COMP10002 Workshop Week 5

Note: Download tis slide: `week5.pdf` from `github.com/anhvir/c102`

Focuses	Pointers, pointers as function arguments Arrays, arrays as pointer constants Arrays: working with arrays, searching, sorting
2	Do Together: 7.7
4	Lab: Implement 7.6, 7.7, 7.8, 7.9, 7.10, 7.11
Mandatory Works as in Canvas	Discuss Exercises 7.6 and 7.7; and through them Confirm that you understand arrays and the operations that manipulate them; then Design and implement solutions to as many as you can get through of Exercises 7.3, 7.4, 7.6, 7.7, 7.8, 7.9, 7.10, and 7.11; The more, the better! (You can do more of them in the Week 6 Workshops).
NOTE	**Quiz 2** (another 10% of final grade) will take place between 4:15pm and 5:00pm Melbourne time on Friday next week (11/SEP). Covering up to lecture recordings lec05-j

Function: can it have more than one output?

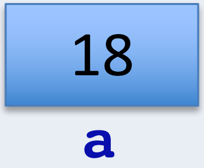
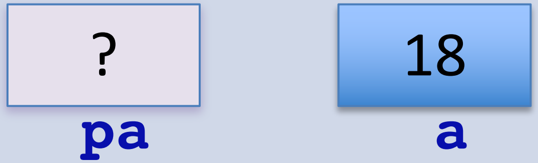
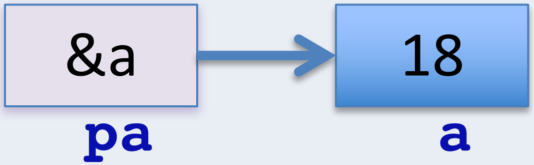
How many outputs we get with the call:

```
    scanf(“%d”, &n)
```

Data type: “pointer to int”, “address of int”

`int a= 18;`	18 **a**	**a** is a cell in the memory, interpreted as an **int**, with value of 18
`int *pa;`	? **pa** 18 **a**	**pa** is an **int pointer**, it can hold the address of an **int**

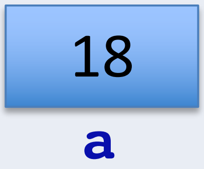
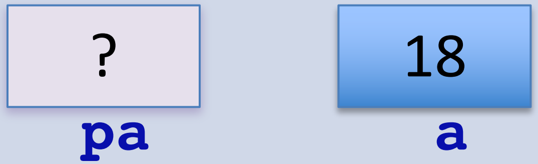
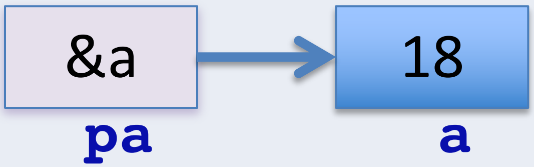
Data type: “pointer to int”, “address of int”

`int a= 18;`		**a** is a cell in the memory, interpreted as an **int**, with value of 18
`int* pa;`		**pa** is an **int pointer**, it can hold the address of an **int** **pa** is of data type **int***
`pa= &a;`		**pa** now holds the address of **a**, or, it "points" to **a**. using the value of **pa** we can have *indirect access* to **a**

Should we write, say, `pa = 5;` or `pa = 2 * pa;` ?

Without using the name `a`, how can we change a to `19`?

Data type: “pointer to int”, “address of int”

`int a= 18;`		**a** is a cell in the memory, interpreted as an **int**, with value of 18
`int *pa;`		**pa** is an **int pointer**, it can hold the address of an **int**
`pa= &a;`		**pa** now holds the address of **a**, or, it "points" to **a**. using the value of **pa** we can have *indirect access* to **a**

Data type: "pointer to int", "address of int"

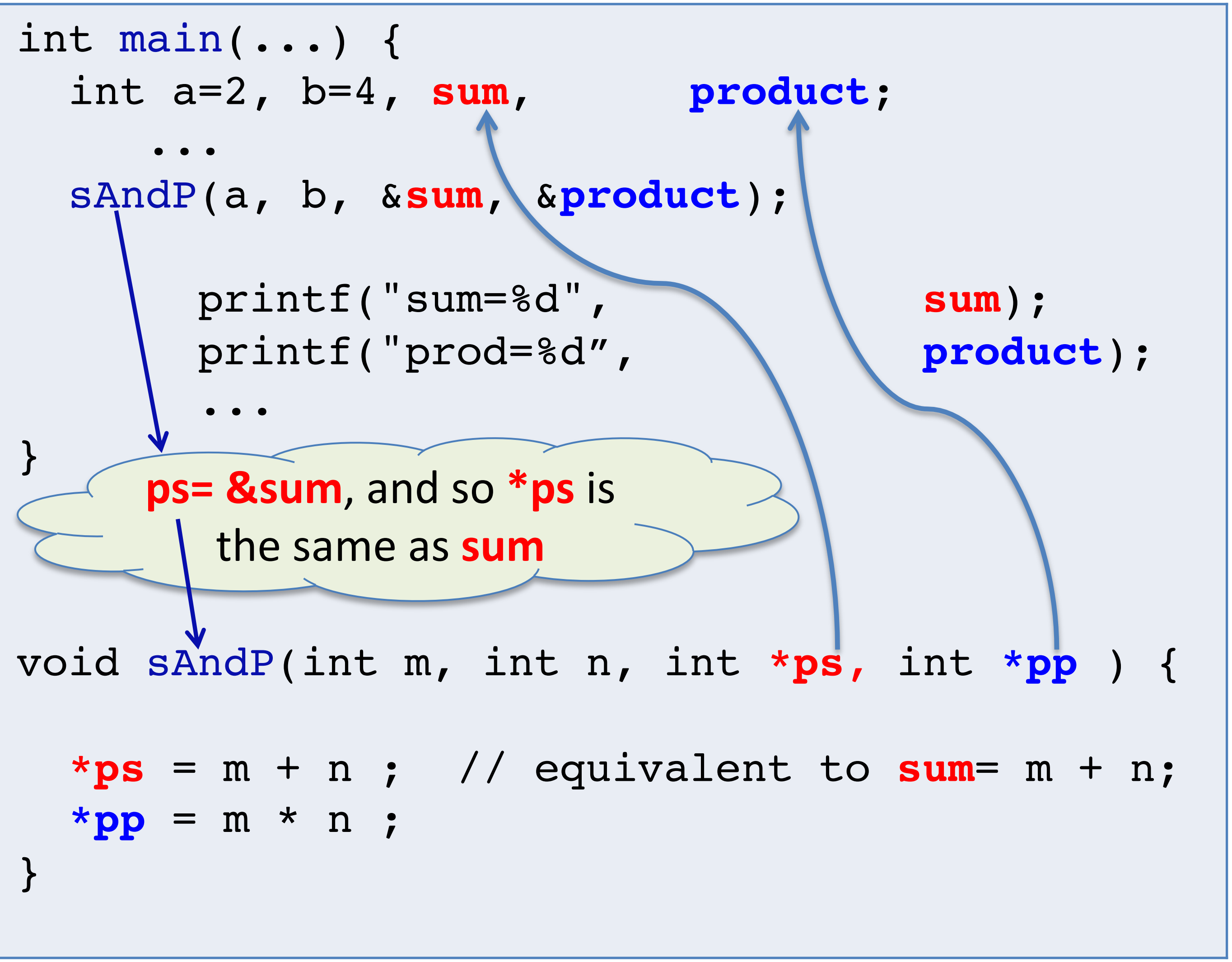
`int a= 18;`	18 **a**	**a** is a cell in the memory, interpreted as an **int**, with value of 18
`int *pa;`	? 18 **pa** **a**	**pa** is an **int pointer**, it can hold the address of an **int** ***pa** is of data type **int**
`pa= &a;`	&a → 18 **pa** **a**	**pa** now holds the address of **a**, or, it "points" to **a**. using the value of **pa** we can have *indirect access* to **a**

`(*pa)++;`	&a → 19
	pa **a**
		***pa** is now equivalent to **a**
		Equivalent statement:
		`*pa = *pa + 1;`
		`a = a + 1;`

pointers as function parameters

Function call in line 4 leads to the change of value of **sum** and **product**.

In this way, function sAndP can access local variables **sum** and **product** of main().



```
1    int main(...) {
2       int a=2, b=4, sum,      product;
3           ...
4       sAndP(a, b, &sum, &product);
5
6             printf("sum=%d",                sum);
7             printf("prod=%d”,               product);
8             ...
9    }

11   void sAndP(int m, int n, int *ps, int *pp ) {

12      *ps = m + n ;  // equivalent to sum= m + n;
13      *pp = m * n ;
14   }
```

ps= &sum, and so *ps is the same as sum

Have you fully understood function int_swap?

Header:

```
    void int_swap(int *a, int *b);
```

Example of using:

```
   int a=10, b= 5;
   int_swap(a, b);     ???
                      now a=    , b=
   int_swap(&a, &b);   ???
                      now a=    , b=
   if (a<b) int_swap(&a, &b); ???
                      now a=    , b=
```

Quiz 1

With the fragment:

```
int x= 10;
f(&x);
```

which function below will set `x` *to zero?*

A:

```
int f(int n) {
   return 0;
}
```

B:

```
void f( int *n) {
   &n= 0;
}
```

C:

```
void f (int *n) {
   n= 0;
}
```

D:

```
void f( int *n) {
   *n= 0;
}
```

Quiz 2

Given function:

```
void f(int a, int *b) {
   a= 1;
   *b = 2;
}
```

what are printed after the following fragment:

```
m= 5;
n= 10;
f(m, &n);
printf(“%d and %d\n”, m, n);
```

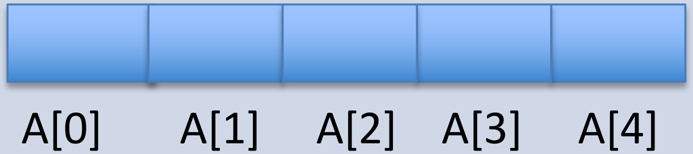
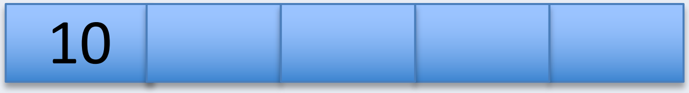
A) 5 and 10	B) 1 and 2	C) 5 and 2	D) 1 and 10

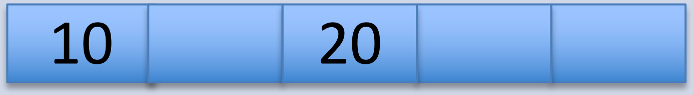
Arrays

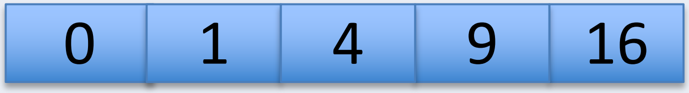
Can we have a chunk of 10 contiguous `int`, and make `A` be the address of the first of those `int` ? YES

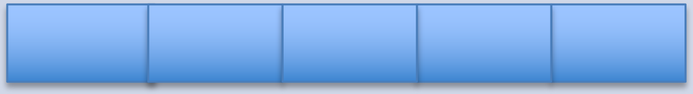
```
int A[10];
```

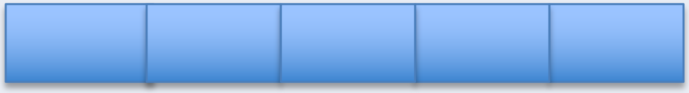
(one-dimensional) arrays

1	The statement `int A[5];`	
2	is equivalent to declaring 5 variables, each is of data type **`int`**:	
3	`A[0] = 10;`	

4	`i= 2;`
	`A[i]= 20;`
		

5	`for (i=0; i<5; i++) {`
	`    A[i]= i*i;`
	`}`
		

6	`for (i=1; i<5; i++) {`
	`    A[i]= A[i-1];`
	`}`
		

7	`for (i=0; i<5; i++) {`
	`    scanf ( “%d”, &A[i] );`
	`}`
		

Motto: “an array is a pointer constant”

With declaration:

```
#define MAX_N 1000
int A[MAX_N]= {10, 20, 30}, n= 3, i= 1;
int *p= A;
```

- `A[i]` is just an `int` variable
- `A` is: a constant? an array? a pointer? an address?
- `p` is a variable, and now `p` can be used as a substitute for `A`

- `p` ⇔ `A`, `p[i]` ⇔ `A[i]`

  but `p++` is valid, while `A++` isn’t

- `A+0` ⇔ `A` , `A+1` ⇔ “the `int*` next to `A`”
- `*A` ⇔ `A[0]` , `*(A+i)` ⇔ `A[i]`

Arrays

Passing array A to a function? We can just pass:

- A: the array name, and
- n: the actual number of elements in the array A.

Example:

```
void square_it(int A[], int n) {
   int i;
   for (i=0; i<n; i++) {
      A[i] *= A[i];
   }
}

...
   int A[]= {1,2,3,4,5}; // now A[4] has value 5
   square_it(A, 5);      // now A[4] has value 25
```

Searching

The Task:

Input: Given an array of n elements.

Output: A specific element satisfying some criteria.

Searching

The Task:

Input: Given an array of n elements.

Output: A specific element satisfying some criteria.

The function might look like

```
??? search( ??? A[], int n, <other inputs>) {
   int i;
   for(i=0; i<n; i++) {
      if ( A[i] satisfies the criteria ) {
         return A[i]; // or return i;
      }
   }
}
```

Note: Sometimes the criteria are just complicated, we might need to write a function for that...

Example: criteria = “is the k-th smallest of A”

Searching

The Task:

Input: Given an array of n elements.
Output: A specific element satisfying some criteria.

Note 2: Sometimes the criteria dictate a selection the best one from all elements, and if we already examined A[0] to A[i], we should have a solution-so-far which is the best of A[0..i].

Example: criteria: the minimal value

The function might look like

```
int min( int A[], int n) {
   int i=0;
   int soln= A[i];            // soln is min of A[0..0]
   for(i=1; i<n; i++) {
      if ( A[i] < soln ) {    // A[i] is a better solution ) {
         soln= A[i];          // updates soln to keep track
      }
                              // soln is min of A[0..i]
   }
   return soln;
}
```

Sorting

The Task:

Input: Given an array of n elements.

Output: The same input array, but the its elements are re-arranged in some *order* (such as *increasing*)

Example: sort array in non-decreasing order)

Input : `int A[5]= {7,2,5,5,6}, n= 5;`

Output: `A[ ]` becomes `{2,5,5,6,7}`

Algorithms:

- Insertion Sort
- Quick Sort
- …

7.6

An alternative sorting algorithm is selection sort. It goes like this: scan the array to determine the location of the largest element, and swap it into the last position. Then repeat the process, concentrating at each stage on the elements that have not yet been swapped into their final position.

Write a function

```
   void selection_sort (int A[], int n)
```

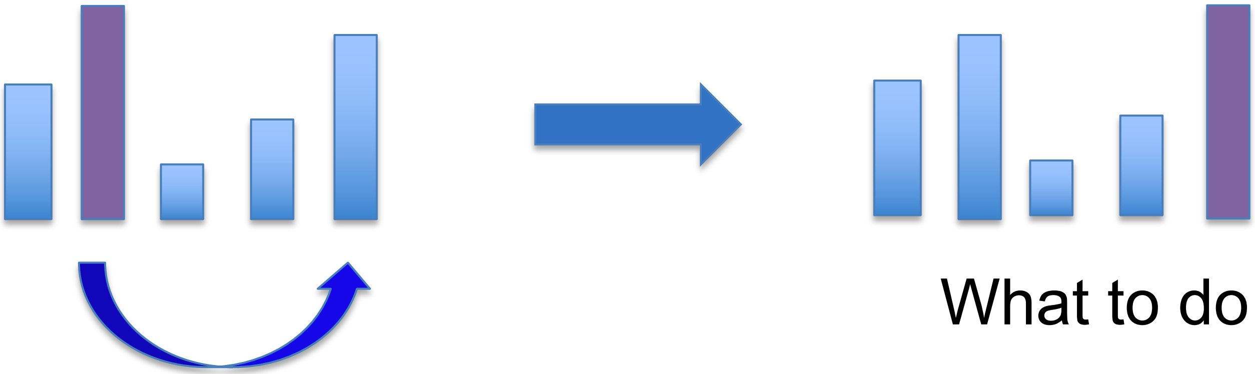
that orders the `n` *elements in array* `A`*.*

7.6: understanding

<u>Selection sort</u>: scan the array to determine the location of the largest element, and swap it into the last position. Then repeat the process …

So in the first round:

- *look at all elements from* `A[0]` *to* `A[n-1]`,
- *determine the location (?) of the largest element,*
- *swap it into the last position, ie.* `A[n-1]`.



What to do next?

7.6: recursive version (DIY: iterative (non-recursive) version)

Selection sort: scan the array to determine the location of the largest element, and swap it into the last position. Then repeat the process …

```
void rec_sel_sort(int a[], int n) {
  // base case

  // general case




}
```

DIT DoItTogether: Ex. 7.7 (grok W05) and a testbed for it

7.7: Write a function that takes as arguments an integer array `A` and an integer `n` that indicates how many elements of `A` may be accessed, and returns the value of the integer in `A` that appears most frequently. If there is more than one value in `A` of that maximum frequency, then the smallest such value should be returned. The array `A` may not be modified.

Function header= ?

Basic task=? Is that similar to sorting? to searching?

Lab: Group+Individual Works: Ex 7.7-7.9 and others.
Write a *function* that

7.2: [W06] sorts an array in deceasing order

7.3: [W06] sorts an array and removes duplicates

7.4: [W06] computes frequency of each value in an array

7.6: [W06] performs selection sort

7.7: **[W05]** returns the value that appears most frequently in an array `A` of integers. On tie, returns the smallest one. The array `A` may not be modified.

7.8: [W5X] returns the *k*-smallest of `int A[]`, not modifying `A`

7.9: [W05] returns the number of *ascending runs* in `int A[].` For example, array **`{10,13,16,18,15,22,21}`** has 3 runs.

7.10: [W5X] returns the number of inversions in `int A[].` For example, the above array has 3 inversions: 2 caused by **`15`**, and 1 by **`21`**.

`***` and all other exercises from `W05, W05X, W06, W06X`