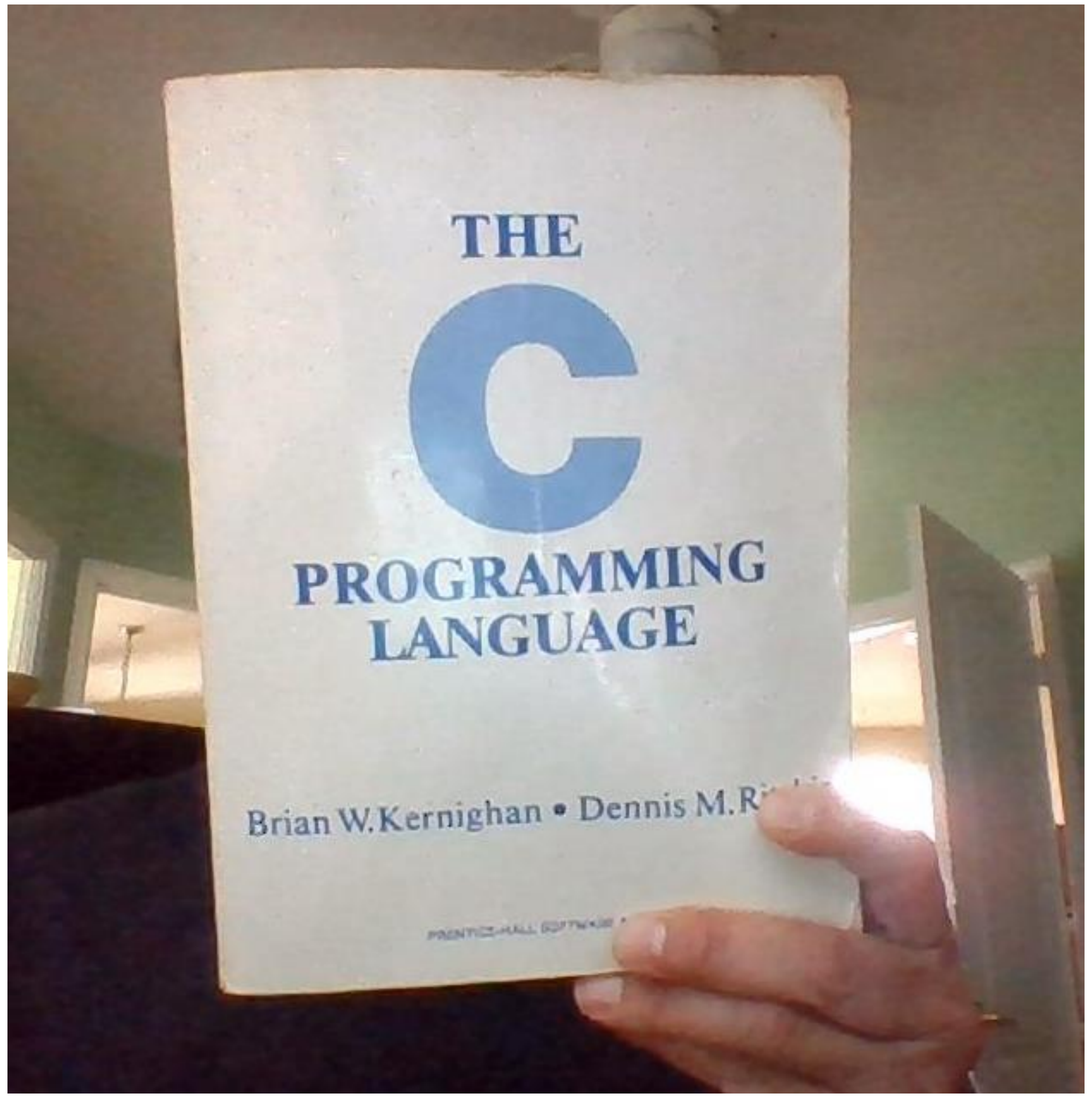
CSE 30 Spring 2023
Computer Organization and Systems Programming
Intro to C

Bryan Chin and Leo Porter



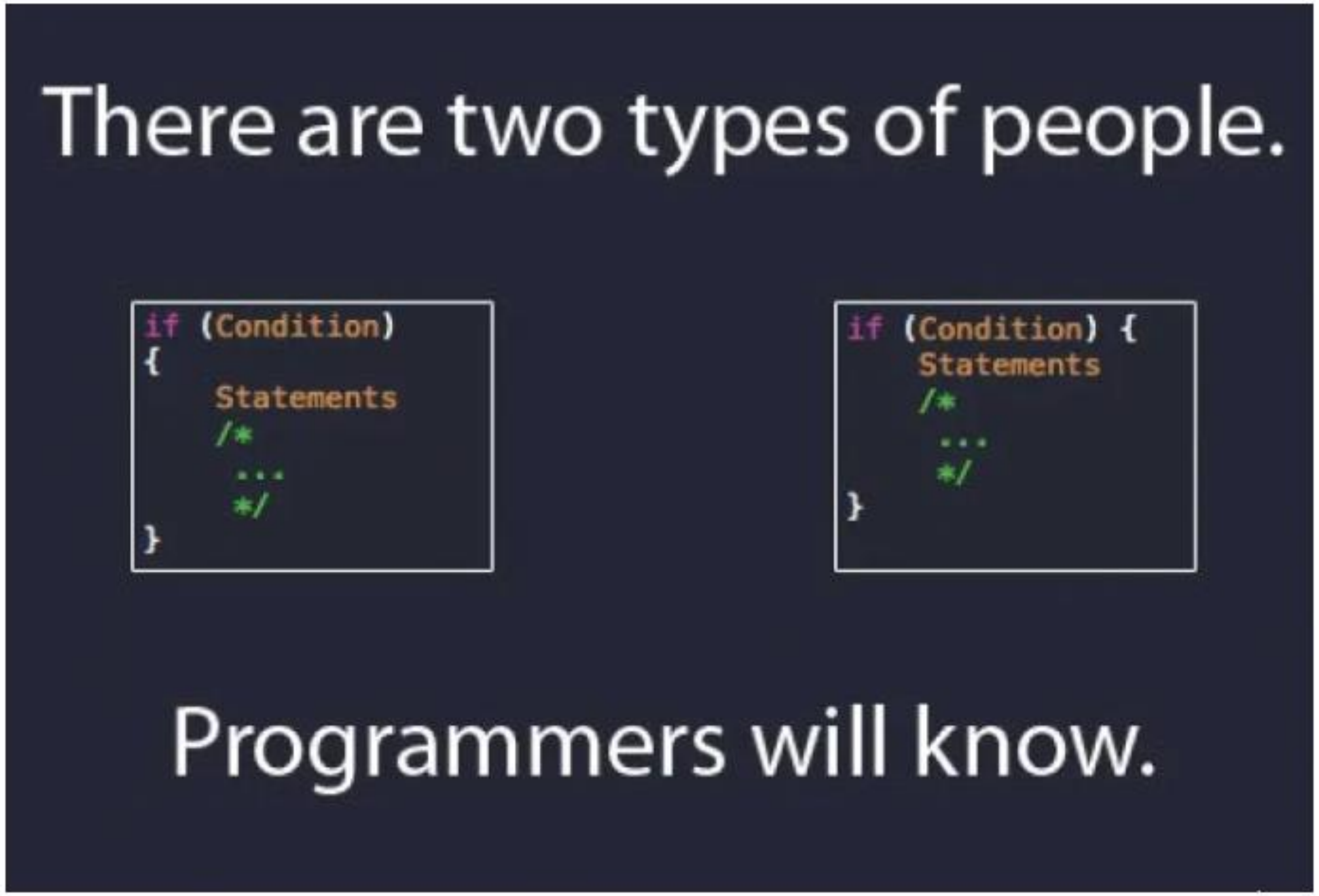


Image Credit: The9gag.com

Boundary between Normal and Denormal

sign	Exponent	mantissa
1	3	4
0	001	0000

$+ \quad 2^{-3} \quad x \quad 1.0000$

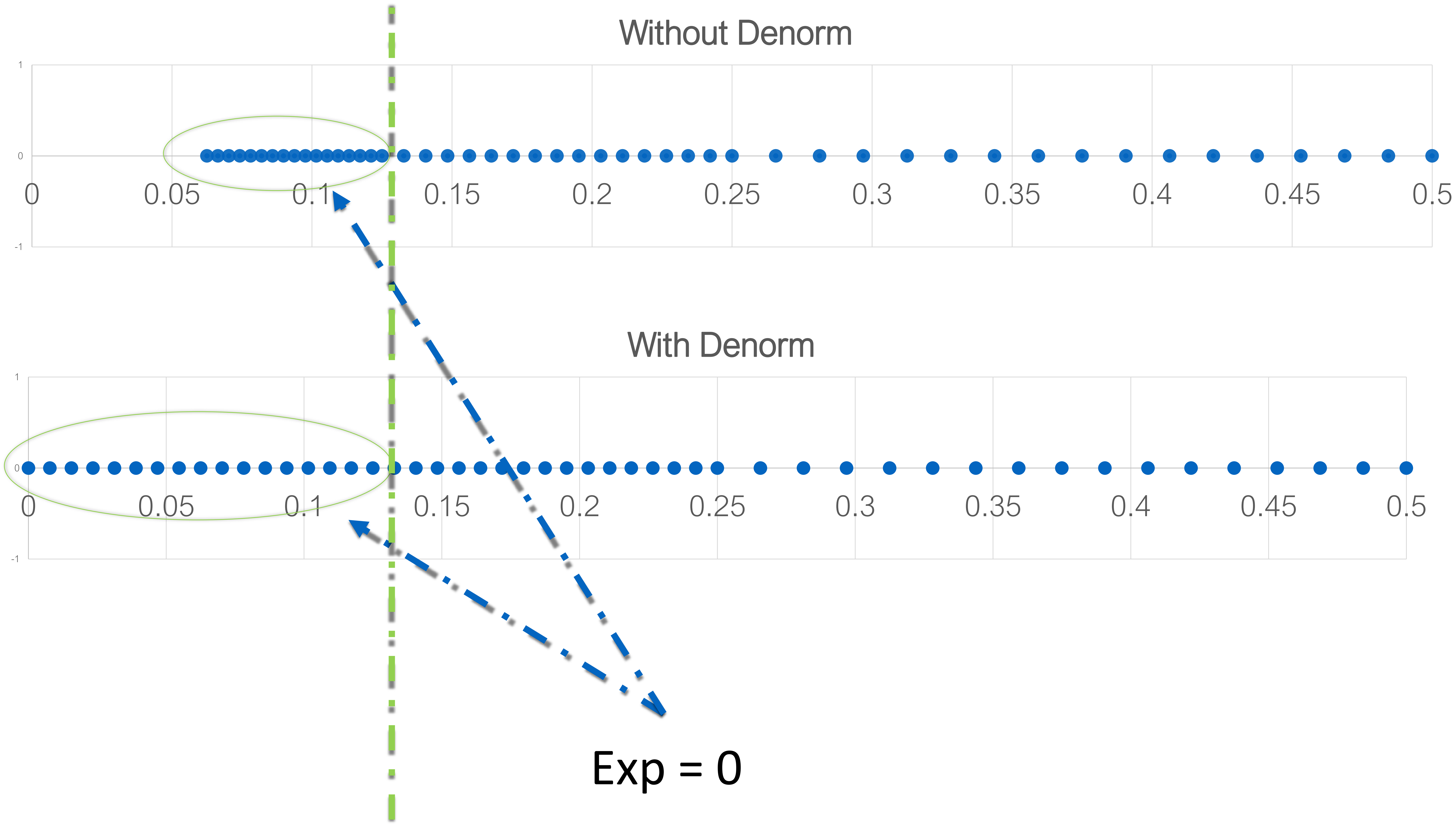
$= 0.0010000 = 1/8 = 0.125_{10}$

sign	Exponent	mantissa
1	3	4
0	000	1111

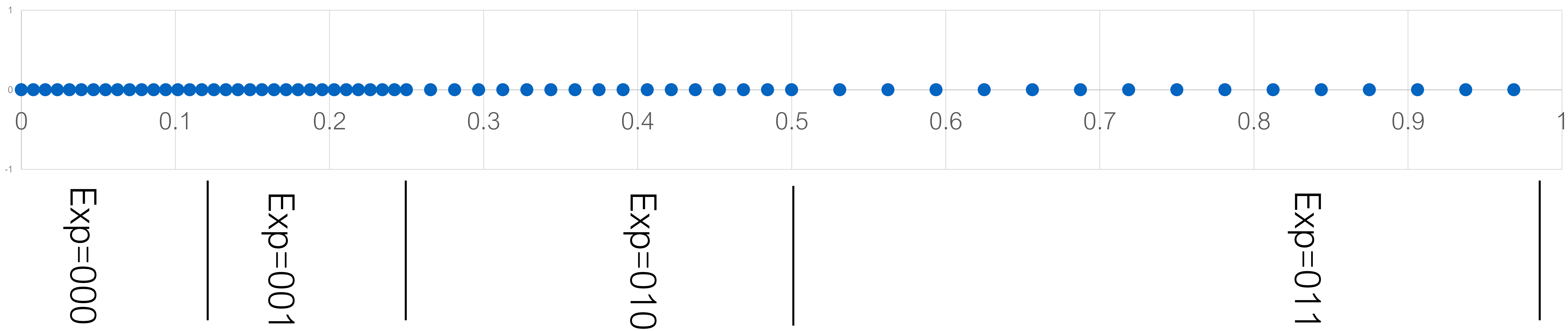
$+ \quad 2^{-3} \quad x \quad 0.1111$

$= 0.0001111 = 1/16 + 1/32 + 1/64 + 1/128 = 0.1171875_{10}$

No Denorms Vs Denorms



Linearity and Floating Point Don't Mix



Why the non-linearity?

Summary FP_{sample_8}

- **Our 8-bit FP standard**
- Exponent: -3 bias
- DeNormal Numbers
  - Exp 0 means no leading 1

sign	Exponent	mantissa
1	3	4

Bit layout

Exponent field	**Bias Represents Exponent**
111	3
110	2
101	1
100	0
011	-1
010	-2
001	-3
000*	-3 (denoms)

IEEE 754 Floating Point

- Evolving Standard
- **Single – 32 bit "C Float"**
- **Double – 64 bit "C Double"**
- Half – 16 bit
- Quad – 128 bit
- Special Encodings
  - NAN – not a number (quiet, signaling)
  - +∞ and -∞ (biggest positive #, and smallest negative number)
- DeNormal Numbers

sign	Exp = E +127	mantissa
1	8	23

sign	Exp = E + 1023	mantissa
1	11	52

What decimal # is represented by this "float"?

sign	Exp = E + 127	mantissa
1 bit	8 bits	23 bits
1	1000 0000	0110 0000 0000 0000 0000 000

A. 2.75

B. -2.75

C. -1.375

D. 1.375

E. None of these

IEEE Floating Point and “C” FP Types.

IEEE Type		sign	exponent	mantissa	“C” name
half	16	1	5	10	
single	32	1	8	23	
double	64	1	11	52	
quad	128	1	15	112	

Comparing Floating Point Numbers

- Compare Sign bit –
- If both positive
  - Compare like integers
- If one positive, one negative, obviously, positive is bigger
- If both negative,
  - Compare like integers only smaller integer is “bigger”

Which of the following lists order FP #'s from **largest to smallest**?

A. 0xfe762322, 0x7f112222, 0xbe762320

B. 0x11223344, 0x01223334, 0xf0001111

C. 0x77776666, 0x77771111, 0x78900000

D. None of the above

Results May Vary

- FP hardware has to round when it cannot represent values with full accuracy.
  - The order of operations then may change an answer:
    - (X+Y)+Z may not equal X + (Y+Z)

- Combining large and small numbers may change the result since the range of numbers represented is determined by the exponent. Combining 2 numbers with different exponents will lose precision

Results May Vary (2)

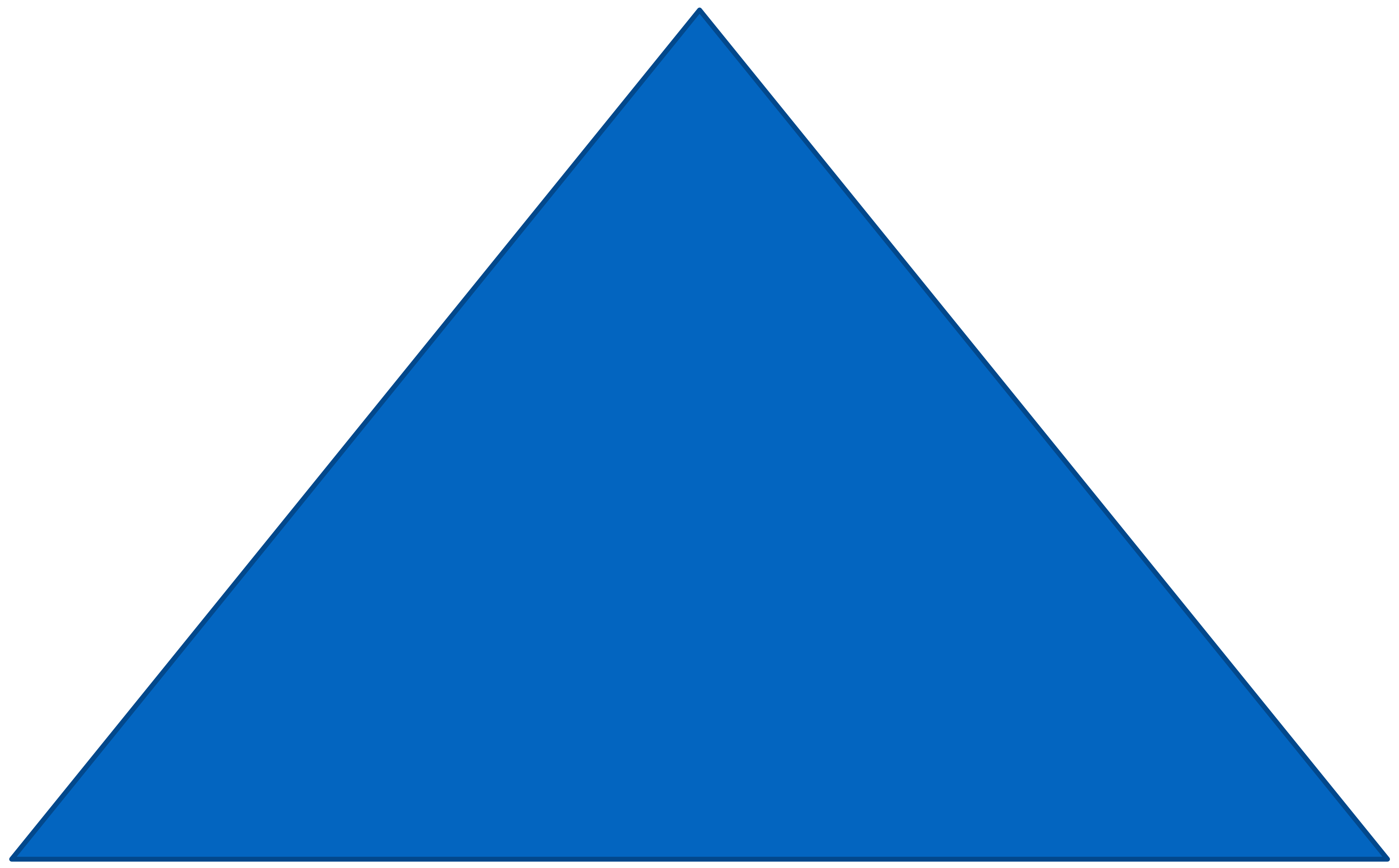
```
#include <stdio.h>
int main(int argc, char **argv){
  float a, b, c;


  a = 10000;
  b = 0.333333;
  c = 0.333333;
  printf("a = %f\n", a);
  printf("b = %f\n", b);
  printf("c = %f\n", c);
  printf("%f  (a + b) + c)\n", (a+b)+c);
  printf("%f  a + (b + c)\n", a+(b+c));
}
```

```
% ./a.out
a = 100000.000000
b = 0.333333
c = 0.333333
100000.671875  (a + b) + c)
100000.664062  a + (b + c)
%
```

Key Points

- Representing floating point numbers using binary is an interesting challenge
  - We want both small numbers and large numbers!
- IEEE754 FP standards are highly useful
  - However, they have limitations as well (e.g., non-linearity in values represented, still limited in range of values possible)
- FP operations often need to round and when they do, results may change.

C



What are the top 3 most popular programming languages in 2019 according to stackify.com?

A. Python, Java, C++

B. Python, JavaScript, Java

C. Java, C++, C

D. Java, C, Python

E. None of these are correct

UC San Diego

Why C?

- Brian Kernighan and Dennis Ritchie at Bell Labs
- Used to write an Operating System – UNIX

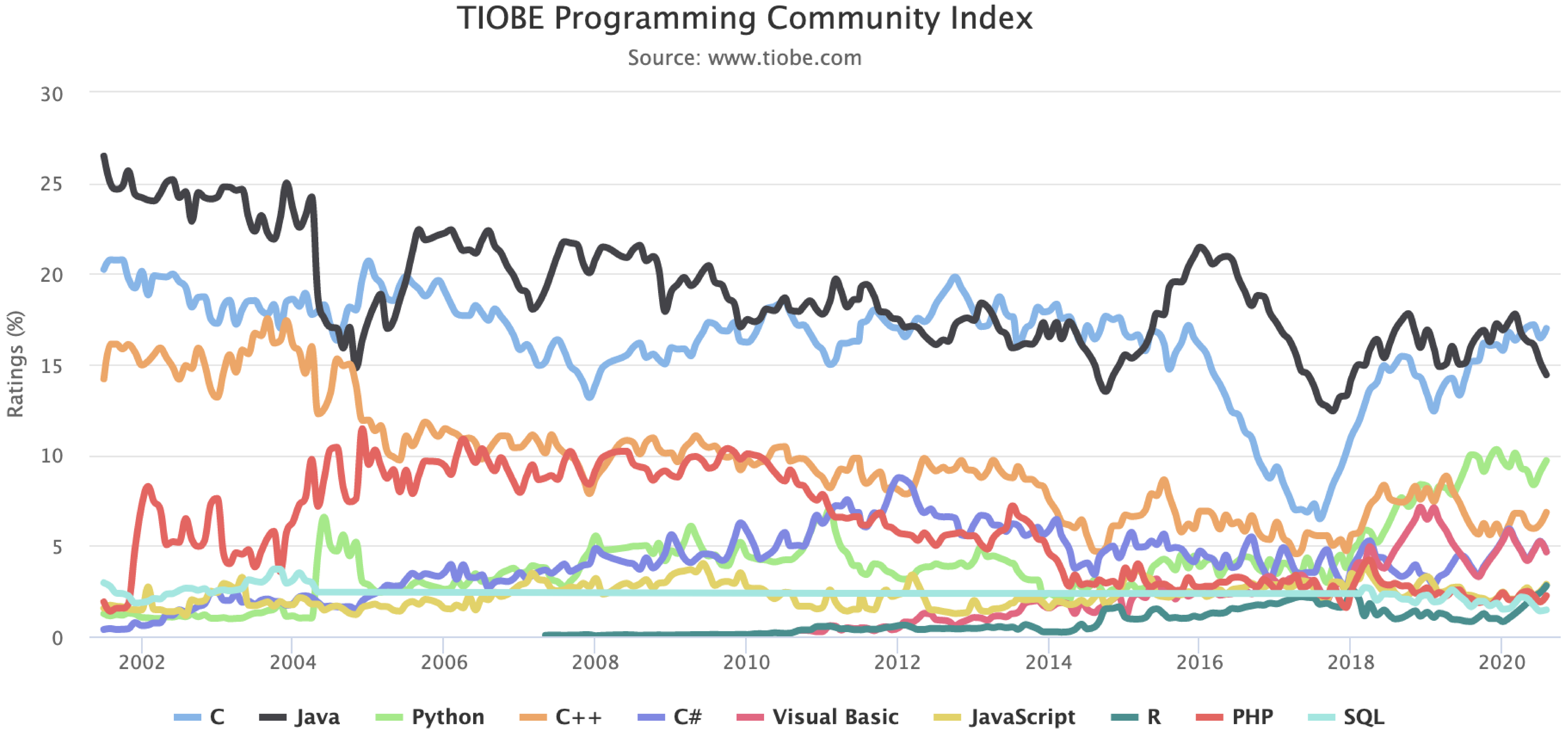
Programming Language	Ratings	Change
Java	16.028%	-0.85%
C	15.154%	+0.19%
Python	10.020%	+3.03%
C++	6.057%	-1.41%
C#	3.842%	+0.30%
Visual Basic .NET	3.695%	-1.07%
JavaScript	2.258%	-0.15%

Most Popular Programming Languages

Programming Language	Ratings	Change
Java	16.028%	-0.85%
C	15.154%	+0.19%
Python	10.020%	+3.03%
C++	6.057%	-1.41%
C#	3.842%	+0.30%
Visual Basic .NET	3.695%	-1.07%
JavaScript	2.258%	-0.15%
PHP	2.075%	-0.85%
Objective-C	1.690%	+0.33%
SQL	1.625%	-0.69%
Ruby	1.316%	+0.13%
MATLAB	1.274%	-0.09%
Groovy	1.225%	+1.04%
Delphi/Object Pascal	1.194%	-0.18%
Assembly language	1.114%	-0.30%
Visual Basic	1.025%	+0.10%
Go	0.973%	-0.02%
Swift	0.890%	-0.49%
Perl	0.860%	-0.31%
R	0.822%	-0.14%

 https://stackify.com/popular-programming-languages-2018/

Programming Language Popularity



https://www.tiobe.com/tiobe-index/

UC San Diego

Very Long Term History

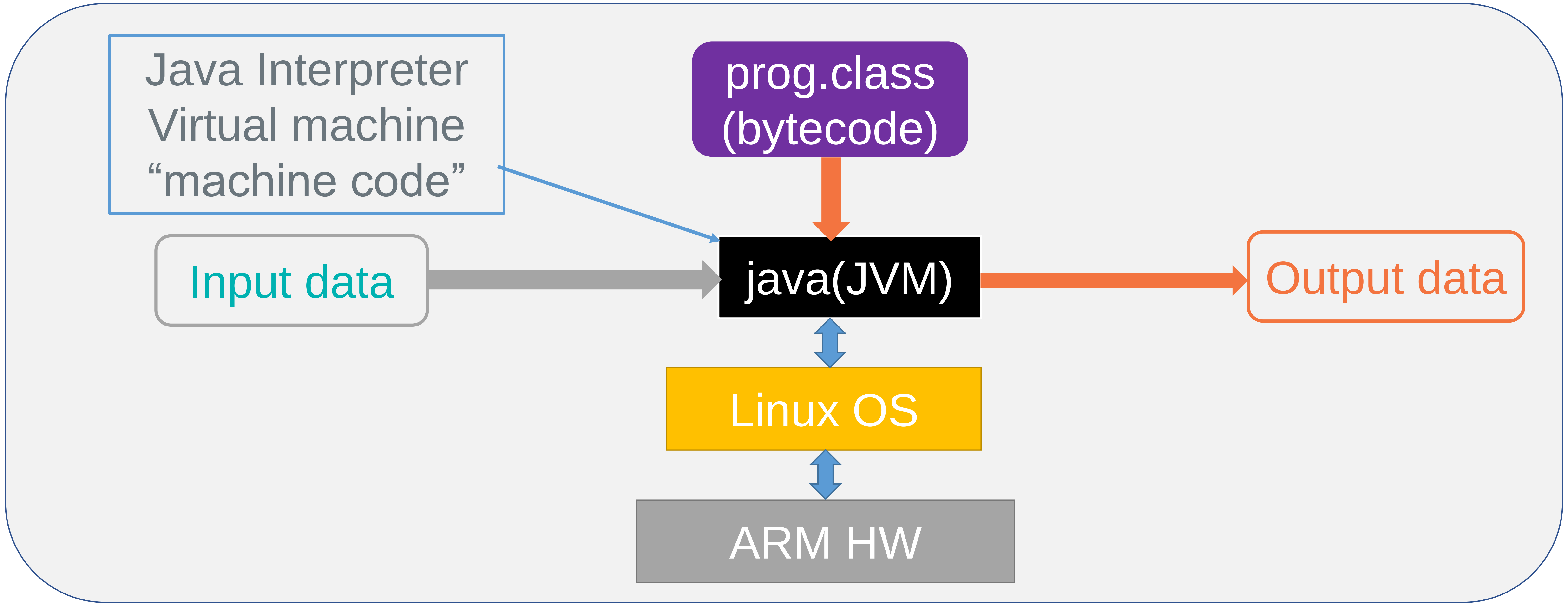
To see the bigger picture, please find below the positions of the top 10 programming languages of many years back. Please note that these are *average* positions for a period of 12 months.

Programming Language	2022	2017	2012	2007	2002	1997	1992	1987
C	1	2	2	2	1	1	1	1
Python	2	5	8	8	18	28	-	-
Java	3	1	1	1	2	18	-	-
C++	4	3	3	3	3	2	2	4
C#	5	4	4	7	12	-	-	-
Visual Basic	6	14	-	-	-	-	-	-
JavaScript	7	7	10	9	9	21	-	-
Assembly language	8	10	-	-	-	-	-	-
PHP	9	6	5	5	8	-	-	-
SQL	10	-	-	-	35	-	-	-
Prolog	24	33	45	28	29	15	10	3
Ada	28	30	17	17	17	11	3	14
Lisp	32	28	13	13	11	8	12	2
(Visual) Basic	-	-	7	4	4	3	7	5

https://www.tiobe.com/tiobe-index/

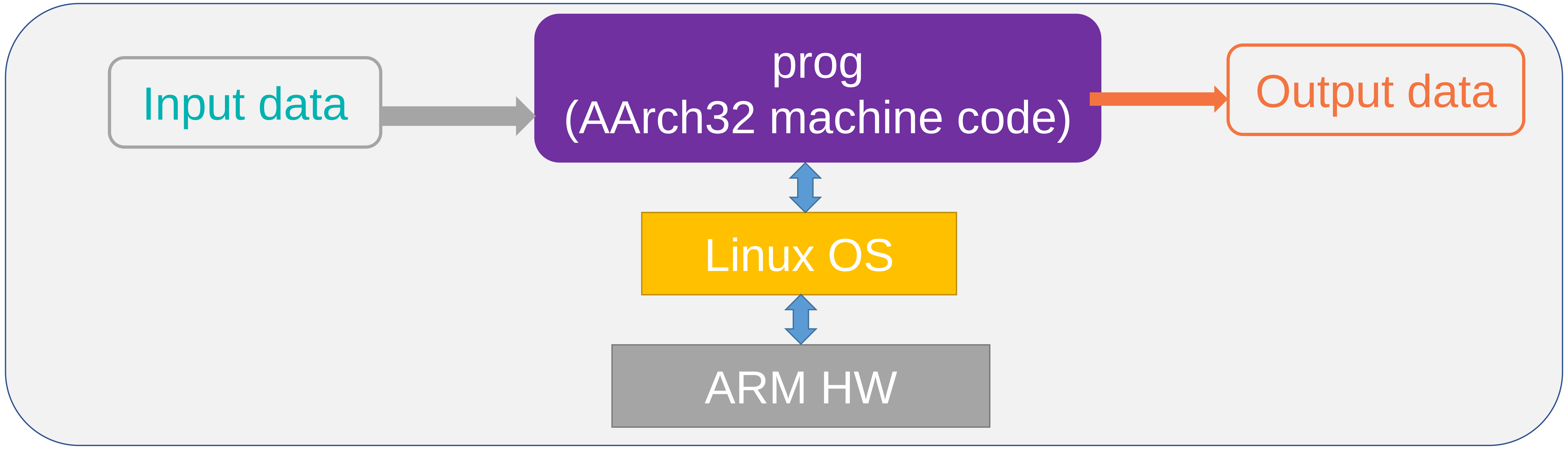
Java vs C/Assembly: Program Execution

- Java Virtual Machine **insulates the programmer**
  - machine independence
  - Java is common in user-interaction (application-level) programming
- Java: Portability over efficiency



how to run under linux → **% java prog**

- C was designed to replace assembly language
  - UNIX (BCPL -> B –> C)
  - architecture specific (compile to the machine)
- C: Runtime Performance focused
  - Very little runtime checks (OS mostly)
  - Underlying assumptions - **you know what you are doing**



how to run under linux → **% ./prog**

In which of these applications should you choose C over Java?

1. appointment reminder application that needs to run on cell phones from many different makers as well as on laptops and tablet computers
2. a communications driver for an Apple Macbook that needs very predictable performance and latency
3. graphics code for an airplane cockpit display system that is required to update at 60 times a second.

A. 1
B. 2
C. 3
D. 1 & 2
E. 2 & 3
F. 1, 2 & 3

C Programming

- Portable – many platforms
- Procedural thought process
- No built-in objects

  data separate from methods/functions

- Low memory overhead v. Java
  - No overhead of classes
  - No abstract machine – compile directly to ISA
  - Fast – write OS kernel in C
- Heap memory management manual
- Pointers can manipulate shared data (but with few checks)

The C Runtime Environment - Overview

- In Java, the compiler is javac and the executable are java byte codes
- In C, the compiler is cc (or gcc) and the executable are machine instructions

Src code:

```
#include <stdio.h>
int main (int argc, char **argv){
        printf("Hello!!\n");
}
```

Compiler

cc hello.c

Executable (a.out)

The C Runtime Environment - compilation

- **cc** (or gcc – the GNU c compiler)
- The C compiler takes the source and converts it to machine code:
- 2 stages
  - Compiling
  - Linking – link all compile output together and associate with libraries
  - (plus assembly)
- By default, cc does both compile and link phase.

Src code:

```
#include <stdio.h>
int main (int argc, char **argv){
        printf("Hello!!\n");
}
```

Compiler

cc hello.c

Executable (a.out)

The C Runtime Environment - Execution

- By default, binary (executable) is called a.out.
- The “1” and “0” ‘s represent machine instructions

```
bwc@bryanWindoze:~/cse30/tmp$ ./a.out
Hello!!
bwc@bryanWindoze:~/cse30/tmp$
```

Src code:

```
#include <stdio.h>
int main (int argc, char **argv){
        printf("Hello!!\n");
}
```

Compiler

Executable (a.out)

```
0002560 f839 8948 74e5 4819 058b 0a5a 0020 8548
0002600 74c0 5d0d e0ff 2e66 1f0f 0084 0000 0000
0002620 c35d 1f0f 0040 2e66 1f0f 0084 0000 0000
0002640 8d48 693d 200a 4800 358d 0a62 0020 4855
0002660 fe29 8948 48e5 fec1 4803 f089 c148 3fe8
0002700 0148 48c6 fed1 1874 8b48 2105 200a 4800
0002720 c085 0c74 ff5d 66e0 1f0f 0084 0000 0000
0002740 c35d 1f0f 0040 2e66 1f0f 0084 0000 0000
```

Getting Started in C

- Lots will be familiar:
  - Declaring variables (mostly)
  - Loops
  - Conditionals
  - Functions
  - Including libraries
- But there are big differences (some)
  - Pointers
  - Memory management
  - Strings (or lack thereof)
  - No objects (structs only)
    - No polymorphism/inheritance/etc.
  - Print syntax is different
  - Compiler directives
  - Function prototypes

somecode.h

```
int getMax (int, int);
```

somecode.c

```
#include <stdio.h>
#include "somecode.h"
#define A 5
#define B 10
int getMax(int a, int b)
{
        if(a > b)
            return a;
        else
            return b;
}
int main(){
    printf("%d\n", getMax(A, B));
}
```

Common Practices Seen in C Source

- Sequence Operator ,

  expr1*,*expr2

- Evaluates *expr1* and then *expr2* evaluates/returns to *expr2*

```
for (i = 0, j = 0; i < 10; i++, j++)
    …
```

- Assignment inside conditional test (this is very common!)

```
if ((i = SomeFunction()) != 0)
    statement1;
else
    statement2;
```

assignment returns the value that is placed into the variable to the left of the = sign, then the test is made

What does this code print when run as ./a.out 2?

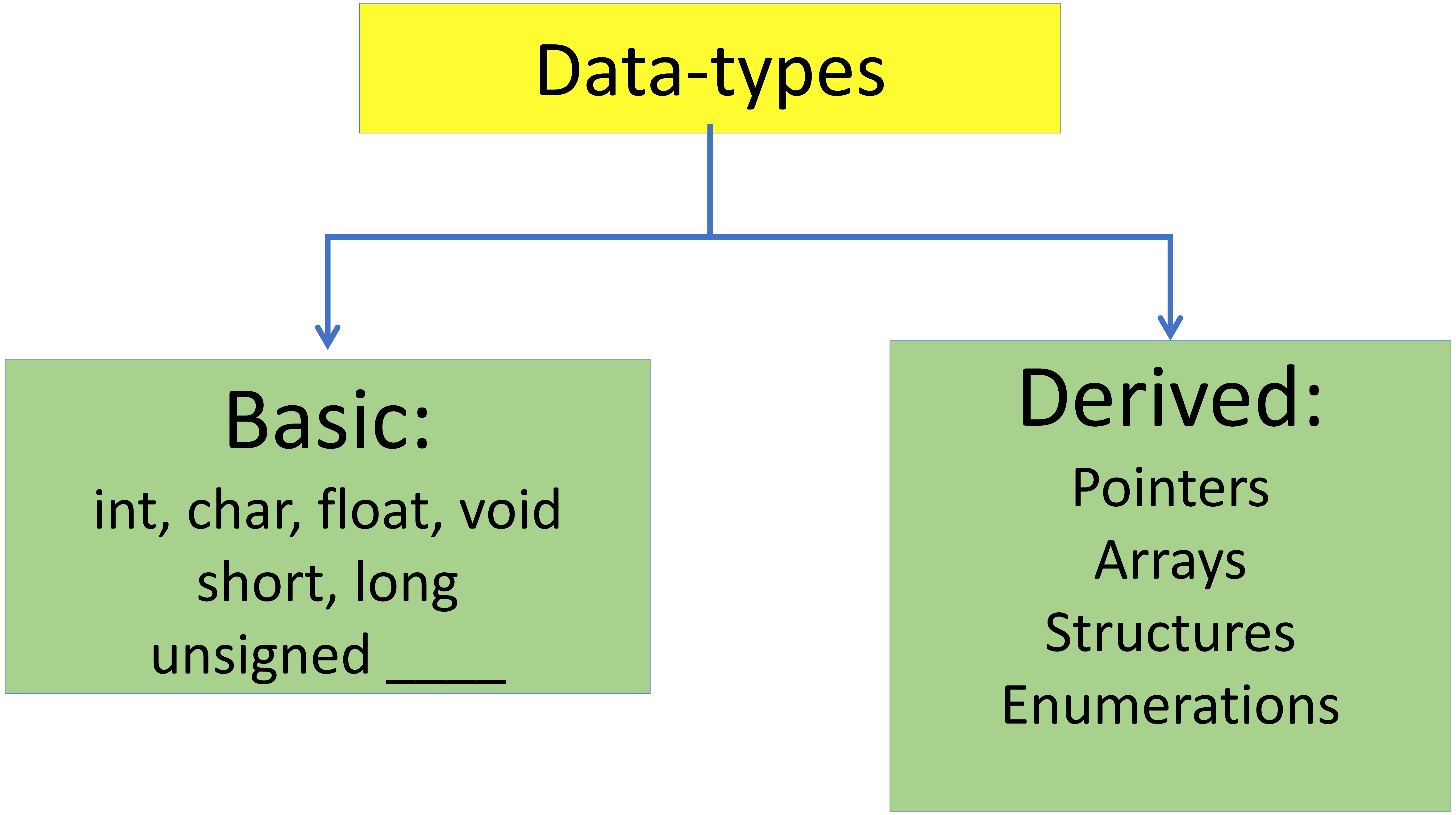
```
#include <stdio.h>
#include <stdlib.h>
int someFunction(int x){
   if (x = 4){
       x++;
   }
   return (x);
}

int main(int argc, char *argv[]){
  int someNum = atoi(argv[1]);
  printf("%d\n", someFunction(someNum));
}
```

A. 2
B. 3
C. 5
D. Won’t compile
E. none of these

Data objects in C

Old IBM Disk Drive with visible platters

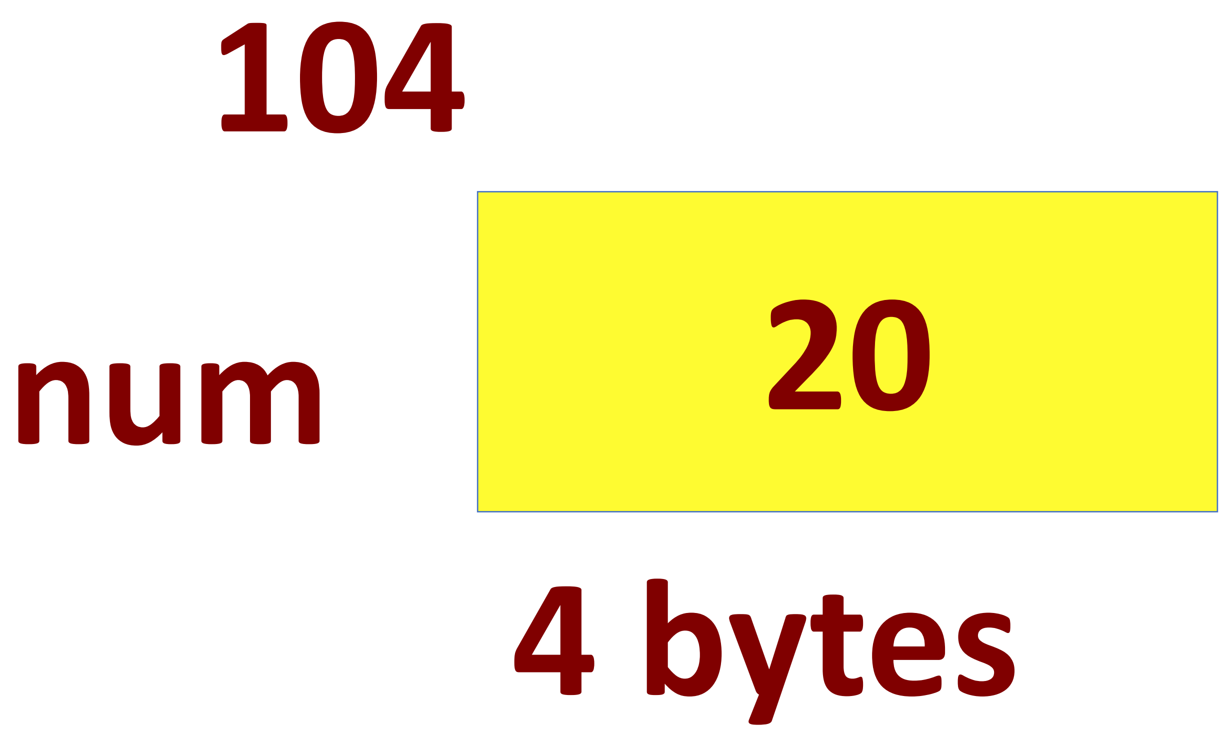
How we manipulate variables depends on *data-type*



Basic data object in memory

A region in memory that contains a value and is associated with name/identifier

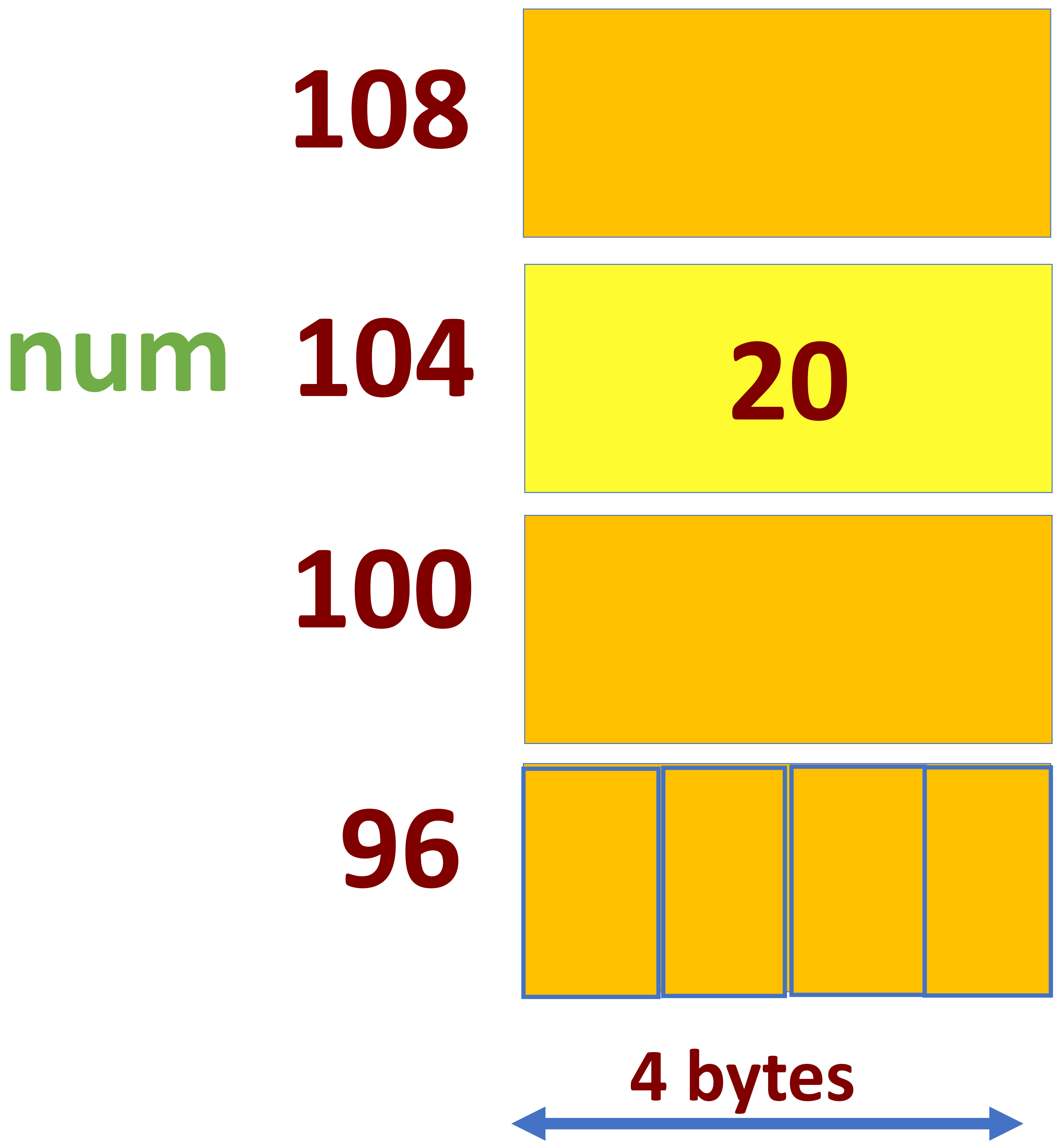
```
int
main(int argc, char**argv){

    int num;
    num = 20;
}
```



Basic data object in memory

A region in memory that contains a value and is associated with name/identifier

```
int
main(int argc, char**argv){
   int num;
   num = 20;
}
```



Attributes of a Data-object/variable:

- Name/identifier
- Value
- Address: Location in memory
- Size
- *data-type*
- Lifetime
- Scope

Definition vs Declaration

Definition

what is <it> and create an instance.

- function: create storage for it and corresponding code
- variable: create storage and put value there (optional)
- **only define once!**

```
int a;
short sum(short a, short b){
  return (a + b);
}
```

Declaration

describe <it>

- e.g function prototype
- variable named but defined elsewhere), containers for data (called structs)

```
extern int a;
short sum(short, short);
```

Declaration & Definition

What are these statement(s)?

```
extern int func(int, int);   // I
```

```
int func2(int a, int b) {    // II
    return a-b;              // II
}                            // II
```

Example definitions (some with initialization)

```
char c='a';          // 1 byte
short s;             // 2 bytes
int a;               // usually 4 bytes  - signed
unsigned int a=0;    // usually 4 bytes
float f;             // 4 bytes use sizeof(float)
double d;            // 8 bytes use sizeof(double)
long double d;       // quad fl. pt. usually 16 bytes)
```

Header Files

- Include Header files (.h) that contain function declarations - the function interface
  - Function declaration (return type, argument types)
- Some other .c files contain the actual code (definition)
- Include files (.h) contain variables referenced here but defined elsewhere (later)

somecode.h

```
int getMax (int, int);
extern int someGlobalVar;
```

somecode.c

```
#include <stdio.h>
#include "somecode.h"
#define A 5
#define B 10
int someGlobalVar = 10;
int getMax(int a, int b)
{
     if(a > b)
         return a;
     else
         return b;
}
int main(){
    printf("%d\n", getMax(A, B));
}
```

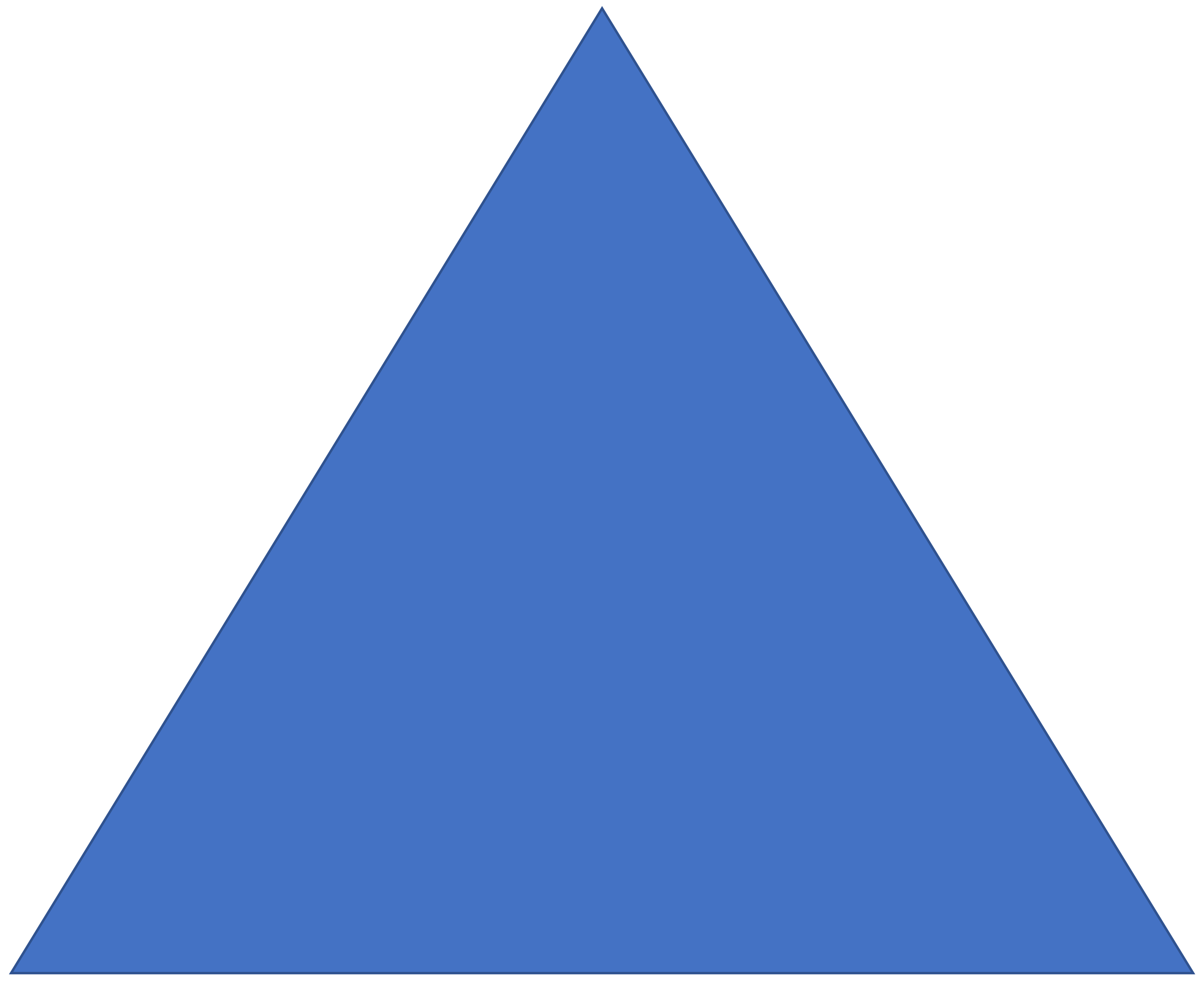
function definition

Which of the following are NOT appropriate for a header file?

```
int a = 10;             // I
int b;                  // II
extern int c;           // III
char rotateMe(char c); // IV
```

A. I.
B. II.
C. I. && II.
D. III. && IV.
E. IV.

Simple I/O



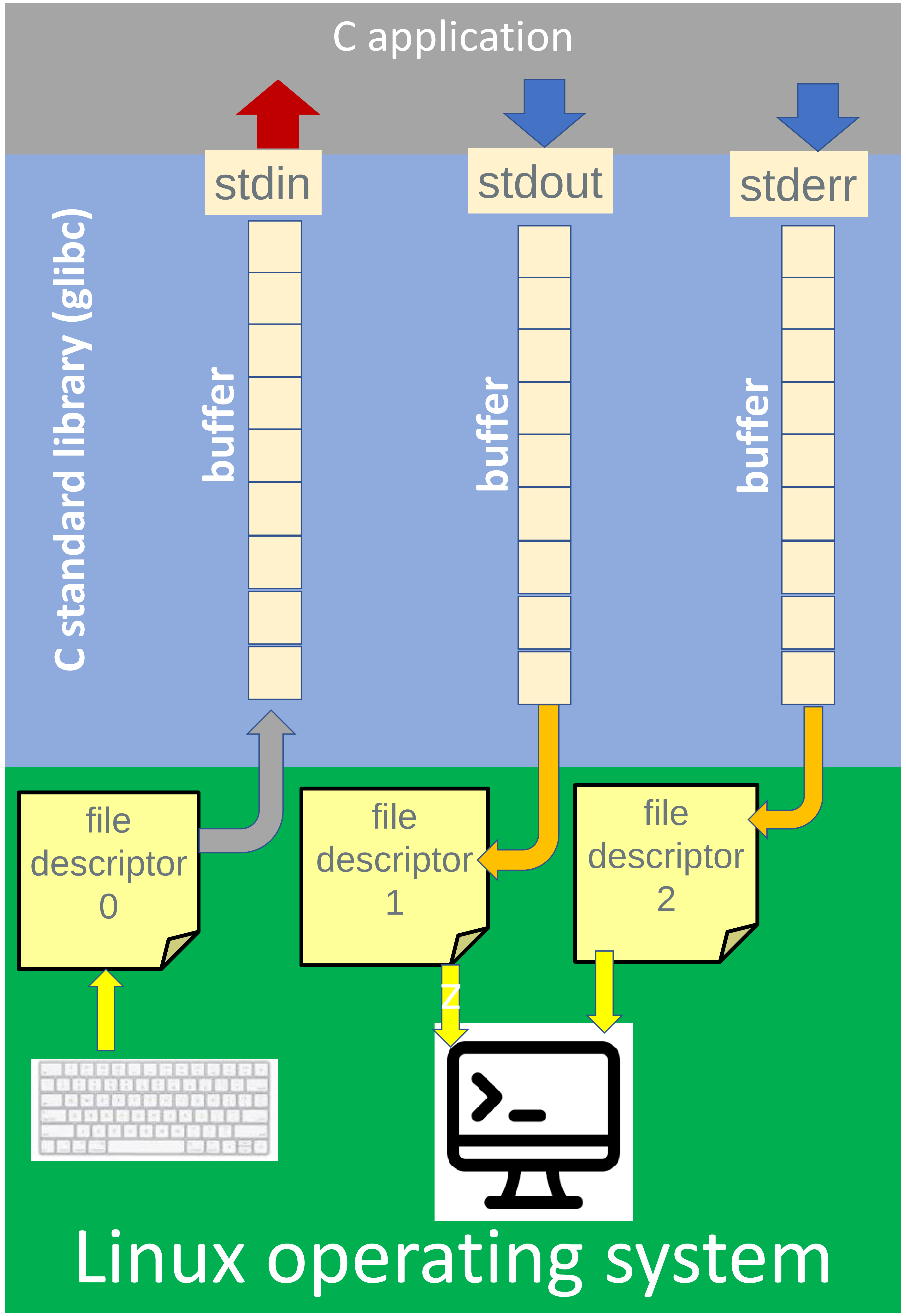
HW 2 and I/O

```
./encrpter inputfile_name
```

- Program reads characters from a file stream
- Program writes output to a stream called stdout
- Program writes error message to a stream called stderr

C Runtime: stdio streams (simplified)

- C's **stdio** library : notion of a stream
    - Sequence of bytes flow **to** and **from** a device
    - *text* or *binary*, Linux does not distinguish
- Most streams : *fully buffered*, reading/writing copy data from and to area of memory : *buffer*
    - Copying to and from a memory *buffer* is very fast
- *buffer* for output stream is *flushed* (physically written) when it becomes full or *fflush()* is called **Why**: do this?
- Input buffers refilled when empty by reading next large **chunk** of input from device or file into buffer



Streams

In addition to `stdin`, `stdout` and `stderr`

`fopen` associates a stream with a file

```
FILE *fopen(char *str, int mode);   // declaration
```

- str is string representing the file name
- mode is "r", "w", "rw" and others (man 3 fopen for more information)

Example:

```
FILE *fp = NULL;
if ((fp = fopen("inpfile", "r")) == NULL){
    // print an error to stderr
    // exit program
};
```

Specifying Streams

- `fgetc(stdin)`
- `fputc(stdout)`
- `printf(    )` same as `fprintf(stdout,     )`

```
#include <stdio.h>
#include <stdlib.h>
int
main(void)
{
    printf("An output message - this message is going to stdout\n");
    fprintf(stderr, "An error message - this message is going to stderr\n");
    exit EXIT_SUCCESS;
}
bwc@bwcsurface:~/tmp$
```

```
bwc@bwcsurface:~/tmp$ ./a.out > out 2> err
bwc@bwcsurface:~/tmp$ cat out
An output message - this message is going to stdout
bwc@bwcsurface:~/tmp$ cat err
An error message - this message is going to stderr
bwc@bwcsurface:~/tmp$
```

File Input and stdout Example

```
FILE *fopen(char *str, int mode);   // declaration
```

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

int main(int argc, char **argv) {
    FILE *fp = NULL;
    if ((fp = fopen(argv[1], "r")) == NULL) {
        fprintf(stderr, "Couldn't open file %s\n", argv[1]);
        return EXIT_FAILURE;
    }
    int c;
    while((c = fgetc(fp)) != EOF){
        fputc(c, stdout);
    }
    fputc('\n', stdout);
    return EXIT_SUCCESS;
}
```

https://edstem.org/us/courses/37726/workspaces/ - basicFileIO

C Arrays

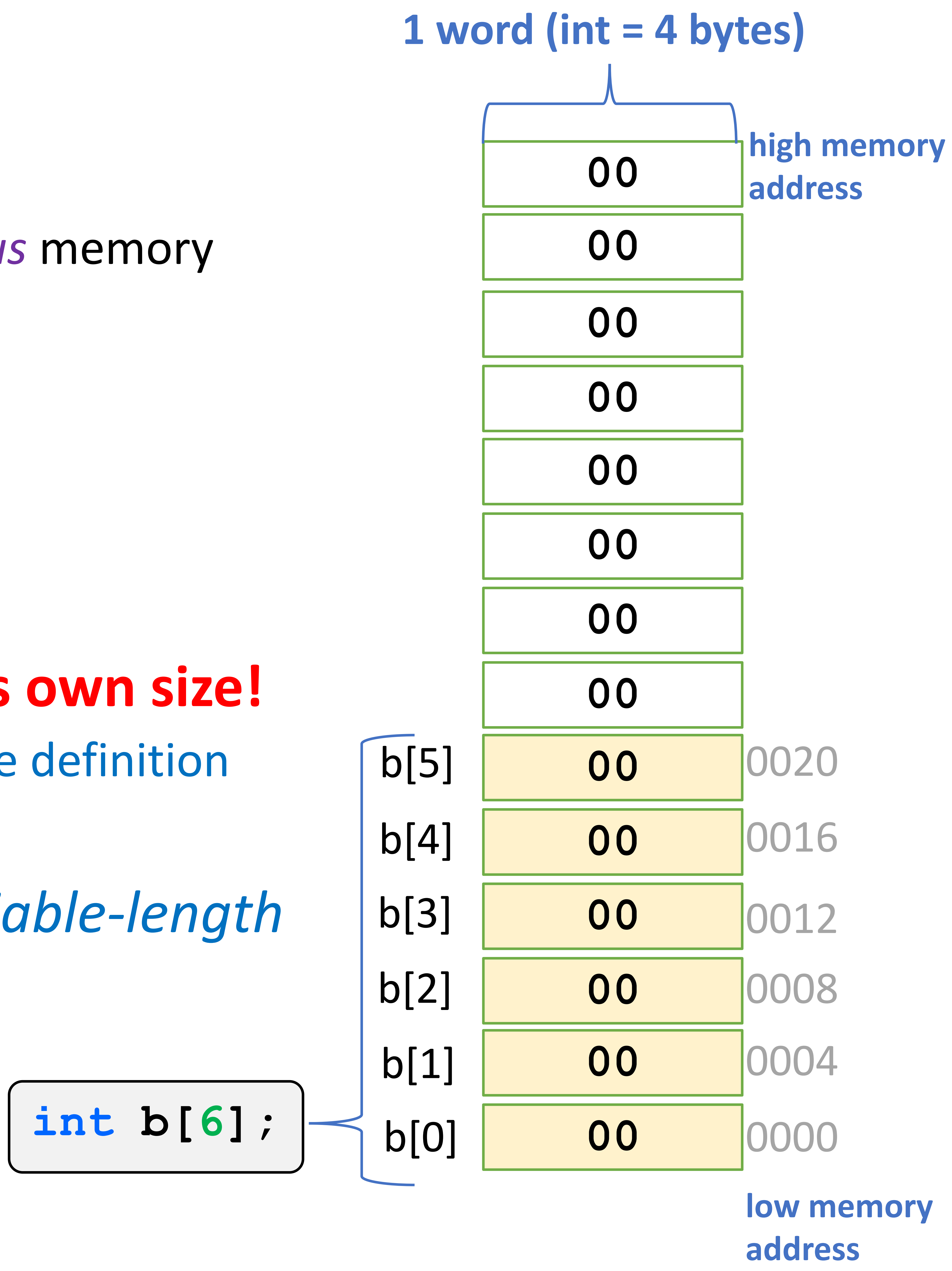
Arrays in C

- Definition: `type name[count]`
  - **Arrays are indexed starting with 0**
  - Allocates (`count * sizeof(type)`) bytes of *contiguous* memory
  - Common usage specifies compile-time constant for `count`

```
#define BSZ   6
int b[BSZ];
```

- Size of an array
  - Not stored anywhere – **an array does not know its own size!**
    - `sizeof(array)` **only works** in **scope** of array variable definition
  - Modern C versions (*not* C++) allow *automatic variable-length arrays*

```
int n = 175;
int scores[n];   // OK in C99
```



Initializing an Array in C

```
int b[5] = {2, 3, 5, 7, 11};
```

```
int b[5] = {2, 3, 5, 7, 11, 13};
```

- 13 is ignored

```
int b[] = {2, 3, 5,
           7, 11};
```

- let compiler determine the array count

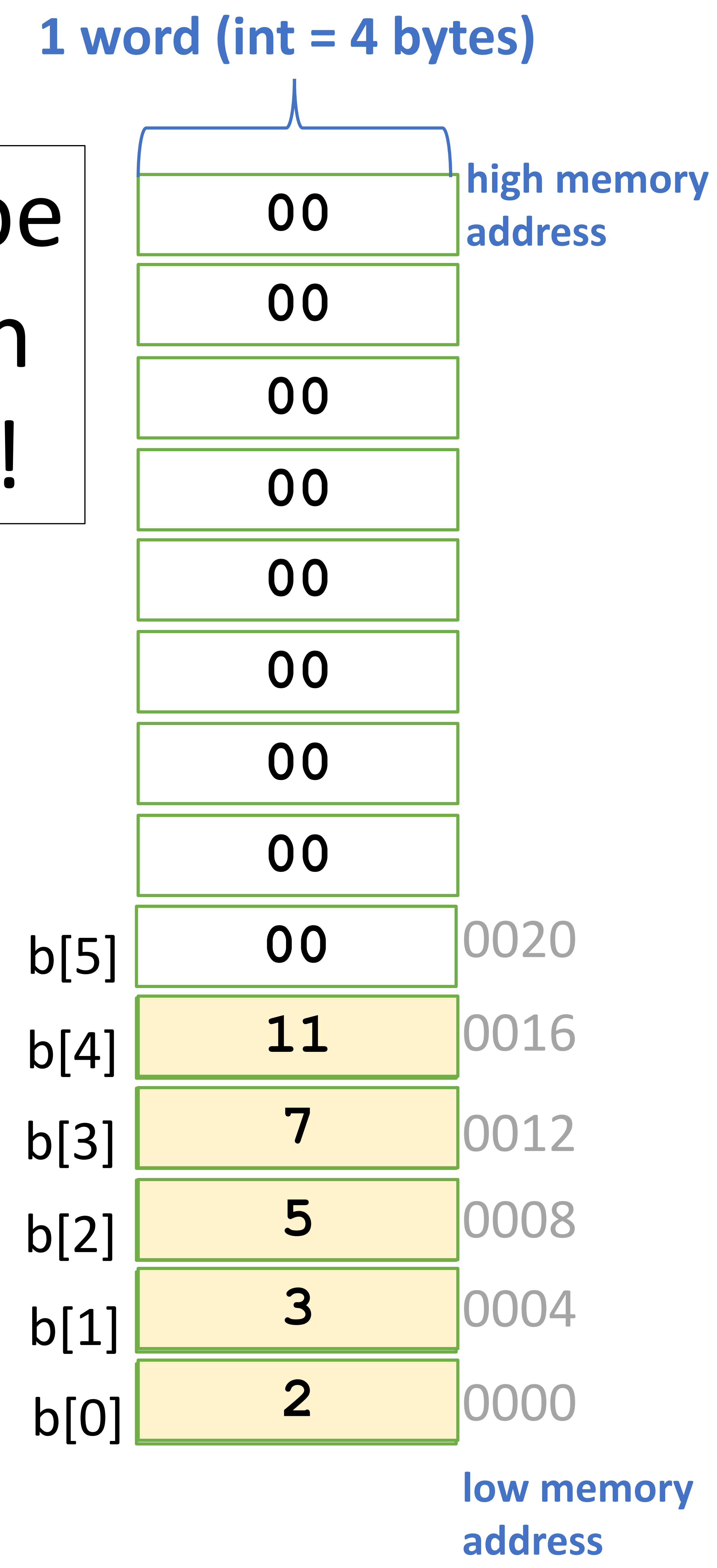
```
int arr[10] = {};
```

- fills array with 0's.

```
int arr[10];
```

- maybe initialized or not.

Arrays can be declared on the stack!!!



Working with Arrays

The size of arrays is not available readily like in Java/python. If you pass an array to a function, you also have to pass along its size.

```
int func(int [] arr, int size);
```

Arrays cannot be copied the way shown below!

```
int a[5];
int b[] = {2, 3, 5,
           7, 11};
a=b;
```

C Strings

Intermission – Strings in C

- Normally, we'd cover Strings *after* covering pointers.
- But… we want to get you up and running doing some programming in C and you need to know how C handles Strings to do almost anything in C.
- So….
    - First, a crash course on Strings, then back to our regularly scheduled lesson

Chars

- Char

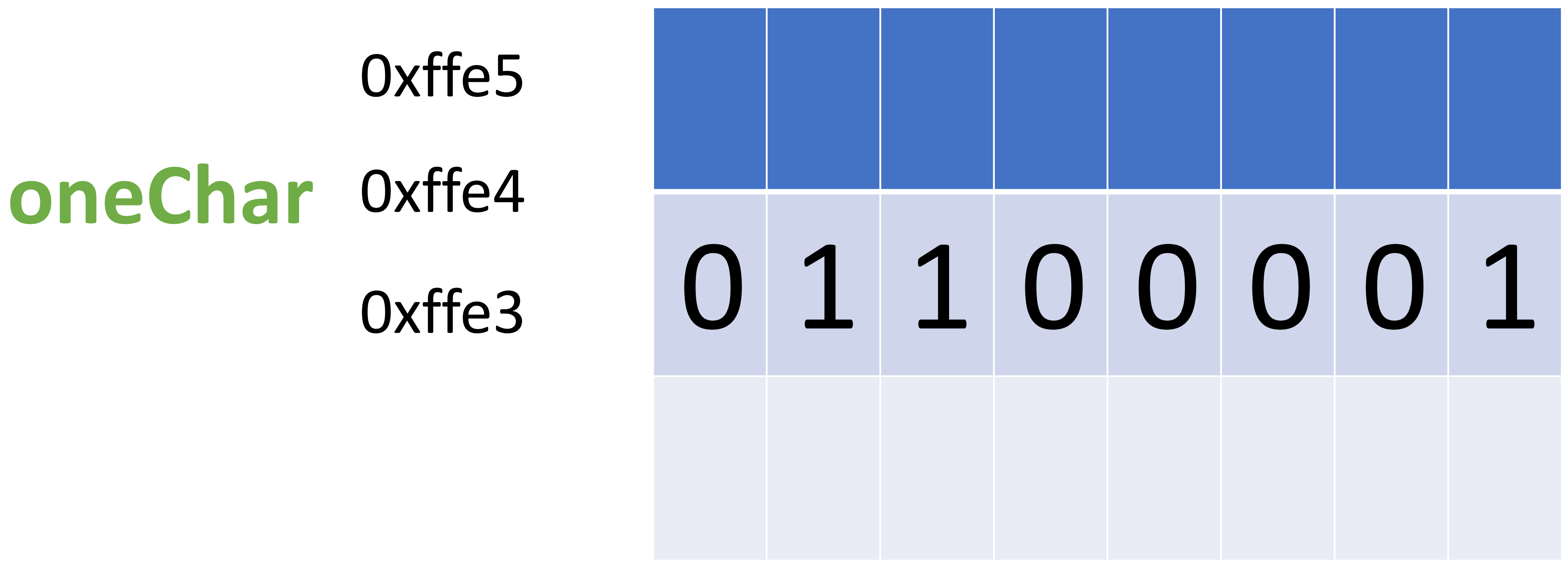
  basic data type (one byte)

- ASCII (UTF-8) character is delimited by single quotes (‘ ‘)

- Char is just a number, so you can do math on it.

```
char oneChar = ‘a’;
```

```
char oneChar = 0x61;  // same as ‘a’
```




```
oneChar =  oneChar + 1;  // same as ‘b’
                         // same as 0x62
```

C Strings

- C has no dedicated variable type for strings
  - Instead, a string is represented as an **array of characters** with a special ending sentinel with a value '\0' (zero)

`"Hello"`

index	*0*	*1*	*2*	*3*	*4*	*5*
char	`'H'`	`'e'`	`'l'`	`'l'`	`'o'`	`'\0'`

- `'\0'` is the **null-terminating character (zero - do not confuse with '0')**
  - you always need to allocate one extra space in an array for it
  - a string does not always have '\n' (do not depend on '\n' being right before the '\0')
- Strings are **<u>not</u>** objects
  - They do not embed additional information (e.g., string length). You must calculate this!
- You can use the C string library **`strlen`** function to calculate string length
  - The null-terminating character does *not* count towards the length.

```
int length = strlen(myStr);          // length = 5
```

Caution: `strlen` is `O(N)` because it must scan the entire string!
You should save the value if you plan to refer to the length later.

C Strings

- mess1 is an array with enough space to hold the string + '\0'
  - you can change array contents but not what mess1 points at

```
char mess1[] = "Hello World";
```

- mess2 is an array with enough space to hold the characters but does not have space for the '\0' SO IT IS NOT A VALID STRING
  - Since this is NOT '\0' terminated, string library functions will not work properly.

```
char mess2[] = {'H','e','l','l','o',' ','W','o','r','l','d'};
```

'\0'

	contents	Byte address
	00000000	0x0100000f
	00110000	0x0100000e
	00110011	0x0100000d
	01000101	0x0100000c
	00000000	0x0100000b
	'd'	0x0100000a
	'l'	0x01000009
	'r'	0x01000008
	'o'	0x01000007
	'W'	0x01000006
	' '	0x01000005
	'o'	0x01000004
	'l'	0x01000003
	'l'	0x01000002
	'e'	0x01000001
mess1	'H'	0x01000000

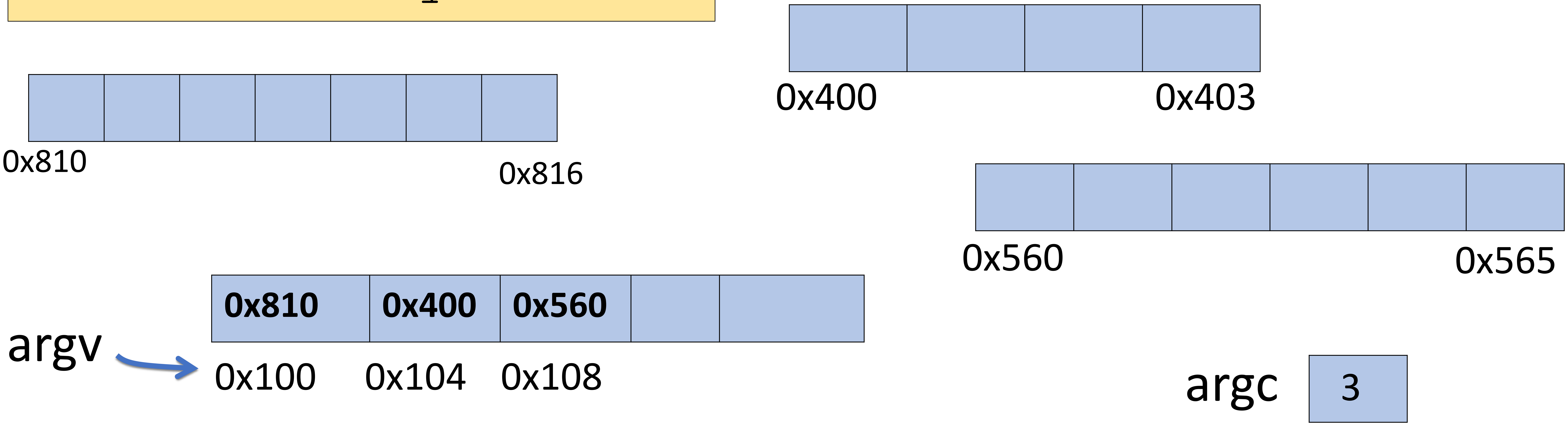
C Standard String Library (some useful functions)

- `size_t strlen(const char *s);`
- `char *strcpy(char *s0, const char *s1)`
- `char *strncpy(char *s0, const char *s1,size t n)`
- `char *strcat(char *s0, const char *s1);`
- `char *strncat(char *s0, const char *s1, size t n);`
- `int strcmp(const char *s0, const char *s1);`

Argv is a Pointer to Pointers

```
int main (int argc, char **argv){
    …
}
```

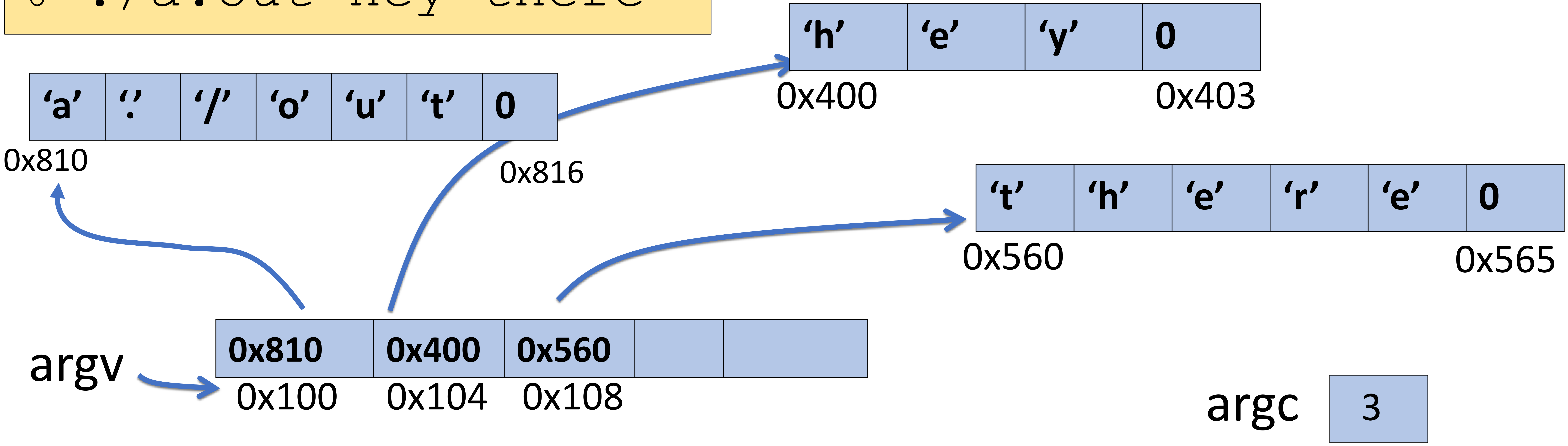
```
% ./a.out hey there
```



Argv is a Pointer to Pointers

```
int main (int argc, char **argv){
    …
}
```

```
% ./a.out hey there
```



Good news – array [] syntax works for pointers to arrays!!

Because char **argv is a pointer to an **array of char pointers**

- So argv[0] gives you a char *, which is a pointer to **an array of chars**
- Which means argv[0] gives you the first “string” in the array

Because argv[0] is a char * that is a pointer to **an array of chars**

- You can say argv[0][0] to get the first character in the first “string”

What is the output of this code?

```
int main (int argc, char **argv){
    printf(“%s”, argv[2]);
}
```

```
% ./a.out how are you?
```

A. ./a.out
B. how
C. are
D. you?
E. a

What is the output of this code?

```
int main (int argc, char **argv){
    printf("%c", argv[1][2]);
}
```

```
% ./a.out how are you?
```

A. a
B. h
C. w
D. r
E. None of the above

What is the output of this code?

```
int main (int argc, char **argv){
    printf("%c", argv[1][3]);
}
```

```
% ./a.out how are you?
```

A. .

B. ←Null char

C. ←space

D. a

E. segfault

Let’s look at this in more detail

```
int main (int argc, char **argv){
    printf(“%c”, argv[1][3]);
}
```

```
% ./a.out how are you?
```

C Strings As Parameters

- When we pass a string as a parameter, it is passed as a **char ***
- C passes the location of the first character rather than a copy of the whole array

```
int doSomething(char *str) {
        ...
        str[0] = 'c’;          // modifies original string!
        printf("%s\n", str);   // prints cello
}

char myString[] = “Hello”;  // defines space and initializes
...
doSomething(myString);
```

Summary

- C is a valuable language that offers high performance
- Many programming constructs are similar between Java/C
  - Loops, if statements, etc.
- C programs have .h files in addition to .c files
- Arrays and Strings have important differences in C
  - Arrays can be allocated on the stack in C
  - Strings (just char[]) require null termination