Xi'an Jiaotong-Liverpool University
西交利物浦大學

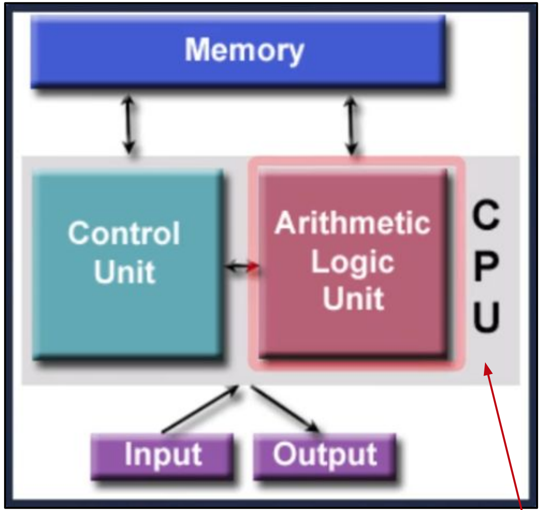
CPT104 - Operating Systems Concepts

Lab 4

Memory, Address, Pointer(1)

The Von Neumann Architecture

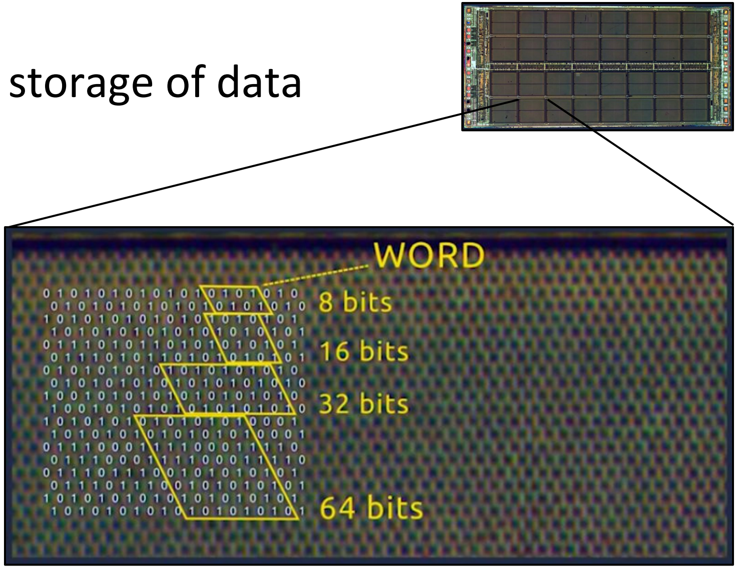
- The von Neumann architecture, used in modern computers, is an abstract model that splits a computer into 4 distinct parts:
  - ALU (Arithmetic Logic Unit)
    - perform arithmetic and logical operations
  - Control Unit
    - coordinates operations and data movement between other parts
  - Memory
    - stores program and data
  - Input/Output
    - communication with external world



Control Unit and ALU together form the CPU

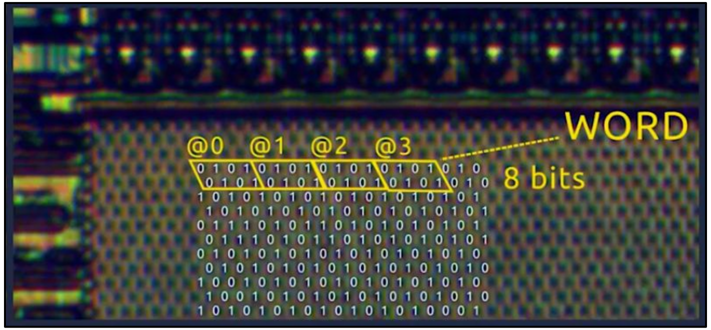
Computer Memory

- Two types of memory:
    - RAM (Random Access Memory) : temporary memory used to execute program and store values of variables
    - Storage (Non-volatile Memory) : permanent storage of data
- We represent RAM as a sequence of binary memory cells, each populated with a 0 or a 1
    - We call each such cell **1 bit**
    - We group several cells to create a word
    - The *size of a word* is in bits or in bytes
        - **1 byte** equals 8 bits
    - Modern computers and processors tend to use ***8, 16, 32,*** or even ***64 bits*** to form one word



Memory Address

- We group memory cells into words to allow for the addressing of memory
  - An address is assigned to each word
- **A memory address** is a whole number that describes the location of the word in the memory
  - For example, imagine the computer whose word length is 8 bits
  - Suppose that this computer can store a total of four words
  - The address 0 would be used for the first word: the first 8 memory cells
  - The address 1 would be used for the second word: the next 8 memory cells and so on



- In C programming language, it is possible *to get these memory addresses* during the execution of a program: if we use a variable to store a value, we could ***obtain*** the address where the value is stored

Space used in memory (1)

- How many space in memory is used to store a char? an integer? a double?
  - We use the function `sizeof`

lonog

```
#include <stdio.h>
int main() {
    char c;
    int i;
    double d;
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(double));
    return 0;
}
```

format specifier used to printf sizeof output

you can also input these with the variables: c, i, d

the output will depend on your system, but most likely (in **bytes**) :
1
4
8

Space used in memory (2)

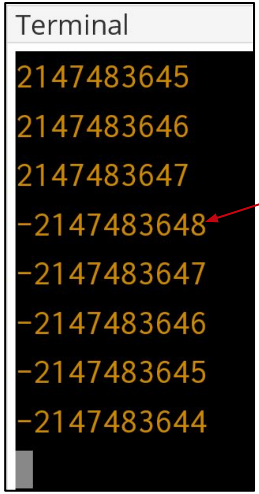
- How many space in memory is used to store an array of chars? of ints? of doubles?
  - ○ Try yourself in Codecast!

```
#include <stdio.h>
int main() {
    char arrChar[5];
    int arrInt[5];
    double arrDouble[5];
    printf("%zu\n", sizeof(arrChar));
    printf("%zu\n", sizeof(arrInt));
    printf("%zu\n", sizeof(arrDouble));
    return 0;
}
```

Wrapped Around

- Now we know we can only store limited number of integers, because space is limited
  - What happen if we keep adding to the largest possible number?

```c
#include <stdio.h>
int main() {
    int num = 2147483645;
    int i;
    for (i=0; i<8; i++) {
        printf("%d\n", num);
        num++;
    }
    return 0;
}
```



```
2147483645
2147483646
2147483647
-2147483648
-2147483647
-2147483646
-2147483645
-2147483644
```

it wraps around and becomes a negative large number

Large Integer in Memory (1)

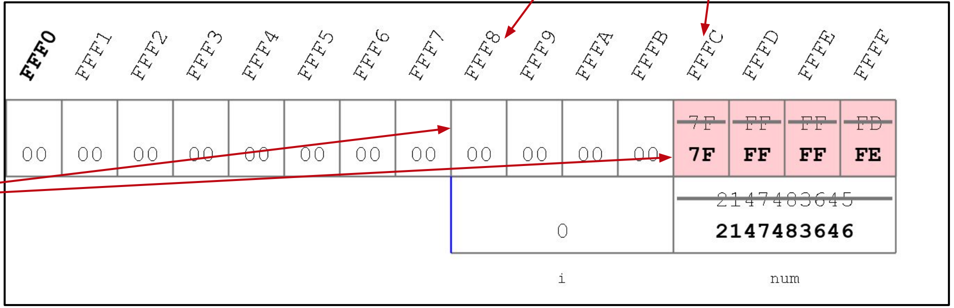
- Let us now see what happen in memory

```c
#include <stdio.h>
int main() {
    //! showMemory(start=65520)
    int num = 2147483645;
    int i;
    for (i=0; i<8; i++) {
        printf("%d\n", num);
        num++;
    }
    return 0;
}
```

add this line, then compile in Codecast, and then run step-by-step using Step Into

this part of memory is called **the stack**

the addresses of i and num in hex

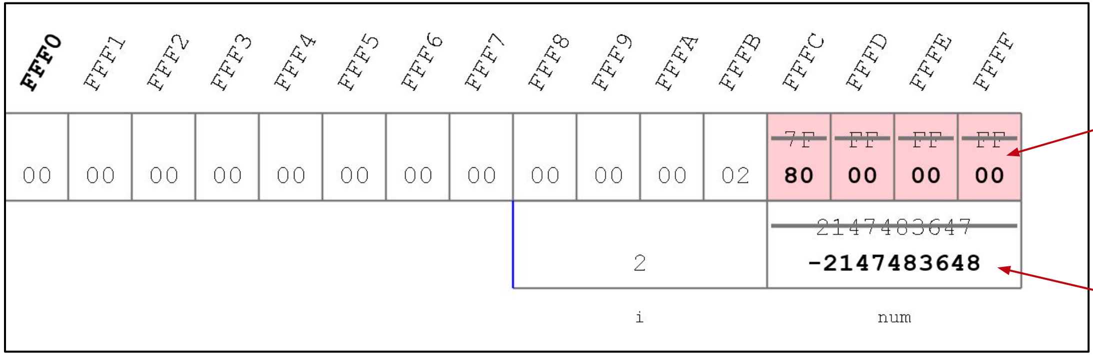


FFF0	FFF1	FFF2	FFF3	FFF4	FFF5	FFF6	FFF7	FFF8	FFF9	FFFA	FFFB	FFFC	FFFD	FFFE	FFFF
00	00	00	00	00	00	00	00	00	00	00	00	~~7F~~ **7F**	~~FF~~ **FF**	~~FF~~ **FF**	~~FD~~ **FE**
								0				~~2147483645~~ **2147483646**			
								i				num			

1 box = 1 bytes
an integer 4 boxes = 4 bytes

Large Integer in Memory (2)

- Wrapped around in memory



FFF0	FFF1	FFF2	FFF3	FFF4	FFF5	FFF6	FFF7	FFF8	FFF9	FFFA	FFFB	FFFC	FFFD	FFFE	FFFF
00	00	00	00	00	00	00	00	00	00	00	02	~~7F~~ **80**	~~FF~~ **00**	~~FF~~ **00**	~~FF~~ **00**
								2				~~2147483647~~ **-2147483648**			
								i				num			

when i = 2,
hexadecimal 7F FF FF FF
is incremented to become
hexadecimal 80 00 00 00

in C, that is interpreted as
this negative number

- Integers (whole number) can be positive or negative
  - when we store them in a limited memory, some combinations of 0s and 1s must also represent the negatives

Variables and Arrays in The Stack

- Run step-by-step in Codecast to see how various variables and arrays are stored in the stack

```c
#include <stdio.h>
int main() {
    //! showMemory(start=65520)
    char c = '!';
    short s = 1024;
    int i = 987654;
    double d = 25.52;
    char lChar[5] = {'a','b','c', 'd', 'e'};
    short lShort[4] = {1, 2, 3, 4};
    int lInt[3] = {10, 20, 30};
    double lDouble[2] = {76.543, 234.5678};
    return 0;
}
```

Pointers (1)

- Write a program to get addresses of variables, and print it, using pointers

```c
#include <stdio.h>
int main() {
    //! showMemory(start=65520)
    int i = 5;
    double d = 12.34;
    char c = 'a';
    int * address_i = &i;
    printf("address of i: %p\n", address_i);
    double * address_d = &d;
    printf("address of d: %p\n", address_d);
    char * address_c = &c;
    printf("address of c: %p\n", address_c);
    return 0;
}
```

variable address_i has type int *

type int * is a pointer to an address
where in that address
an int is stored

how do we fill in that variable?
using operator ampersand &
we get the address of integer
variable i, and put it inside variable
address_i

similarly for pointer to a double
and pointer to a char

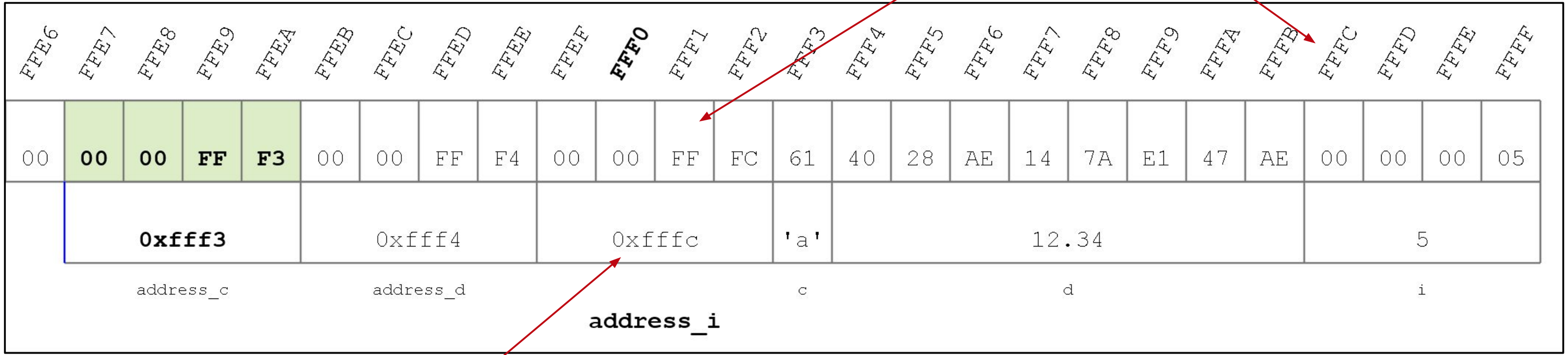
use %p to printf an address

Pointers (2)

- Write a program to get addresses of variables, and print it, using pointers

run step-by-step in Codecast

the address of i



FFE6	FFE7	FFE8	FFE9	FFEA	FFEB	FFEC	FFED	FFEE	FFEF	**FFF0**	FFF1	FFF2	FFF3	FFF4	FFF5	FFF6	FFF7	FFF8	FFF9	FFFA	FFFB	FFFC	FFFD	FFFE	FFFF
00	**00**	**00**	**FF**	**F3**	00	00	FF	F4	00	00	FF	FC	61	40	28	AE	14	7A	E1	47	AE	00	00	00	05
	0xfff3				0xfff4				0xfffc				'a'	12.34								5			
	address_c				address_d				**address_i**				c	d								i			

0x means it's followed by a hexadecimal

addresses are 4 bytes long here, since Codecast is 32 bit system for 64 bit system, we would have 8 bytes for the addresses

Dereference a Pointer

- Dereference: you have an address and want to access **the value** inside that address

```c
#include <stdio.h>
int main() {
    //! showMemory(start=65520)
    double d = 12.34;
    double * addr_d = &d;
    printf("address %p value %.2lf\n", addr_d, * addr_d);
    char c = 'a';
    char * addr_c = &c;
    char b = * addr_c;
    * addr_d = 5.0;
    * addr_d = * addr_d + 1.0;
    printf("address %p value %.2lf\n", addr_d, * addr_d);
    return 0;
}
```

also with * , but this is **not** declaring a pointer

dereference the pointer addr_d, a double

same as using d

read value in address addr_c, put in b

replace the value in address addr_d with 5.0

then add to 1.0 to it

(Not) Swap with Function (1)

- Can the function swap below really swap its two inputs?

```c
#include <stdio.h>
void swap(int, int);
int main() {
    int a = 2;
    int b = 5;
    swap(a, b);
    printf("%d %d\n", a, b);
    return 0;
}
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

want to swap a and b,
so that a = 5 and b = 2
what gets printed in Codecast?

a void function has no output

(Not) Swap with Function (2)

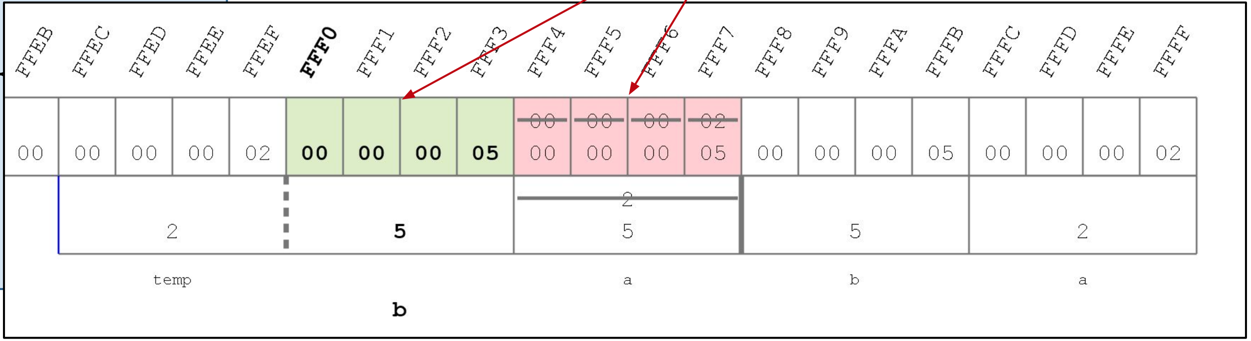
- Run step-by-step in Codecast using comment to see what happen in the memory

```c
#include <stdio.h>
void swap(int, int);
int main() {
    //! showMemory(start=65520)
    int a = 2;
    int b = 5;
    swap(a, b);
    printf("%d %d\n", a, b);
    return 0;
}
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

there are two new variables a, b different from the a, b in main

the a, b in swap gets swapped, but ***not*** the a, b in main

these are two **local variables**, visible only within swap function



FFEB	FFEC	FFED	FFEE	FFEF	**FFF0**	FFF1	FFF2	FFF3	FFF4	FFF5	FFF6	FFF7	FFF8	FFF9	FFFA	FFFB	FFFC	FFFD	FFFE	FFFF
00	00	00	00	02	**00**	**00**	**00**	**05**	~~00~~ 00	~~00~~ 00	~~00~~ 00	~~02~~ 05	00	00	00	05	00	00	00	02
	2				**5**				~~2~~ 5				5				2			
	temp				**b**				a				b				a			

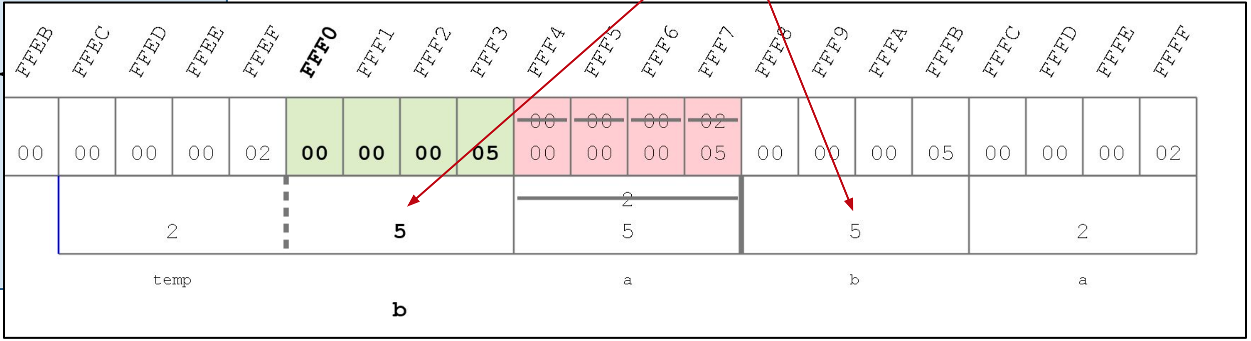
(Not) Swap with Function (3)

- Run step-by-step in Codecast using comment to see what happen in the memory

```c
#include <stdio.h>
void swap(int, int);
int main() {
    //! showMemory(start=65520)
    int a = 2;
    int b = 5;
    swap(a, b);
    printf("%d %d\n", a, b);
    return 0;
}
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

this is called **pass by value**

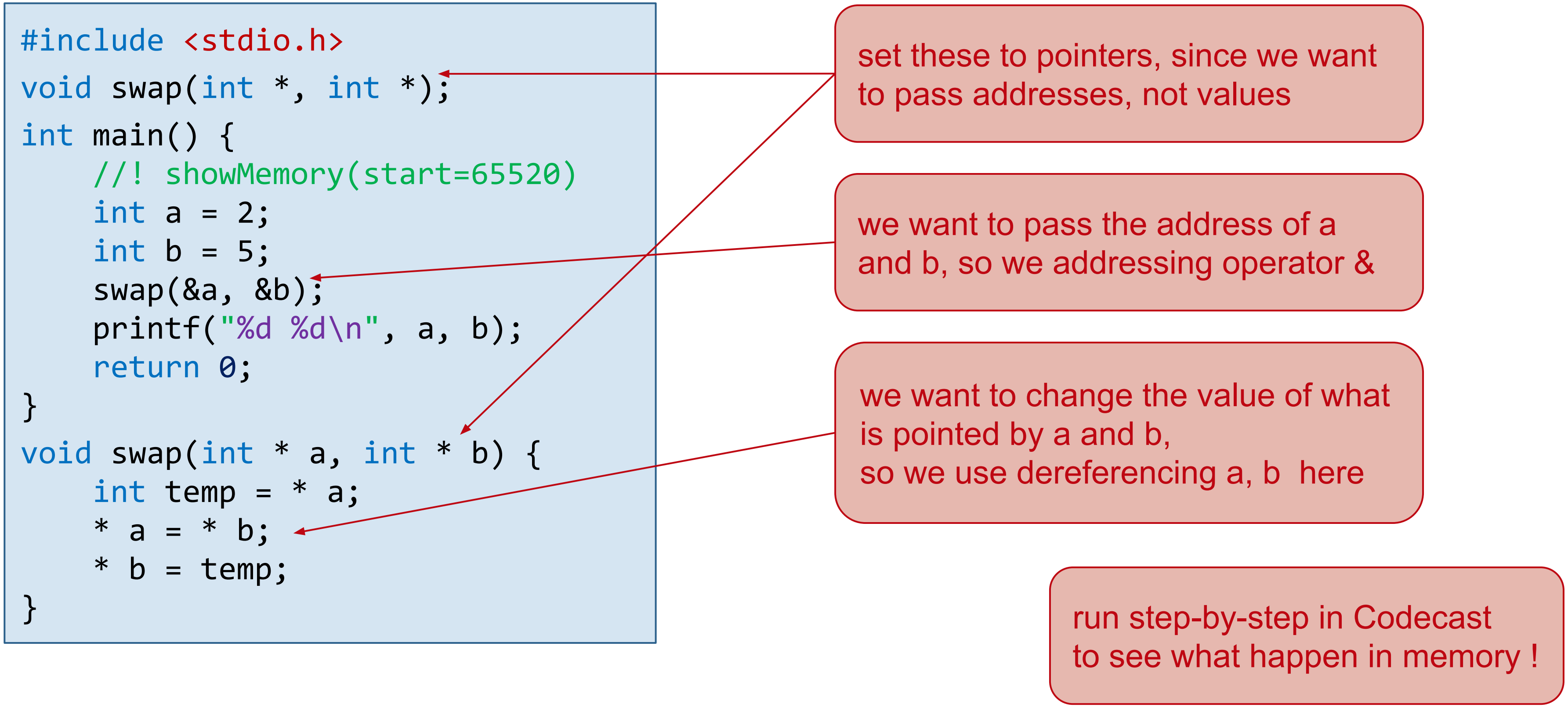
only the values of a, b in main gets passed to a, b in swap



FFEB	FFEC	FFED	FFEE	FFEF	**FFF0**	FFF1	FFF2	FFF3	FFF4	FFF5	FFF6	FFF7	FFF8	FFF9	FFFA	FFFB	FFFC	FFFD	FFFE	FFFF
00	00	00	00	02	**00**	**00**	**00**	**05**	~~00~~ 00	~~00~~ 00	~~00~~ 00	~~02~~ 05	00	00	00	05	00	00	00	02
	2				**5**				~~2~~ 5				5				2			
	temp				**b**				a				b				a			

Swap with Function and Pointer (1)

- How do we modify the code so that swap function works ?



```c
#include <stdio.h>
void swap(int *, int *);
int main() {
    //! showMemory(start=65520)
    int a = 2;
    int b = 5;
    swap(&a, &b);
    printf("%d %d\n", a, b);
    return 0;
}
void swap(int * a, int * b) {
    int temp = * a;
    * a = * b;
    * b = temp;
}
```

set these to pointers, since we want to pass addresses, not values

we want to pass the address of a and b, so we addressing operator &

we want to change the value of what is pointed by a and b,
so we use dereferencing a, b here

run step-by-step in Codecast
to see what happen in memory !

Swap with Function and Pointer (2)

- How do we modify the code so that swap function works ?

```c
#include <stdio.h>
void swap(int *, int *);
int main() {
    //! showMemory(start=65520)
    int a = 2;
    int b = 5;
    swap(&a, &b);
    printf("%d %d\n", a, b);
    return 0;
}
void swap(int * a, int * b) {
    int temp = * a;
    * a = * b;
    * b = temp;
}
```

this is called **pass by reference**

we pass not the value of the variables, but the address of (or reference to) the variables

a in swap contains address of a in main

note that temp here is just an integer variable
it's not storing an address, so we don't need a star

Add with Function and Pointer

- Another simple example to modify a variable from a function by pass by reference

```
#include <stdio.h>

void addTen(int *);

int main() {
    //! showMemory(start=65520)
    int a = 5;
    addTen(&a);
    printf("%d\n", a);
    return 0;
}

void addTen(int * aPtr) {
    *aPtr = *aPtr + 10;
    printf("%d\n", *aPtr);
}
```

run step-by-step in Codecast

Thank you for your attention !

- In this lab, you have learned:
  - Memory
    - Word Size
    - Memory Address
    - sizeof() Function
  - Pointer
    - Declare, assign, dereference a pointer
    - Address Operator
    - Pass by Value vs Pass by Reference
    - Void function with Pass by Reference

- **For more information:**

✓ refer to book chapter 5, 5.1-5.2