Topic 7. Pointers in C

COMP ENG 2SH4

Principles of Programming

McMaster University, 2015

Instructor: Sorina Dumitrescu

Required Textbook Reading

- Chapter 7
- Sections: 1-6
- The remaining sections will be covered by the following two topics

Memory Concepts

- Memory is organized as a sequence of bytes (1byte=8bits).
- Each byte has an **address** – a nonnegative integer value.

Memory Concepts

- When a variable is defined, a specific location in memory is reserved for it (a number of consecutive bytes, depending on the type of the variable).

- Its value will be stored at that location.

- **int** n; **/* 4 bytes of memory are allocated to var n*/**
- n = 10; **/* computer reads: go to the memory location reserved for n; replace whatever value was there by 10 */**

Address Operator: &

- C allows us to refer to the address of the location where some variable **var** is stored by using **&var**
- Address operator: **&**

```
#include <stdio.h>
int main(void){
     int n=6;
     printf("The memory location for n is at address %p.\n", &n);
     printf("The value of n is  %d.\n", n);
     return 0;
}
```

- Conversion specifier %p is used to output an address using hexadecimal notation.

```
#include <stdio.h>
int main(void){
     int n=6;
     printf("The memory location for n is at address %p.\n", &n);
     printf("The value of n is  %d.\n", n);
     return 0;}
```

```
The memory location for n is at address 0x22aadc.
The value of n is  6.
```

0x22aadc	0x22aadd	0x22aade	0x22aadf

Address Operator “&”

- “&” can be applied only to an **lvalue**.
- “&” cannot be applied to constants or expressions that are not lvalues.
- &(y+5)   // syntax error.
- &1934   // syntax error.

Arrays in memory

```
#include <stdio.h>
int main(void){
  int b[3]={-1,-2,-3};
  printf("The memory location for b[0] has address: %p\n",&b[0]);
  printf("The memory location for b[1] has address: %p\n",&b[1]);
  printf("The memory location for b[2] has address: %p\n",&b[2]);
  return 0;
}
```

```
The memory location for b[0] has address: 0x22aad0
The memory location for b[1] has address: 0x22aad4
The memory location for b[2] has address: 0x22aad8
```

What Does the Name of an Array Represent?

- In C the **name** of an array represents the **address** of the array , in other words, the address of its first element.

```
#include <stdio.h>
int main(void){
        int b[3]={-1,-2,-3};
        printf("The address of element b[0]: %p\n", &b[0]);
        printf("The address of the array is %p\n", b);
        return 0;
        }
```

```
The address of element b[0]: 0x22aad0
The address of the array: 0x22aad0
```

How Does C Access Elements of an Array?

- **int** b[4]={3,4}; **// b: 3,4,0,0**
- b[3] means: variable stored 3 locations (of the size of an **int**) to the right of address b.
- Unfortunately, C allows us to refer outside the array boundaries. The following are syntactically correct
  - b[5]
  - b[-2] **/* var stored 2 locations to the left of address b*/**

- **Pay attention not to refer outside the array boundaries!**
- If you do refer outside array boundaries
  - Program may crash or
  - Program may run to completion, but with incorrect results.
  - Only if you are extremely, extremely lucky, nothing bad happens – Don’t rely on that!!!

Pointers

- Pointer is a new kind of **data type**.
- What is a data type?
- A **data type specifies**:
  - the **set of values** its variables can take
  - the **operations** that can be performed on its variables.
  - the amount of **storage** for each variable
  - the way the values are **represented** in memory

Pointers

- The **value of a pointer variable** is the **address of another variable.**

- There are **different pointer types** depending on the type of the variable pointed to:
  - Pointer to **int** or (**int***) (can hold the address of a variable of type **int**)
  - Pointer to **double** (**double***) (its value can be the address of a variable of type **double**)
  - Pointer to **char** (**char***)
  - …

Declaration of a Pointer Variable

- The **value of a pointer variable** is the **address of another variable**.

Declaration of a pointer variable:

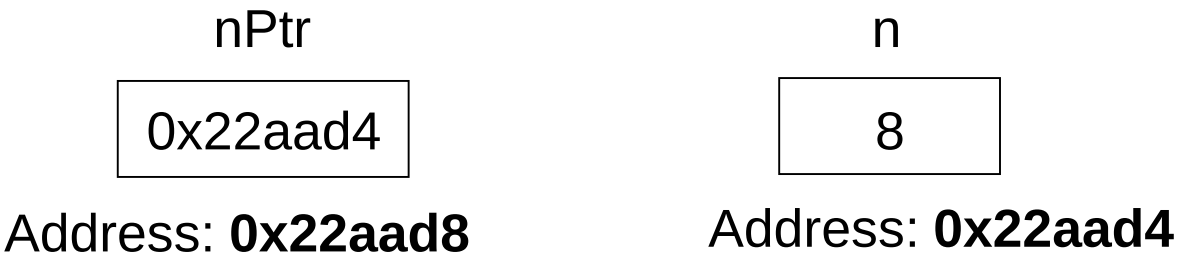
- **int ***nPtr;   /* declares variable **nPtr** of type **pointer to int**, thus it can hold the address of a variable of type **int** */
- **int*** nPtr;   /*   equivalent declaration */

- **char*** cPtr; /* declares variable **cPtr** of type **pointer to char**, thus it can hold the address of a variable of type **char** */

Pointers

- int *nPtr; /* declares variable **nPtr** of type **pointer to int,**
  thus it can hold the address of a variable of type int */
- int n=8; /* declares and initializes variable **n** of type int */
- nPtr = &n; /* assigns to **nPtr** the address of variable **n**;
  by this assignment pointer nPtr "**points to**" variable **n** */

- **In memory:**



nPtr

0x22aad4

Address: **0x22aad8**

n

8

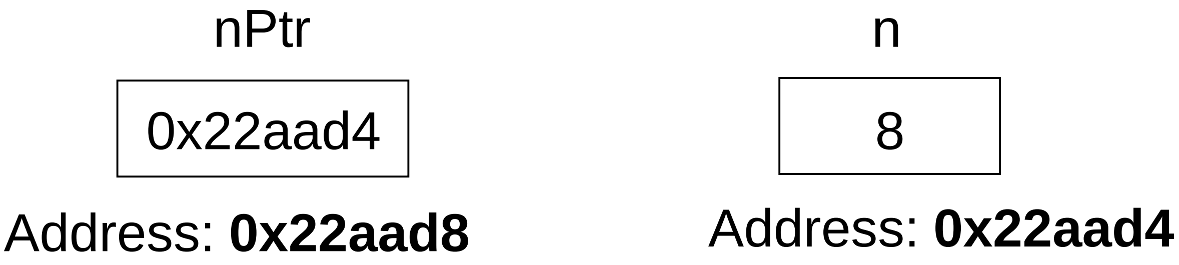
Address: **0x22aad4**

Pointers

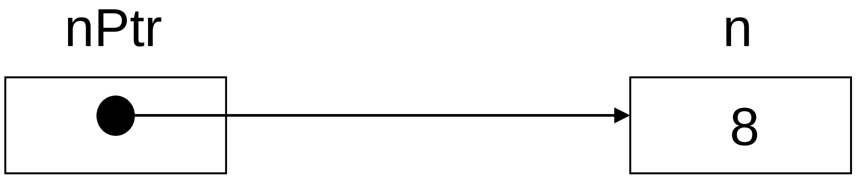
- **int** ***nPtr; /* declares variable **nPtr** of type **pointer to int,** thus it can hold the address of a variable of type **int** */
- **int** n=8; ***/*** declares variable **n** of type **int** */
- nPtr = &n; /* assigns to **nPtr** the address of variable **n**;

  by this assignment pointer nPtr "**points to**" variable **n** */

In memory:



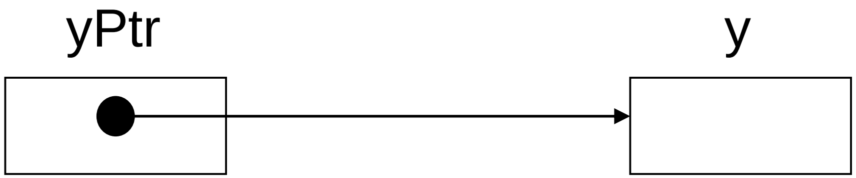
Higher level graphical representation:



Indirection (“*”) Operator

- The **indirection** (or **dereferencing**) operator can be applied to a pointer (or an expression of pointer type) : ***yPtr**
- It returns the **variable** the pointer points to.

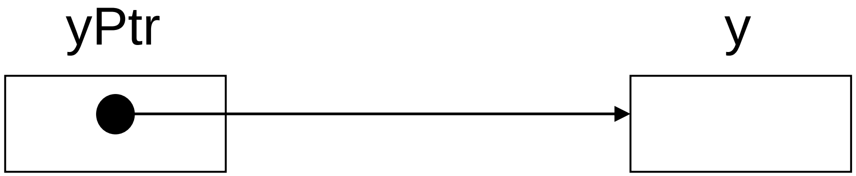
```
double *yPtr=NULL;   /* yPtr is declared as a pointer to a
                        double; it is initialized to point to
                        nothing  */
double y=4.0;
yPtr=&y;                /*   yPtr points to y  */
printf(“%f ”, *yPtr);   /* prints value of y */
```



Indirection (“*”) Operator

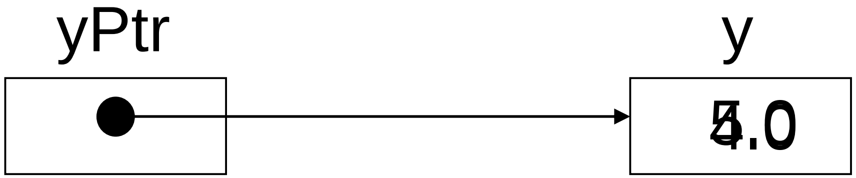
- The **indirection** (or **dereferencing**) operator applied to a pointer returns the **variable** the pointer points to.

- If **yPtr** points to variable **y**, then *yPtr is **another name** for y, thus
  - ***yPtr** can be used in any expression where **y** can be used.
  - ***yPtr** can be used on the left of an assignment.
  - By modifying ***yPtr**, variable **y** is actually modified.



Indirection (“*”) Operator

```
double *yPtr=NULL;   /* yPtr is declared as a pointer to a
                        double; it is initialized to point to nothing  */
double y=4.0;
yPtr=&y;                    /*   yPtr points to y  */
*yPtr = 5.0;                  /* the value of y is changed to 5.0 */
printf(“y=%.2f”, y);          /* value printed is 5.00 */
```



Using a Pointer

```
1.    int *bPtr = NULL;
2.    int a, b = 25;
3.    bPtr = &b; /* pointer bPtr points
                    to variable b */
4.    a = *bPtr;
5.    *bPtr = 0;
```

		initially	line 3	line 4	line 5
bPtr	0x22aad8	00000000	0x22aad0	0x22aad0	0x22aad0
a	0x22aad4	?	?	25	25
b	0x22aad0	25	25	25	0
	...	...	...	...	...

Assigning Values to Pointers

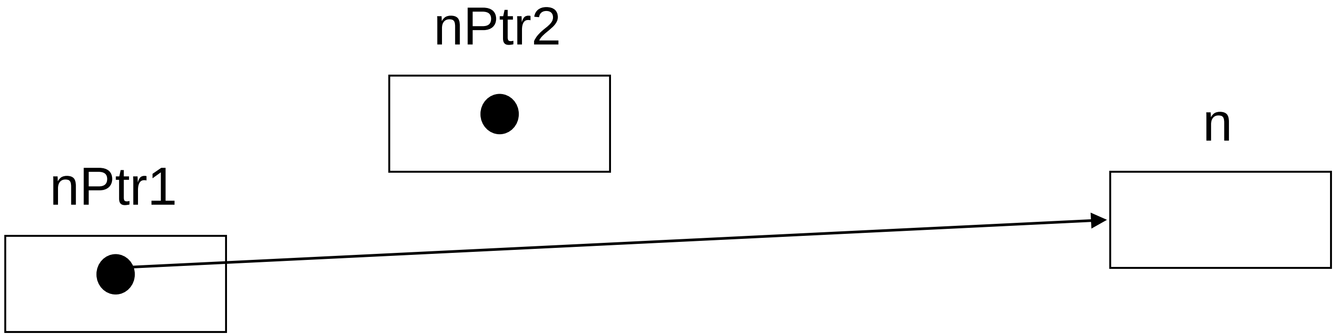
- A pointer variable can be assigned:
  - the address of a variable;
  - a NULL pointer (points to nothing);
  - **a pointer of the same type.**

int ***nPtr1=NULL, *****nPtr2=NULL, n; // declares **nPtr1** and

// **nPtr2** of type **int*** and **n** of type **int**

nPtr1=&n;   /* **nPtr1** points to variable **n**   */

nPtr2=nPtr1;



Assigning Values to Pointers

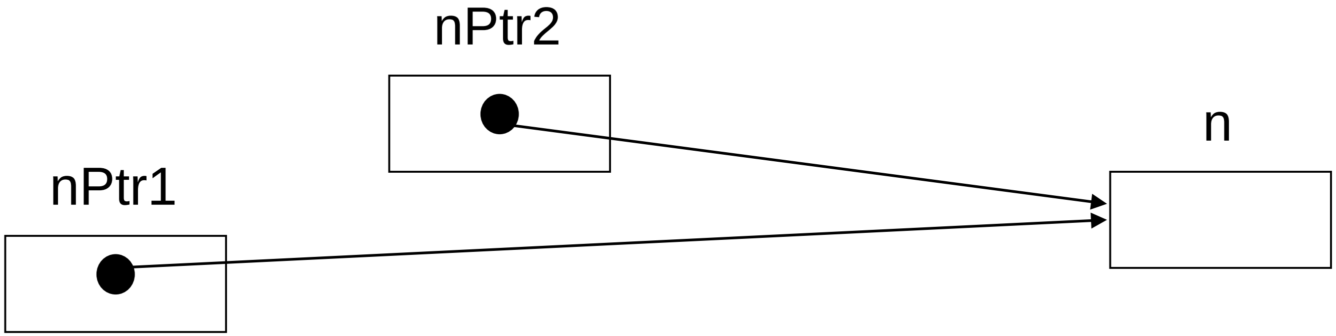
- A pointer variable can be assigned:
  - the address of a variable;
  - a NULL pointer (points to nothing);
  - **a pointer of the same type.**

int *nPtr1=NULL, *nPtr2=NULL, n; // declares **nPtr1** and

// **nPtr2** of type **int*** and **n** of type **int**

nPtr1=&n; /* **nPtr1** points to variable **n** */

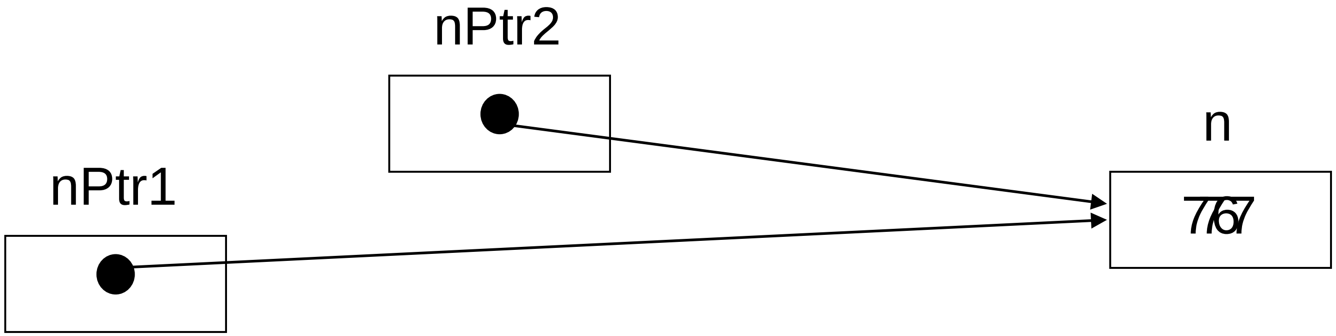
nPtr2=nPtr1; /* **nPtr2** will point to the same variable **nPtr1** points to */



Assigning Values to Pointers

```
int n;
int *nPtr1=NULL;
int *nPtr2=NULL;
nPtr1=&n;
nPtr2=nPtr1;
*nPtr2 =76;
++*nPtr1;
```

unary operators appearing in front of the operand have the same precedence level and associate from right to left



Operators Precedence

()	[]	++ (post)	-- (post)					
++ (pre)	-- (pre)	+	-	!	(type)	&	sizeof	*
*	/	%						
+	-							
<	<=	>	>=					
==	!=							
&&								
||								
? :								
=	+=	-=	*=	/=	%=			
,								

Assigning Values to Pointers

- A pointer variable can be assigned:
  - the address of a variable
  - a pointer of the same type
  - a NULL pointer (points to nothing)

- **double** *p2=NULL;
- **double** *p3=0;

  /* both p2 and p3 point to nothing */

- A NULL pointer **cannot be dereferenced**

Passing Pointers to a Function

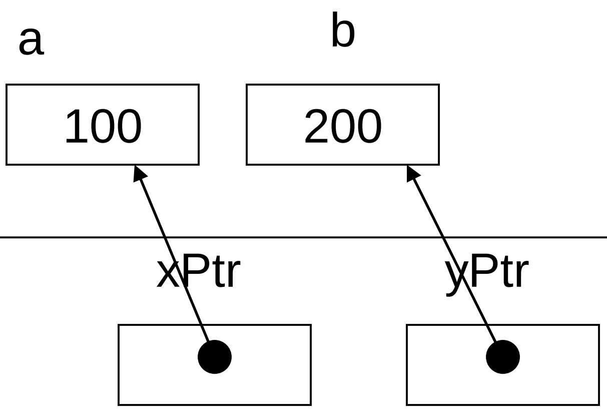
- Pointer **arguments** are passed to a function **by value.**
- However, the value of the pointer is the **address** of the variable pointed to.
- Since the function knows the address of the variable pointed to, it can **access and modify this variable**!!
- Thus, we can say that the **variable pointed to** is passed **by reference**.
- In other words, pointers can be used to **simulate** **passing by reference**.

Passing Pointers to a Function

- If we want to write a function to **change the value** of a variable in the caller, we need to pass to the function the **address** of that variable.
- Therefore, the corresponding parameter has to be of **pointer type**.

Swapping 2 Variables in the Caller

```
/* In main(): */
int a = 100, b = 200;
swap( &a, &b );
```



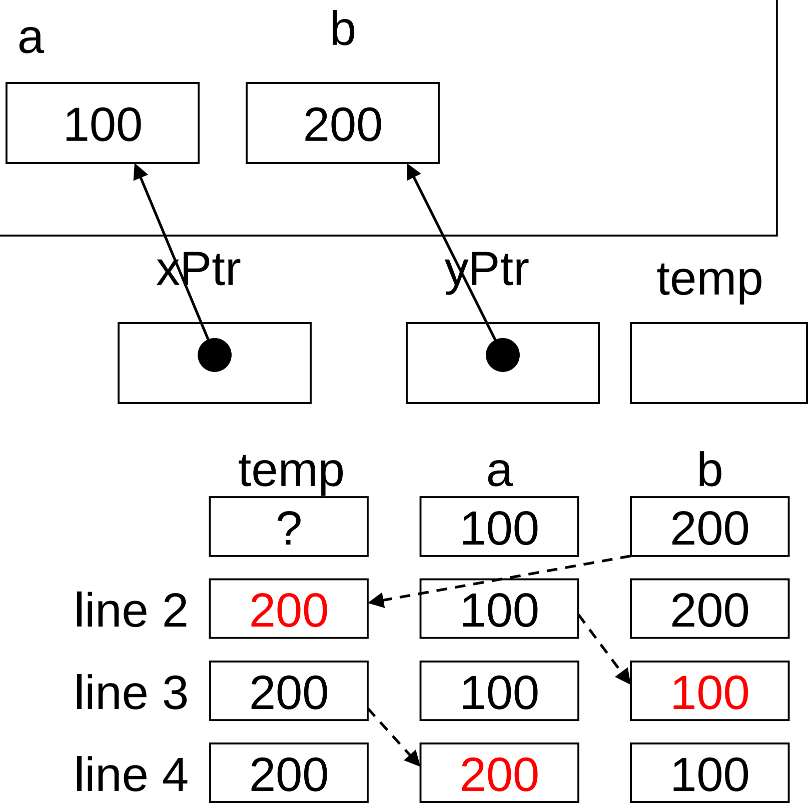
```
void swap(int *xPtr, int *yPtr )
{
}
```

- When **swap()** is called, local variables **xPtr** and **yPtr** (pointers to **int**) are created.
- **xPtr** is assigned the value **&a ,** thus **xPtr** will point to **a**. Likewise **yPtr** will point to **b**.
- ***xPtr** becomes **another name** for variable **a**. ***yPtr** becomes **another name** for variable **b**.
- The names **a** and b are not known inside the function **swap()**, but the names ***xPtr and *yPtr** are known;
- **swap()** can use these other names (***xPtr, *yPtr**) to access and modify the variables **a** and **b** in the caller.

Swapping 2 Variables in the Caller

```
/* In main(): */
int a = 100, b = 200;
swap( &a, &b );
```

```
void swap(int *xPtr, int *yPtr )
{
1.        int temp;
2.        temp = *yPtr;
3.        *yPtr = *xPtr;
4.        *xPtr = temp;
}
```



	temp	a	b
	?	100	200
line 2	200	100	200
line 3	200	100	100
line 4	200	200	100

After swap() ends execution, variables **xPtr**, **yPtr** and **temp** are destroyed (deallocated).

Simulating Returning More Values from A Function

- We need to write a single function to compute the **sum and difference** of two variables (x and y)

❑ Functions in C can return only a single value.

❑ What can we do?

❑ Define variables **sum** and **diff** in the caller.

❑ Write a function which is able to modify **sum** and **diff** in the caller.

```
int x = 100, y = 200;
sum_diff( ?? );
```

Simulating Returning More Values from A Function

- Define variables **sum** and **diff** in the caller.
- Write a function which is able to modify **sum** and **diff** in the caller.
- What values should we pass to the function?
  - Value of x
  - Value of y
  - Address of **sum**
  - Address of **diff**

```c
int x = 100, y = 200;
int sum, diff;
sum_diff( ?? );
```

Simulating Returning More Values from A Function

```
int x = 100, y = 200;
int sum, diff;
sum_diff(x, y, &sum, &diff);
```

- Since we decided what values to pass to the function, we know what parameters our function should have:
  - □ int a
  - □ int b
  - □ int *sumPtr
  - □ int *diffPtr

```
void sum_diff(int a, int b, int* sumPtr, int* diffPtr);
```

Simulating Returning More Values from A Function

- How do we write our function?
  - **Tip:** Do not think of `sumPtr` and `diffPtr` as pointers!
  - Instead
    - think of `*sumPtr` as a variable to store the sum of **a** and **b**;
    - think of `*diffPtr` as a variable to store the difference of **a** and **b**.

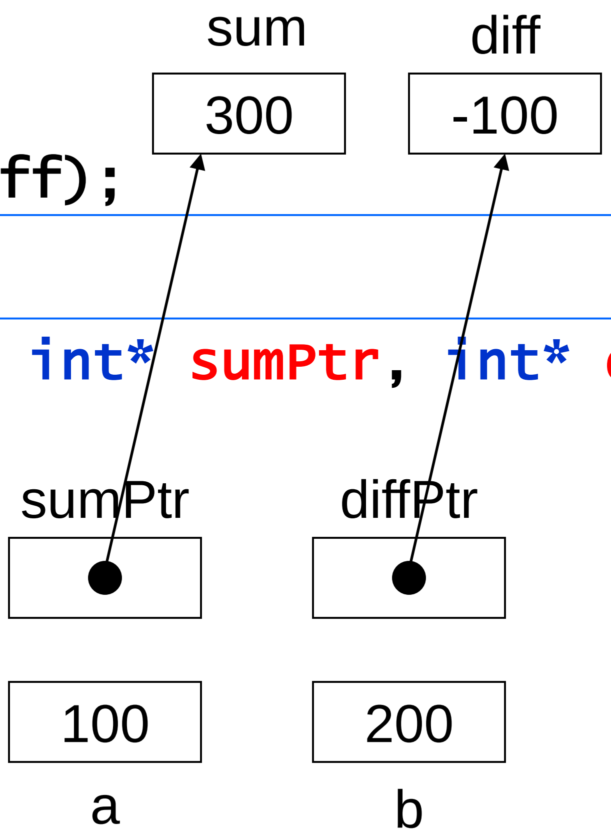
```
int x = 100, y = 200;
int sum, diff;
sum_diff(x, y, &sum, &diff);
```

```
void sum_diff(int a, int b, int* sumPtr, int* diffPtr){
     ???
}
```

Simulating Returning More Values from A Function

```
int x = 100, y = 200;
int sum, diff;
sum_diff(x, y, &sum, &diff);
```

```
void sum_diff(int a, int b, int* sumPtr, int* diffPtr)
{
      *sumPtr= a + b;
      *diffPtr = a - b;
}
```


sum
diff
300
-100
sumPtr
diffPtr
100
200
a
b