}
```

- And so is this

```
for( ; ; ) {
    printf("This loop will run forever.\n");
}
```

- These are invalid in C

```
for (int i=0; i< 10; i++ { …}
for i in 0..10 {…}
```
```

While Loops

- Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

```
while (expression) {
   statements;
}
```

- Like a while statement, except that it tests the condition at the end of the loop body, so the statement or group of statements execute at least once

```
do
    statements;
while (expression);
```

Loop Control Statements

`Break`

- Terminates the `loop` or `switch` statement and transfers execution to the statement immediately following the loop or switch.

`Continue:`

- Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

`Goto`

- Transfers control to the labeled statement.

```
if (expression) { goto error; }
…
 error: printf("This is an error");
```

`printf` function to print output

- C language decision to use the same function for all data types
- The `printf()` function is used to print the "character, string, float, integer, octal and hexadecimal values" onto the output screen.
- We use `printf()` function with `%` format with a specifier to display the value of specific data types
  - `%d` for decimal, `%f` for floats, `%s` for strings, `%lf` for double, `%x` for hex, etc.).
- To generate a newline, we use `"\n"`

```
printf("Character is %c \n", ch);
printf("String is %s \n" , str);
printf("Float value is %f \n", flt);
printf("Integer value is %d\n" , no);
printf("Double value is %lf \n", dbl);
printf("Octal value is %o \n", no);
printf("Hexadecimal value is %x \n", no);
```

printf and fprintf

```
#include <stdio.h>

int printf( const char *format, ...) ;

int fprintf( FILE *fp, const char *format, ...);
```

`fprintf` format

format

`%[flags][width][.precision][length] specifier`

`flags` Justification, padding, leading +/-

`width` The minimum total field width (sig. info not truncated)

`precision` The number of decimals (might be truncated)

`length` Modifier for the type specifier

`specifier` Specifies the data type, and presentation

`fprintf` output - specifier

specifier

```
%[flags][width][.precision][length] specifier
```

`c` character

`d, i` signed decimal int

`f` float

`e` scientific notation

`g` shorter of `%e` or `%f`

`s` string

`o` unsigned octal

`p` pointer

`u` unsigned decimal int

`x` unsigned hex

`fprintf` output - flags

flags

```
%[flags][width][.precision][length] specifier
```

- Left-justify

+ Forces leading sign, even for positive numbers

With o, x, or X specifier, output is preceded by 0, 0x, or 0X

0 Pads with 0, rather than space

`scanf` to read from the keyboard

- C language design to use a single function for all data types
- The `scanf()` function is used to read character, string, numeric data from keyboard

```
#include <stdio.h>

int main()

{

   char ch;

   char str[100];


   printf("Enter any character \n");

   scanf("%c", &ch);

   printf("Entered character is %c \n", ch);

   printf("Enter any string ( upto 100 character ) \n");

   scanf("%s", &str);

   printf("Entered string is %s \n", str);

}
```

File handles – FILE*

```
#include <stdlib.h>

FILE* fopen( const char* pathname, const char* mode );

int fflush( FILE* stream );

int fclose( FILE* stream );
```

Modes include:

`r` Open for reading (the default)

`r+` Open for reading and writing. Pointer

positioned at beginning of file

`w` Truncate to zero length, open for writing

`a` Open for appending (writing at end of file). File pointer to end of file

`a+` Open for reading and appending. File pointer to end of file. Created, if needed

EOF

- EOF is not a character, it is -1
- EOF is int, not char
- How do we know when we reached the end of the file?
- The OS knows the size of the file (it keeps such metadata info) so we can calculate the last character
- If text files, files are made of lines and we can see the END-OF-LINE representations at the end of each line
  - a LF (\n) in Unix
  - a CR-LF (\r\n) in Windows, Vax.
- If reading from `stdin`, when we see `Ctrl^D` (end of input)

C Code to Read/Write to a file

```
.
.
.

FILE *in_file  = fopen("name_of_file", "r"); // read only
FILE *out_file = fopen("name_of_file", "w"); // write only

// test for files not existing.
if (in_file == NULL || out_file == NULL)
  {
    printf("Error! Could not open file\n");
    exit(-1); // must include stdlib.h
   }

// write to file vs write to screen
fprintf(out_file, "this is a test %d\n", integer); // write to file

fprintf(stdout, "this is a test %d\n", integer); // write to screen
printf(         "this is a test %d\n", integer); // write to screen

// read from file/keyboard. remember the ampersands!
fscanf(in_file, "%d %d", &int_var_1, &int_var_2);

fscanf(stdin, "%d %d", &int_var_1, &int_var_2);
scanf(        "%d %d", &int_var_1, &int_var_2);

.
.
.
```

C Code to Read One Line From a File

use `fgets` command

```
char line[100];

while ( fgets( line, 100, stdin ) != NULL )
  {
    fprintf("The line is: %s\n", line);
  }
```

usethe `getline` command

```
FILE *fp = fopen( "someTextFile" ) ;
char *buff = NULL ;
size_t len = 0;

while( getline( &buff, &len, fp ) != -1 )
{
/* overwrite newline */
buff[ strlen(buff)-1 ] = '\0' ;
printf( "%z chars: %s\n", len, buff ) ;
}
```

Lessons

- Lesson 1: Learn C to become a power programmer
- Lesson 2: C / C++ are the defacto systems programming languages

Resources

- These notes
- K&R Book http://tinyurl.com/yaemm9vh (it covers an older version of C but it is a great way to learn the basics)