Arrays and Pointers

2020 Spring: Introduction to C

July 8th, 2020

Today

- Arrays in C
- Strings in C
- Arrays and Functions
- Pointers
- & and * Operators
- Call by Value
- Pointers as Function Parameters

Programming Problem

- **#4344 평균은 넘겠지**

- **For simplicity, only consider the problem for a single test case.**

  - **Input**

```
5
50
50
70
80
100
```

  - **Output**

```
40.000%
```

- **Try to solve this problem!**

Why is it hard?

- **We need each input value twice**
  - To compute the average
  - To count how many were above average

- **We could read each value into a variable, but we**
  - don't know how many values are needed until the program runs
  - don't know how many variables to declare

- **Need a way to declare many variables in one step.**

Arrays

- **array: A data structure that stores many values of the same type**
  - *element*: One value in an array
  - *index*: A 0-based integer to access an element from an array

index	0	1	2	3	4	5	6	7	8
value	12	-1	49	0	5	7	-19	128	1

- **Declaration**
  - `type name[length];`

- **Example**

```
int arr[10];
```

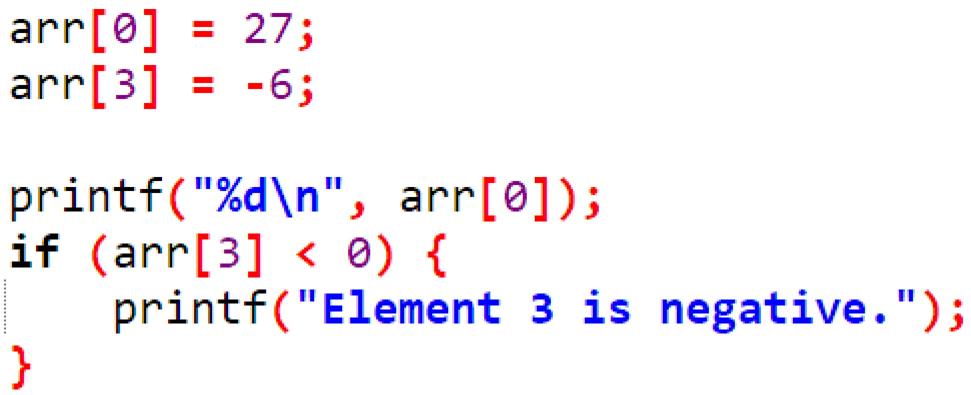
index	0	1	2	3	4	5	6	7	8	9
value	?	?	?	?	?	?	?	?	?	?

Arrays

- **Accessing elements**
  - Can be used like a variable

```
name[index]                // access
name[index] = value;       // modify
```

- **Example**

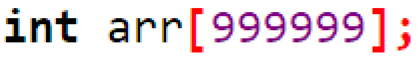


```
arr[0] = 27;
arr[3] = -6;

printf("%d\n", arr[0]);
if (arr[3] < 0) {
    printf("Element 3 is negative.");
}
```

index	0	1	2	3	4	5	6	7	8	9
value	27	?	?	-6	?	?	?	?	?	?

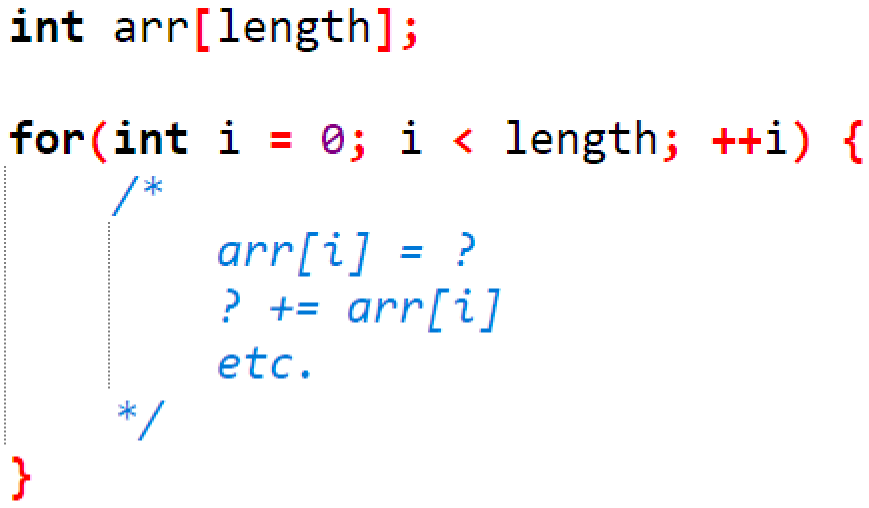
Benefits of using Arrays

- Can declare multiple variables at once



```c
int arr[999999];
```

- Can apply similar pattern to each of the elements



```c
int arr[length];

for(int i = 0; i < length; ++i) {
    /*
        arr[i] = ?
        ? += arr[i]
        etc.
    */
}
```

Array Initialization

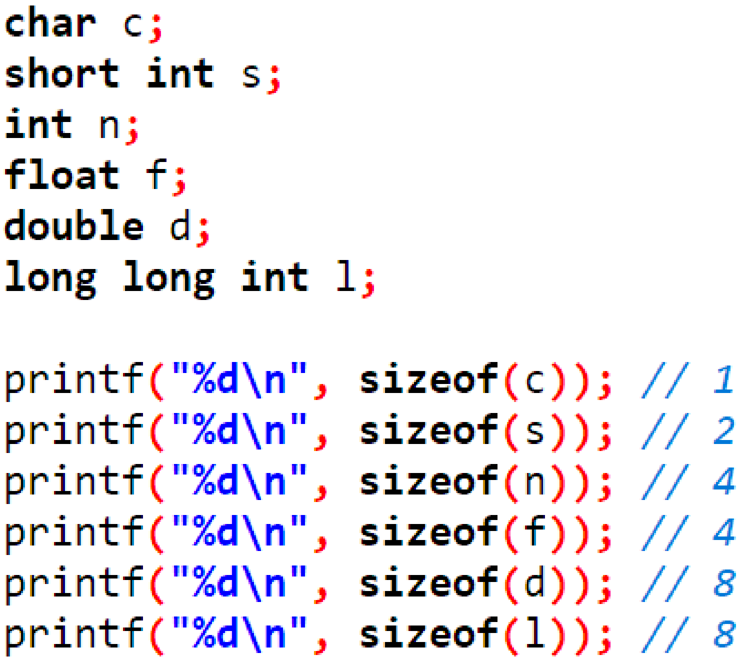
- **Arrays declared *locally* might contain random values**
  - Initialization is necessary before accessing the array
  - But repeating `name[index] = value` is troublesome

- **Initializer list method**
  - `int arr1[5] = {1, 2, 3, 4, 5};`
    - Initializes the array to the given values
  - `int arr2[] = {1, 2, 3, 4};`
    - If the length is unspecified, the length is checked automatically
  - `int arr3[5] = {1, 2, 3};`
    - Uninitialized elements will be given *zero-equivalent* values
  - `int zeros[length] = {0}` will zero-initialize the whole array

- ***Global arrays* will be automatically initialized to zero-equivalent values**

sizeof() Function

- sizeof(x) returns the size of the variable x



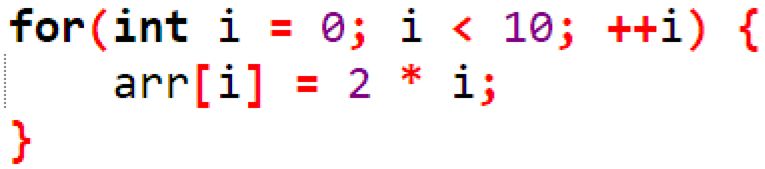
```
char c;
short int s;
int n;
float f;
double d;
long long int l;

printf("%d\n", sizeof(c)); // 1
printf("%d\n", sizeof(s)); // 2
printf("%d\n", sizeof(n)); // 4
printf("%d\n", sizeof(f)); // 4
printf("%d\n", sizeof(d)); // 8
printf("%d\n", sizeof(l)); // 8
```

Machine dependent!

- Calculation of array length
  - `type arr[length];`
  - `int len = sizeof(arr) / sizeof(type);`

Arrays and for Loops

- **It is common to use for loops to access array elements**
  - The loop counter is used as an index
  - We can also assign each element a value in a loop

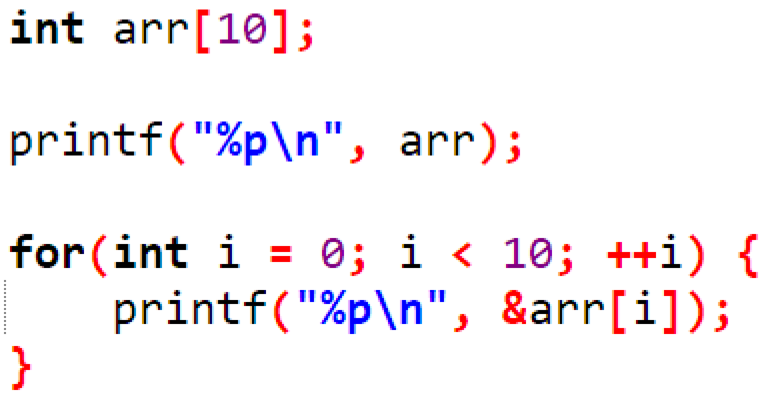


```c
for(int i = 0; i < 10; ++i) {
    arr[i] = 2 * i;
}
```

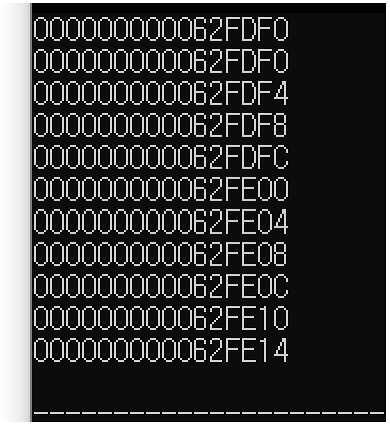
  - *Will be used very often!*

Element Access

- `type arr[length];`
- **The symbol `arr` contains the *starting address of the array***
- **The address of element at index i is calculated as follows**
  - `arr + i * sizeof(type)`



```
int arr[10];

printf("%p\n", arr);

for(int i = 0; i < 10; ++i) {
    printf("%p\n", &arr[i]);
}
```



```
000000000062FDF0
000000000062FDF0
000000000062FDF4
000000000062FDF8
000000000062FDFC
000000000062FE00
000000000062FE04
000000000062FE08
000000000062FE0C
000000000062FE10
000000000062FE14

--------------------
```

- **C does not do index bound checking**
  - Can access elements that are out of bounds, but may cause segmentation faults during runtime

Comparing and Copying Arrays

- **Comparing arrays cannot be done with ==**
  - `int arr1[10], arr2[10];`
  - `arr1 == arr2` will compare the *starting address* of each array
  - Will probably be different
  - Must compare each element one by one

- **Cannot copy arrays with = (assignment operator)**
  - `int arr1[3] = {1, 2, 3}, arr2[3];`
  - `arr2 = arr1;     // error`
  - Must copy each element one by one
  - `memcpy` function

Exercise

- **#4344 평균은 넘겠지**
  - Use an integer array to store all the scores
- **#10818 최소, 최대**
  - Use an integer array to store all the values
  - Traverse the array to find the maximum/minimum
- **#2562 최댓값**
- **#2577 숫자의 개수 (★)**

C Strings

- ***A string in C is an array of characters!***
- **Always ends with the null character '\0'**
  - Used as an 'end of string' delimiter

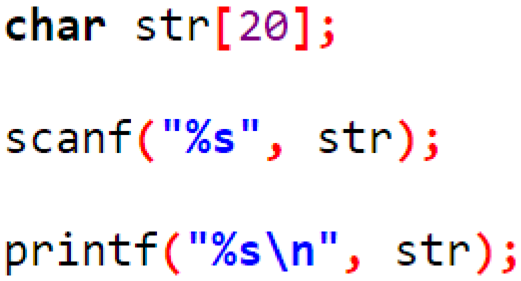
- **Declaration**
  - `char str[length + 1];` // + 1 for the null character

- **Example**
  - `char str[10] = "Hello, C!";`

index	0	1	2	3	4	5	6	7	8	9
value	H	e	l	l	o	,		C	!	\0

`scanf`, `printf` with C Strings

- **char array *must be longer than the maximum input length***
- **Uses %s as format specifier**

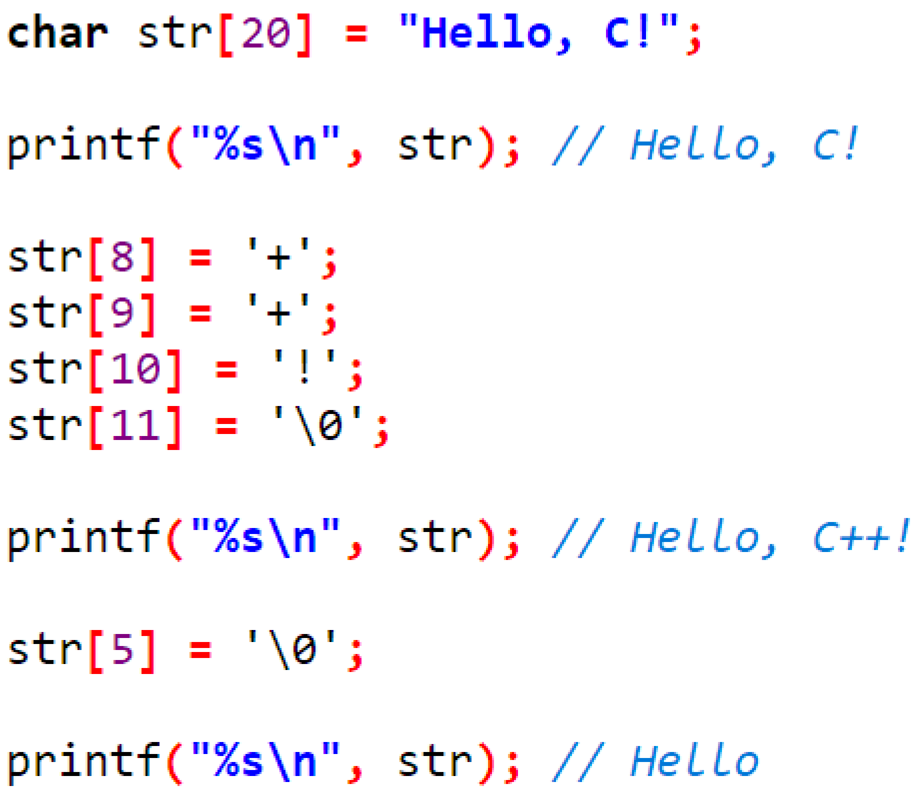


```
char str[20];

scanf("%s", str);

printf("%s\n", str);
```

- **Symbol `str` contains the *starting address* of the string**
- **`scanf` will**
  - Store each character starting from address `str`
  - If input is finished, automatically pad '\0' at the end
- **`printf` will**
  - Print each character starting from address `str`
  - Print until '\0' (end of string) is found

Modifying Strings

- Can access or modify each character of a string, just like an array

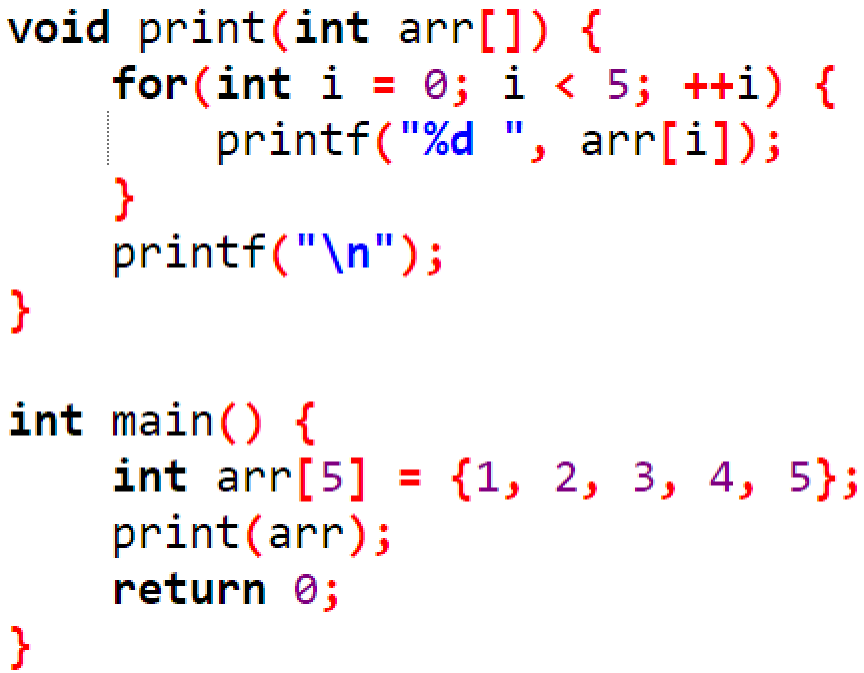


```
char str[20] = "Hello, C!";

printf("%s\n", str); // Hello, C!

str[8] = '+';
str[9] = '+';
str[10] = '!';
str[11] = '\0';

printf("%s\n", str); // Hello, C++!

str[5] = '\0';

printf("%s\n", str); // Hello
```

Passing an Array as Function Parameter

- **Function declaration**
  - `type func(`**`type arr`**`[])`

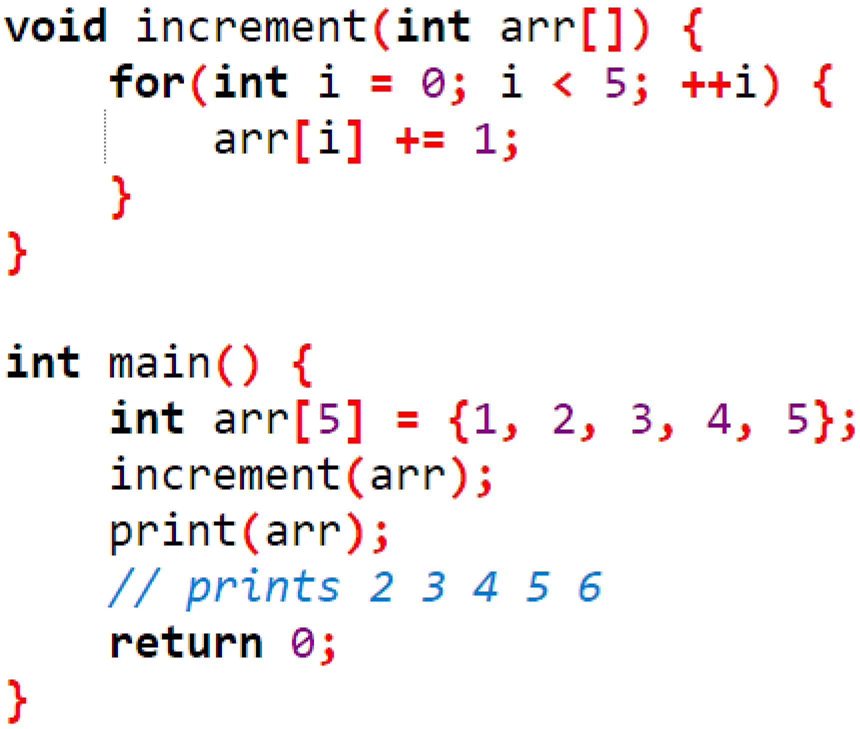
- **Example**



```c
void print(int arr[]) {
    for(int i = 0; i < 5; ++i) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    print(arr);
    return 0;
}
```

Passing an Array as Function Parameter

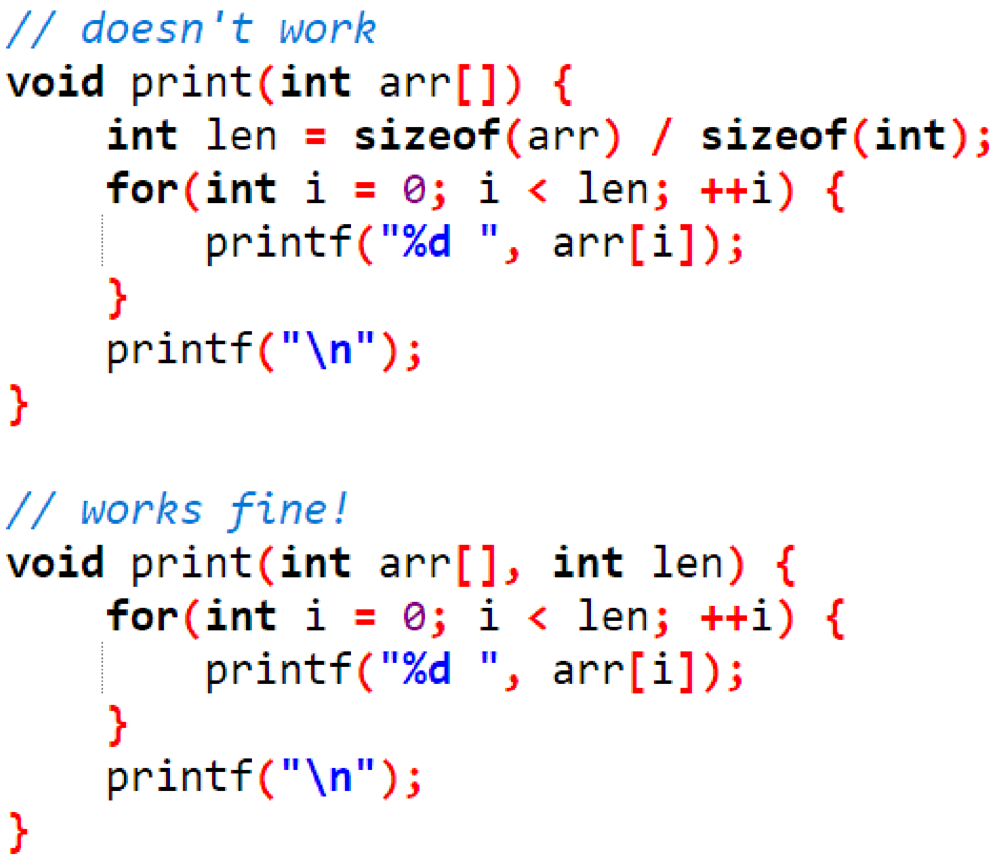
- **Be careful! The function may change the elements of the array**



```
void increment(int arr[]) {
    for(int i = 0; i < 5; ++i) {
        arr[i] += 1;
    }
}

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    increment(arr);
    print(arr);
    // prints 2 3 4 5 6
    return 0;
}
```

Array Length as Parameter

- **Declaration *doesn't specify the length* of array**
  - `sizeof(arr) / sizeof(type)` does not work here
  - Must pass the length as a second parameter



```c
// doesn't work
void print(int arr[]) {
    int len = sizeof(arr) / sizeof(int);
    for(int i = 0; i < len; ++i) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// works fine!
void print(int arr[], int len) {
    for(int i = 0; i < len; ++i) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

Pointers

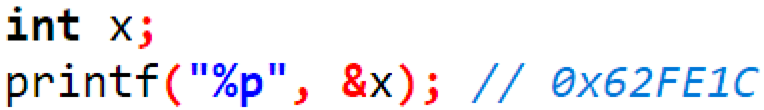
- **pointer: a variable that contains a memory address, and information about that memory address**

- **Declaration**
  - **`type`** `* ptr; // type* ptr, type *ptr is also OK`
  - Meaning: `ptr` is a pointer to a variable of type **`type`**

- **Think of `type*` as a new data type that contains addresses**

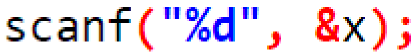
& and * Operators

- **&x gives the memory address of the variable x**



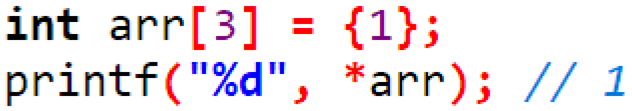
```
int x;
printf("%p", &x); // 0x62FE1C
```

  - Scan the integer and store it to location &x



```
scanf("%d", &x);
```

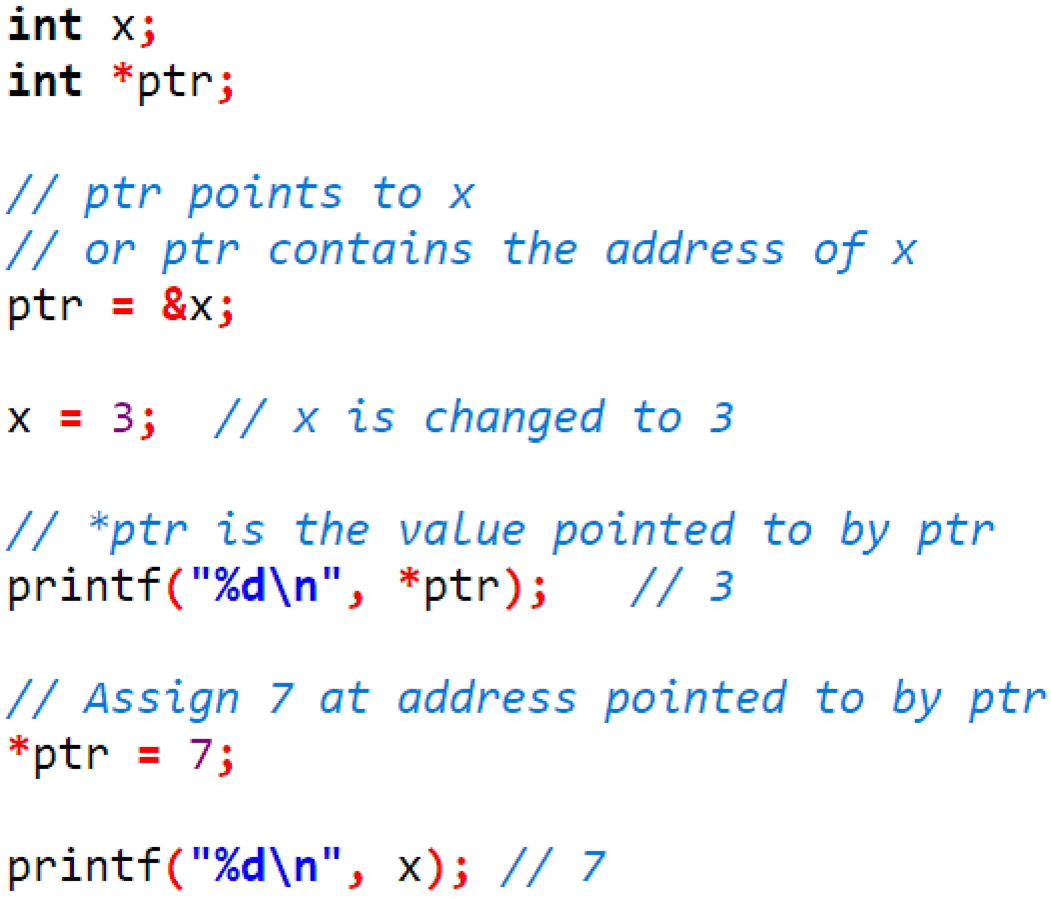
- ***x gives the value pointed to by x**



```
int arr[3] = {1};
printf("%d", *arr); // 1
```

& and * Operators

- **Example: pointer ptr points to x**



```c
int x;
int *ptr;

// ptr points to x
// or ptr contains the address of x
ptr = &x;

x = 3;  // x is changed to 3

// *ptr is the value pointed to by ptr
printf("%d\n", *ptr);   // 3

// Assign 7 at address pointed to by ptr
*ptr = 7;

printf("%d\n", x); // 7
```

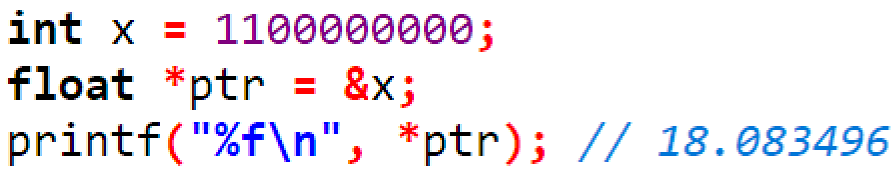
Necessity of Pointers

- **Pointers enable memory references**
  - Pointers can be used for managing arrays and strings
  - Pointers can be used as *writable function parameters*

- **Addresses are always *integers*, why do we need pointers? We could just store the address to an integer variable.**
  - Integer variable will have no information about that address
  - Pointer type contains the information about *how to interpret the data at that address*
  - `double *ptr;`
    - Data pointed to by `ptr` will be read as a `double`

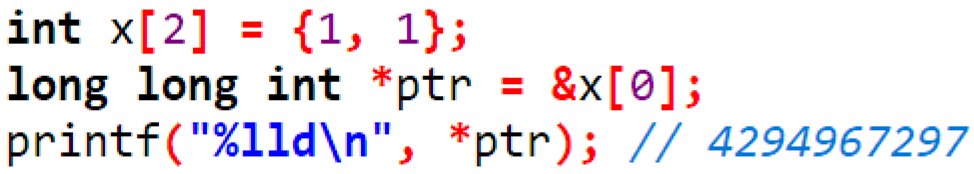
Interpreting Data in Memory

- **Reading an integer as a float**



```
int x = 1100000000;
float *ptr = &x;
printf("%f\n", *ptr); // 18.083496
```

- **Reading two integers as a long long int**

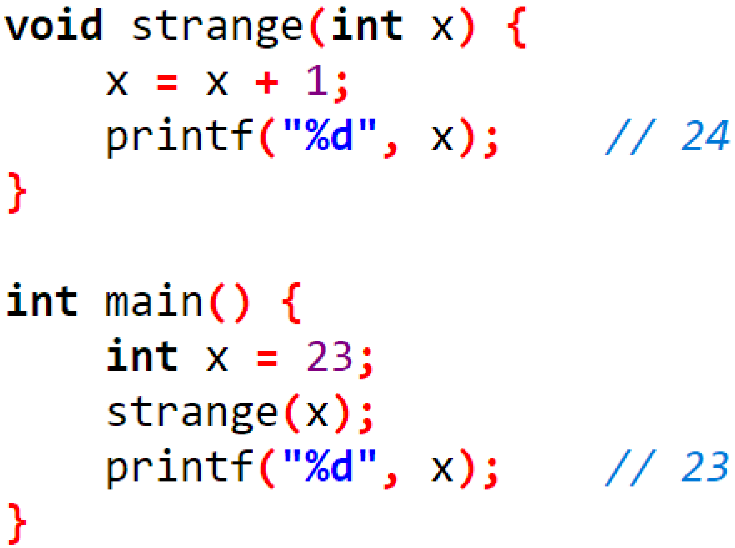


```
int x[2] = {1, 1};
long long int *ptr = &x[0];
printf("%lld\n", *ptr); // 4294967297
```

Call by Value

- **When a function is called:**
  - The value is stored into the parameter variable
  - The function's code executes, using that value (inside variable)

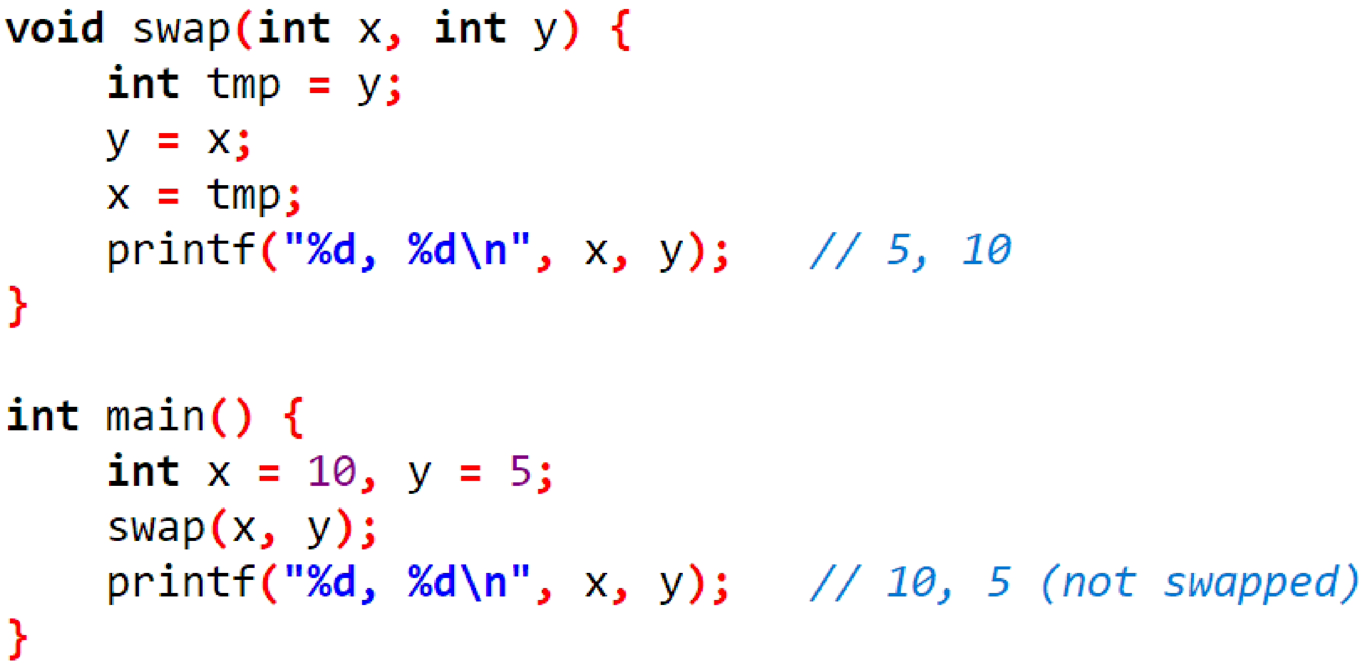
- **call by value: When values are passed as parameters, *their values are copied***
  - Modifying the parameter **will not affect the variable passed in**



```
void strange(int x) {
    x = x + 1;
    printf("%d", x);    // 24
}

int main() {
    int x = 23;
    strange(x);
    printf("%d", x);    // 23
}
```

Call by Value

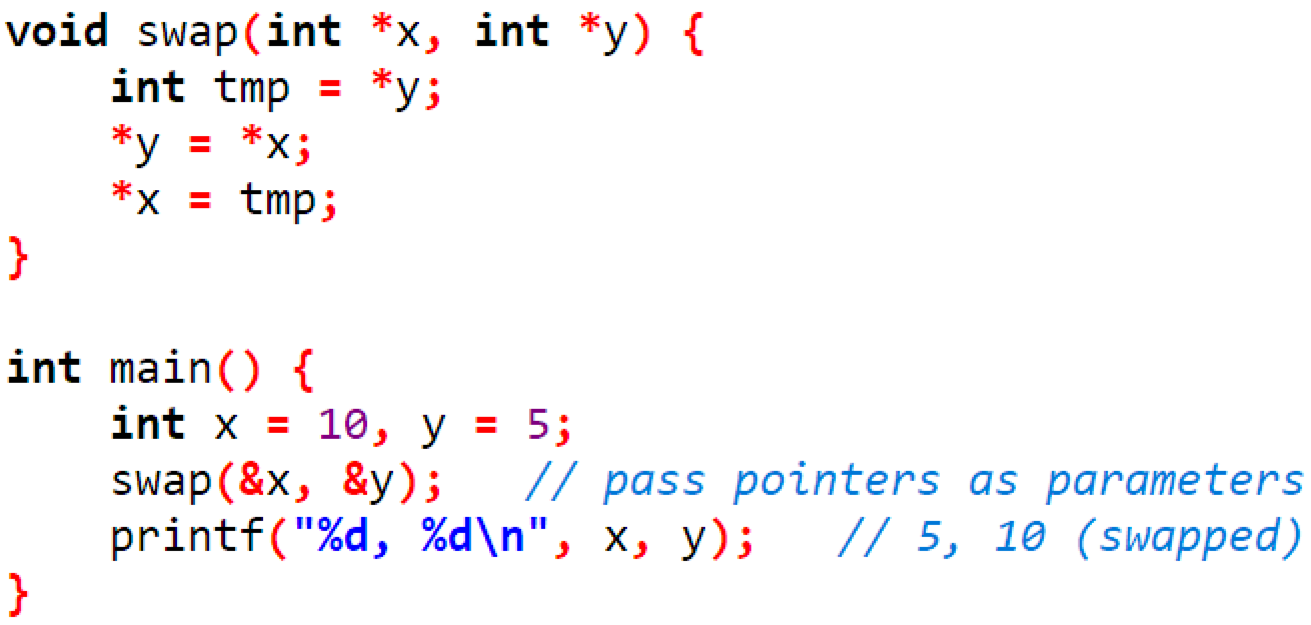
- Example



```c
void swap(int x, int y) {
    int tmp = y;
    y = x;
    x = tmp;
    printf("%d, %d\n", x, y);   // 5, 10
}

int main() {
    int x = 10, y = 5;
    swap(x, y);
    printf("%d, %d\n", x, y);   // 10, 5 (not swapped)
}
```

Swap with Pointers

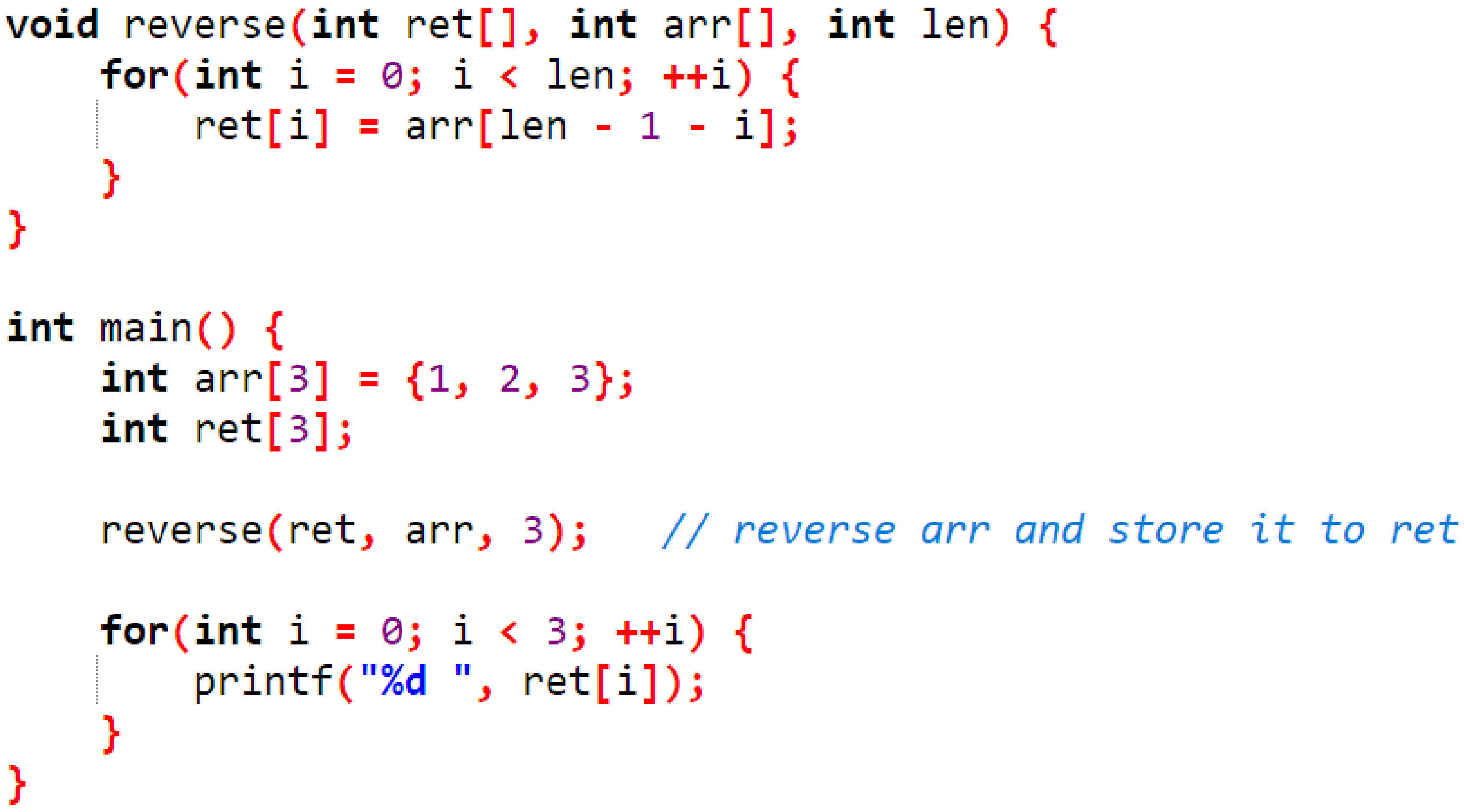
- **Working version of swap with pointers as parameters**
- ***Writable function parameter***



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

int main() {
    int x = 10, y = 5;
    swap(&x, &y);   // pass pointers as parameters
    printf("%d, %d\n", x, y);   // 5, 10 (swapped)
}
```

Writeable Function Parameter

- **Can write to the passed parameter**
  - Can be used to modify arrays



```c
void reverse(int ret[], int arr[], int len) {
    for(int i = 0; i < len; ++i) {
        ret[i] = arr[len - 1 - i];
    }
}

int main() {
    int arr[3] = {1, 2, 3};
    int ret[3];

    reverse(ret, arr, 3);   // reverse arr and store it to ret

    for(int i = 0; i < 3; ++i) {
        printf("%d ", ret[i]);
    }
}
```