ECE 220: Computer Systems & Programming

Lecture 9: Pointers and Arrays

Thomas Moon

February 13, 2024

Swap Function

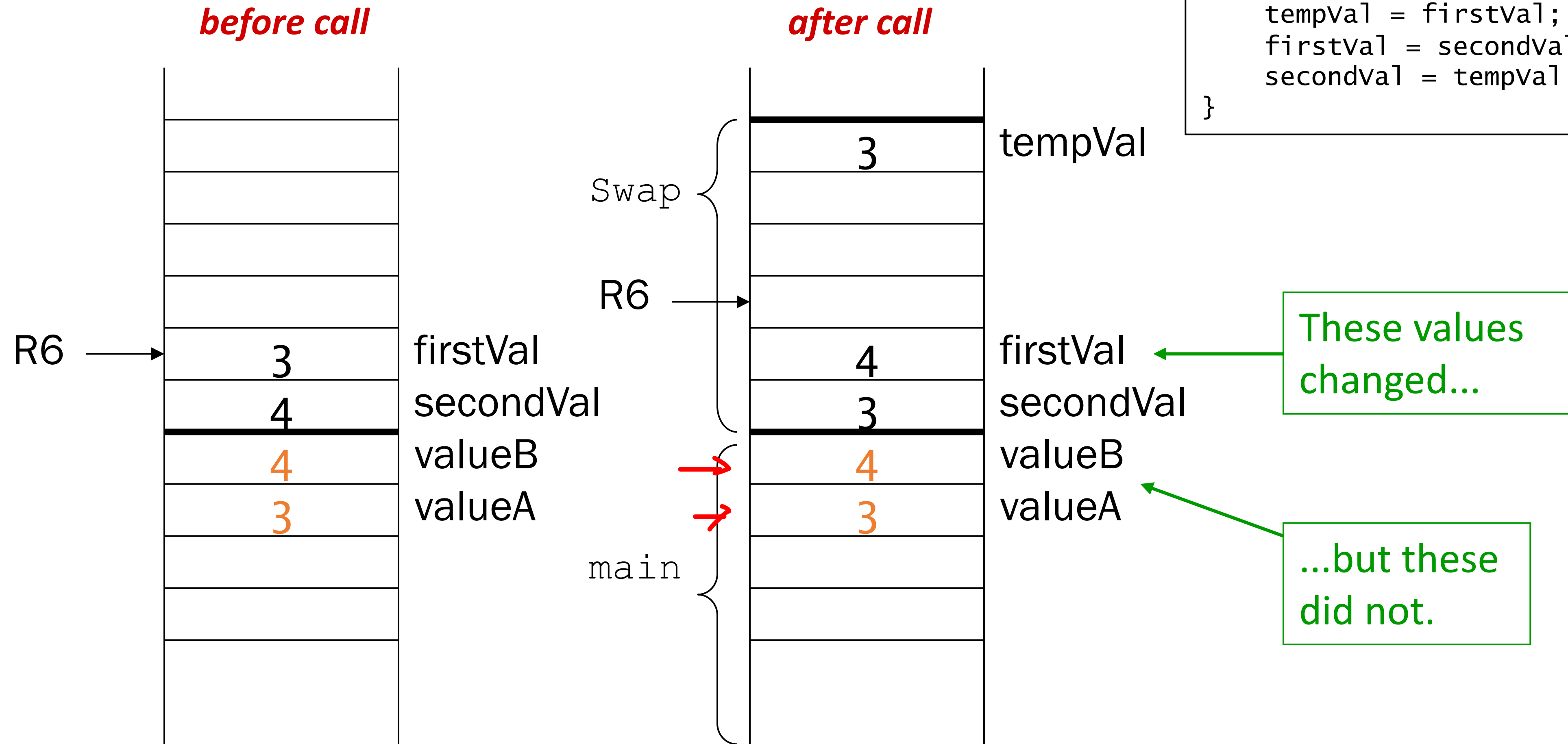
```
void Swap(int firstVal, int secondVal);
int main()
{
    int valueA = 3;
    int valueB = 4;

    Swap(valueA, valueB);
}

void Swap(int firstVal, int secondVal)
{
    int tempVal;

    tempVal = firstVal;
    firstVal = secondVal;
    secondVal = tempVal;
}
```

Goal:

Swap `valueA` and `valueB` in `main`.

Swap Function – Activation Record

```
int main(){
    int valueA = 3;
    int valueB = 4;
    Swap(valueA, valueB);
}
void Swap(int firstVal, int secondVal){
    int tempVal;
    tempVal = firstVal;
    firstVal = secondVal;
    secondVal = tempVal;
}
```



- Swap needs *addresses* of variables (valueA and valueB) outside its own activation record. Pointers solve this problem!

Pointers and Arrays

Pointer

- Stores the *address* of a variable in memory
- Allows us to indirectly *access/change* variables
- In other words, we can talk about its *address* rather than its *value*

Array

- A list of values arranged *sequentially* in memory
- Example: a list of telephone numbers
- Expression a[4] refers to the 5th element of the array a (index starts from 0)

Pointers and Arrays can be used *interchangeably*!

Pointers in C

Declaration

```
    int *ptr;   /* ptr is a pointer to an int */
or  int* ptr;
```

A pointer in C is always a pointer to a particular data type:
`int*, double*, char*, etc.`

'*' used in different purposes

Operators

`*ptr` → <u>dereference operator</u>: access the content pointed to by `ptr`

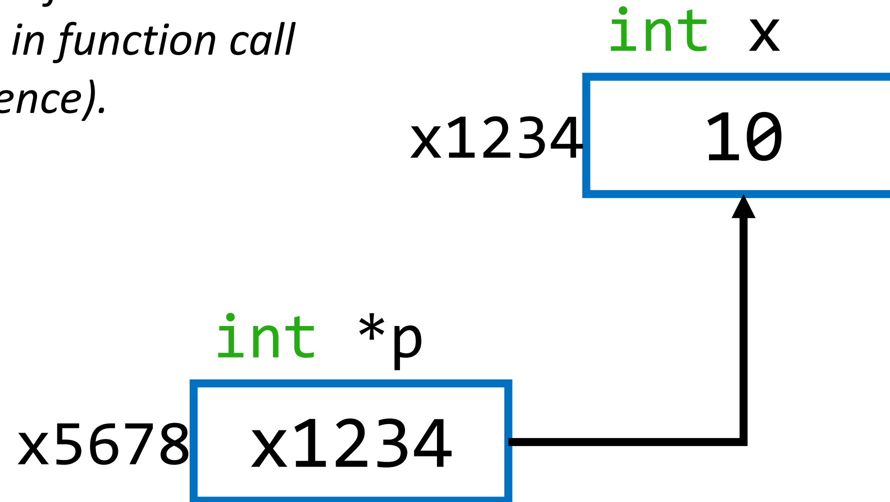
`&val` → <u>address operator</u>: returns the address of variable `val`

Pointer

```
int x = 10;
int *p;
p = &x;
```

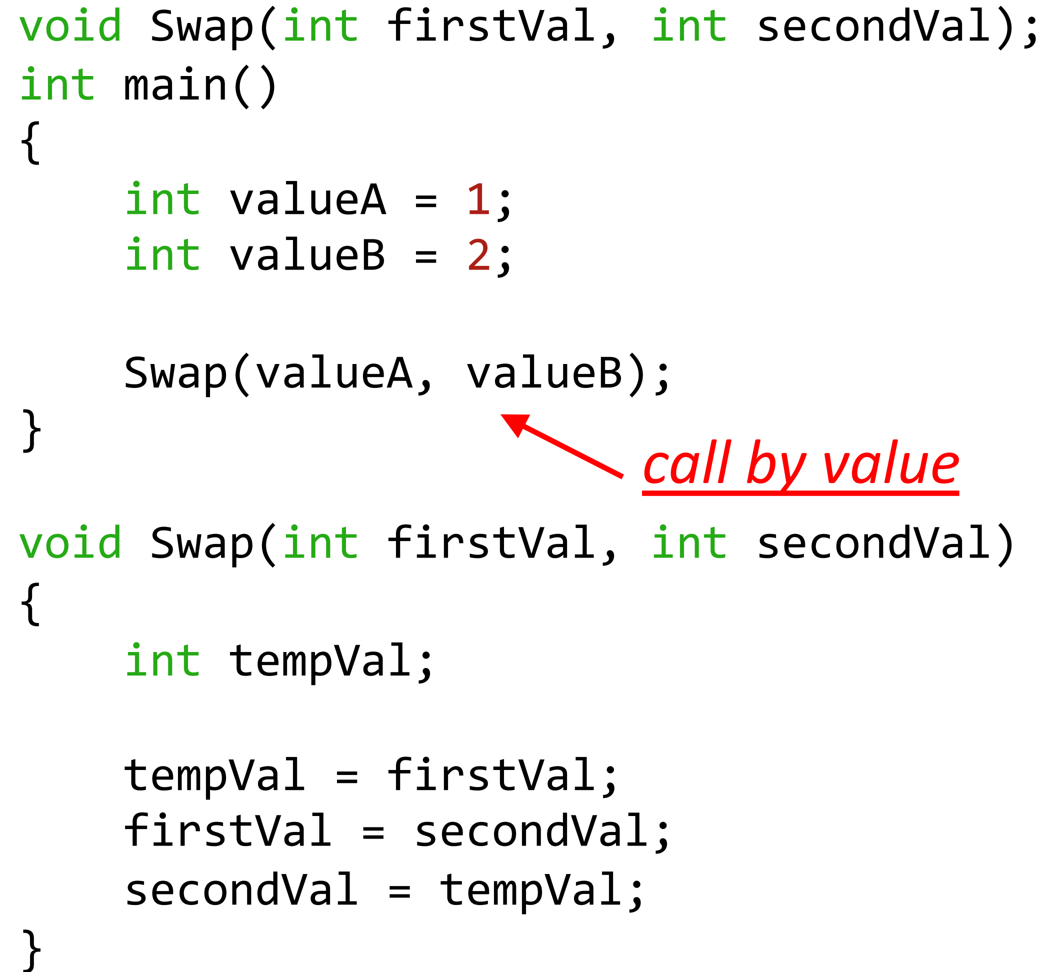
or `int *p = &x;`

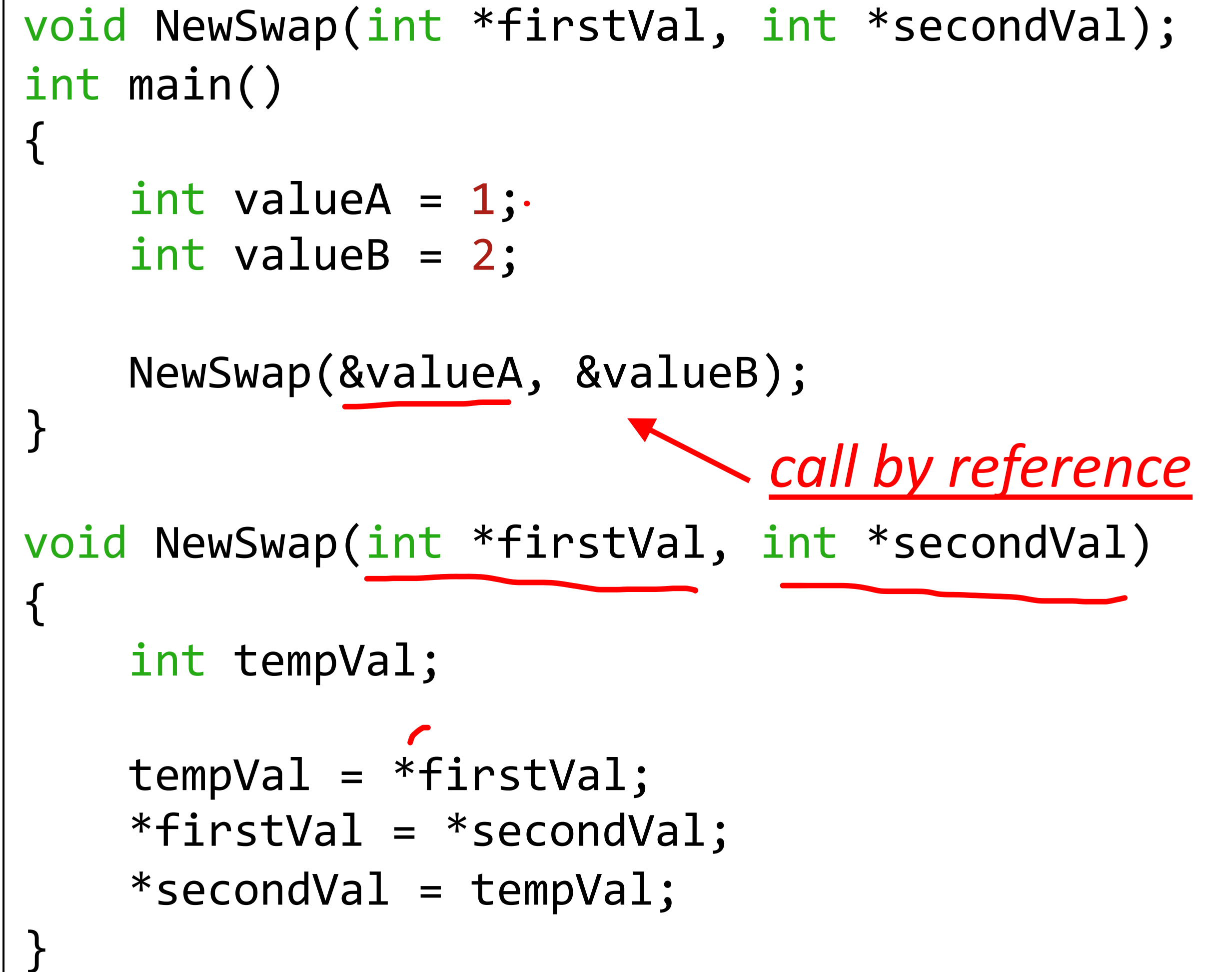
Remember this form.
We'll use this in function call (call-by-reference).




```
printf("x%X\n", &x);     x1234
printf("x%X\n", p);      x1234
printf("x%X\n", &p);     x5678
printf("%d\n", *p);      10

*p = *p + 10;
printf("%d\n", *p);      20
printf("%d\n", x);       20
```

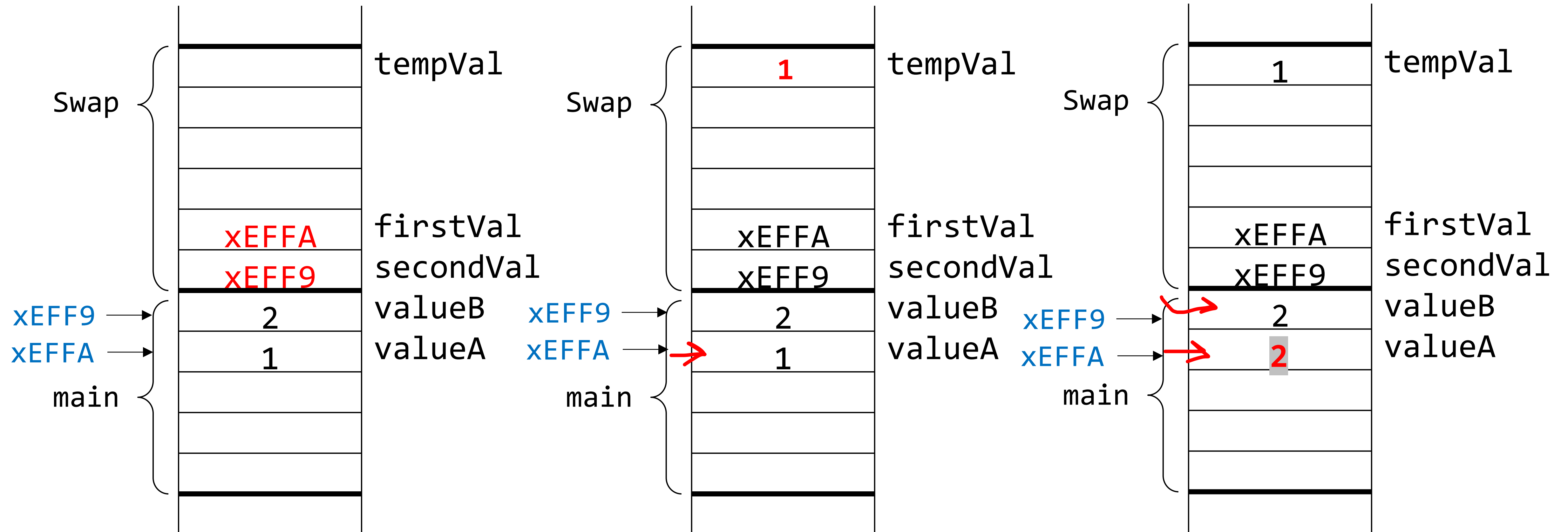
Solve Swap Problem





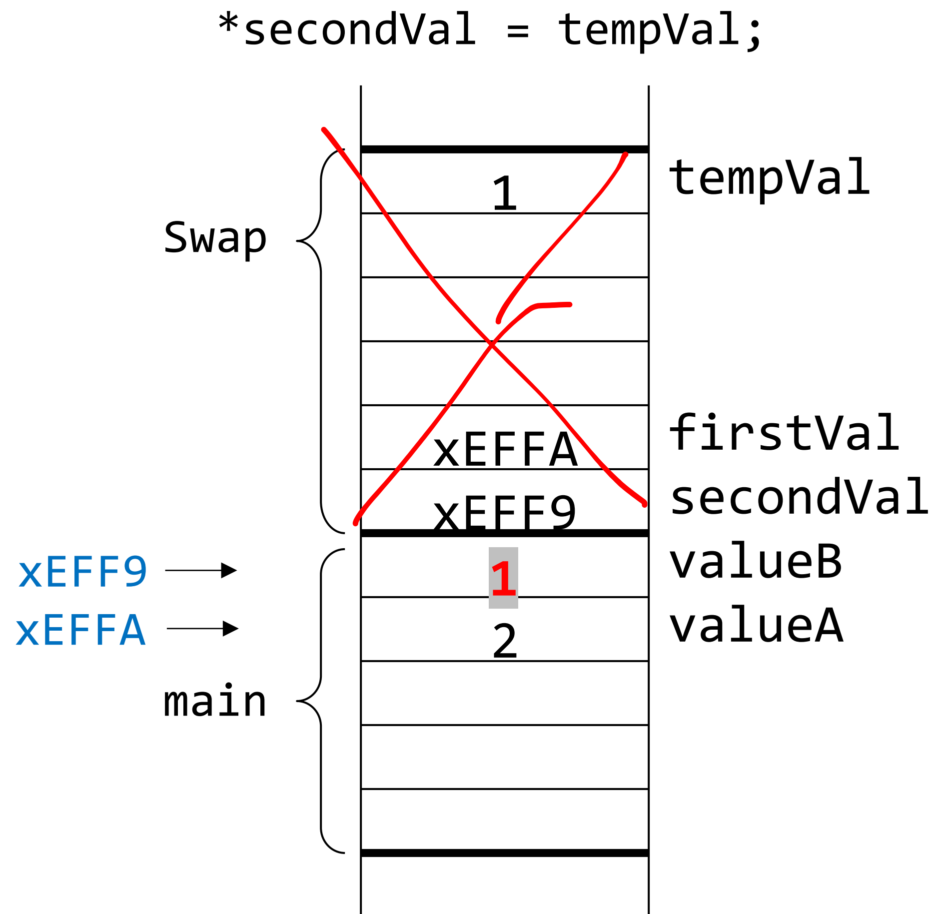
New Swap – Activation Record

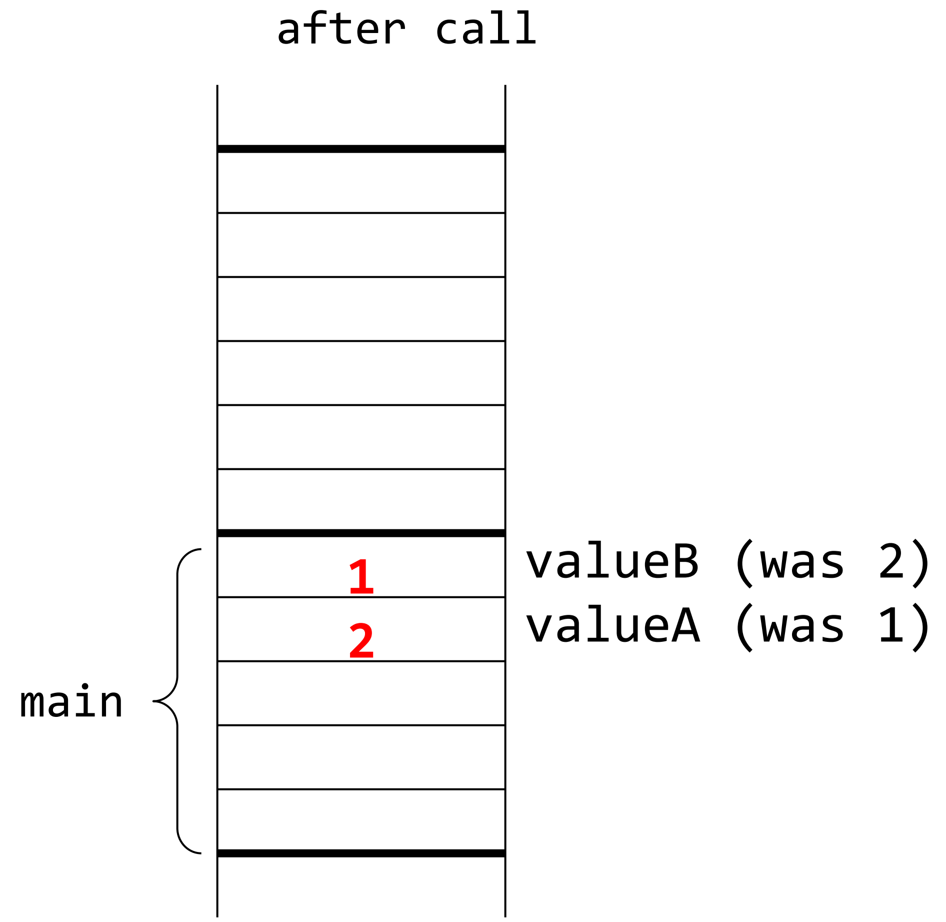
```
void NewSwap(int *firstVal, int *secondVal)
{
    int tempVal;

    tempVal = *firstVal;
    *firstVal = *secondVal;
    *secondVal = tempVal;
}
```



New Swap – Activation Record (continued)

```
void NewSwap(int *firstVal, int *secondVal)
{
    int tempVal;

    tempVal = *firstVal;
    *firstVal = *secondVal;
    *secondVal = tempVal;
}
```





Too many *?

```
int main()
{
    int valueA = 1;
    int valueB = 2;

    NewSwap(&valueA, &valueB);
}

void NewSwap(int *firstVal, int *secondVal)
{
    int tempVal;

    tempVal = *firstVal;
    *firstVal = *secondVal;
    *secondVal = tempVal;
}
```

① ② ③ ④ ⑤ ⑥

Which * is used for the dereference operator?

3,4,5,6

Which * is used for declaring a pointer variable?

1,2

```
int *firstVal = &valueA;
int *secondVal = &valueB;
```

```
int *firstVal;
firstVal = &valueA;

int *secondVal;
secondVal = &valueB;
```

Too many *?

```
int main()
{
    int valueA = 1;
    int valueB = 2;

    NewSwap(&valueA, &valueB);
}
                   ①              ②
void NewSwap(int *firstVal, int *secondVal)
{
    int tempVal;
                   ③
    tempVal = *firstVal;
④   *firstVal = *secondVal; ⑤
⑥   *secondVal = tempVal;
}
```

```
*firstVal = &valueA;
int *firstVal = valueA; ←
```

```
int *firstVal = &valueA;

int *secondVal = &valueB;
```

```
int *firstVal;
firstVal = &valueA;

int *secondVal;
secondVal = &valueB;
```

More on Pointers

- Null pointer: a pointer that points to nothing

```
int *ptr;
int valueA;

ptr = &valueA;
printf("x%X\n", ptr);

ptr = NULL;
printf("x%X\n", ptr);
```

=0

```
x1B2F5F3C
x0
```

- Demystifying '&' in scanf

```
int input;
scanf("%d", &input);
```

- scanf needs to update the variable from the keyboard. Therefore, It needs the address of the variable, not its value.

Common Mistakes on Pointers

```c
int val = 5;
char *ptr;
ptr = &val;
```

Type mismatch

```c
int *ptr;
*ptr = 4;
```

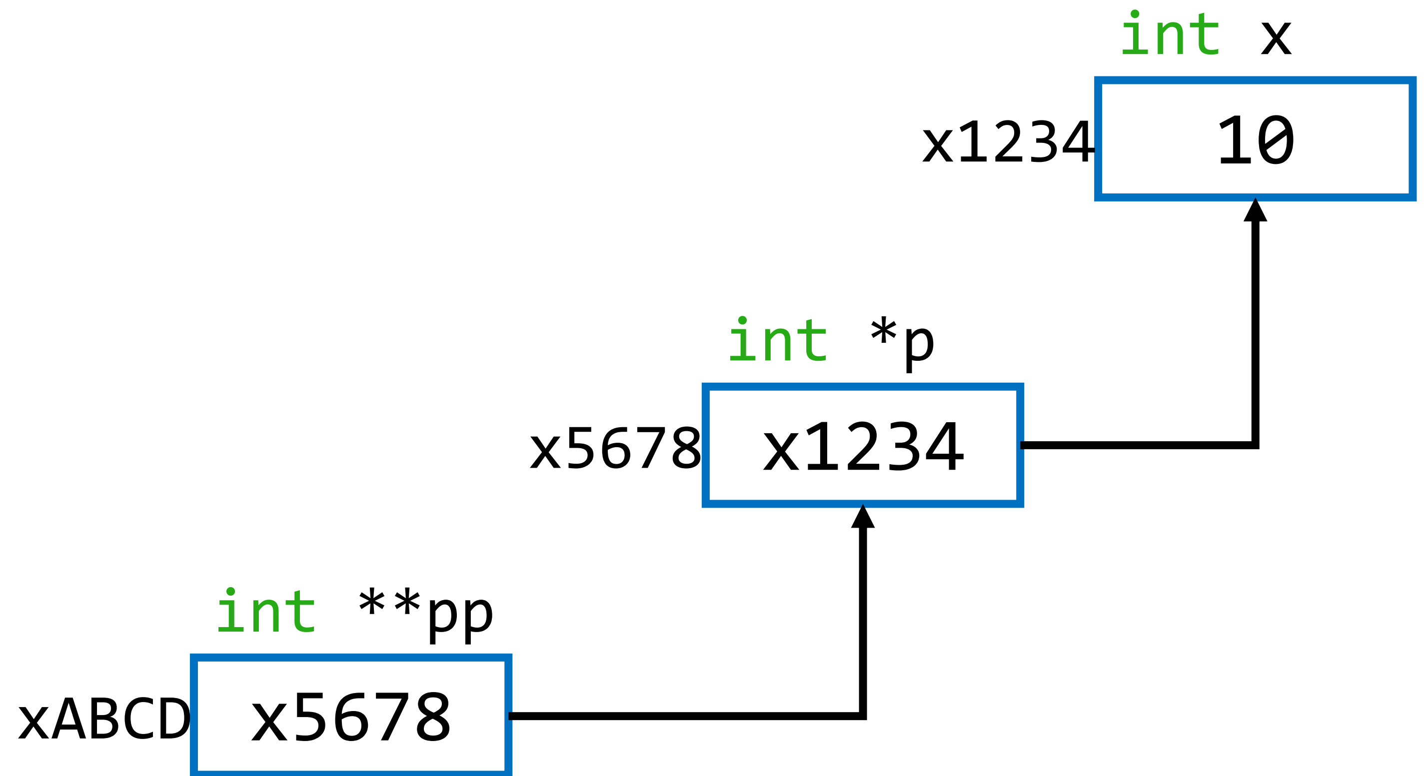
Dereferencing a pointer before initialized

```c
int *ptr;
ptr = 4;
```

Pointing memory address “4”
(exception 0 (NULL))

Double Pointer

```
int x = 10;
int *p;

p = &x;

int **pp;
pp = &p;
```




```
printf(”x%X\n", &pp);     xABCD
printf(”x%X\n", pp);      x5678
printf(”x%X\n", *pp);     x1234
printf(”%d\n", **pp);     10
```

Array

- How do we allocate a group of memory locations?
  - character string
  - table of numbers

```
int num0;
int num1;
int num2;
int num3;
.
.
.
```

- Problem1:
  What if 100 numbers?
- Problem2:
  How to write a loop to process each number?

- Solution: Array

```
int num[100];
num[0]=1;
```

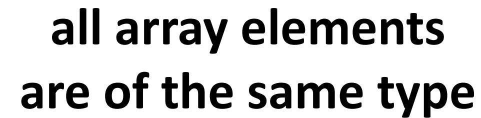
max/min index?

num[100]=1; 99 0

Array Syntax

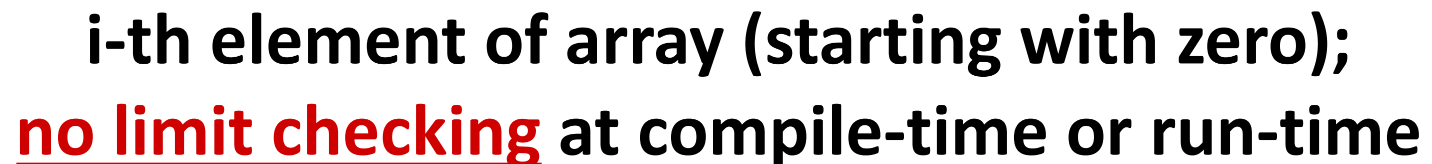
- Declaration

```
type  variable[num_elements];
```



- Array reference

```
variable[index];
```



Example

```
int grid[10] = {2,10,21,3,6,1,0,9,11,12};

grid[6] = grid[3]+1;

int i;
for(i=0;i<10;i++)
    printf("%d\n", grid[i]);
```

```
2
10
21
3
6
1
4
9
11
12
```

Array Size?

- Once you declared an array, you cannot change its size (except using *dynamic allocation*…)
- You need to tell its size to the compiler *somehow.*

```
int array[2];              →OK
int array[] = {1,2};       →OK
int array[];               →Compiler Error
```

- *No bound-checking!*

```
int array[] = {1,2};
printf("%d %d %d\n", array[0], array[1], array[2]);
```
→No error

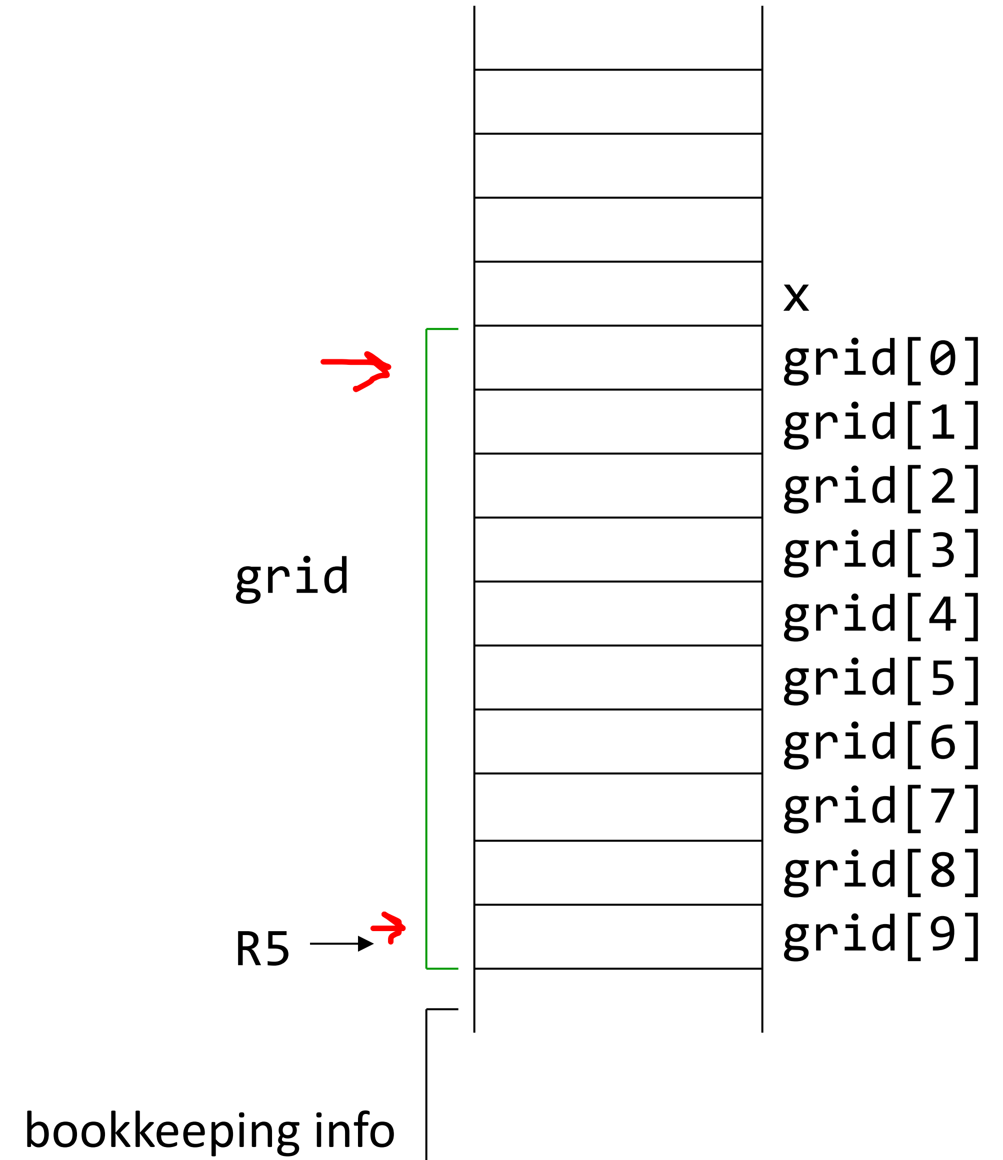
```
1 2 1111558128
```
→garbage value for `array[2]`

Array in Activation Record

```
int grid[10];
int x;
.
.
.
```

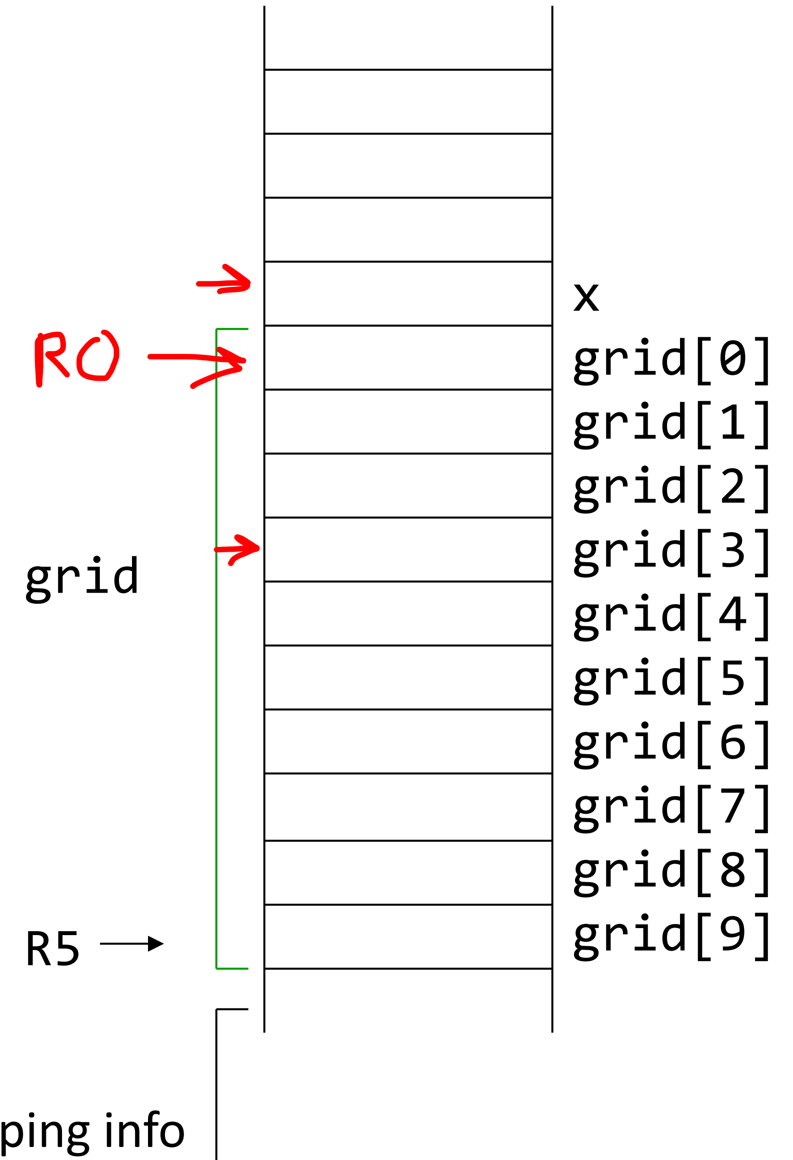
*First element (grid[0]) is at *lowest* address of allocated space.

*If grid is the first local variable, then **R5** points to grid[9].



LC-3 for Array References

```
; x = grid[3] + 1

ADD R0, R5, #-9      ; R0 = &grid[0]
LDR R1, R0, #3       ; R1 = grid[3]
ADD R1, R1, #1       ; plus 1
STR R1, R5, #-10     ; x = R1
```



Relationship between Pointers and Arrays

- An array name points to the first element in the array.

```
int word[10];
int *cptr;

cptr = word; // points to word[0]
```

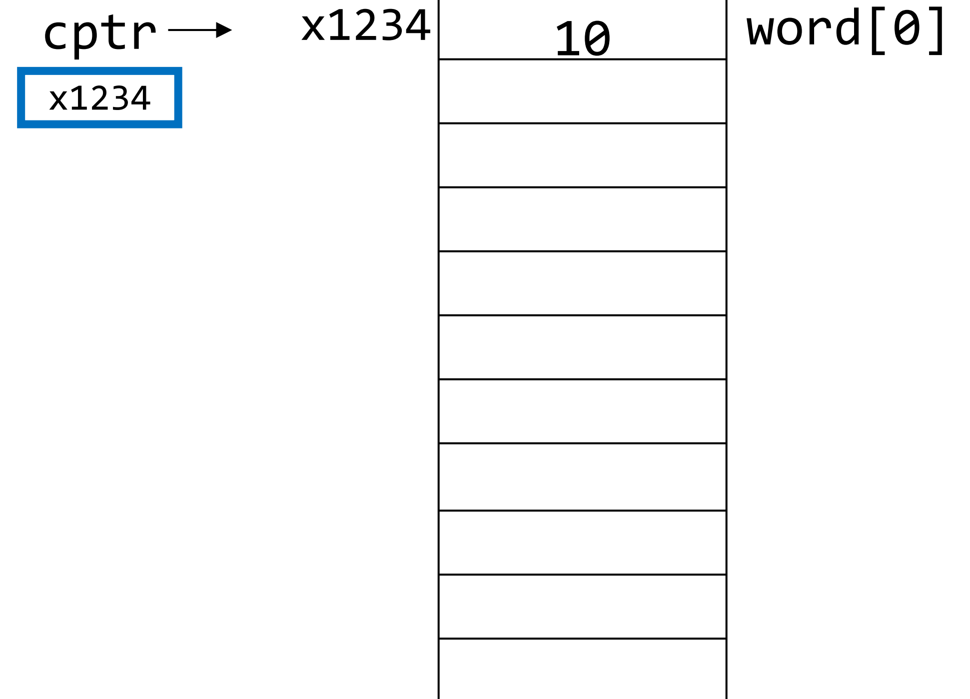
- What is difference?

cptr is a variable, but the name “word” is not a variable.

```
cptr = cptr + 1;
word = word + 1; // compile error
```

Correspondence between Pointer and Array Notation

```
int word[10];
int *cptr;

cptr = word;


printf(”%p\n", cptr);          0x1234
printf(”%p\n", word);          0x1234

printf(”%p\n", &word[0]);      0x1234

printf(”%d\n", word[0]);         10
```



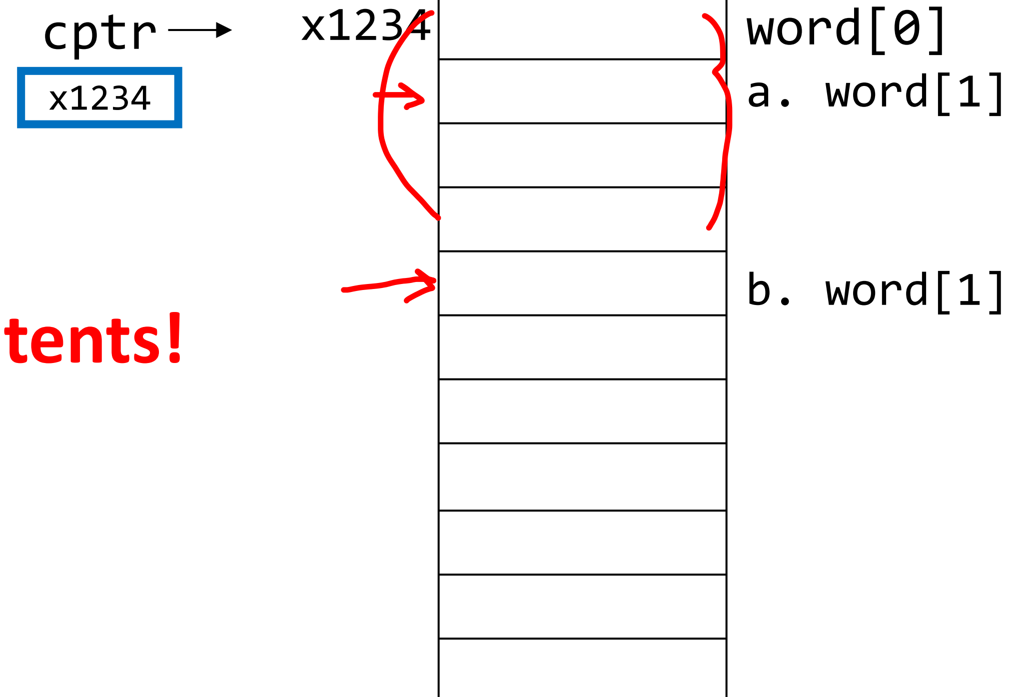
Correspondence between Pointer and Array Notation

```
int word[10];
int *cptr;

cptr = word;

cptr = cptr + 1;
```

incrementing it by the size of **its contents!**

```
printf(”%p\n", cptr);
```

1. 0x1234
2. 0x1235
3. We don’t know



If word[1] starts at a → cptr will be incremented to x1235

If word[1] starts at b → cptr will be incremented to x1238

Correspondence between Pointer and Array Notation

```
int word[10];
int *cptr;

cptr = word; // points to word[0]
```

- Each line below gives three equivalent expressions:

```
cptr            word            &word[0]
cptr + n        word + n        &word[n]
*cptr           *word           word[0]
*(cptr + n)     *(word + n)     word[n]
```

Passing Array as Arguments

- C passes arrays by reference
  - The address of the array (*address of the first element*) is written to the function's activation record.

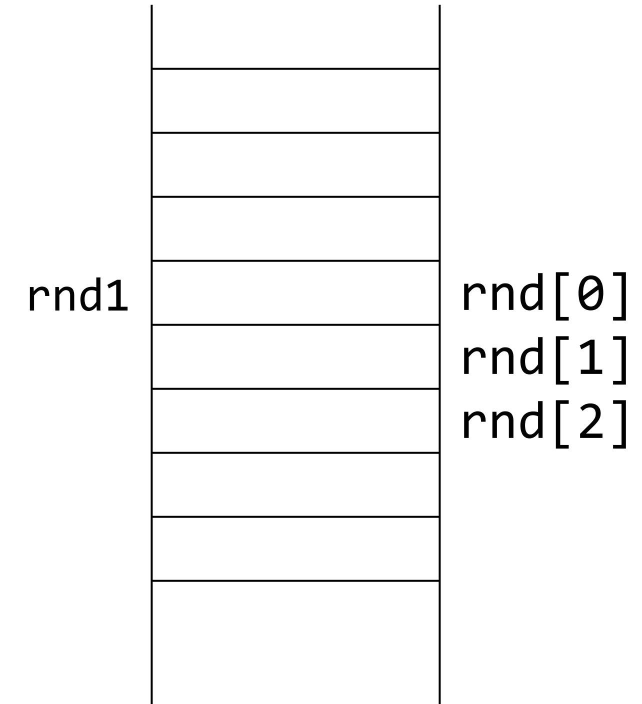
```
#define MAX_NUM 10
int main(){
    int array[MAX_NUM];
    int result;
    result = func(array);
}
```

Which function declaration is correct?

```
int func(int array[]);

int func(int *array);

int func(int array[MAX_NUM]);
```

Example: Roll a dice!

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
int main(){
    srand(time(0));
    int rnd[3];

    // update rnd as 3 integers
    roll_a_dice1(    &rnd[0],    &rnd[1] , &rnd[2] );


    // update rnd as array
    roll_a_dice2(rnd);
```



```c
// generate 3 random numbers, 1-6
// rand(): range between 0 to INT_MAX
void roll_a_dice1(int *rnd1, int *rnd2, int *rnd3){
    *rnd1 = rand()%6 + 1;
    *rnd2 = rand()%6 + 1;
    *rnd3 = rand()%6 + 1;
}

void roll_a_dice2(int array[]) {
    array[0] = rand()%6 + 1;
    array[1] = rand()%6 + 1;
    array[2] = rand()%6 + 1;

    *array = rand()%6 + 1;
    *(array+1) = rand()%6 + 1;
    *(array+2) = rand()%6 + 1;
}
```

Exercise: Implement a function to reverse an array

```
/*array_reverse(): reverses an integer array, such that the first
element will become the last element, the second element will
become the second to last element and so on. This function takes
two arguments: a pointer to an integer array and its size.*/
void array_reverse(int array[], int n)
{

}



void print_array(int array[], int n)
{

}
```