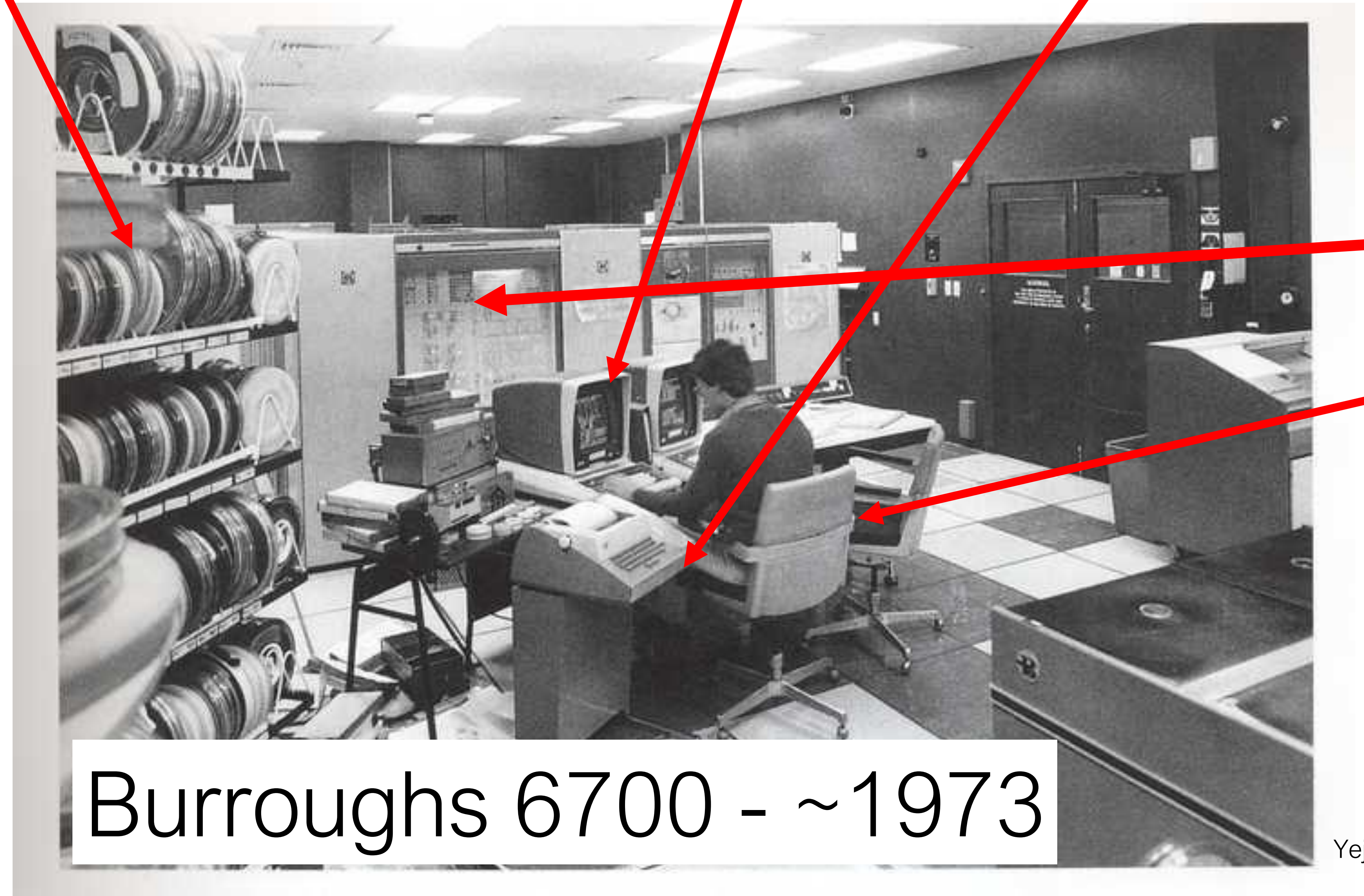
CSE 30
Computer Organization and Systems Programming
C Programming

Magnetic Tapes

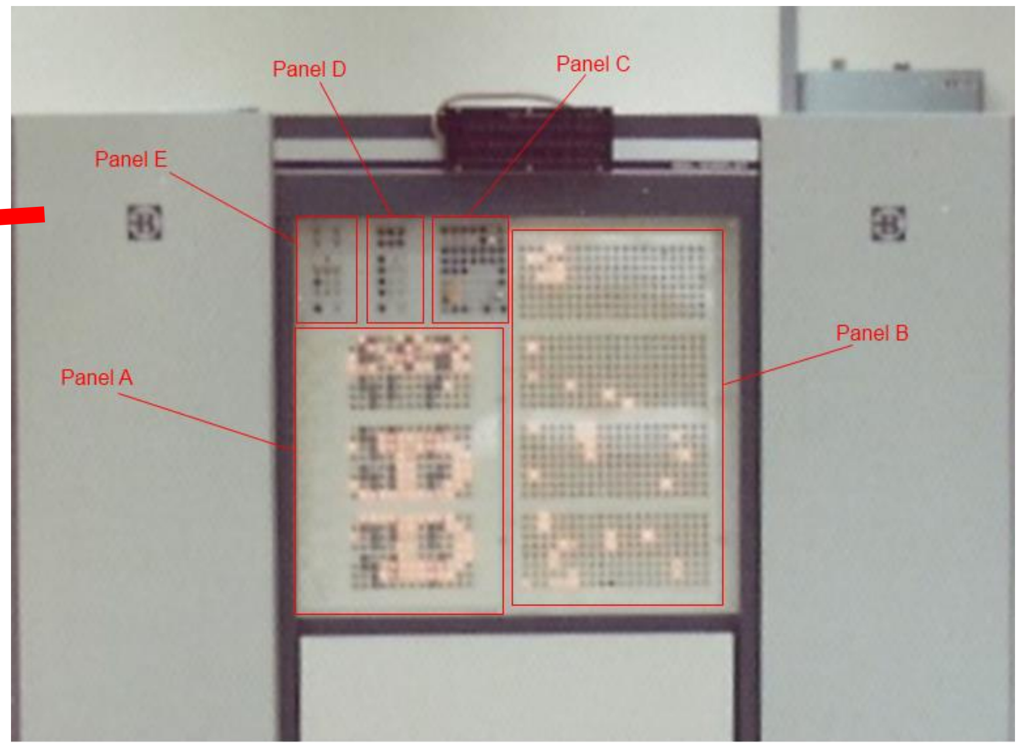
CRT

TTY – 11 or 30 cps

Chair



1 to 3 CPUS
5 to 10 MHz



Console (blinking lights!)

https://4310b1a9-a-b8516d33-s-sites.googlegroups.com/a/retrocomputingtasmania.com/home/home/projects/elliott-503/dsir-503/BarVict185a.jpg?attachauth=ANoY7cr_NoRSxIOHcur8tDlA2au5DvyvQvu1JvSXmg4ymAQupk1qJvcx9SlIJXDIGgaxxDczPzpKVMOWkG-0ujg36RmW0zrtngnns5I5ZUXY9SjotMO8cCzIph74PP-qoxGIgGetouYOtvJB-CB0l8M-Yej8t2QotyZ8rQFQfuJpujywYUnB793uoUR5Bsu5bKZToIK1n2CoMvosn5IwjkCveuoiShDr8MCvn94E_-E1ISVwodlh48y4Jkr6BYKyPa3C9gNzOmlaEj2ZSu7A9JuZ-GqPtg1PWw%3D%3D&attredirects=0

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));
}
```

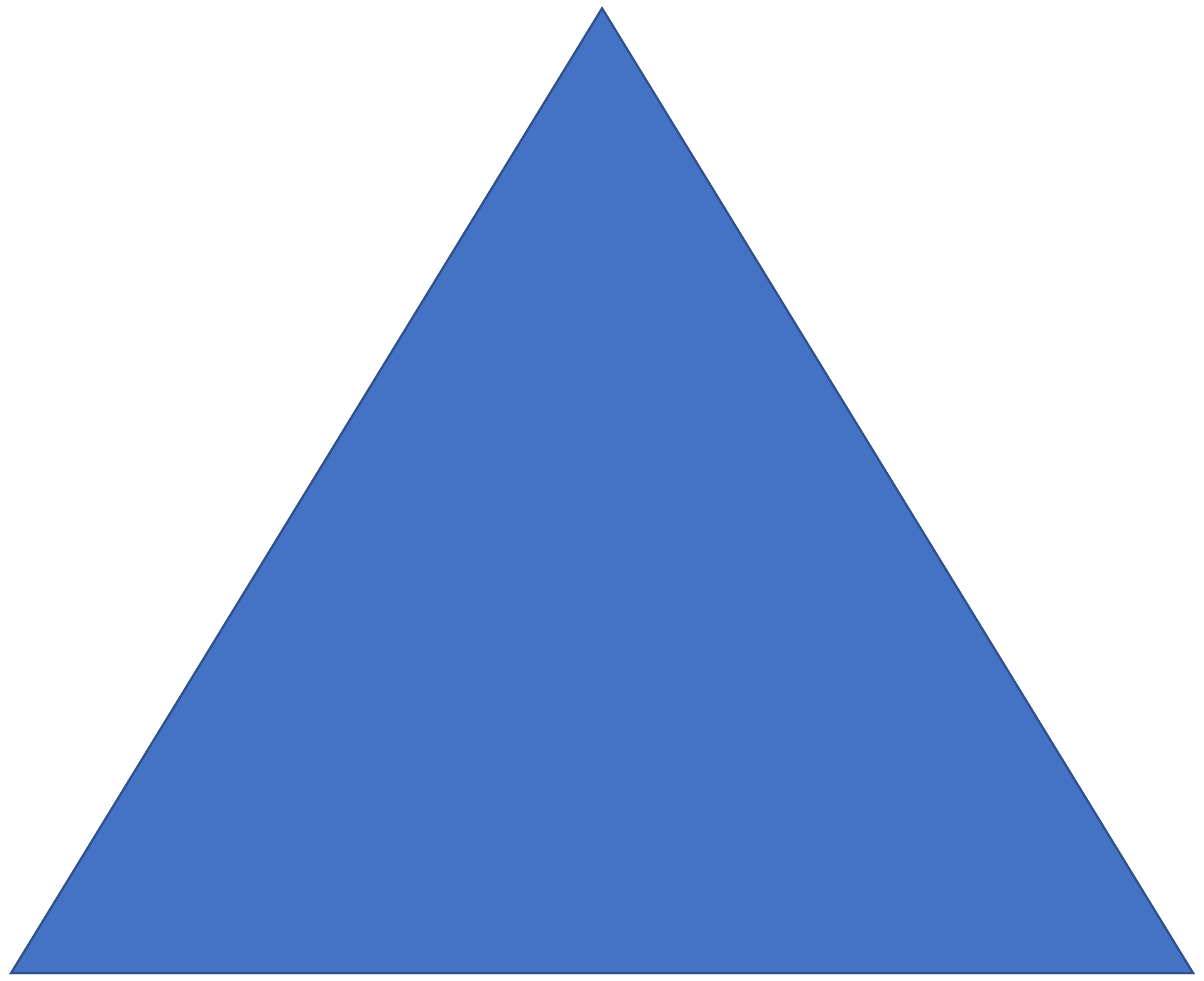
function definition

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



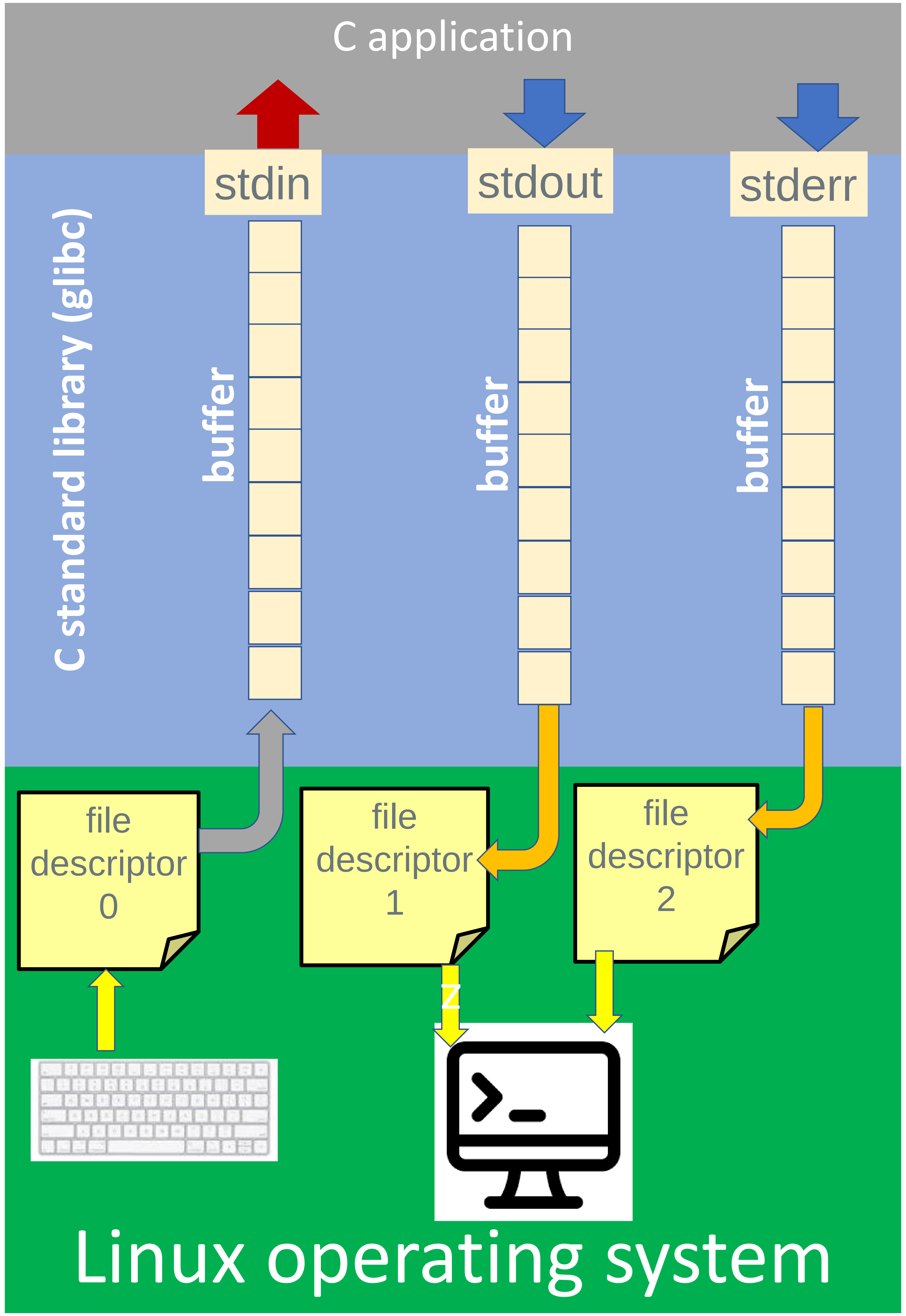
HW 2 and I/O

```
./encrypter 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



Specifying Streams

`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$
```

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
};
```

File Input and stdout Example

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

```
#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);
    fclose(fp);
    return EXIT_SUCCESS;
}
```

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

C Arrays

Arrays in C

ex float a[10]
→ 40 bytes
as float = 4 bytes
10 inputs in array

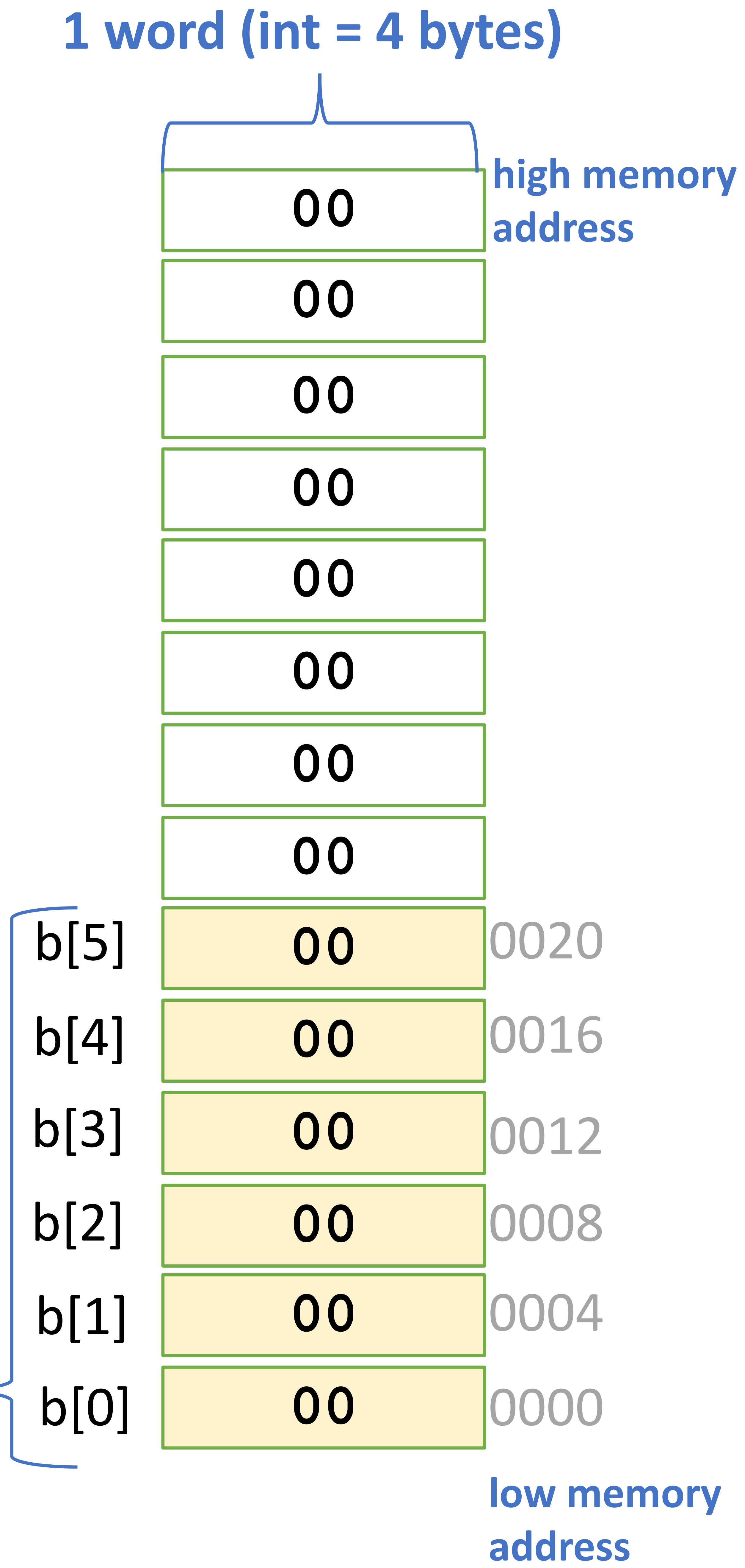
- Definition: `type name[count]`
  - Arrays are indexed starting with 0
  - Allocates (`count * sizeof(type)`) bytes of *contiguous* memory
  - Common usage specifies compile-time constant for `count`

```
#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
```

```
int b[6];
```



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

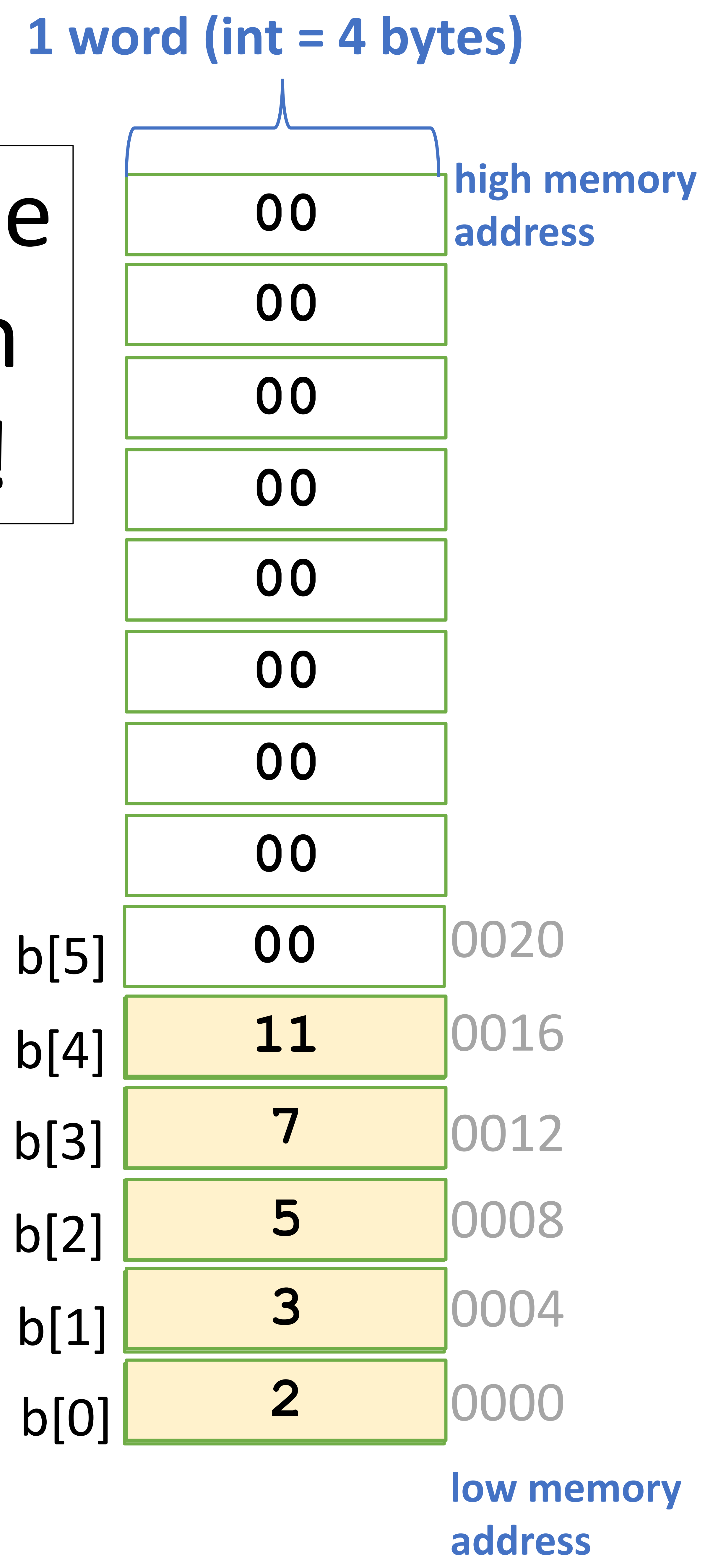
```
int arr[10] = {};
```

- 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

Chars

```
char oneChar = 'a';
char oneChar = 0x61;   // same as 'a'
```

- Char

  basic data type (one byte)

	0xffe5								
oneChar	0xffe4	0	1	1	0	0	0	0	1
	0xffe3								

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

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

```
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)

index	*0*	*1*	*2*	*3*	*4*	*5*
"Hello" *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 **not** 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'};
```

	contents	Byte address
	00000000	0x0100000f
	00110000	0x0100000e
	00110011	0x0100000d
	01000101	0x0100000c
‘\0’	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);`
- `char *strdup(const char *s);`

For the PA: What is the output of this code?

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

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

argv[0] argv[1] argv[2]

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

https://edstem.org/us/courses/37726/workspaces/ argv printargv2.c

We will revisit argv later after we discuss pointers

Variable Scope and Lifetime

```
int my_global_variable = 7;  // file scope, program lifetime

int f(void)
{
    int my_local_variable = 11; // block scope, function lifetime
    //...
}
```

Scope

where can you access it

Lifetime

for how long can it retain its value

Examples – Local and Global

Local variable

```
int foo(int x){
    int aLocalVar;
    aLocalVar = 7;
        {
        // nested block
        int aLocalVar = 4;
    }
    x = x + aLocalVar;
    return (x);
}
```

scope : local to function
lifetime : duration of function

Global variable

```
int GlobalVarX = 10;
int GlobalVarY;

int main(){
    GlobalVarY = GlovalVarX;
    foo();
    printf("%d\n", GlobalVarY);
}

void foo() {
    GlobalVarY = -1;
}
```

scope : entire program
lifetime : entire program

Examples – Static File and Function

Static file variable

```
static int someStaticLocal;
int foo(){
   someStaticLocal++;
   return(someStaticLocal);
}
void foo2(){
  someStaticLocal=5;
  printf("%d\n", someStaticLocal);
  foo();
  printf("%d\n", someStaticLocal);
}
```

scope : file
lifetime : entire program

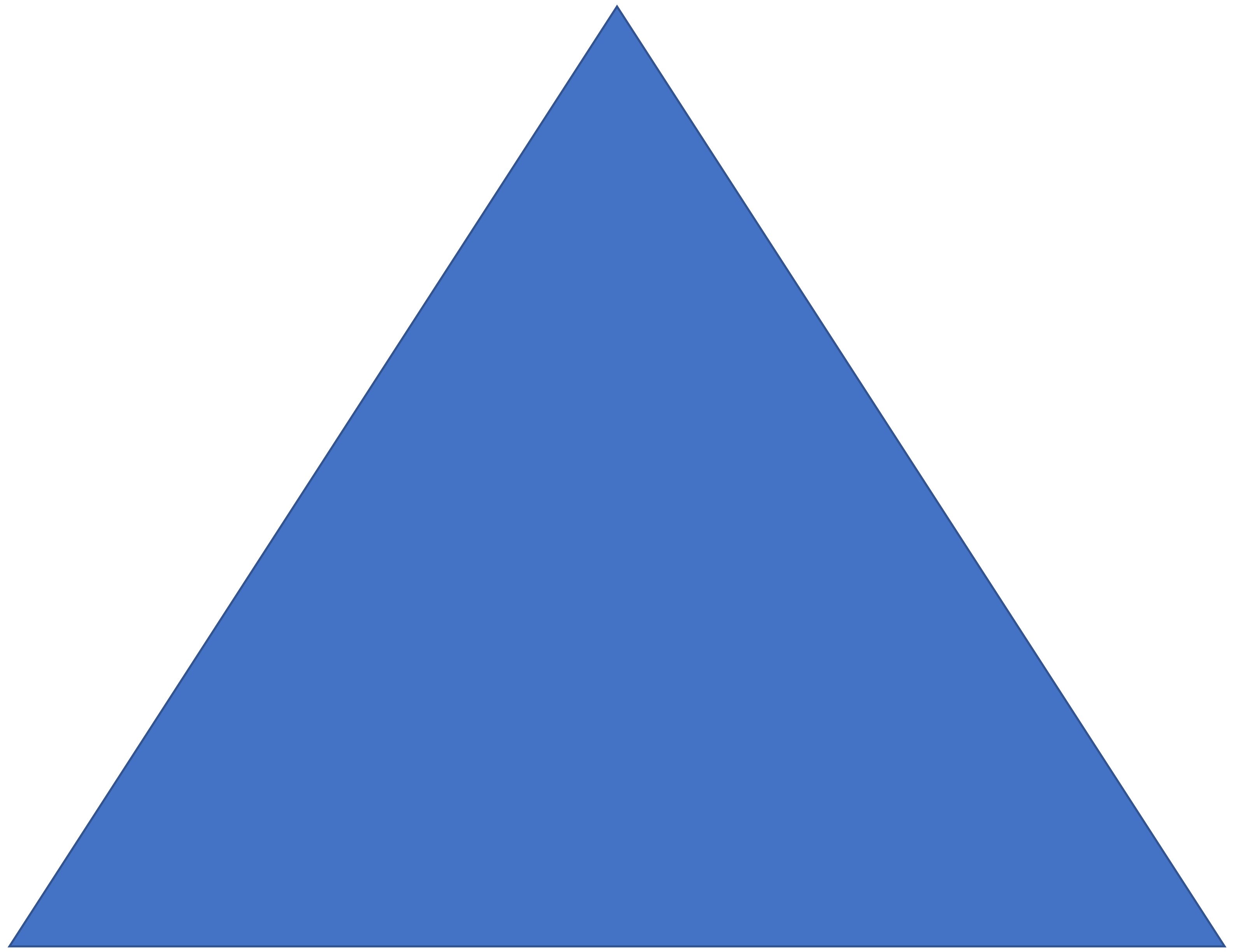
Static local variable

```
int foo(){
   static int staticVarX = 0;
   staticVarX++;
   return(staticVarX);
}

int main(){
  printf("%d\n", foo());
  printf("%d\n", foo());
}
```

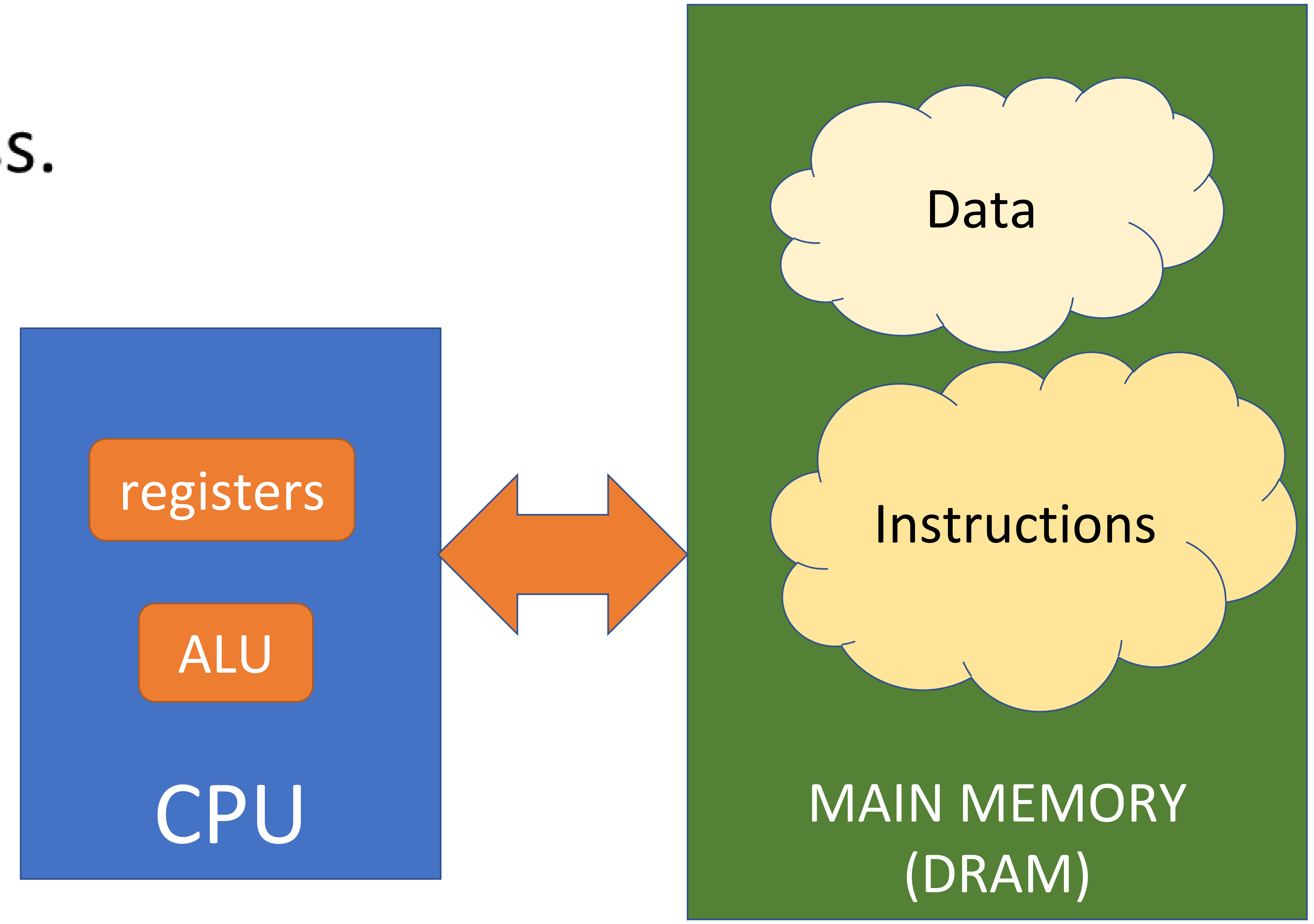
scope : function
lifetime : entire program

C and Memory



Programmer's Model of the Computer

- Sequence of 8-bit Cells (bytes) in a linear arrangement (like an array of bytes)
- Each cell can be accessed by a # called its address.
- Units of Memory
  - Bit
  - Byte (addressable unit)
- Size of memory (powers of 2 so K = 1024)
  - KB - 2^{10}
  - MB - 2^{20}
  - GB - 2
  - TB
  - PB
  - EB



Memory Location

- Range of addressable memory.

- ARMv6 - 32-bit architecture
  - Addresses are 32 bits wide
  - 32 bit registers
  - how much memory could one reach with a 32-bit address?

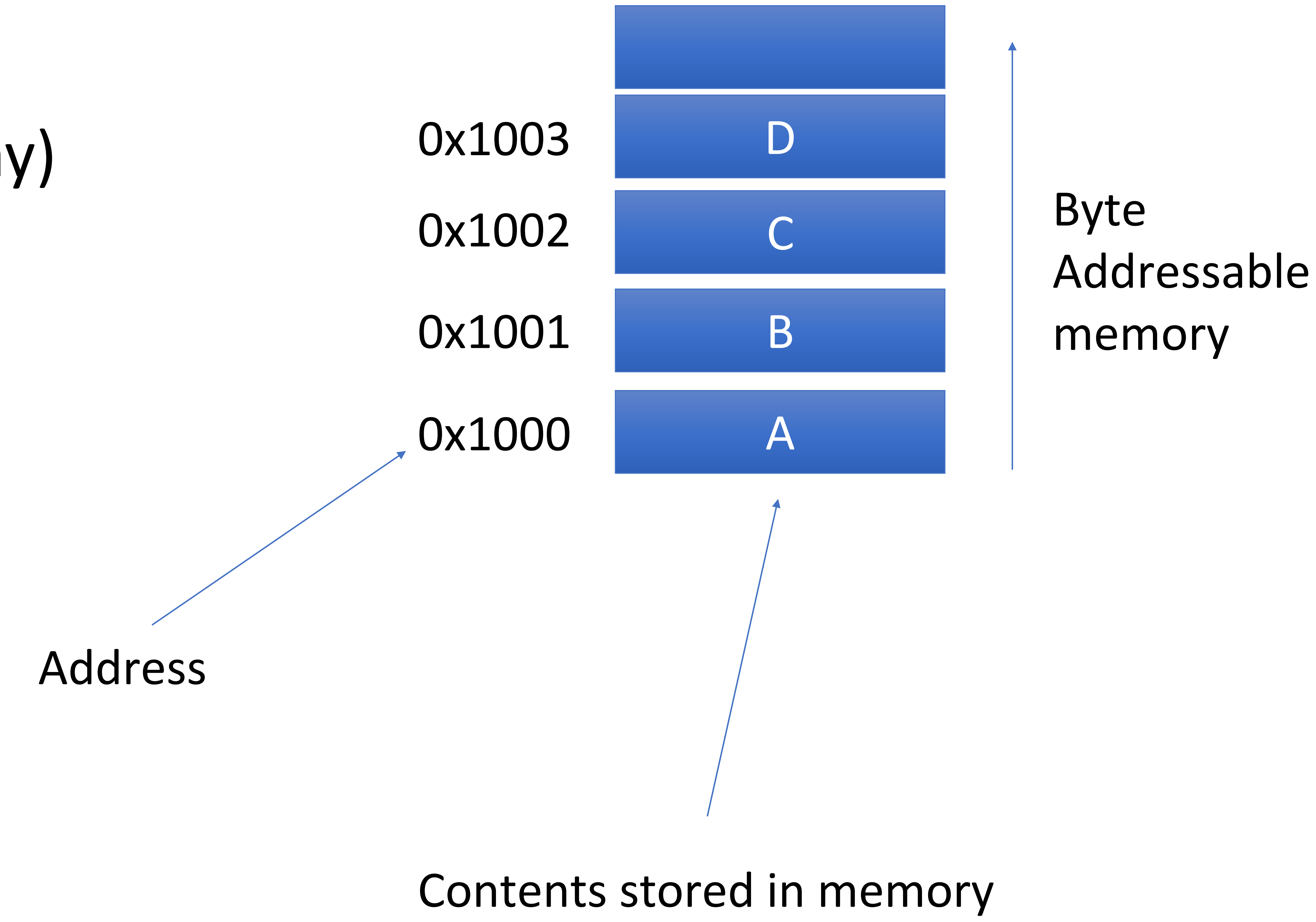
  - Instructions are all 32 bits *
- C - addresses visible to the programmer (Java no!!!)

*There are some arm instructions that are shorter than 32 bits, but we won’t be using them.

- X86 started as a 16 bit address architecture, then extended to 32 then 64
- ARM started as 32 bit address architecture, ARMv8 extends to 64 bits.

Memory Addressing – “hello memory, how are you?”

- Address – name of a memory location
  - like “Fred”, only it’s a number
  - Organized sequentially (like a large array)
- Contents of memory
  - Each location stores 8-bits (1 byte)



How much memory could a 42-bit address access?

A. 1 TB

B. 2 TB

C. 128 TB

D. 4 TB

Can we change the location of a variable after it is **defined**?

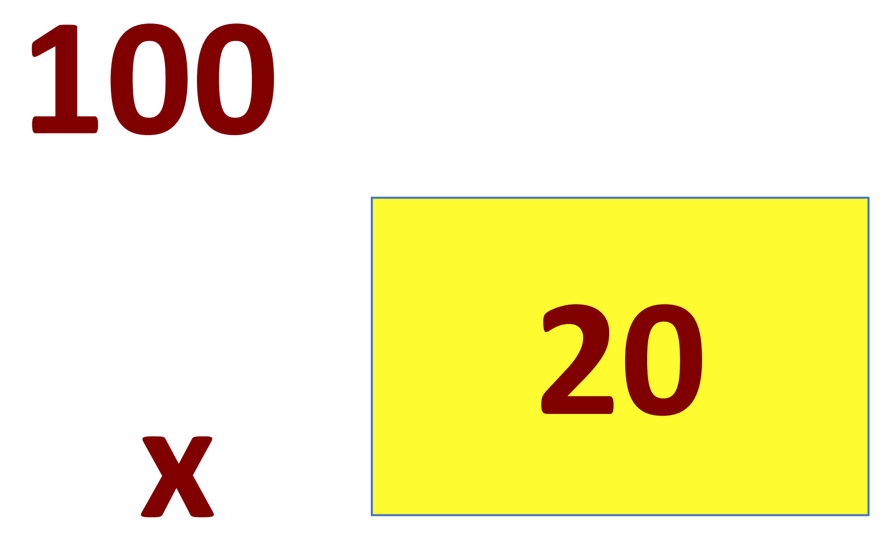
A. Yes

A. No

Accessing value, Lvalue and Rvalue

To access/change the value of a basic type:

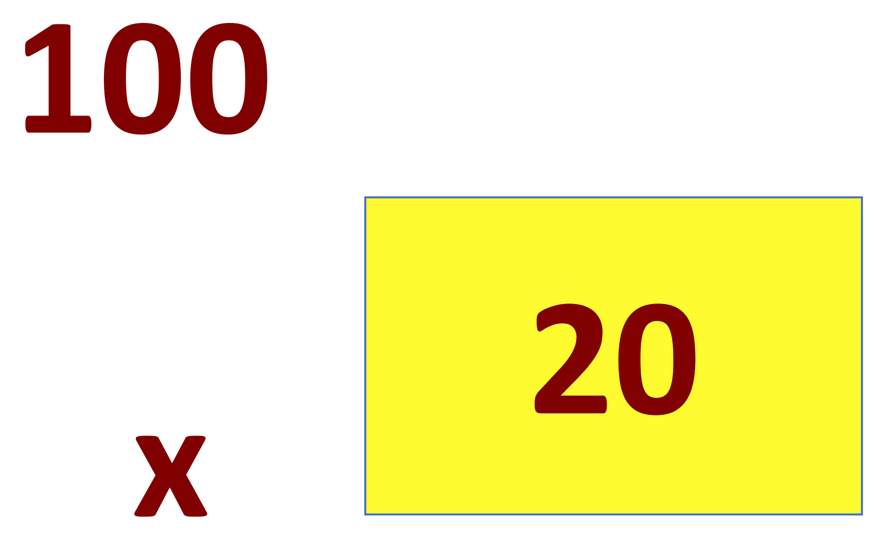
```
int y = x;
x = 10;
y = x > y ? x : y;
```



Accessing location

To access the location/address, use the address operator ‘&’

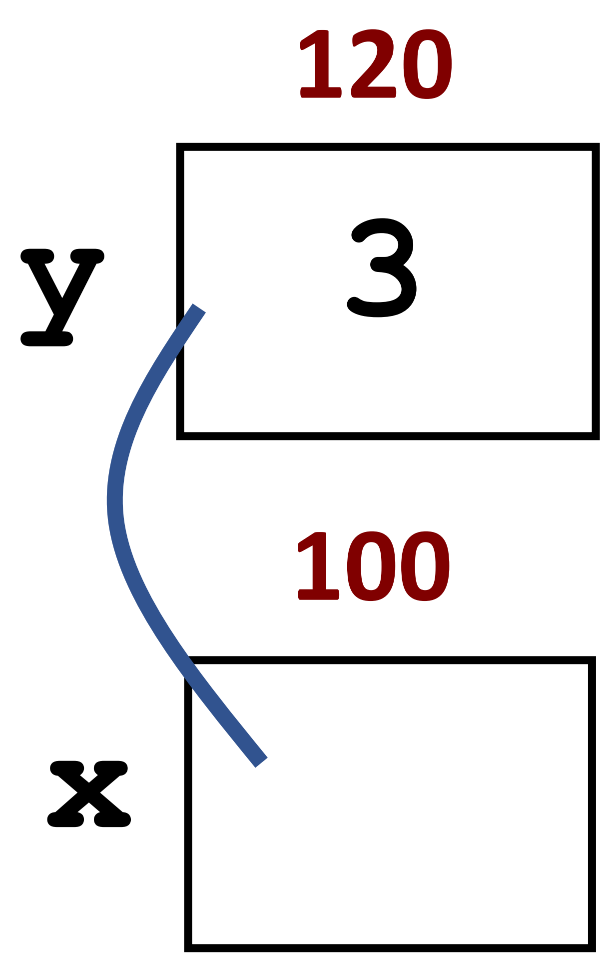
&x is 100



Gernally, a pointer’s width is the address size of the machine

(e.g. in ARM versions v6, pointers are 32-bits)

Getting an address into a pointer




```
int y;
int *x;

x = &y;
```

- In this context "&" means get the address of the variable
- `&y;` returns the address of `y` (not its value)
- `int *x` defines x as a pointer to an int
- `x` is a pointer to an int type and so is compatible with `&y`
- **&y is not an integer**

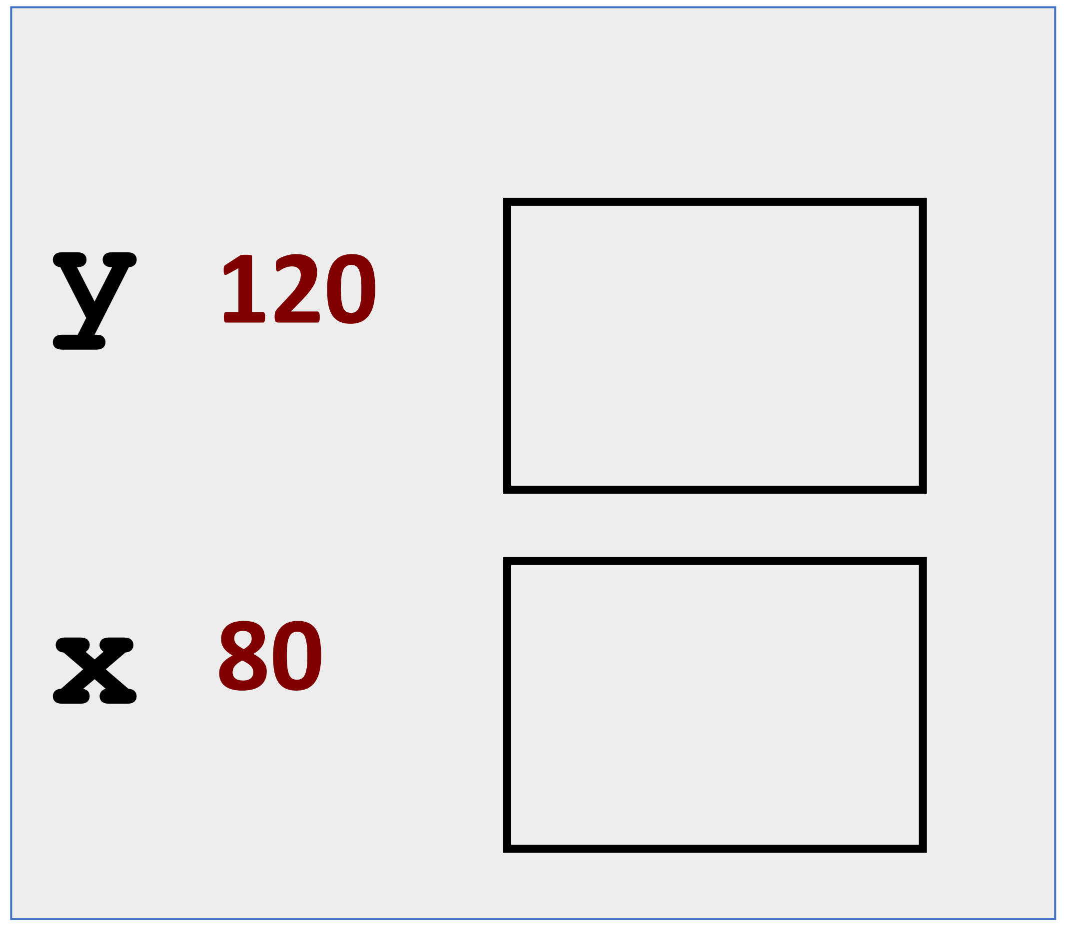
Pointers (1)

- Pointer: A variable that contains the <u>address</u> of a variable

```
int *x, y;

y = 3;

x = &y;
```



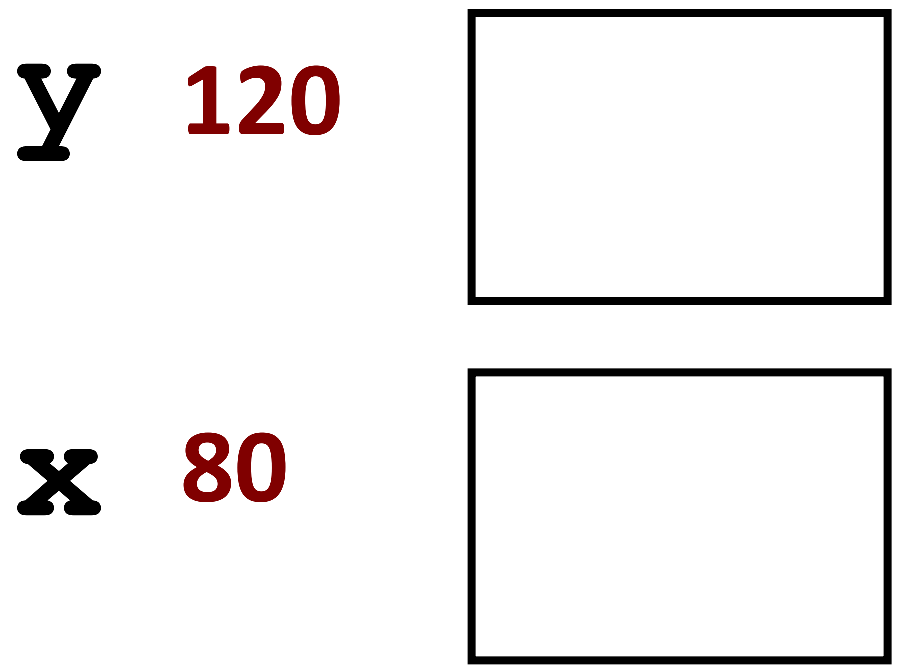
Pointers (2)

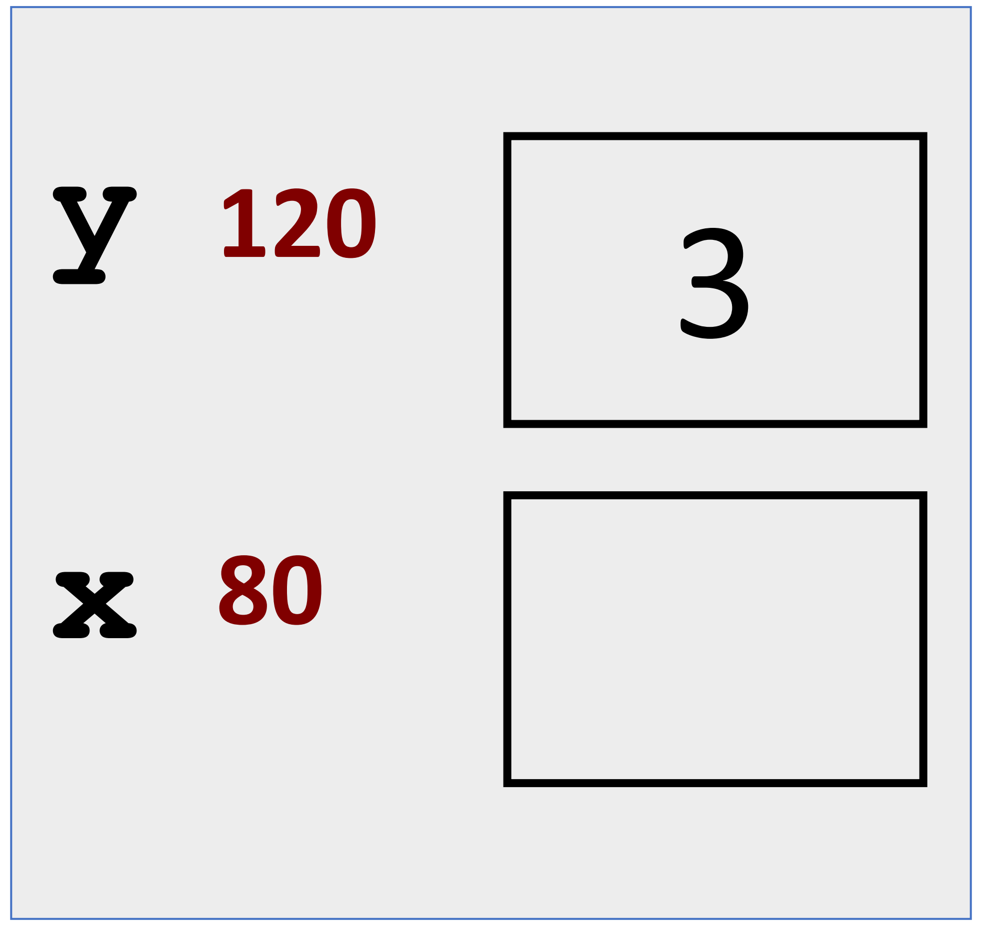
- Pointer: A variable that contains the <u>address</u> of a variable

```
int *x, y;

y = 3;

x = &y;
```





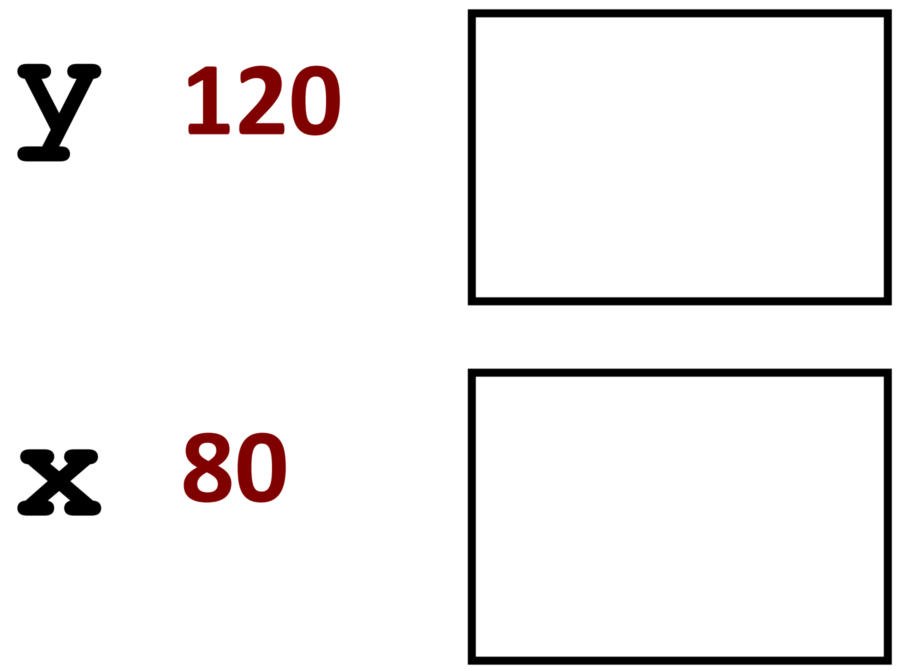
Pointers (3)

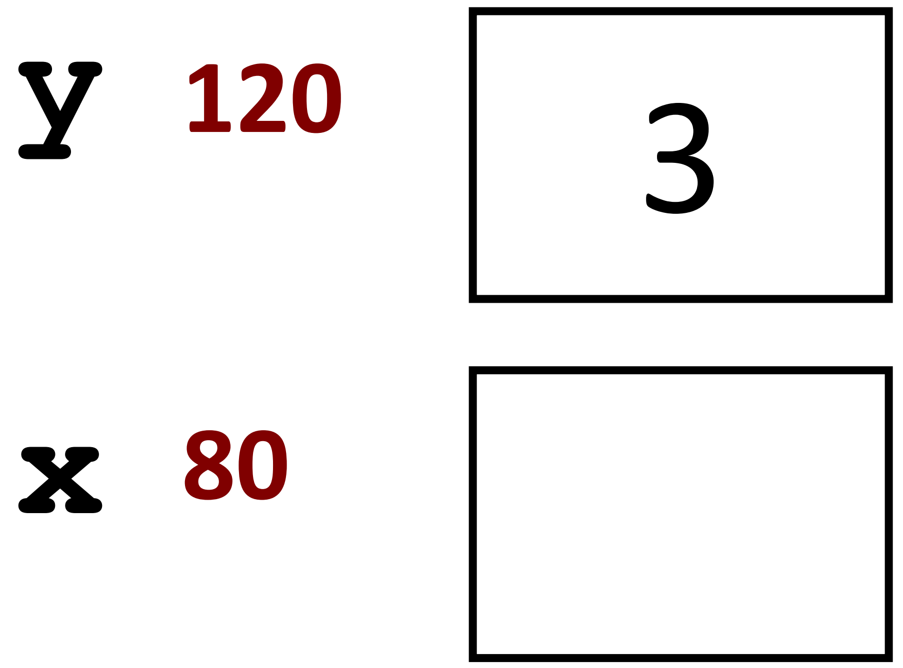
- Pointer: A variable that contains the <u>address</u> of a variable

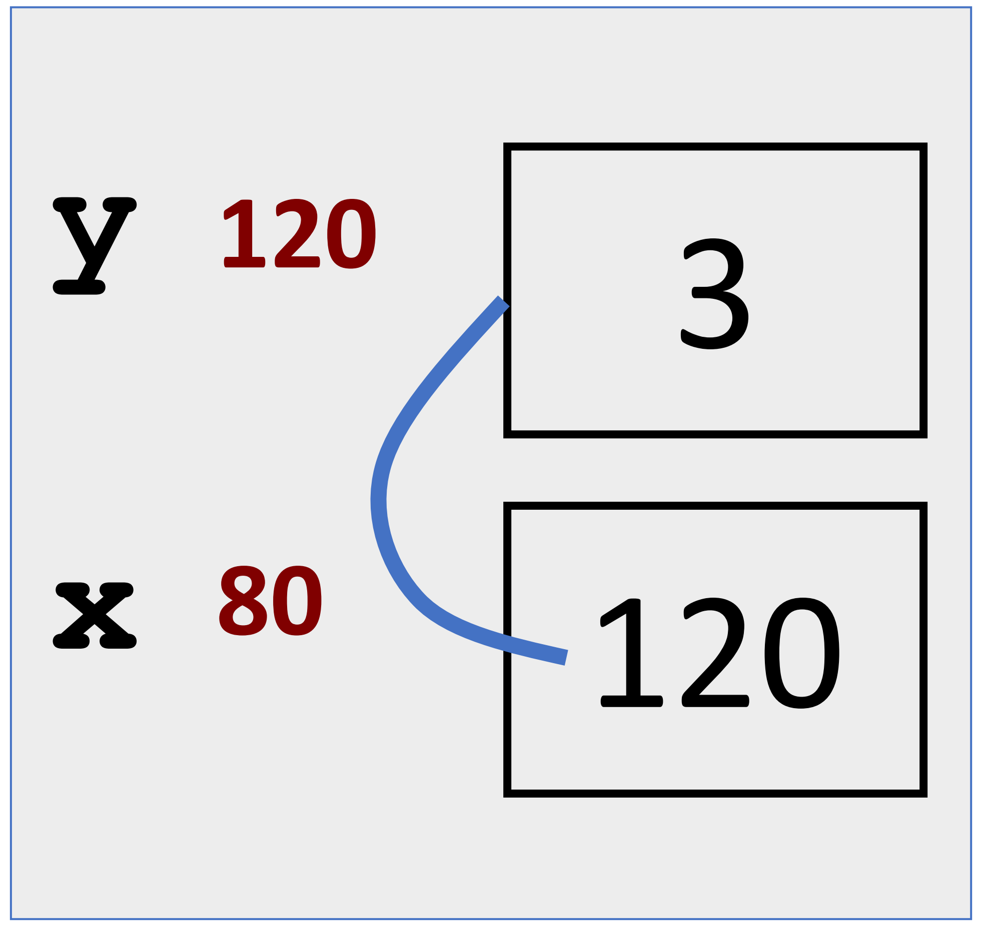
```
int *x, y;

y = 3;

x = &y;
```



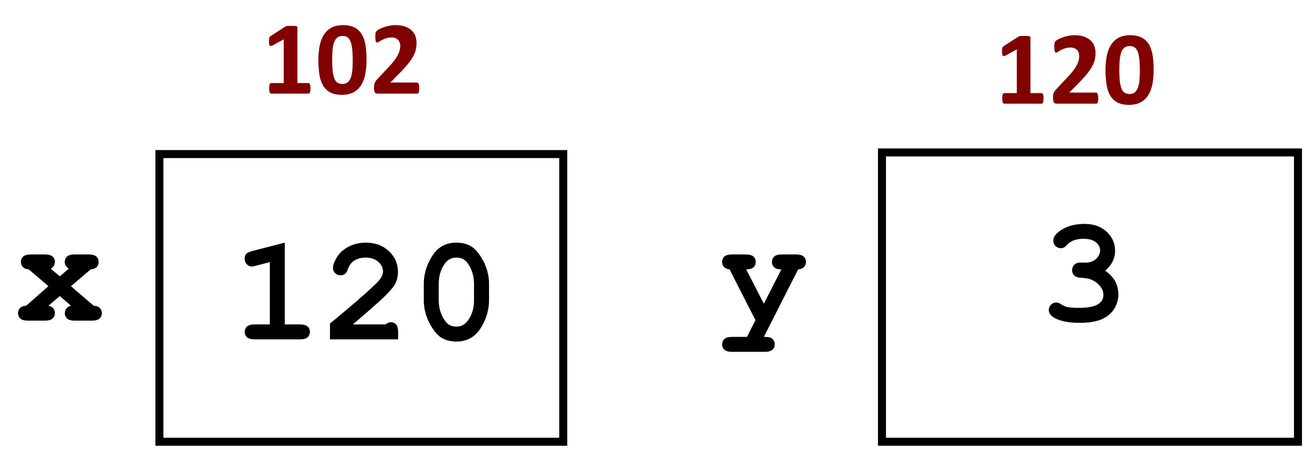


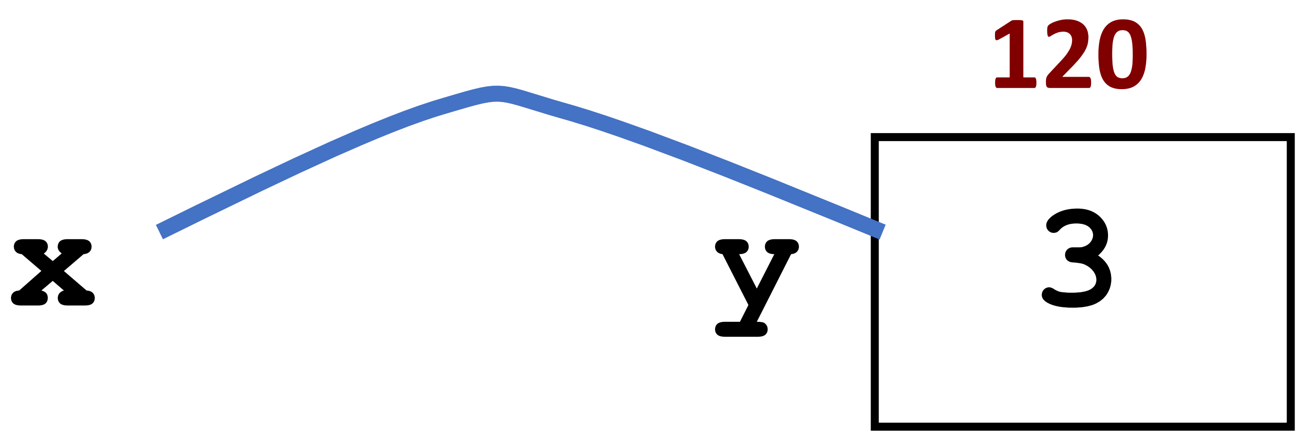


What is sizeof(x)?

Pointer Diagrams

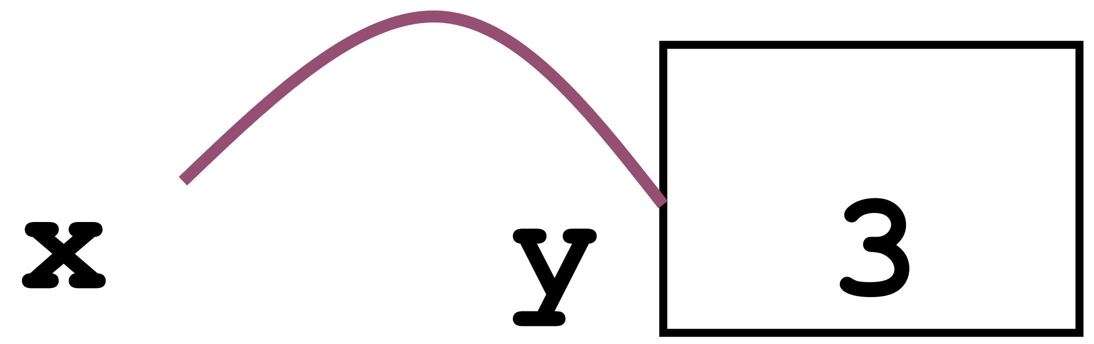
- Short hand diagram for the following scenario



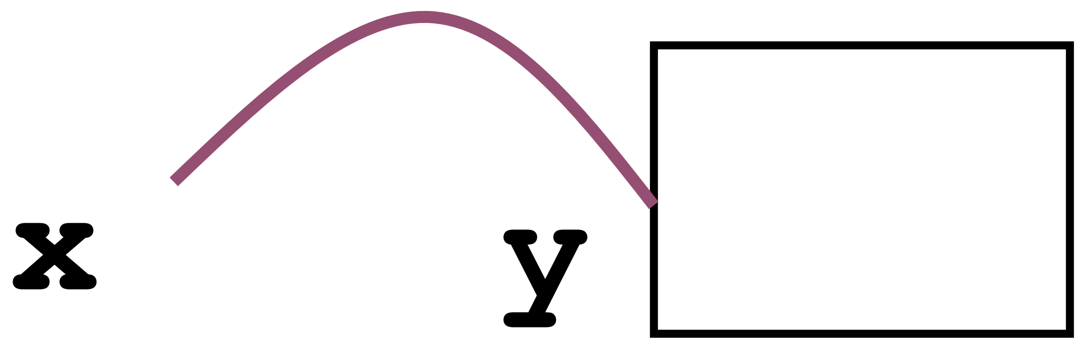


Reference through a pointer

- Use dereference * operator to left of pointer name
- "*" is high precedence.



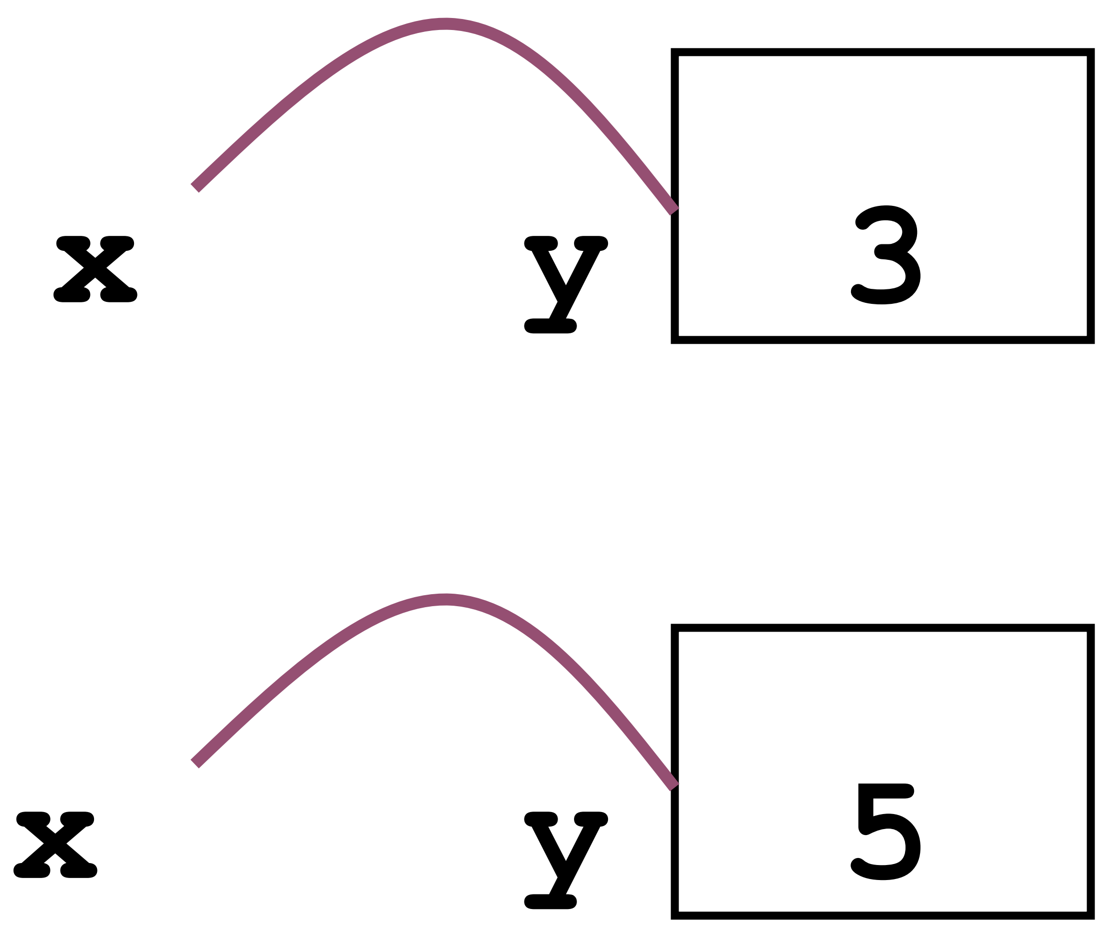

```
*x = 5;
```



Pointer Dereference(2)

- Two ways of changing the value of any variable
- Why - will be clear when we discuss functions and pointers

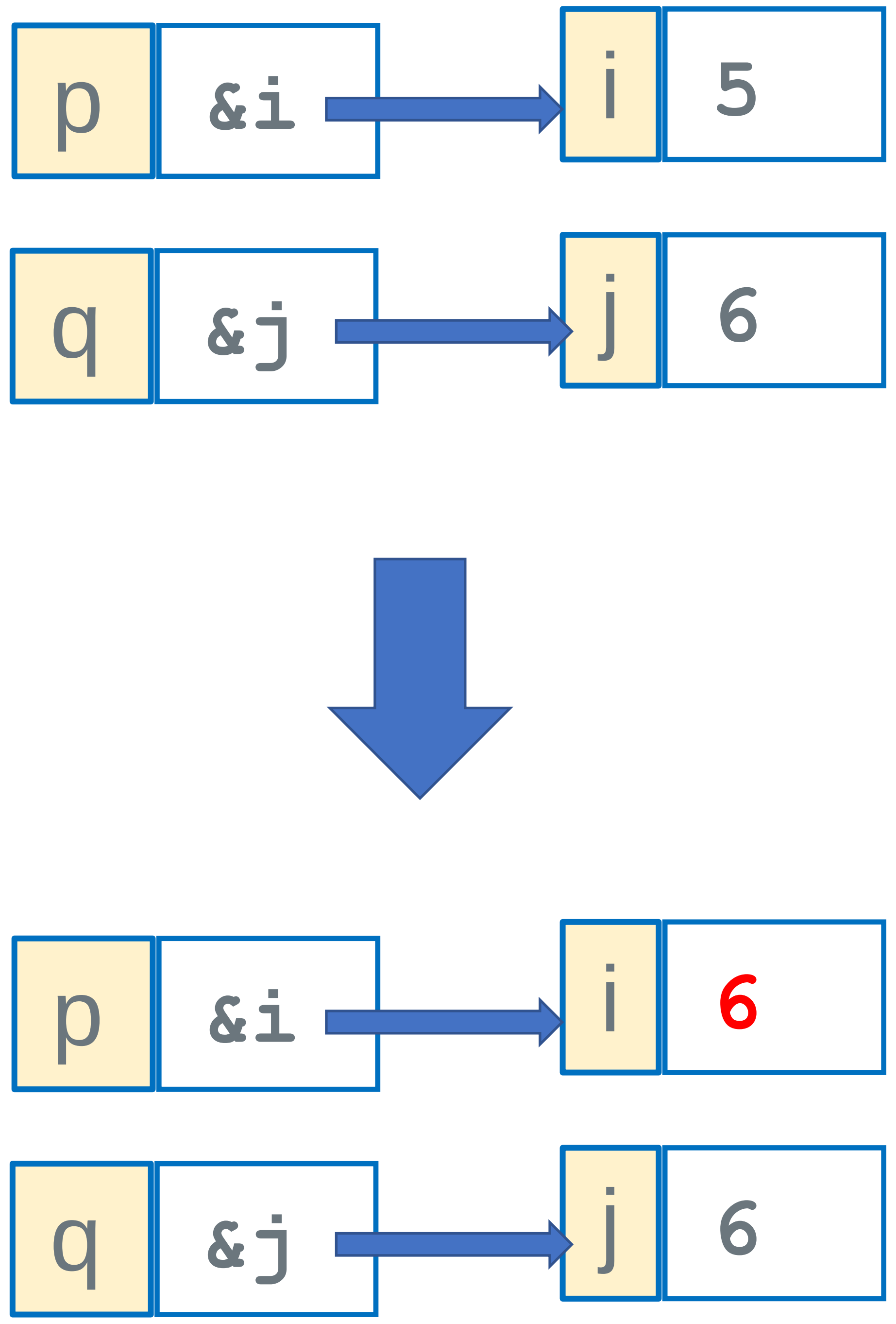
```
*x = 5;
```



Dereferencing On Both Sides of an Assignment

```
int i = 5;
int j = 6;
int *p = &i;
int *q;
q = &j;


*p = *q;
```



***p = *q;**

- ***q** on the **Rside**, reads contents of q to get an address, then * says gets the contents at address
- ***p** on the **Lside** says **destination address** is **in the contents** of p

changes the value *of* ***what p points at*** *to be* ***the value of what q points at***

- ***does not*** *change the contents of p*

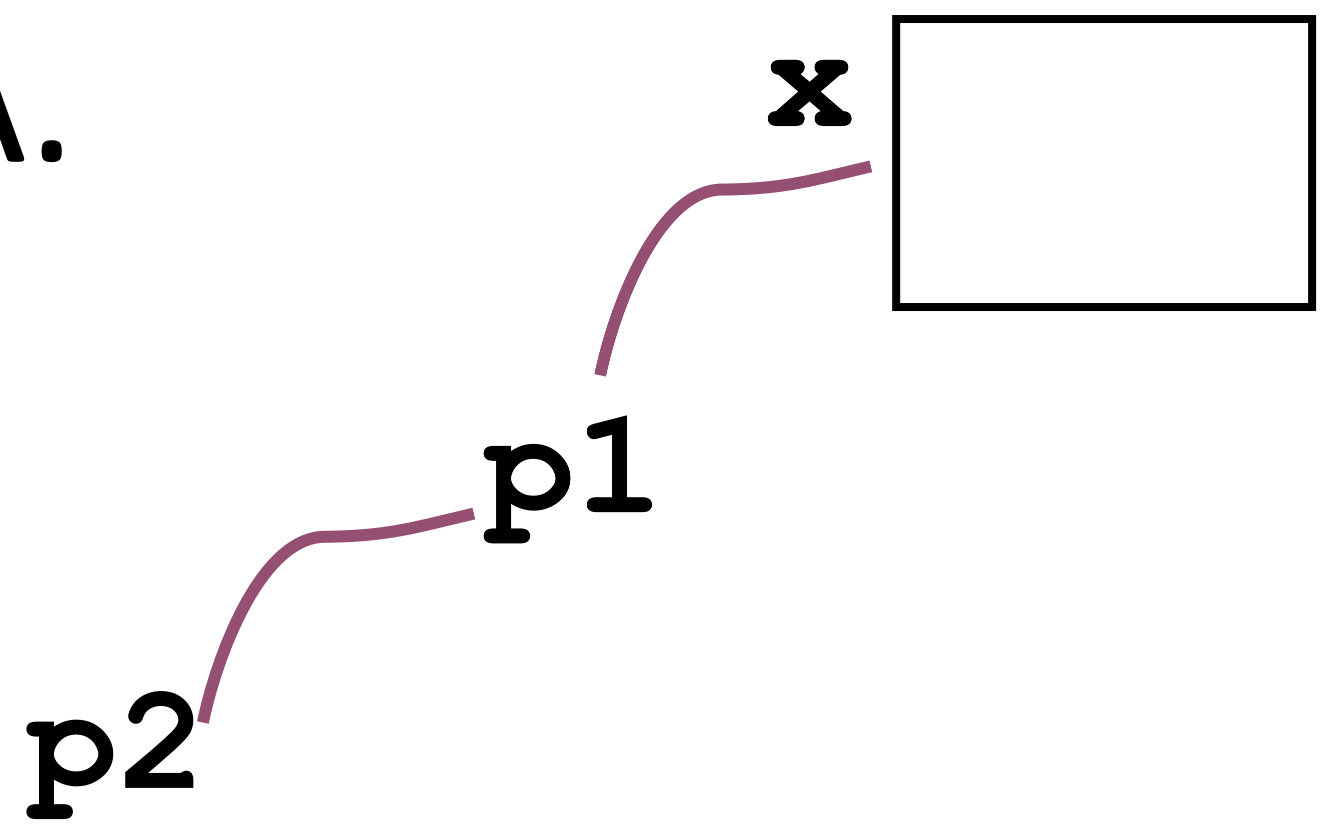
- E.g. **changes the value of i** to 6, it <u>does not change either pointer</u>
  - *i was not used in the statement, its contents were changed*

Pointers and Pointees

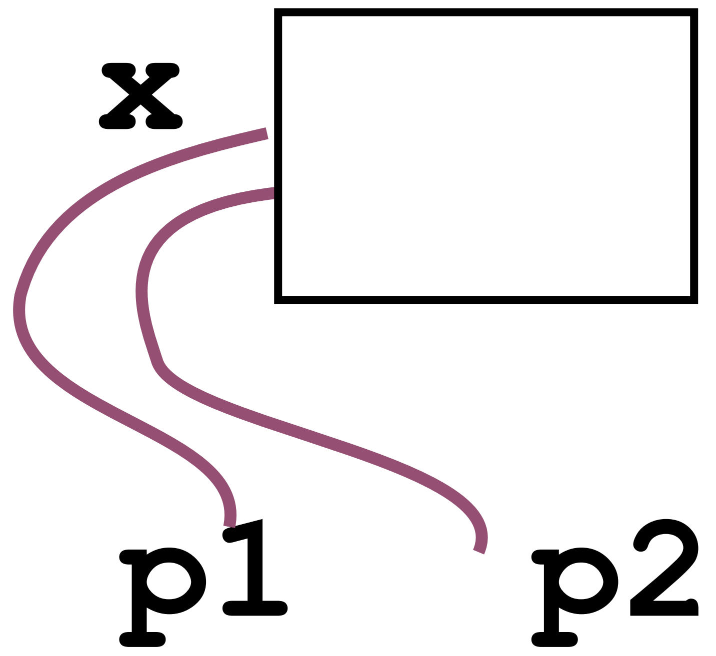
```
int *p1, *p2, x;
p1 = &x;
p2 = p1;
```

Q: Which of the following pointer diagrams best represents the outcome of this code?

A.

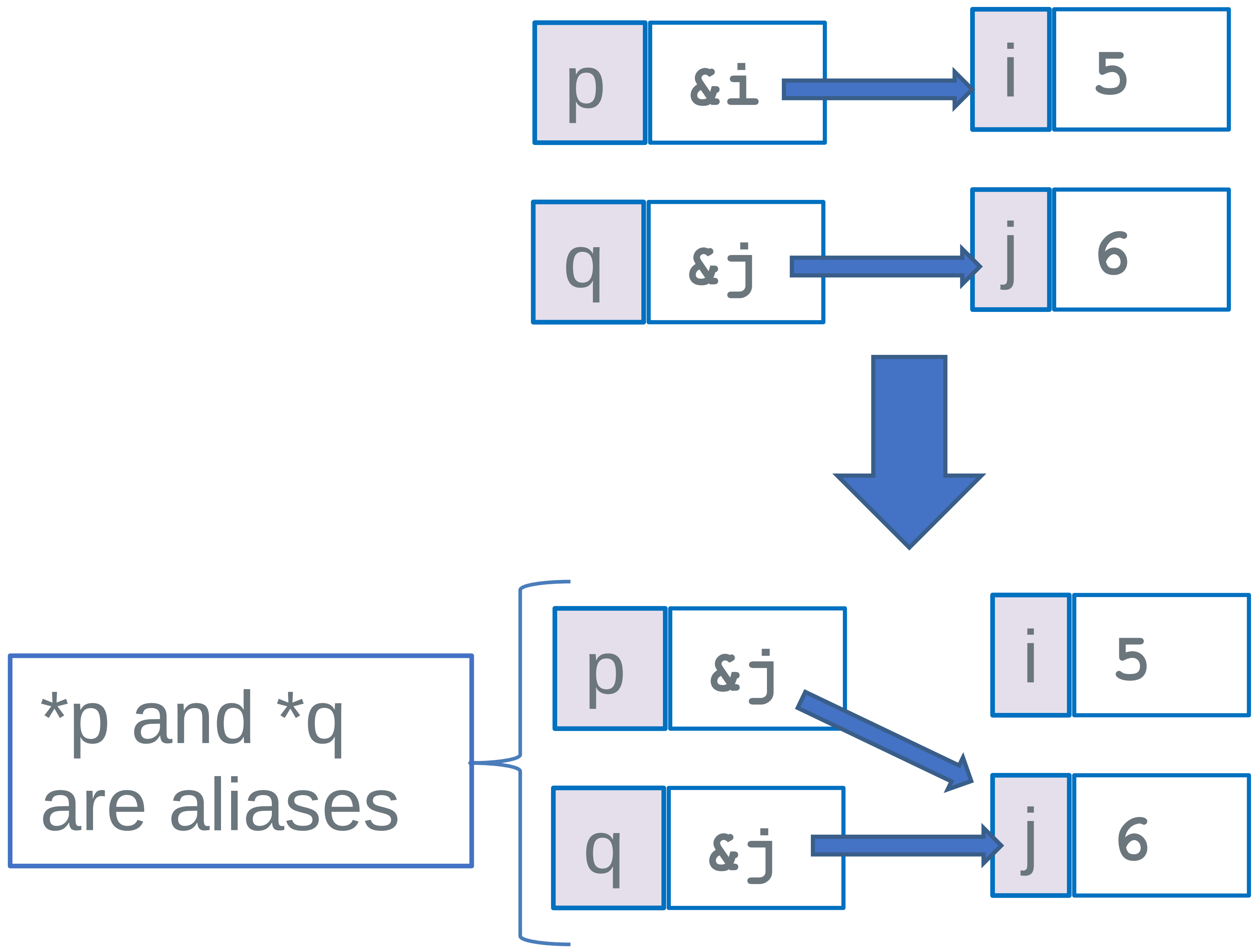


B.



C. Neither, the code is incorrect

What is Aliasing?




```
int i = 5;
int j = 6;
int *p = &i;
int *q;
q = &j;


p = q;
```

- q = &j;
- p = q;
- The operation is called ***aliasing*** *(creates an* ***alias****)*
  - **p and q** are now **aliases of each other**
- **Aliasing** occurs when the **same memory contents can be accessed from more than one variable**
  - **Variable identifiers are aliases** when they are **allocated or point at the same memory location**

The NULL Pointer

- **NULL** is a special pointer **value** to represent that the pointer points to "nothing"
  - If pointer is unknown or no longer points to a valid location THEN assign it to NULL

- A pointer with a value of NULL is often known as a "NULL pointer" (not a valid address!)

```
int *p = NULL;
int *p = (int *)0;     // cast 0 to a pointer type
int *p = (void *)0 ;   // automatically gets converted to the correct type
```

- Some functions return NULL to indicate an error

```
int *func(int p1) {
   int *somePtr;

   // some code . . .
   if (errorCondition){
       somePtr = NULL;
       goto cleanup;
   }
   // some code . . .
cleanup:
   return(somePtr);
}
```

What will this code do?

A. prints 12
B. prints 13
C. may get a SEGMENTATION FAULT
D. print the address of p

```
#include <stdio.h>
int main() {
  int *p;
  *p = 12;
  *p = (*p)++;
  printf("%d\n", *p);
}
```

Q: Which of the following is true when code 1 and 2 are compiled and executed?

1.

```
char *p;
int y;
p = &y;
```

2.

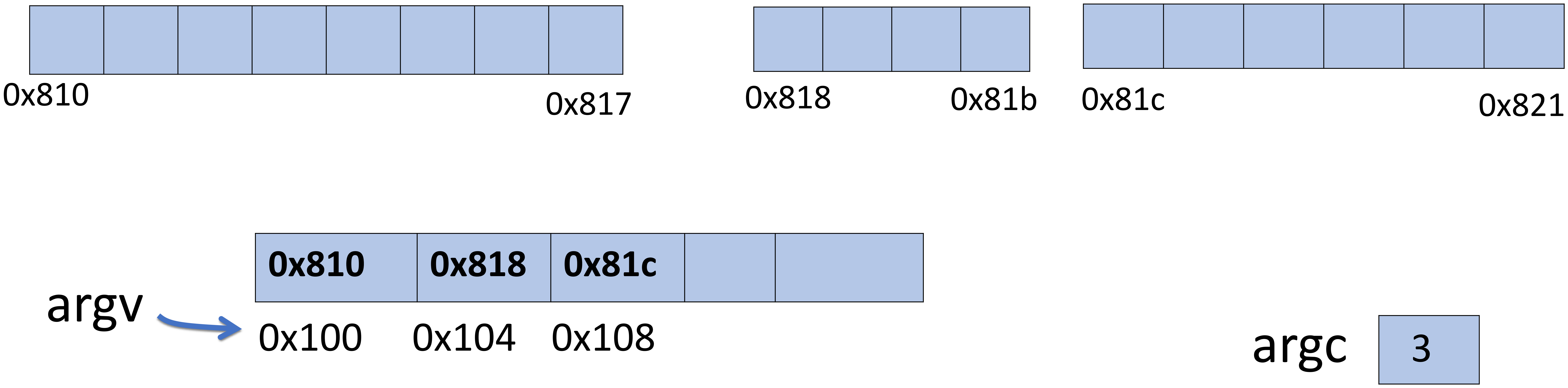
```
int *p;
*p = 5;
```

	Code 1	Code 2
A	Compile time warning	Compile time error
B	Compile time error	Compiler error
C	Compile time warning	Runtime error
D	Compile time error	Runtime error
E	None of the above	

Argv is a Pointer to Pointers

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

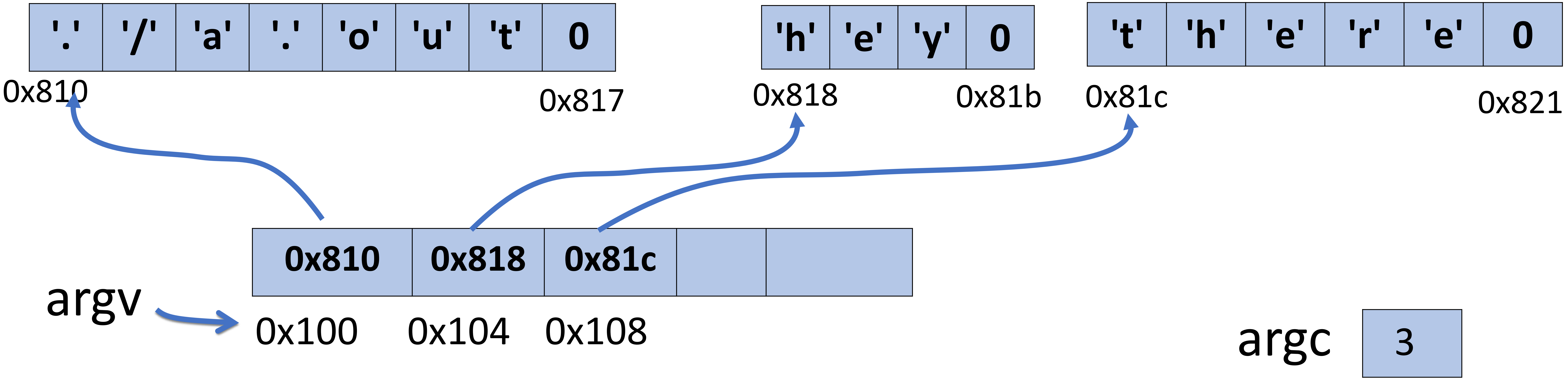
```
% ./a.out hey there
```



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("%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?
```

https://edstem.org/us/courses/37726/workspaces/ argv moredetail.c

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