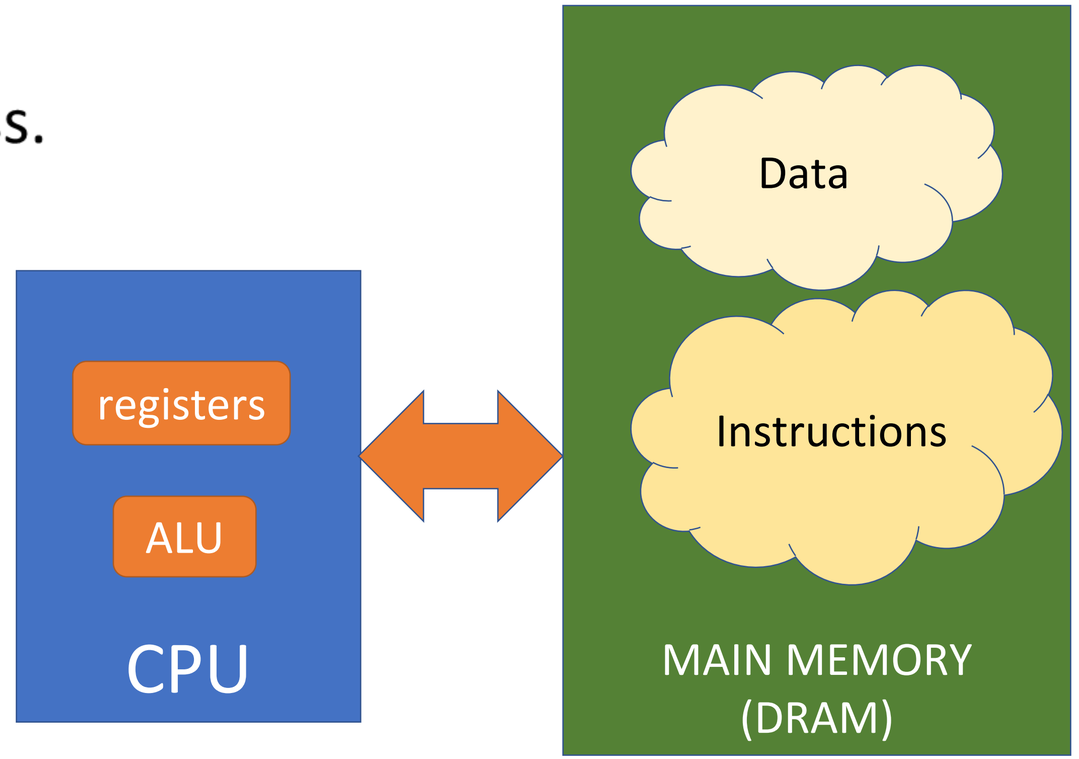
Programmer's Model of the Computer

- 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.

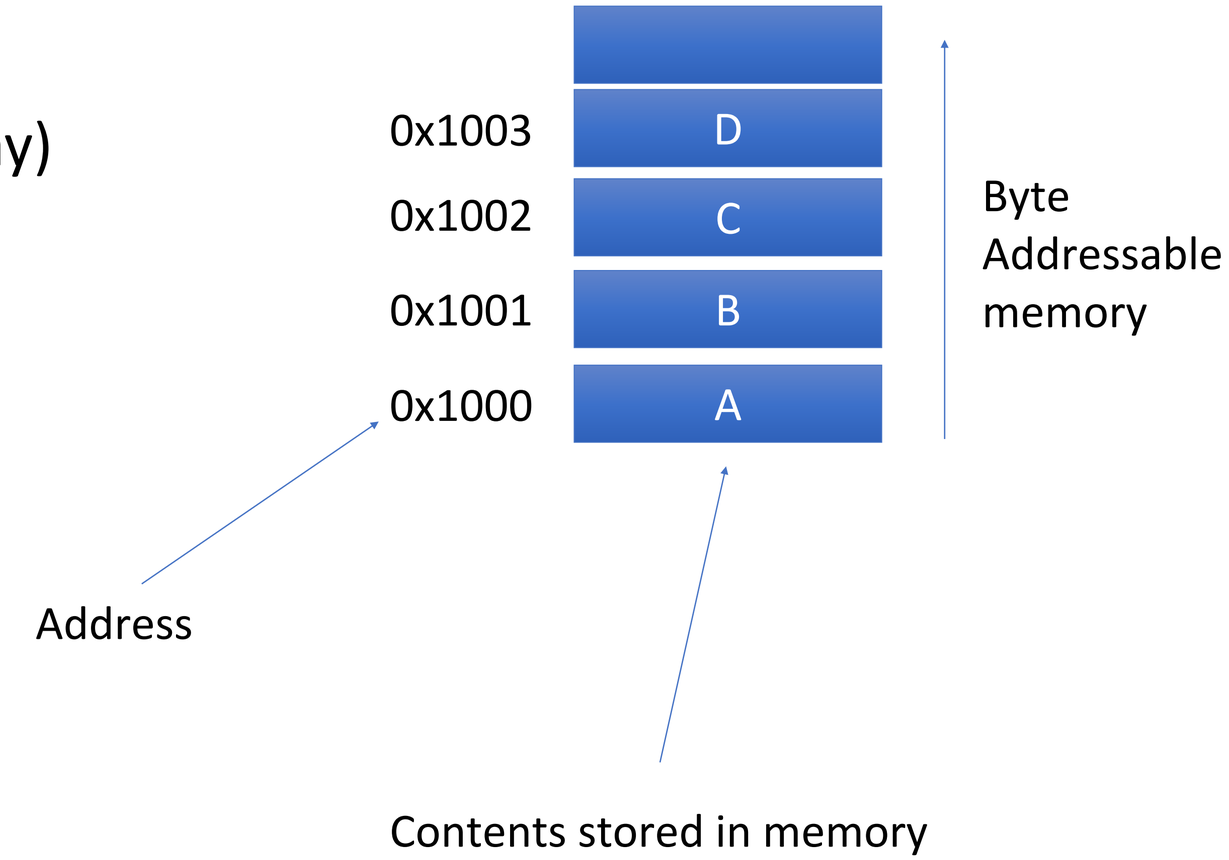
- ARMv6 - 32-bit architecture
  - Addresses are 32 bits wide
  - 32 bit registers
  - how much memory could one reach with a 32-bit address?

  - 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**?

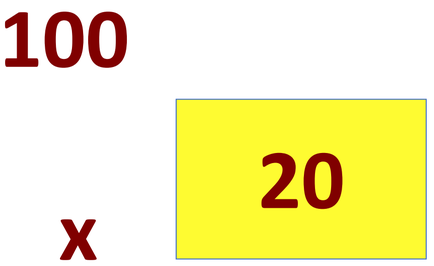
A. Yes

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

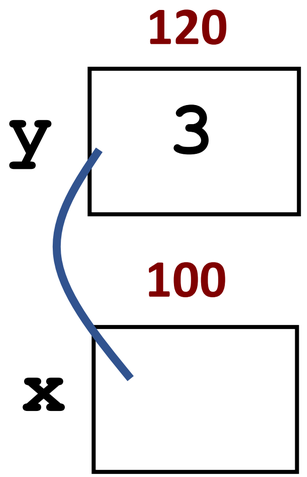
To access the location/address, use the address operator ‘&’

&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**

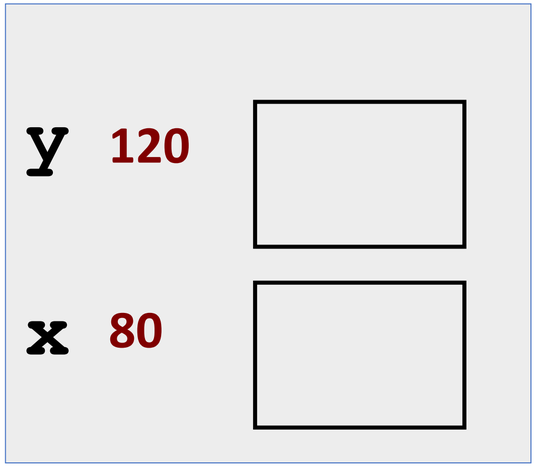
Pointers (1)

- Pointer: A variable that contains the address of a variable

```
int *x, y;

y = 3;

x = &y;
```



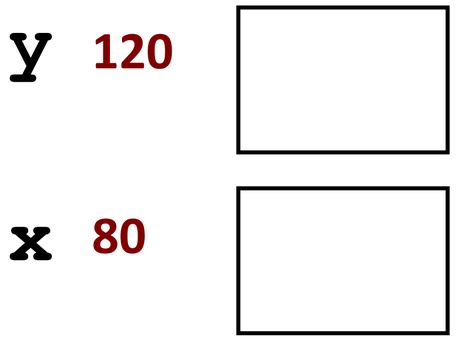
Pointers (2)

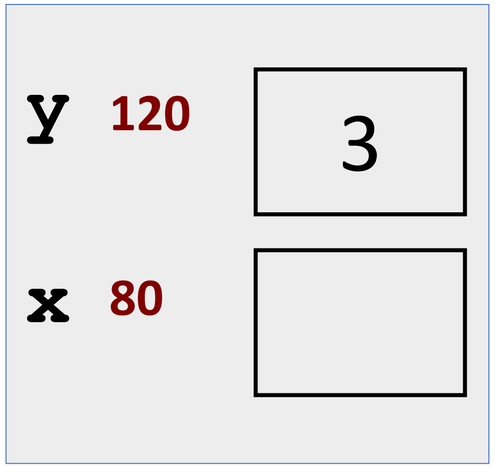
- Pointer: A variable that contains the address of a variable

```
int *x, y;

y = 3;

x = &y;
```





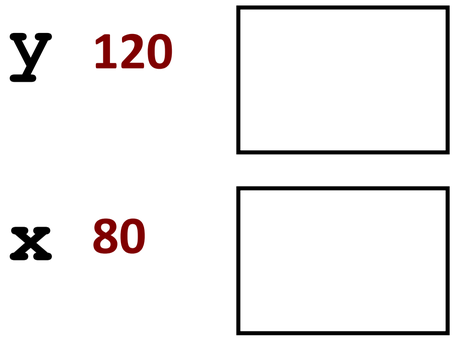
Pointers (3)

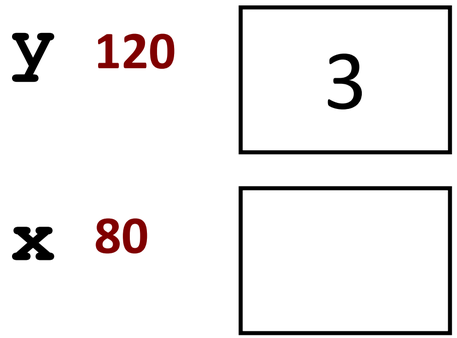
- Pointer: A variable that contains the address of a variable

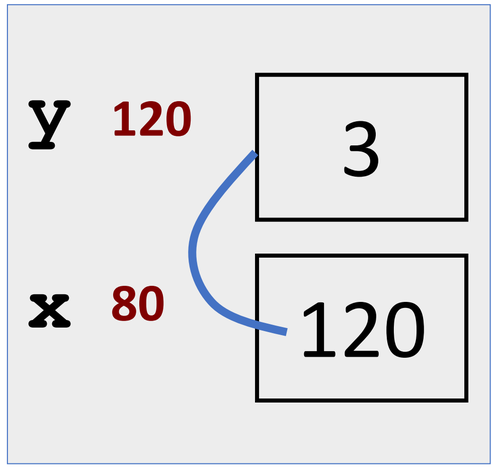
```
int *x, y;

y = 3;

x = &y;
```



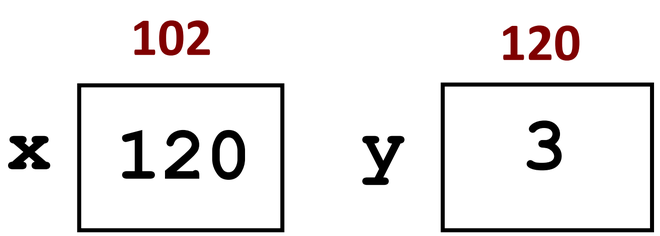


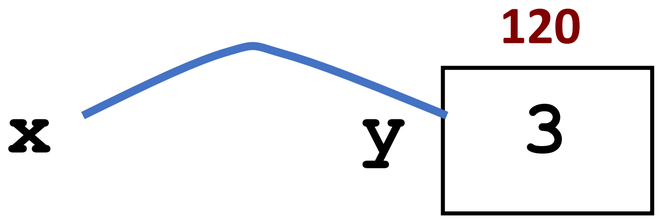


What is sizeof(x)?

Pointer Diagrams

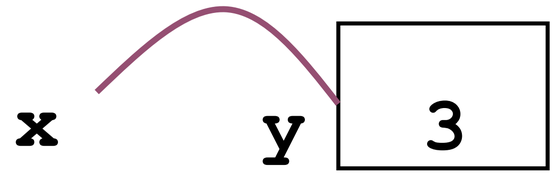
- Short hand diagram for the following scenario



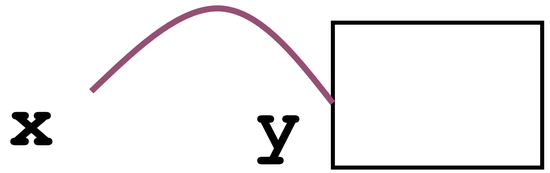


Reference through a pointer

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




```
*x = 5;
```

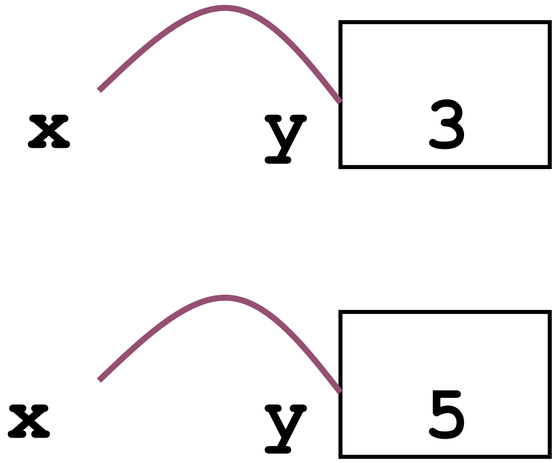


UC San Diego

Pointer Dereference(2)

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

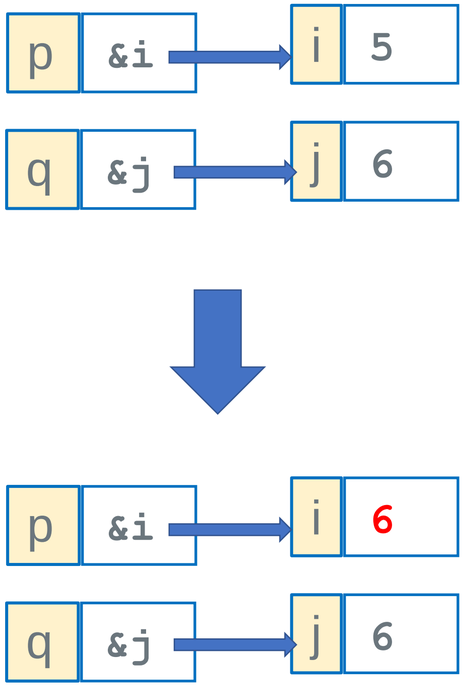
```
*x = 5;
```



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*

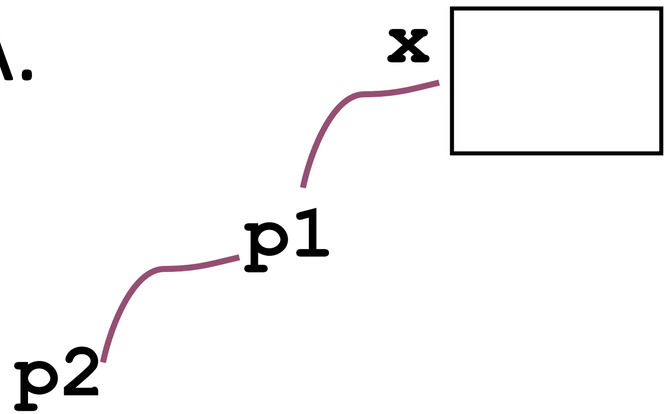
- E.g. **changes the value of i** to 6, it does not change either pointer
  - *i was not used in the statement, its contents were changed*

Pointers and Pointees

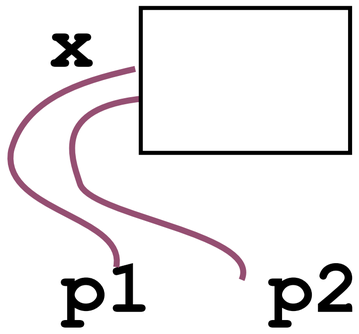
```
int *p1, *p2, x;
p1 = &x;
p2 = p1;
```

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

A.

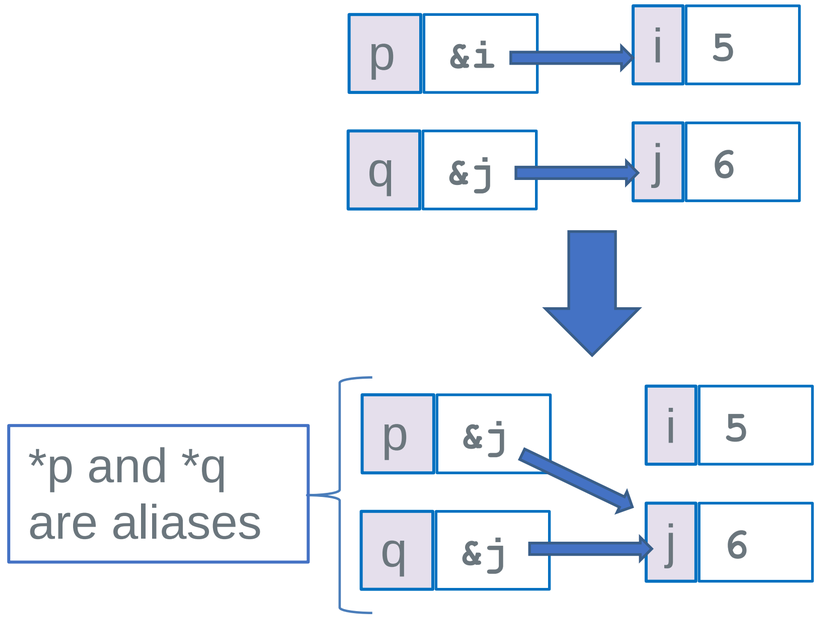


B.



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	