Fundamentals of Programming

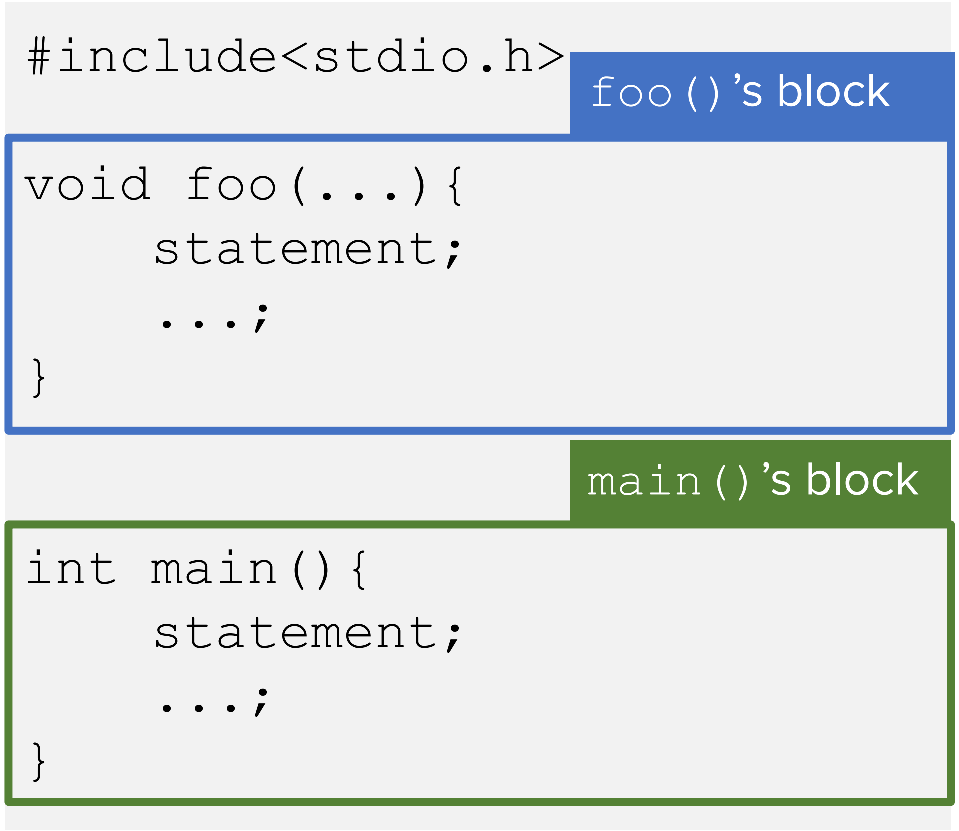
Lecture 12

Akara Supratak
Jidapa Kraisangka
Pilailuck Panphattarasap

RECAP

Function Workspace: Blocks

Every function has its own Workspace (block)



```
#include<stdio.h>
```

foo()'s block

```
void foo(...){
    statement;
    ...;
}
```

main()'s block

```
int main(){
    statement;
    ...;
}
```

Function Workspace: Pass by Value

We usually pass just the “values”, not variables

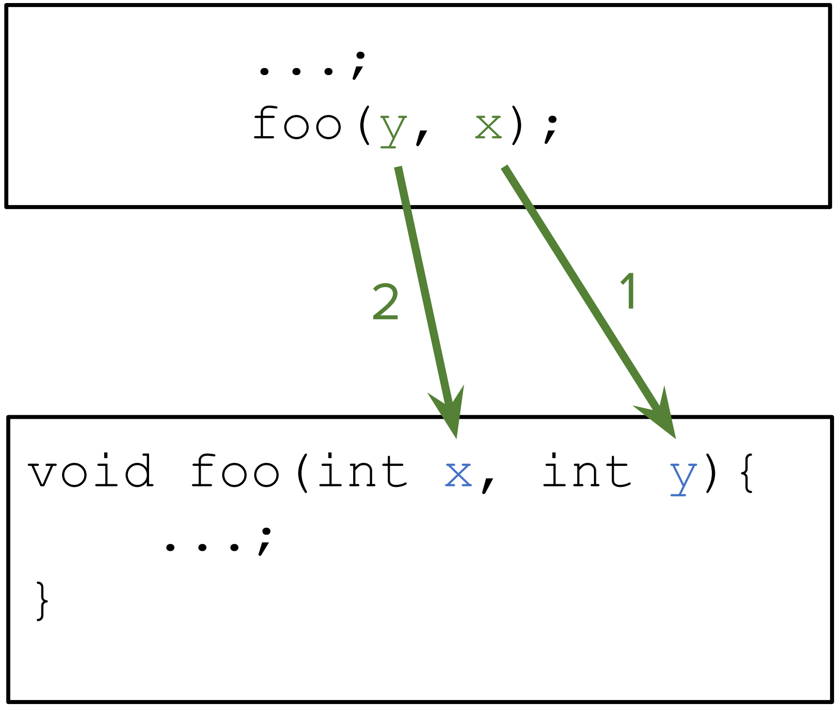
```
#include<stdio.h>

void foo(int x, int y){
    y = 10;
    printf("%d %d", x, y);
}

int main(){
    int x = 1, y = 2;
    foo(y, x);
    printf("%d %d", x, y);
    return 0;
}
```

```
...;
foo(y, x);
    2  1
```

Function Workspace: Pass by Value

We usually pass just the “values”, not variables

```
#include<stdio.h>

void foo(int x, int y){
    y = 10;
    printf("%d %d", x, y);
}

int main(){
    int x = 1, y = 2;
    foo(y, x);
    printf("%d %d", x, y);
    return 0;
}
```



```
        ...;
        foo(y, x);
```

```
void foo(int x, int y){
    ...;
}
```

Variable Scope: Override

- If variables have the same name, the most local one will be used
- For example, a variable declared within a function will override the global variable of the same name.

```
#include<stdio.h>

char a = 'a';

int main(){
    int a;
    a = 10;
    printf("%d", a);
    return 0;
}
```

Output

```
10
```

Today’s Topics

- Pointers and Addresses
- Pass by Reference
- Using Pointer with Array

Pointer and Address

Concept of Variables

Variable: a symbolic name associated with a value; its value can be varied.

- C compiler assigns a specific block of memory within the computer to hold the value of that variable.
- The size of that block depends on the data type.

```
datatype variable_name;
```

Look Inside Computer Memory

T	H	I	S		I	S	
I	T	C	S		2	0	1
	C	L	A	S	S	$	%
&	*	(	+	=	@	!	197
	521	1.6					

Look Inside Computer Memory

```
int x = 24;
```

T	H	I	S		I	S	
I	T	C	S		2	0	1
	C	L	A	S	S	$	%
&	*	(	+	=	@	!	197
24	521	1.6					

Look Inside Computer Memory

T	H	I	S		I	S	
I	T	C	S		2	0	1
	C	L	A	S	S	$	%
&	*	(	+	=	@	!	197
24	521	1.6	…	…	…		
0	1	2	3	4	5	6	7
8	9						

```
int x = 24;
```

```
int grades[10];
grades[0] = 0;
…
grades[9] = 9;
```

Concept of Variables

There are **two** components associated with each variable.

- Value
- Memory address

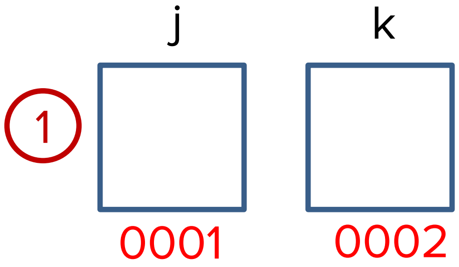
```
int j, k;
k = 2;
j = 7;
k = j;
```

Concept of Variables

There are **two** components associated with each variable.

- Value
- Memory address

1
```
int j, k;
k = 2;
j = 7;
k = j;
```

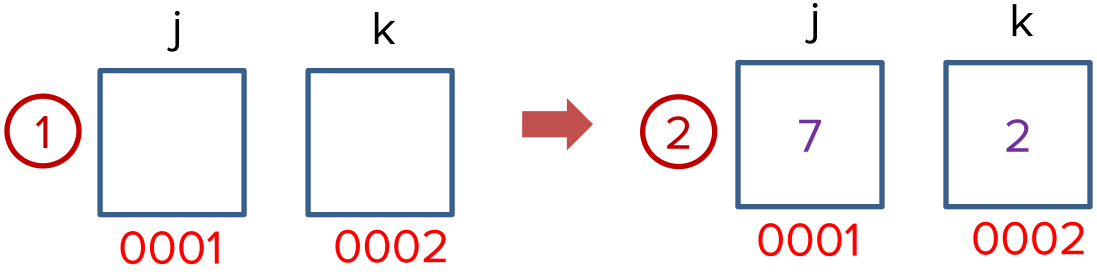


Concept of Variables

There are **two** components associated with each variable.

- Value
- Memory address

```
①  int j, k;
    k = 2;
②
    j = 7;
    k = j;
```

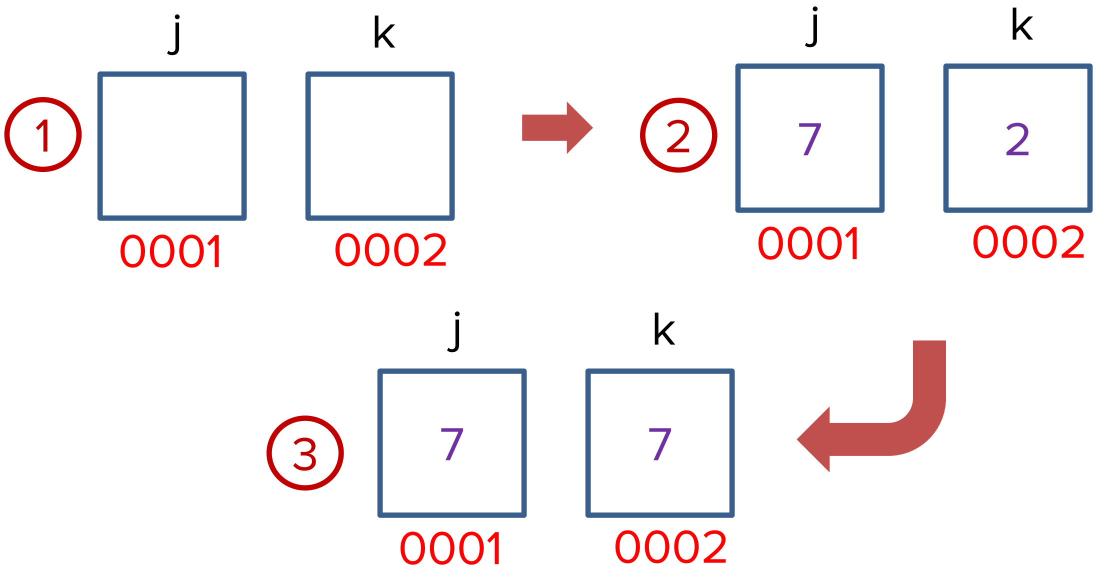


Concept of Variables

There are **two** components associated with each variable.

- Value
- Memory address

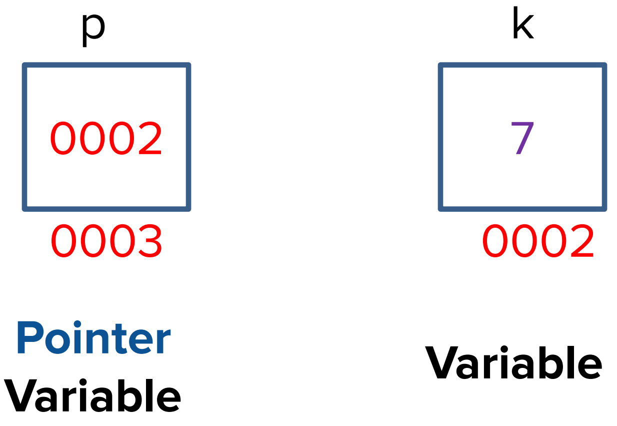
```
1  int j, k;
   k = 2;
2
   j = 7;
3  k = j;
```



What is Pointer?

Pointer: a variable that stores the memory address of another variable located in computer memory.

- A pointer ***references*** a location in memory
- Obtaining the value stored at that location is known as ***dereferencing*** the pointer.



Why Pointer?

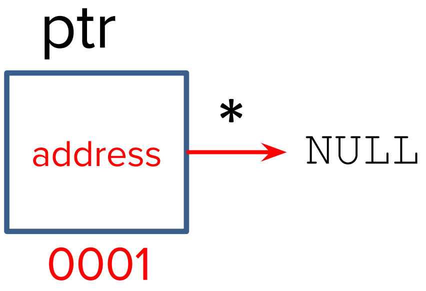
- Modify the values of the variables of the function’s caller.
- Pass an (big) array to a function.
- etc.

Pointer Declaration

```
datatype *variable_name;
```

- `*` (asterisk) is to inform the C compiler that we want a pointer variable.
- `datatype` is used to specify the type of the value that the pointer will point to.
- E.g., a pointer that can point to an integer variable:

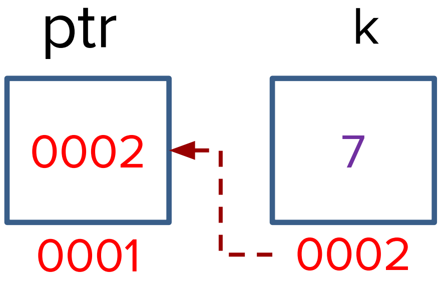
```
int *ptr;
```



Point to a variable

When we want a pointer to point to a variable, we will assign the address of that variable to the pointer.

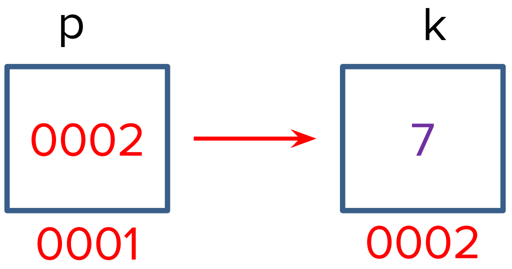
```
ptr = &k;
```



& operator retrieves the address of the variable `k`.

Now, `ptr` is said to "point to k".

Dereferencing



Dereferencing: Obtain the value of the variable that a pointer is pointing to.

- The “dereferencing operator” or “indirection operator” is the asterisk (*****)

```
*ptr
```

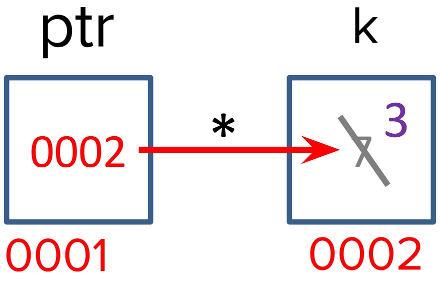
```
printf("%d",*ptr);
int a = *ptr;
int b = a + *ptr;
if (*ptr > 10) {
    …
}
```

```
printf("%d",k);
int a = k;
int b = a + k;
if (k > 10) {
    …
}
```

Dereferencing

We can also change the value of the variable that the pointer is pointing to:

```
*ptr = 3;
```



Example

```
int num = 3;
int *p_num;

// Assign the address of num to p_num
p_num = &num;

// Output?
printf("%d %d", num, *p_num);

// Change num
num = 5;

// Output?
printf("%d %d", num, *p_num);
```

Example

```c
int num = 3;
int *p_num;

// Assign the address of num to p_num
p_num = &num;

// Output? 3 3
printf("%d %d", num, *p_num);

// Change num
num = 5;

// Output? 5 5
printf("%d %d", num, *p_num);
```

Exercise

```
int num = 17;
int *p_num;
p_num = &num;

printf("%d", *p_num);   // Output?

num = 14;
int x = *p_num;
*p_num = -7;

printf("%d", x);   // Output?

printf("%d", num);   // Output?
```

Summary

Pointer Declaration	`int *p_num;`
Point to a variable	`p_num = &x;`
Dereferencing	`a = 15 + *p_num;`

	Variable (int x)	Pointer (int *ptr)
Value	x	*ptr
Address	&x	ptr

Pass by Reference

Function Call

Pass by value: Passing copies of values of variables to a function

Pass by reference: Passing copies of addresses of variables to a function

Function Call

Pass by value: Passing copies of values of variables to a function

Pass by reference: Passing copies of addresses of variables to a function

Pass by Ref. Pass by Val. Pass by Ref.

```
return_type function_name(type1 *name1, type2 name2, type3 *name3)
{
    statement1;  // may define new params
    statement2;  // may use arguments
    …
    return expression;
}
```

Function Call

Pass by value: Passing copies of values of variables to a function

Pass by reference: Passing copies of addresses of variables to a function

Pass by Ref. Pass by Val. Pass by Ref.

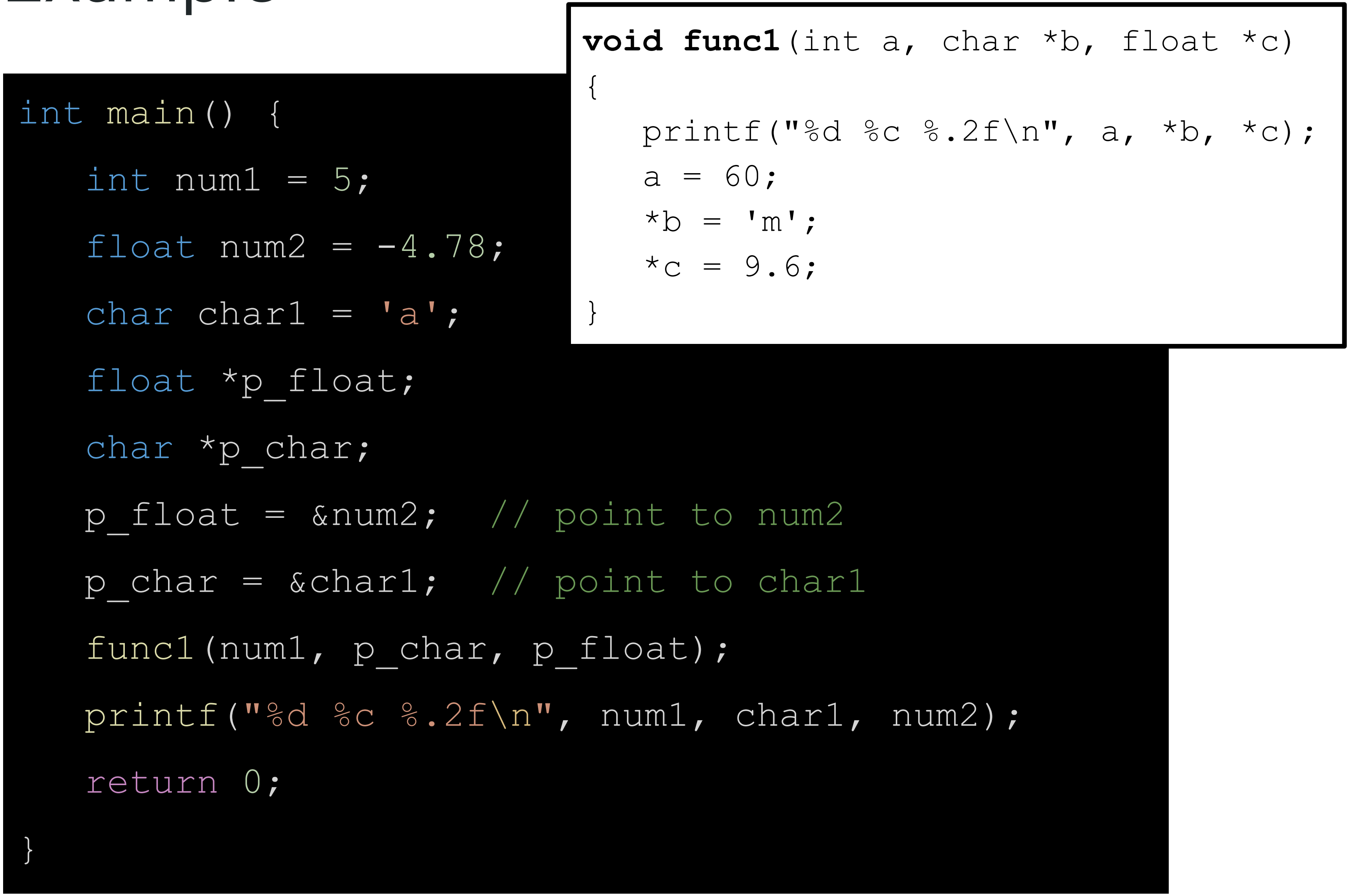
```
return_type function_name(type1 *name1, type2 name2, type3 *name3)
{
    statement1;  // may define new params
    statement2;  // may use arguments
    …
    return expression;
}
```

Inside the function, they can be used in exactly the same way as the pointers.

Why Pass by Reference?

- A function can change the value of the argument
- A function can receive and process arrays (discussed later)

Example



```
void func1(int a, char *b, float *c)
{
    printf("%d %c %.2f\n", a, *b, *c);
    a = 60;
    *b = 'm';
    *c = 9.6;
}
```

```
int main() {
    int num1 = 5;
    float num2 = -4.78;
    char char1 = 'a';
    float *p_float;
    char *p_char;
    p_float = &num2;   // point to num2
    p_char = &char1;   // point to char1
    func1(num1, p_char, p_float);
    printf("%d %c %.2f\n", num1, char1, num2);
    return 0;
}
```

Example

```
int main() {
    int num1 = 5;
    float num2 = -4.78;
    char char1 = 'a';
    float *p_float;
    char *p_char;
    p_float = &num2;  // point to num2
    p_char = &char1;  // point to char1
    func1(num1, p_char, p_float);
    printf("%d %c %.2f\n", num1, char1, num2);
    return 0;
}
```

```
void func1(int a, char *b, float *c)
{
   printf("%d %c %.2f\n", a, *b, *c);
   a = 60;
   *b = 'm';
   *c = 9.6;
}
```

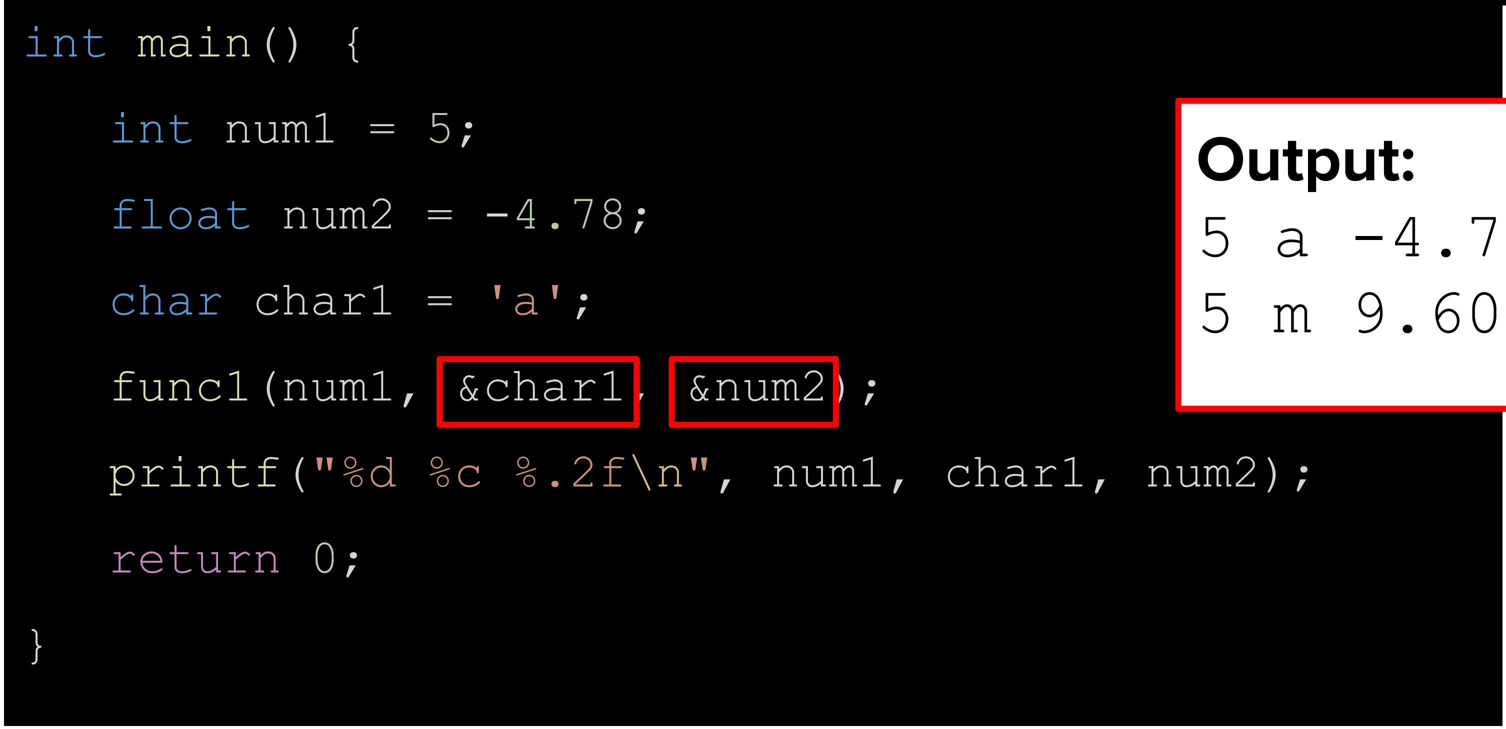
Output:

```
5 a -4.78
5 m 9.60
```

Example

We can also just pass the address of the variables as the input arguments.

```
void func1(int a, char *b, float *c)
{
    printf("%d %c %.2f\n", a, *b, *c);
    a = 60;
    *b = 'm';
    *c = 9.6;
}
```



```
int main() {
    int num1 = 5;
    float num2 = -4.78;
    char char1 = 'a';
    func1(num1, &char1, &num2);
    printf("%d %c %.2f\n", num1, char1, num2);
    return 0;
}
```

Output:

```
5 a -4.78
5 m 9.60
```

Example: Swap values

```
void swap(int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}

int main()
{
    int x, y;
    x = 5;
    y = 10;
    swap(x, y);
    printf("x=%d y=%d\n", x, y);
    return 0;
}
```

Example: Swap values

```
void swap(int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}

int main()
{
    int x, y;
    x = 5;
    y = 10;
    swap(x, y);
    printf("x=%d y=%d\n", x, y);
    return 0;
}
```

DOES NOT exchange the values !!!

Output:

```
x=5 y=10
```

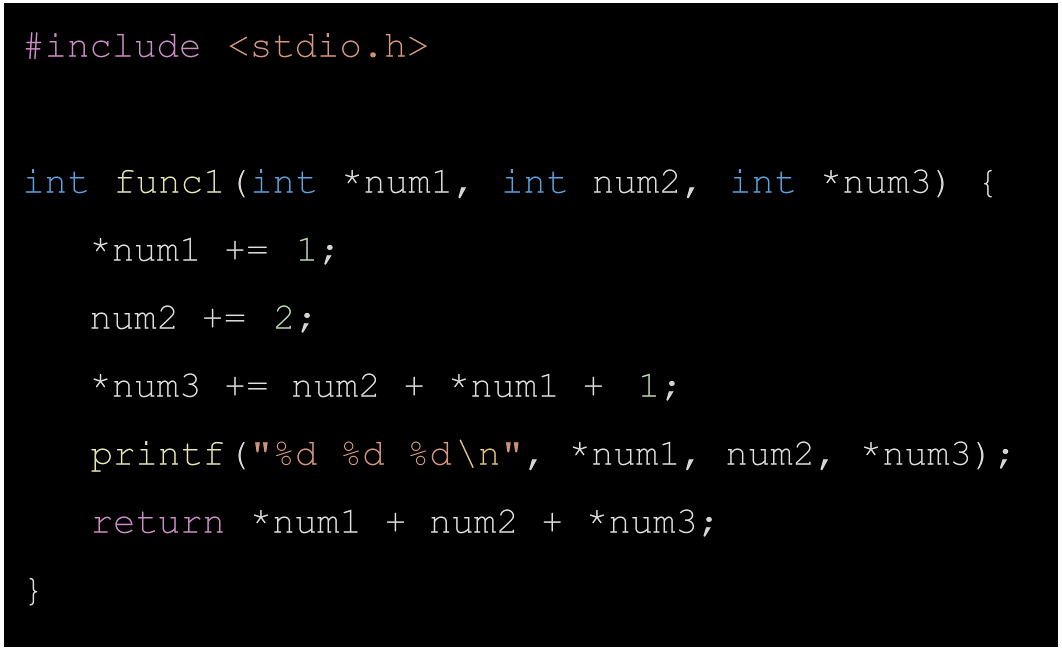
Example: Swap values

```c
void swap(int *x, int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

int main()
{
    int x, y;
    x = 5;
    y = 10;
    swap(&x, &y);
    printf("x=%d y=%d\n", x, y);
    return 0;
}
```

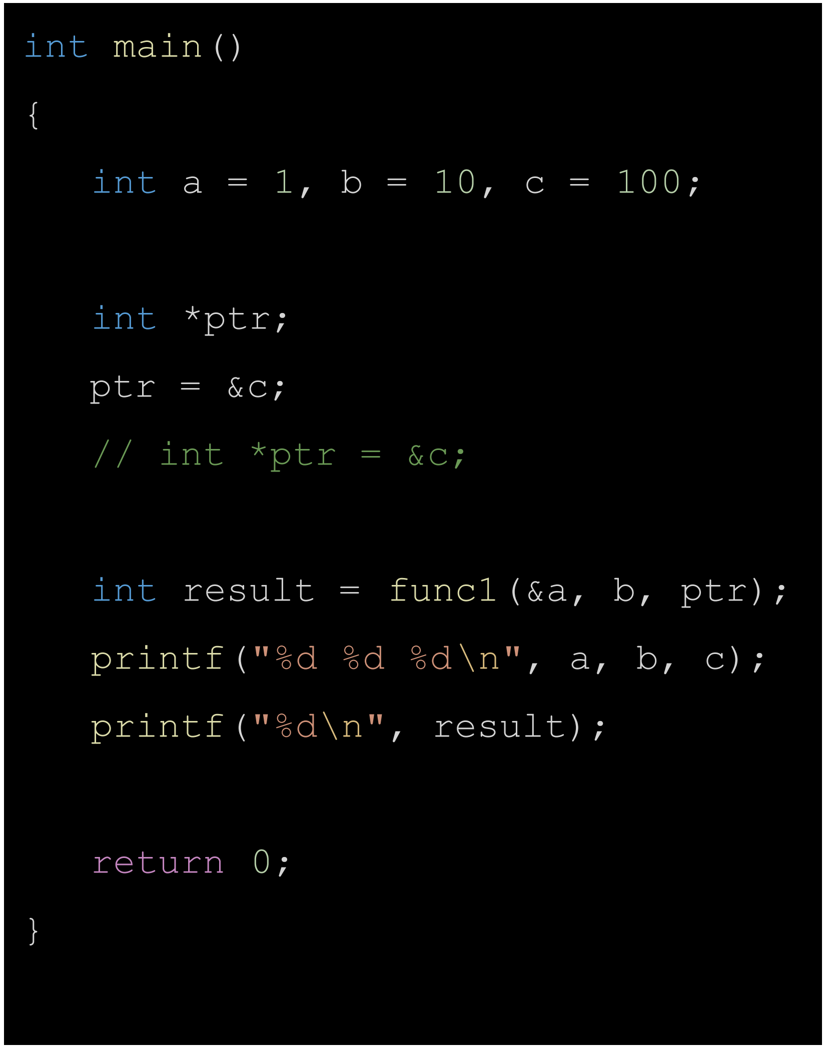
Output:

```
x=10 y=5
```

Exercise



```
#include <stdio.h>

int func1(int *num1, int num2, int *num3) {
    *num1 += 1;
    num2 += 2;
    *num3 += num2 + *num1 + 1;
    printf("%d %d %d\n", *num1, num2, *num3);
    return *num1 + num2 + *num3;
}
```



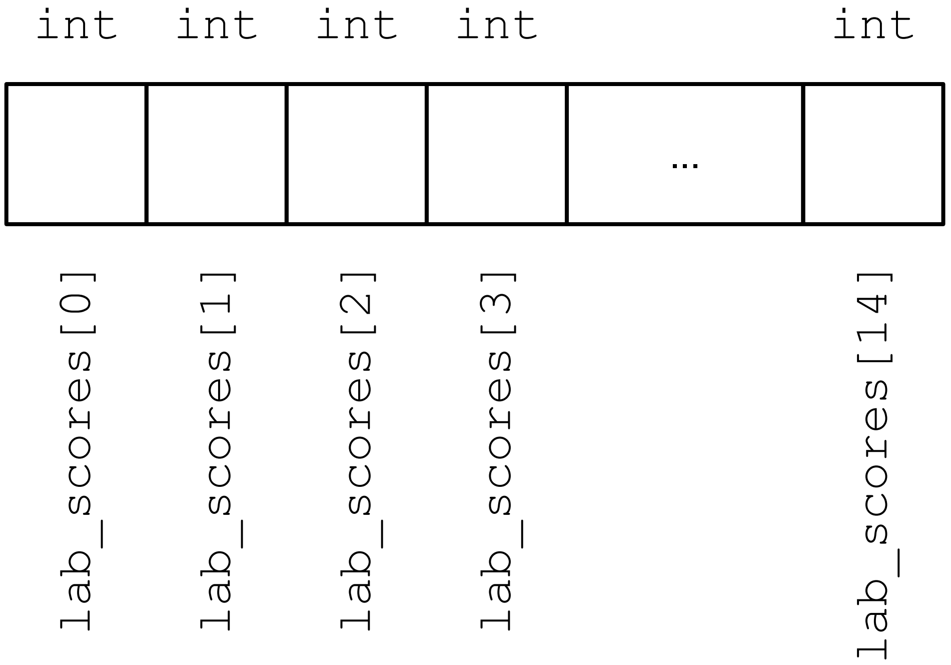
```
int main()
{
    int a = 1, b = 10, c = 100;

    int *ptr;
    ptr = &c;
    // int *ptr = &c;

    int result = func1(&a, b, ptr);
    printf("%d %d %d\n", a, b, c);
    printf("%d\n", result);

    return 0;
}
```

Using Pointer with Array

Recap: Array

Array: a list of values of the same data type that is stored using a single group name.

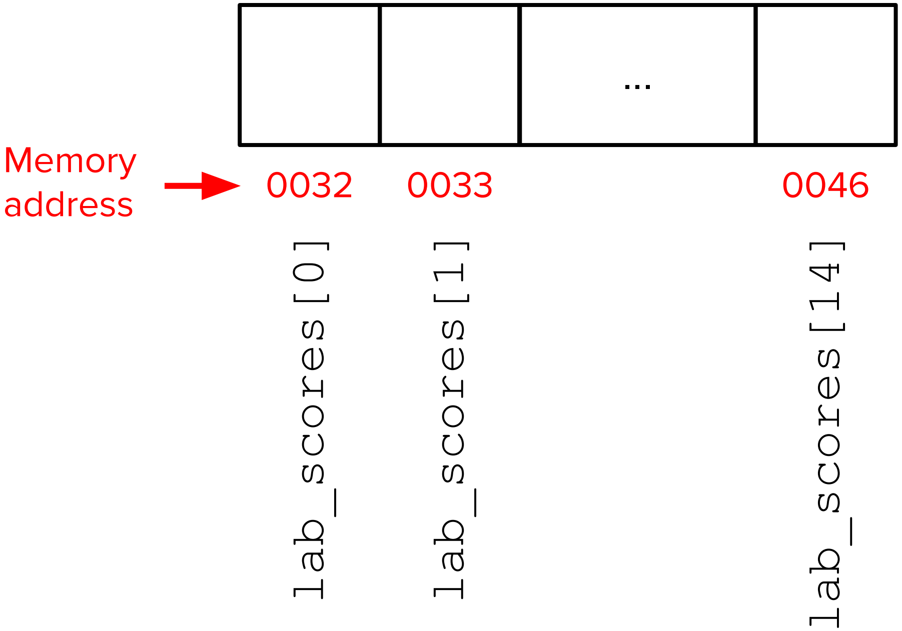
```
int lab_scores[15];
```



Access Array Elements with Pointer

By making a pointer points to the first element of the array, we can use **"pointer + offset"** to access each element in the array.

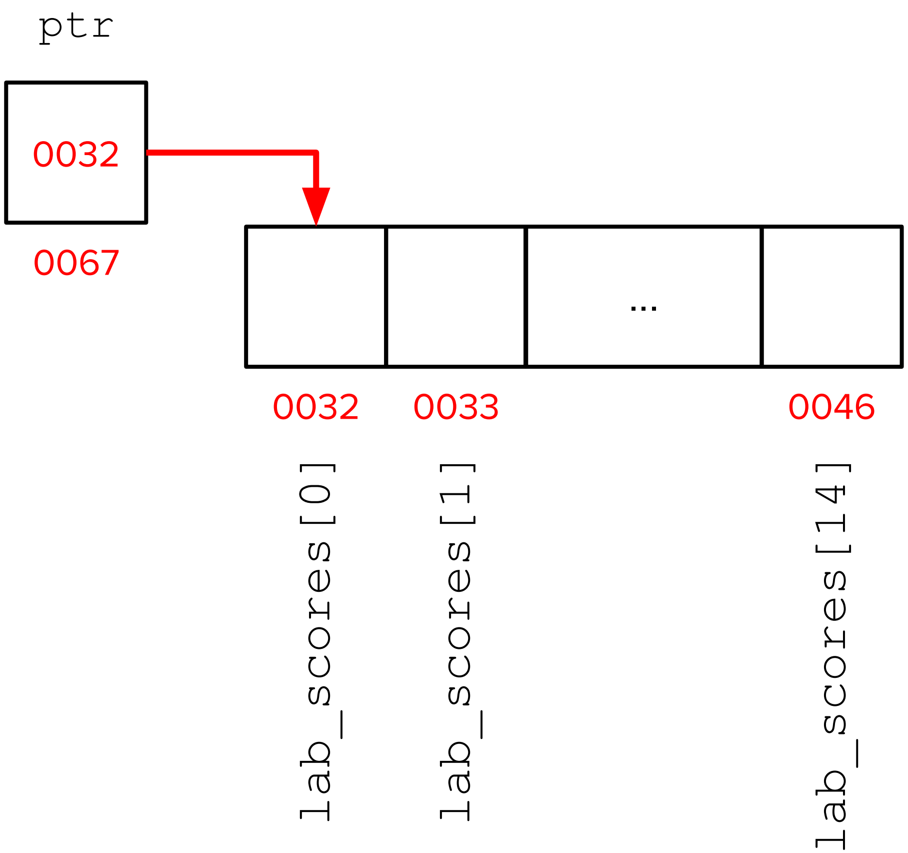
```
int lab_scores[15] = {...};
```



Access Array Elements with Pointer

By making a pointer points to the first element of the array, we can use **“pointer + offset”** to access each element in the array.

```
int lab_scores[15] = {...};

// Point to the 1st element
int *ptr;
ptr = &lab_scores[0];
```

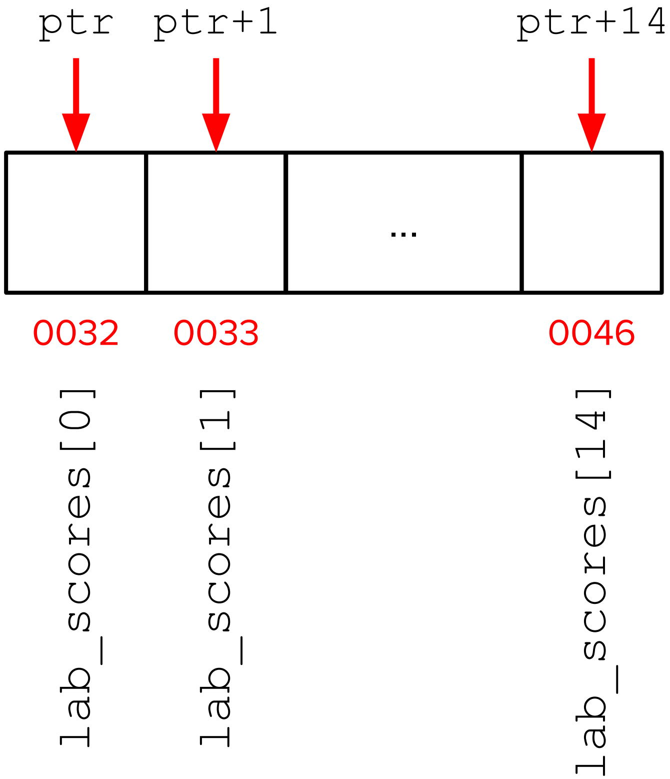


Access Array Elements with Pointer

By making a pointer points to the first element of the array, we can use **“pointer + offset”** to access each element in the array.

```
int lab_scores[15] = {...};

// Point to the 1st element
int *ptr;
ptr = &lab_scores[0];

// Access array elements
int i;
for (i=0 ; i<15 ; i++) {
    printf("%d ", *(ptr+i));
}
```

offset

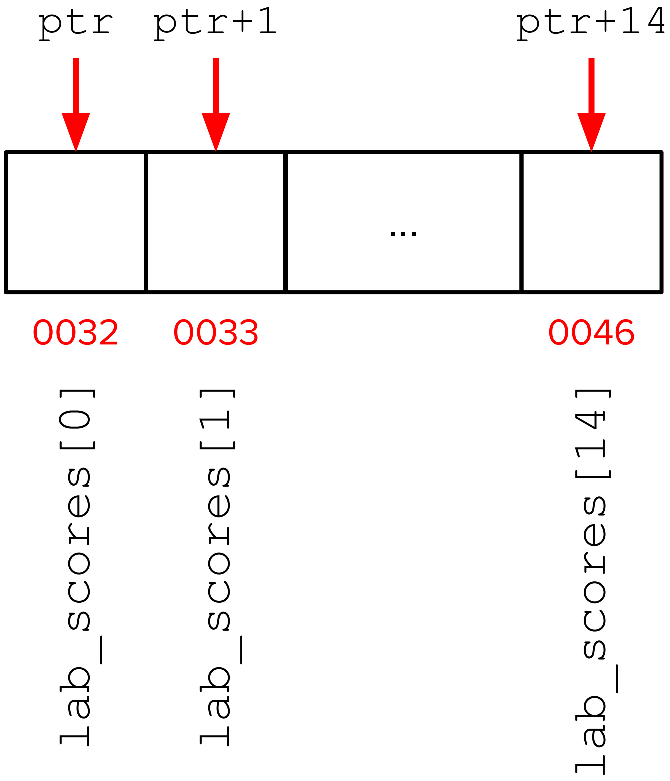


Access Array Elements with Pointer

By making a pointer points to the first element of the array, we can use **"pointer + offset"** to access each element in the array.

```
int lab_scores[15] = {...};

// Point to the 1st element
int *ptr;
ptr = &lab_scores[0];

// Access array elements
int i;
for (i=0 ; i<15 ; i++) {
    printf("%d ", ptr[i]));
}
```

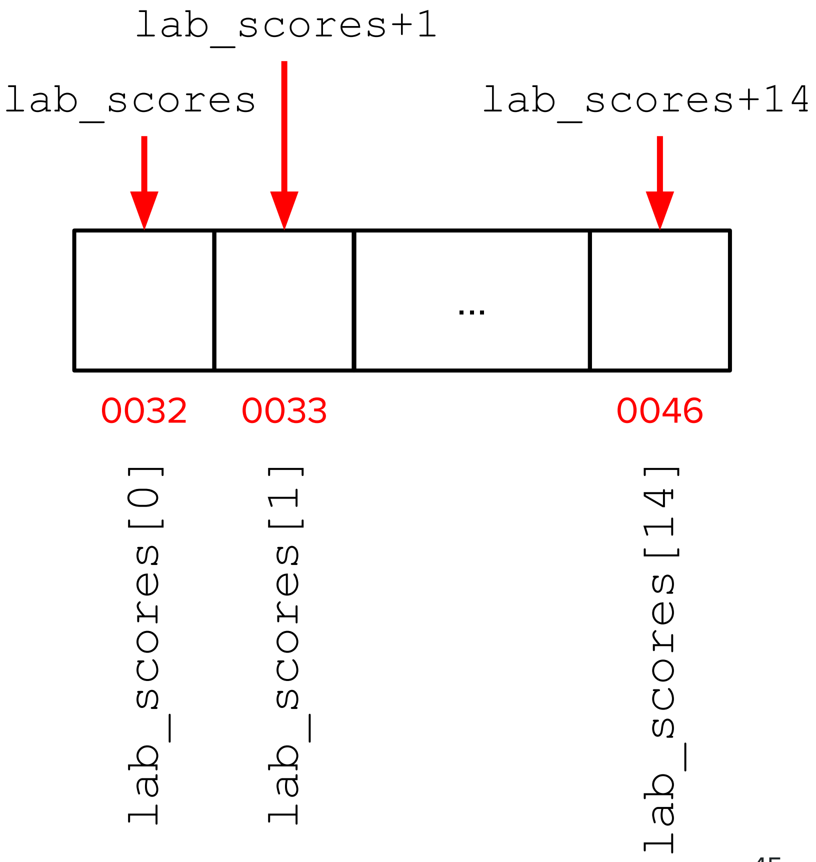
offset



Array Name as Pointer

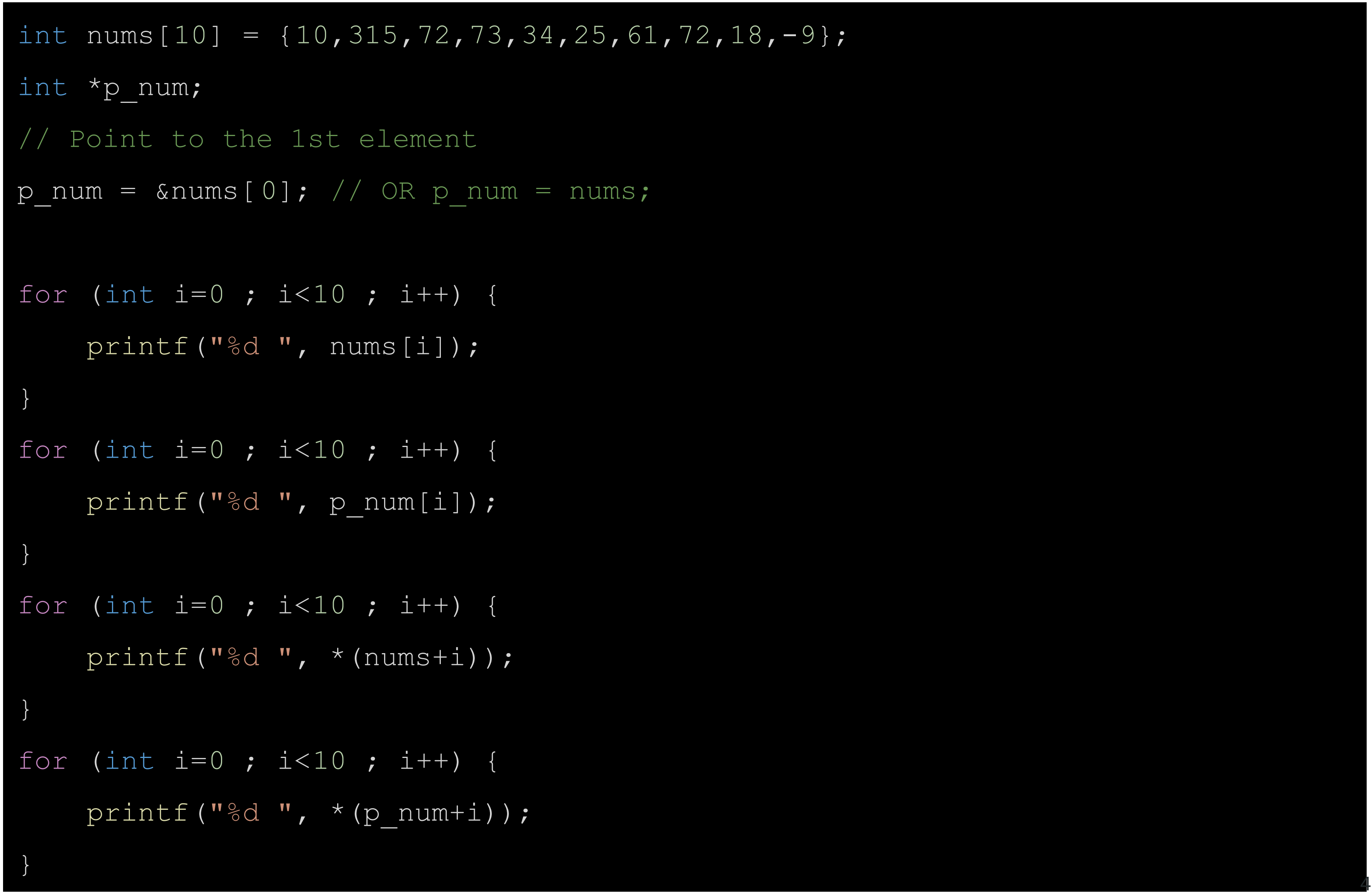
When we use the array name without the index, the name is converted to a pointer to its first element.

```
int lab_scores[15] = {...};

// Access array elements
int i;
for (i=0 ; i<15 ; i++) {
    printf("%d ", *(lab_scores+i));
}
```



Here, `lab_scores` is converted to a pointer to its first element, which is then added with the offset to access each array element.

Example



```c
int nums[10] = {10,315,72,73,34,25,61,72,18,-9};
int *p_num;
// Point to the 1st element
p_num = &nums[0]; // OR p_num = nums;

for (int i=0 ; i<10 ; i++) {
    printf("%d ", nums[i]);
}
for (int i=0 ; i<10 ; i++) {
    printf("%d ", p_num[i]);
}
for (int i=0 ; i<10 ; i++) {
    printf("%d ", *(nums+i));
}
for (int i=0 ; i<10 ; i++) {
    printf("%d ", *(p_num+i));
}
```

Example

```c
int nums[10] = {10,315,72,73,34,25,61,72,18,-9};
int *p_num;
// Point to the 1st element
p_num = &nums[0]; // OR p_num = nums;

for (int i=0 ; i<10 ; i++) {
    printf("%d ", nums[i]);
}
for (int i=0 ; i<10 ; i++) {
    printf("%d ", p_num[i]);
}
for (int i=0 ; i<10 ; i++) {
    printf("%d ", *(nums+i));
}
for (int i=0 ; i<10 ; i++) {
    printf("%d ", *(p_num+i));
}
```

Output:

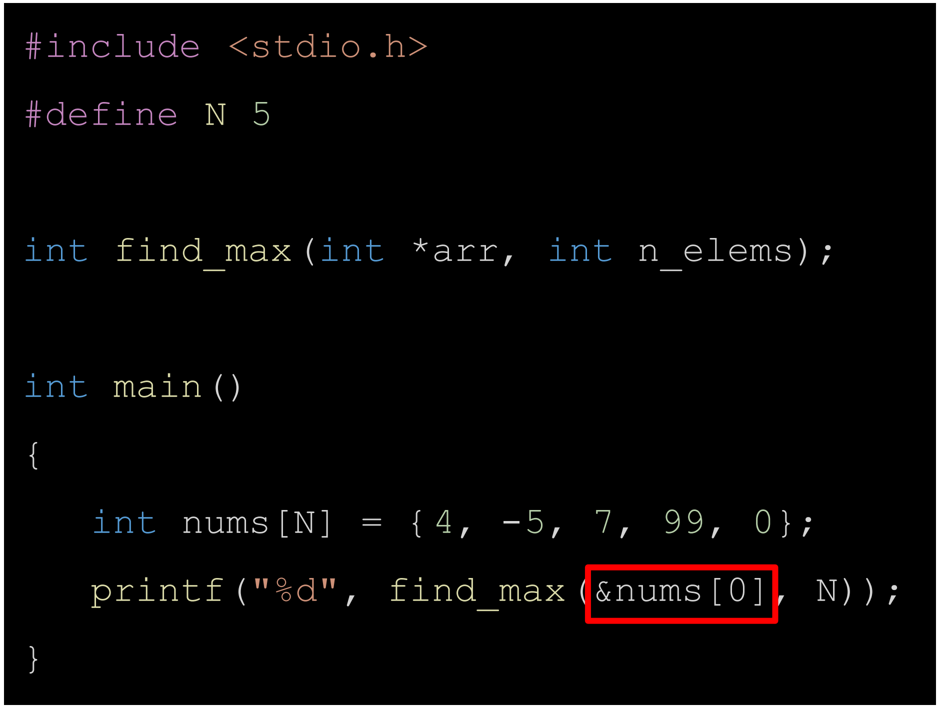
```
10 315 72 73 34 25 61 72 18 -9
10 315 72 73 34 25 61 72 18 -9
10 315 72 73 34 25 61 72 18 -9
10 315 72 73 34 25 61 72 18 -9
```

Exercise

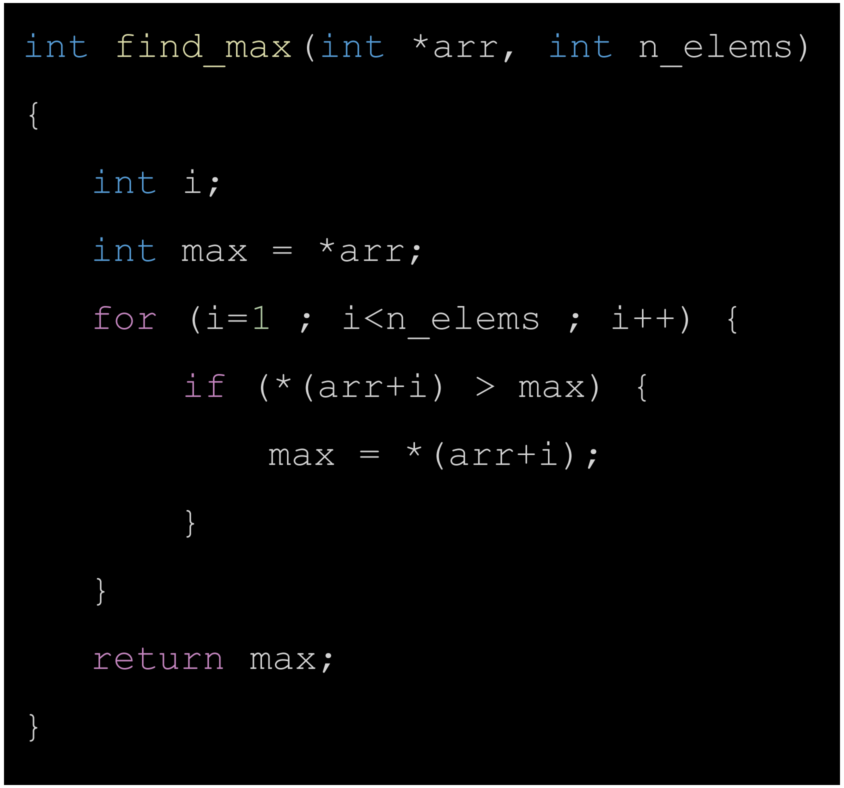
```c
#include <stdio.h>
#define N 5

int main()
{
    int nums[N] = {-4, 15, 91, 34, 0};
    int *ptr_1 = &nums[0];
    int *ptr_2 = &nums[4];
    int *ptr_3 = nums;
    printf("%d\n", *ptr_1);
    printf("%d\n", *(ptr_1+3));
    printf("%d\n", ptr_1[1]);
    printf("%d\n", nums[1]);
    printf("%d\n", *nums);
    printf("%d\n", *ptr_3);
    printf("%d\n", *ptr_2);
    printf("%d\n", *(ptr_2-2));
    printf("%d\n", *(ptr_2+1));
    return 0;
}
```

Pass Array to Function

We can use **“pass by reference”** to pass an array to a function.



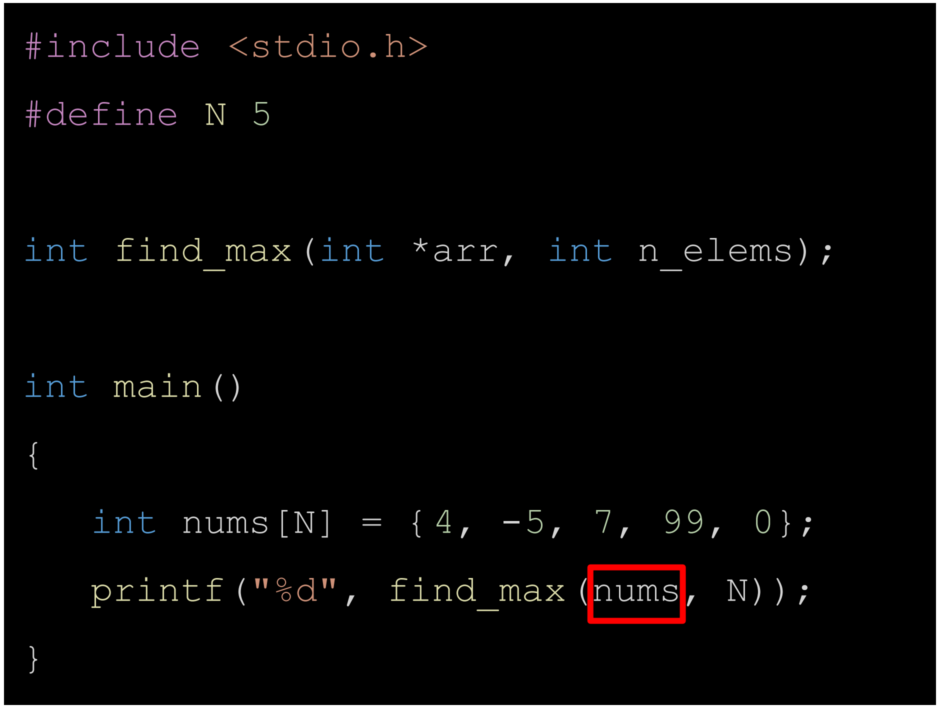
```
#include <stdio.h>
#define N 5

int find_max(int *arr, int n_elems);

int main()
{
    int nums[N] = {4, -5, 7, 99, 0};
    printf("%d", find_max(&nums[0], N));
}
```



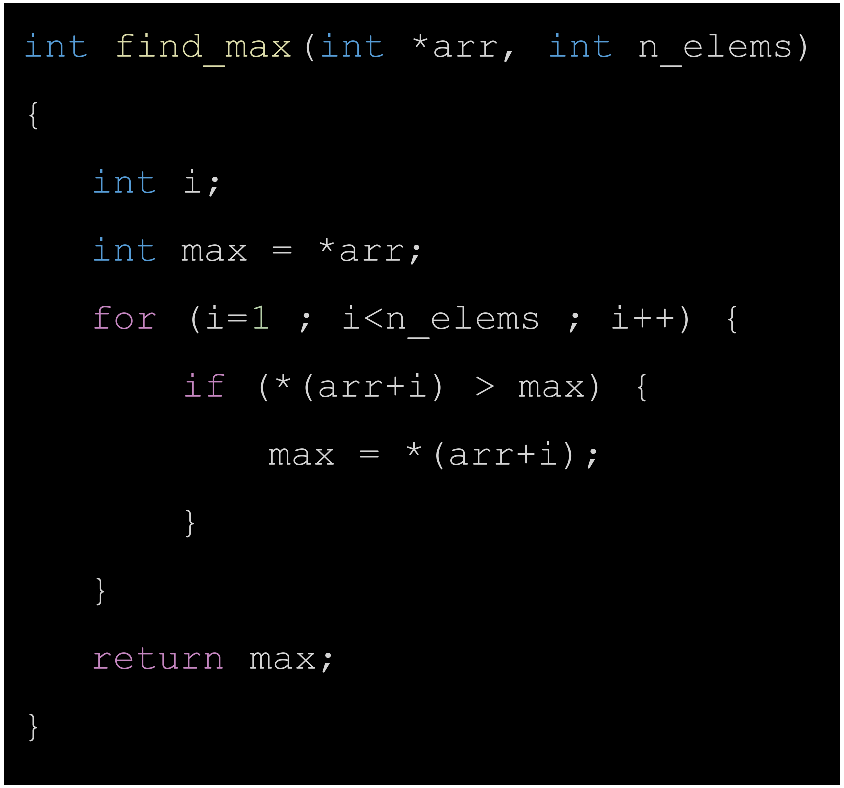
```
int find_max(int *arr, int n_elems)
{
    int i;
    int max = *arr;
    for (i=1 ; i<n_elems ; i++) {
        if (*(arr+i) > max) {
            max = *(arr+i);
        }
    }
    return max;
}
```

Pass Array to Function

We can use **“pass by reference”** to pass an array to a function.



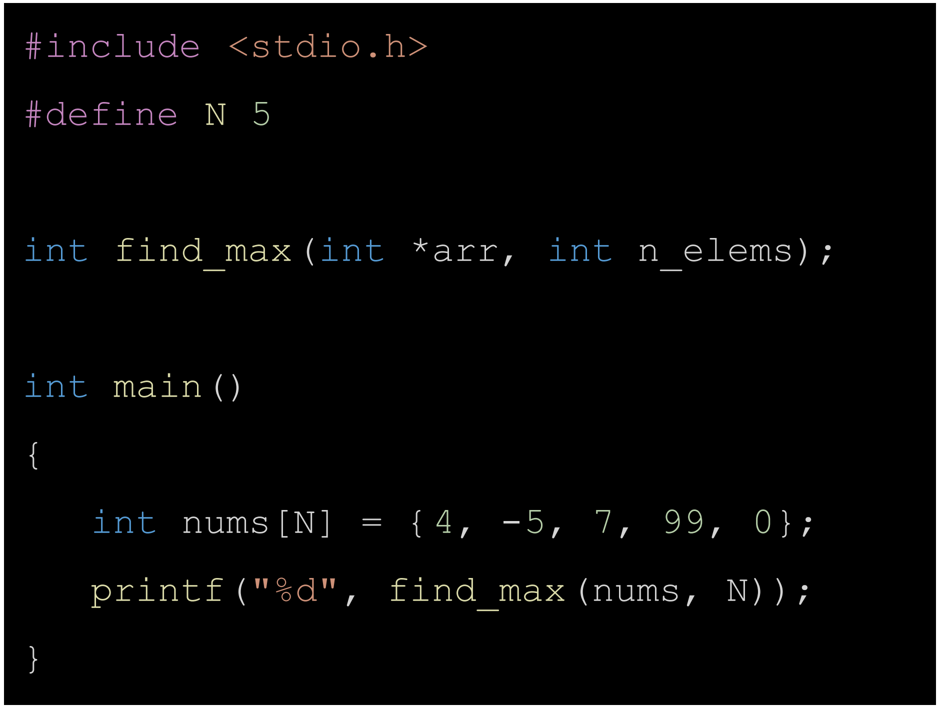
```
#include <stdio.h>
#define N 5

int find_max(int *arr, int n_elems);

int main()
{
    int nums[N] = {4, -5, 7, 99, 0};
    printf("%d", find_max(nums, N));
}
```



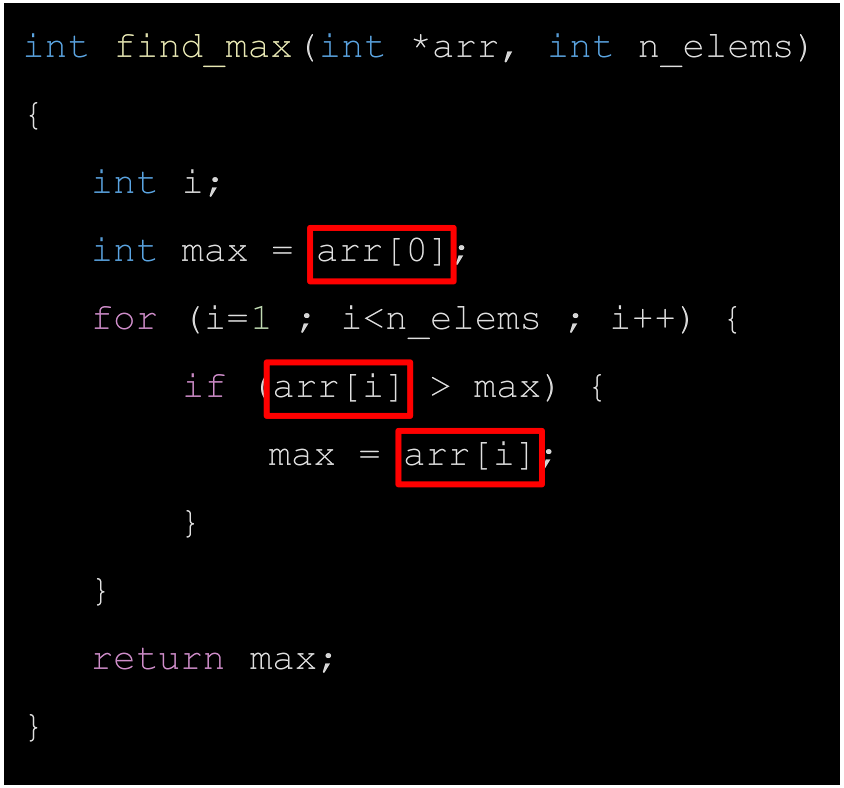
```
int find_max(int *arr, int n_elems)
{
    int i;
    int max = *arr;
    for (i=1 ; i<n_elems ; i++) {
        if (*(arr+i) > max) {
            max = *(arr+i);
        }
    }
    return max;
}
```

Pass Array to Function

We can use **“pass by reference”** to pass an array to a function.



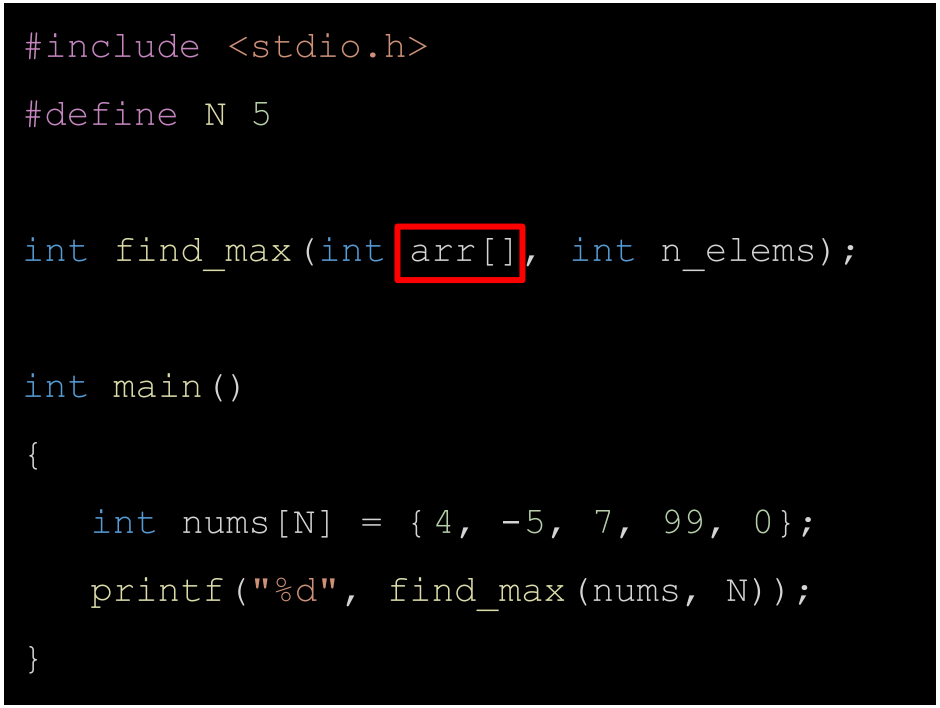
```c
#include <stdio.h>
#define N 5

int find_max(int *arr, int n_elems);

int main()
{
    int nums[N] = {4, -5, 7, 99, 0};
    printf("%d", find_max(nums, N));
}
```



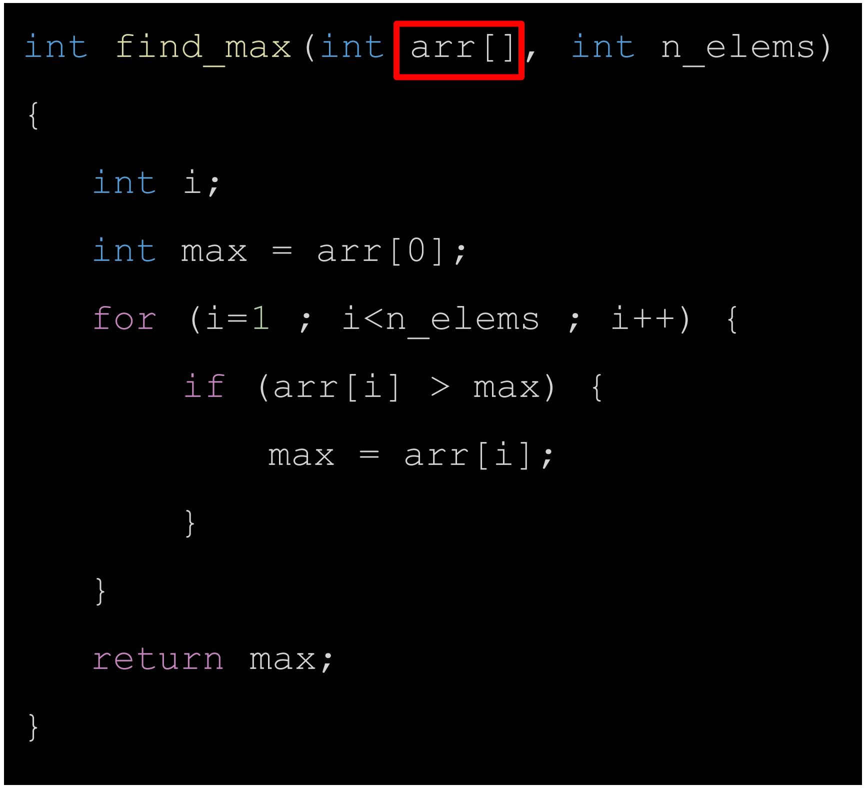
```c
int find_max(int *arr, int n_elems)
{
    int i;
    int max = arr[0];
    for (i=1 ; i<n_elems ; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}
```

Pass Array to Function

We can use **“pass by reference”** to pass an array to a function.



```c
#include <stdio.h>
#define N 5

int find_max(int arr[], int n_elems);

int main()
{
    int nums[N] = {4, -5, 7, 99, 0};
    printf("%d", find_max(nums, N));
}
```



```c
int find_max(int arr[], int n_elems)
{
    int i;
    int max = arr[0];
    for (i=1 ; i<n_elems ; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}
```

Exercise

```
#include <stdio.h>

// Function prototype

int main() {
    int n = 5;
    int arr[n];
    int i;
    for (i=0 ; i<n ; i++) {
        arr[i] = i*i;
    }
    // Call function to compute the sum
    return 0;
}

// Function definition
```

Exercise

What is the output?

```c
#include <stdio.h>

void func1(int *arr, int n);

int main()
{
    int arr[5] = {-5, 3, 4, 1, 8};
    func1(arr, 5);
    int i;
    for (i=0 ; i<5 ; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

```c
void func1(int *arr, int n)
{
    int i;
    for (i=0 ; i<n ; i++) {
        arr[i] = arr[i] * arr[i];
    }
}
```

Let’s move to the lab