COMP0204: Introduction to Programming for Robotics and AI

Arrays and Pointers

Course lead: Dr Sophia Bano

MEng Robotics and AI

UCL Computer Science

Recap (previous week)

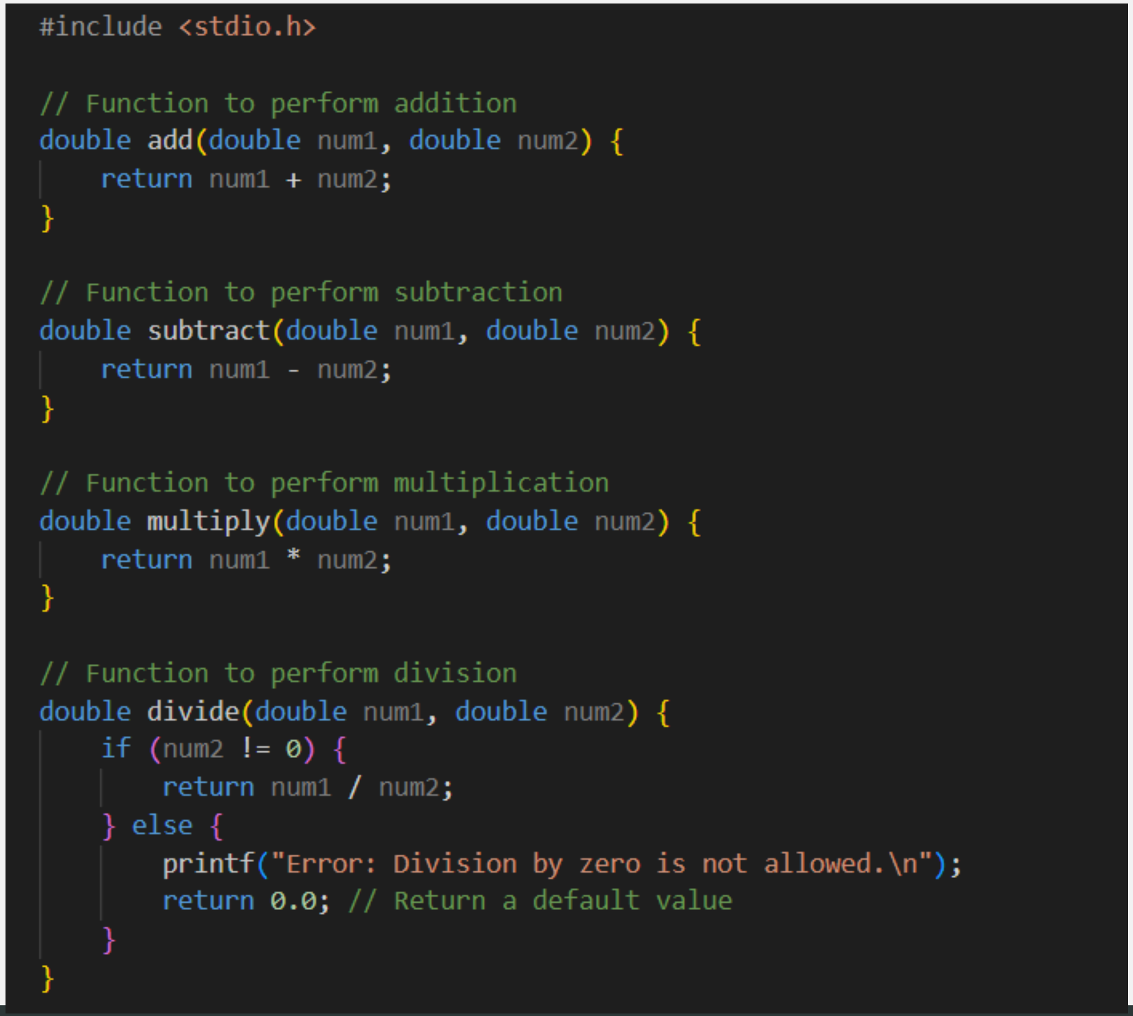
- Random numbers
- Functions
- Variable scope
- Recursion
- Arrays and String

Function – practice exercise

Exercise

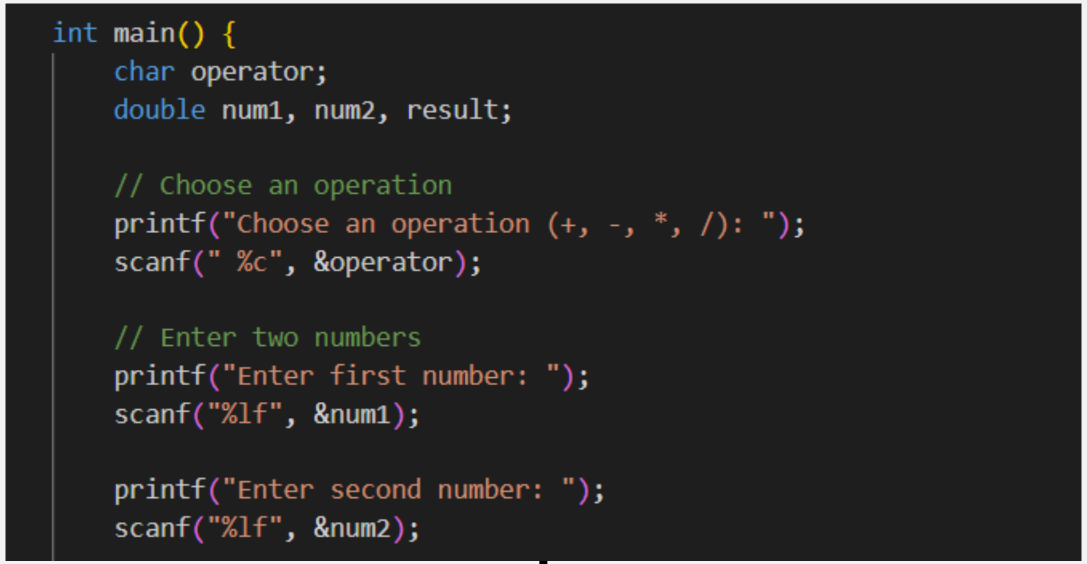
Write a simple calculator program that performs basic arithmetic operations (addition, subtraction, multiplication, and division) using custom functions.

Function – practice exercise

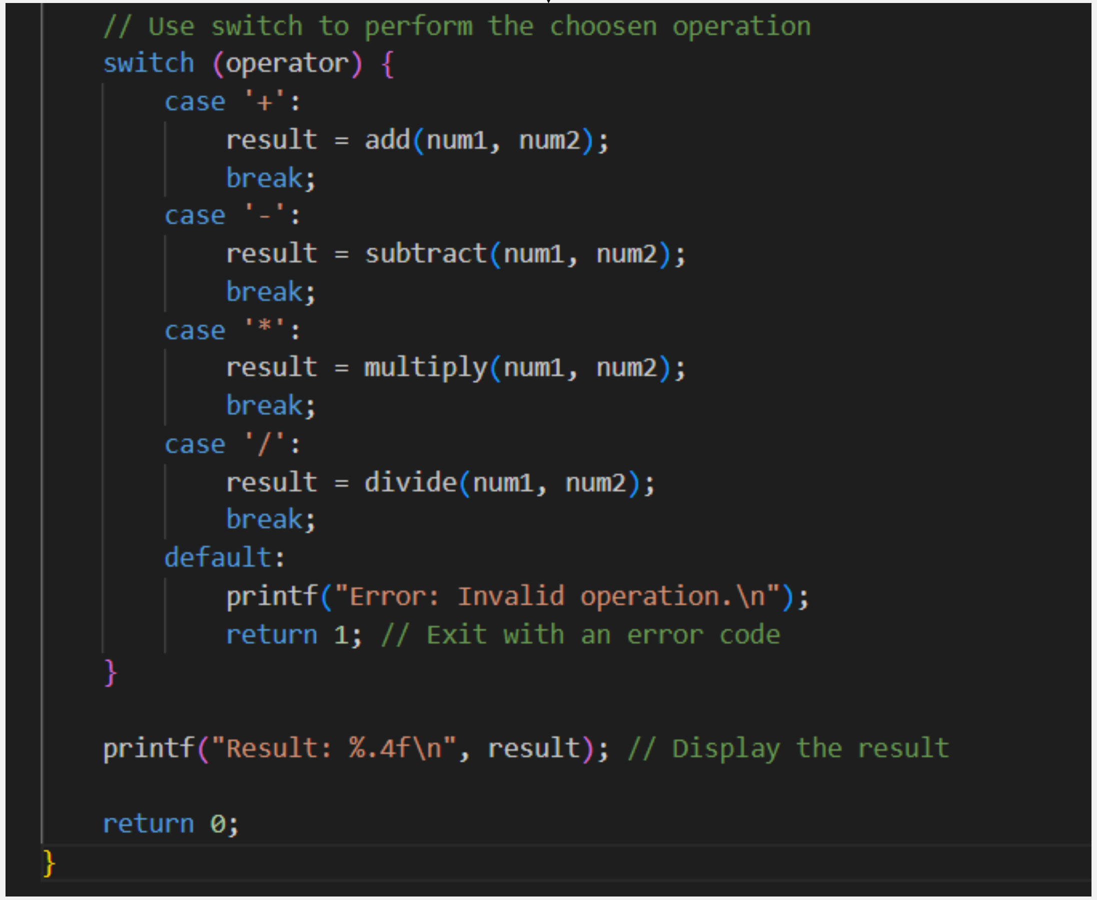


```c
#include <stdio.h>

// Function to perform addition
double add(double num1, double num2) {
    return num1 + num2;
}

// Function to perform subtraction
double subtract(double num1, double num2) {
    return num1 - num2;
}

// Function to perform multiplication
double multiply(double num1, double num2) {
    return num1 * num2;
}

// Function to perform division
double divide(double num1, double num2) {
    if (num2 != 0) {
        return num1 / num2;
    } else {
        printf("Error: Division by zero is not allowed.\n");
        return 0.0; // Return a default value
    }
}
```

Simple calculator – using switch



```c
int main() {
    char operator;
    double num1, num2, result;

    // Choose an operation
    printf("Choose an operation (+, -, *, /): ");
    scanf(" %c", &operator);

    // Enter two numbers
    printf("Enter first number: ");
    scanf("%lf", &num1);

    printf("Enter second number: ");
    scanf("%lf", &num2);
```



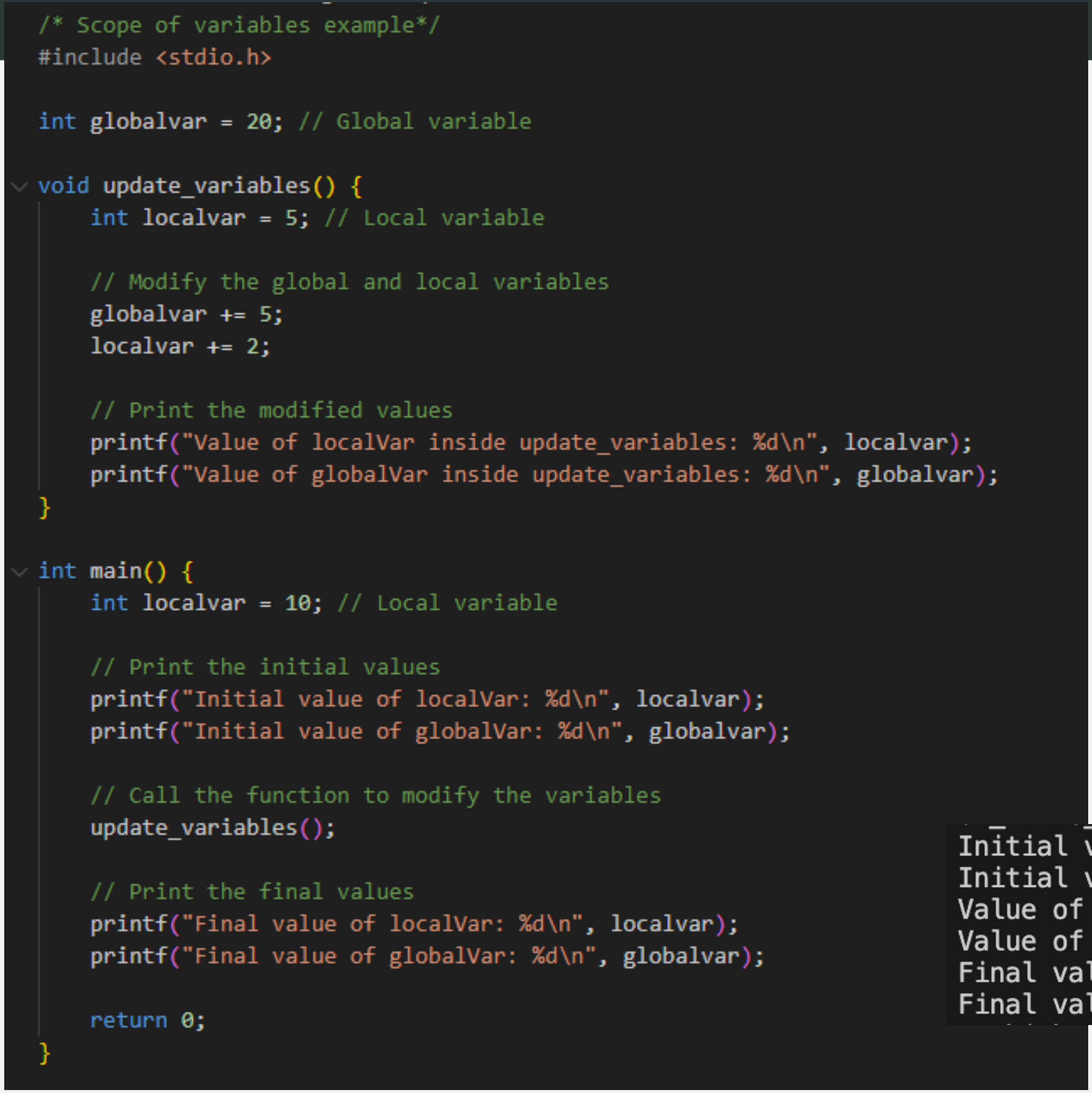
```c
    // Use switch to perform the choosen operation
    switch (operator) {
        case '+':
            result = add(num1, num2);
            break;
        case '-':
            result = subtract(num1, num2);
            break;
        case '*':
            result = multiply(num1, num2);
            break;
        case '/':
            result = divide(num1, num2);
            break;
        default:
            printf("Error: Invalid operation.\n");
            return 1; // Exit with an error code
    }

    printf("Result: %.4f\n", result); // Display the result

    return 0;
}
```

Scope – practice exercise

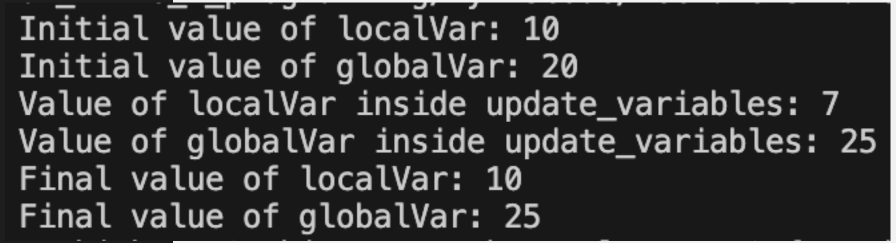
Exercise

- Write a C program to demonstrate the scope of variables. Define a global variable ‘globalvar’ and a local variable ‘localvar’ in the main function.
- Create a user-defined function ‘update_variables’ that attempts to access, modify both the global and local variables, and print them.
- In the main function, print the values of both variables before and after calling the ‘update_variables’ function to observe how the scope affects the accessibility and modification of these variables.

Solution



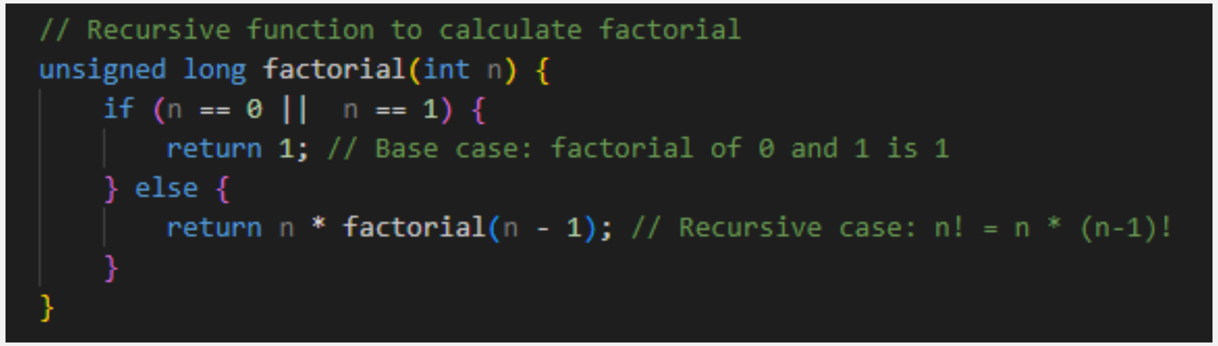
```c
/* Scope of variables example*/
#include <stdio.h>

int globalvar = 20; // Global variable

void update_variables() {
    int localvar = 5; // Local variable

    // Modify the global and local variables
    globalvar += 5;
    localvar += 2;

    // Print the modified values
    printf("Value of localVar inside update_variables: %d\n", localvar);
    printf("Value of globalVar inside update_variables: %d\n", globalvar);
}

int main() {
    int localvar = 10; // Local variable

    // Print the initial values
    printf("Initial value of localVar: %d\n", localvar);
    printf("Initial value of globalVar: %d\n", globalvar);

    // Call the function to modify the variables
    update_variables();

    // Print the final values
    printf("Final value of localVar: %d\n", localvar);
    printf("Final value of globalVar: %d\n", globalvar);

    return 0;
}
```



```
Initial value of localVar: 10
Initial value of globalVar: 20
Value of localVar inside update_variables: 7
Value of globalVar inside update_variables: 25
Final value of localVar: 10
Final value of globalVar: 25
```

Recursion function

- Factorial (*n!*) through recursion

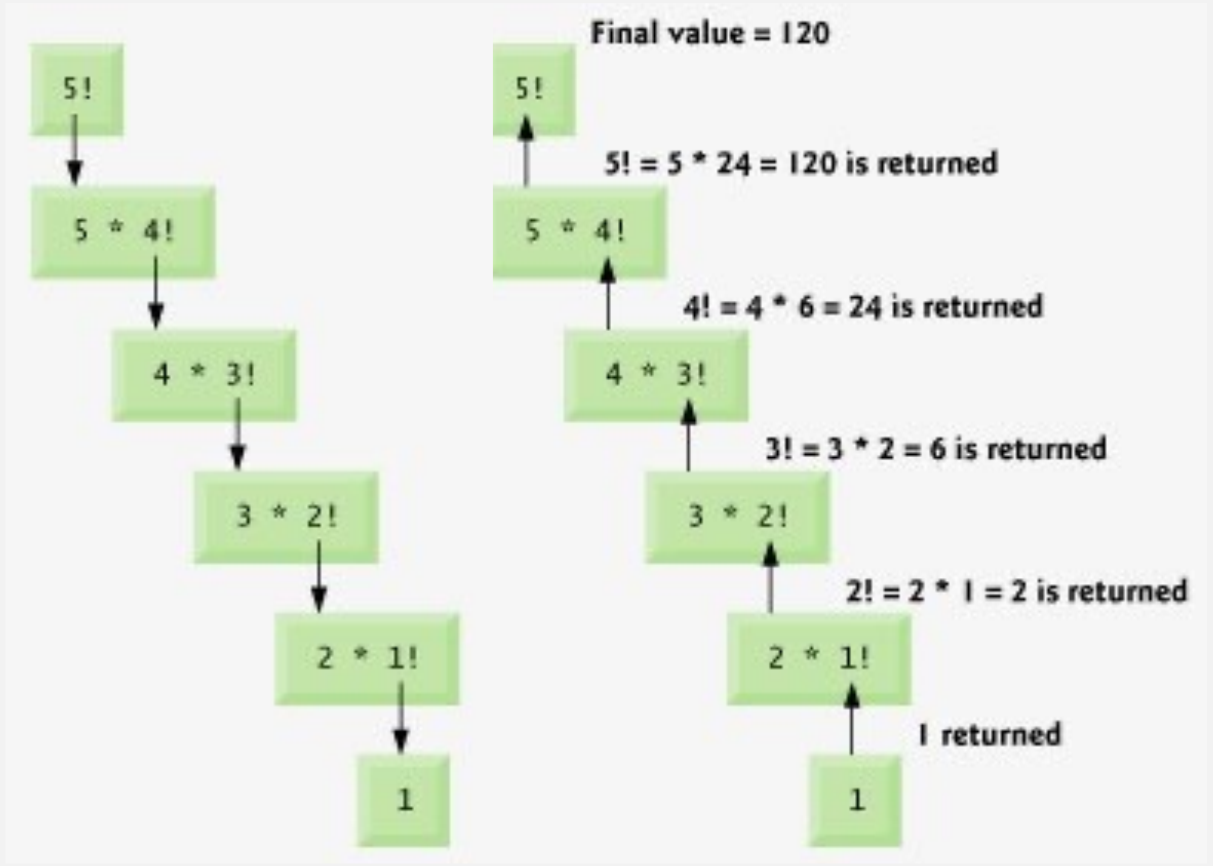


```
// Recursive function to calculate factorial
unsigned long factorial(int n) {
    if (n == 0 ||  n == 1) {
        return 1; // Base case: factorial of 0 and 1 is 1
    } else {
        return n * factorial(n - 1); // Recursive case: n! = n * (n-1)!
    }
}
```

- Combination: $^{n}C_{r} = \frac{n!}{r!(n-r)!}$

- Permutation: $^{n}P_{r} = \frac{n!}{(n-r)!}$

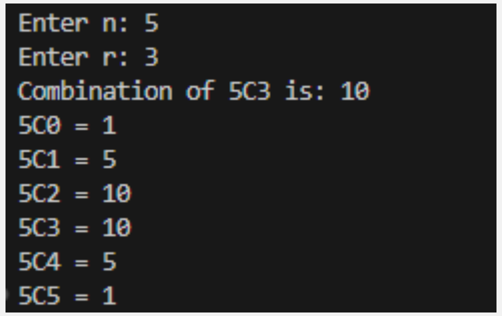
Recursion calls **Values return from each call**


5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1
Final value = 120
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1 returned
1

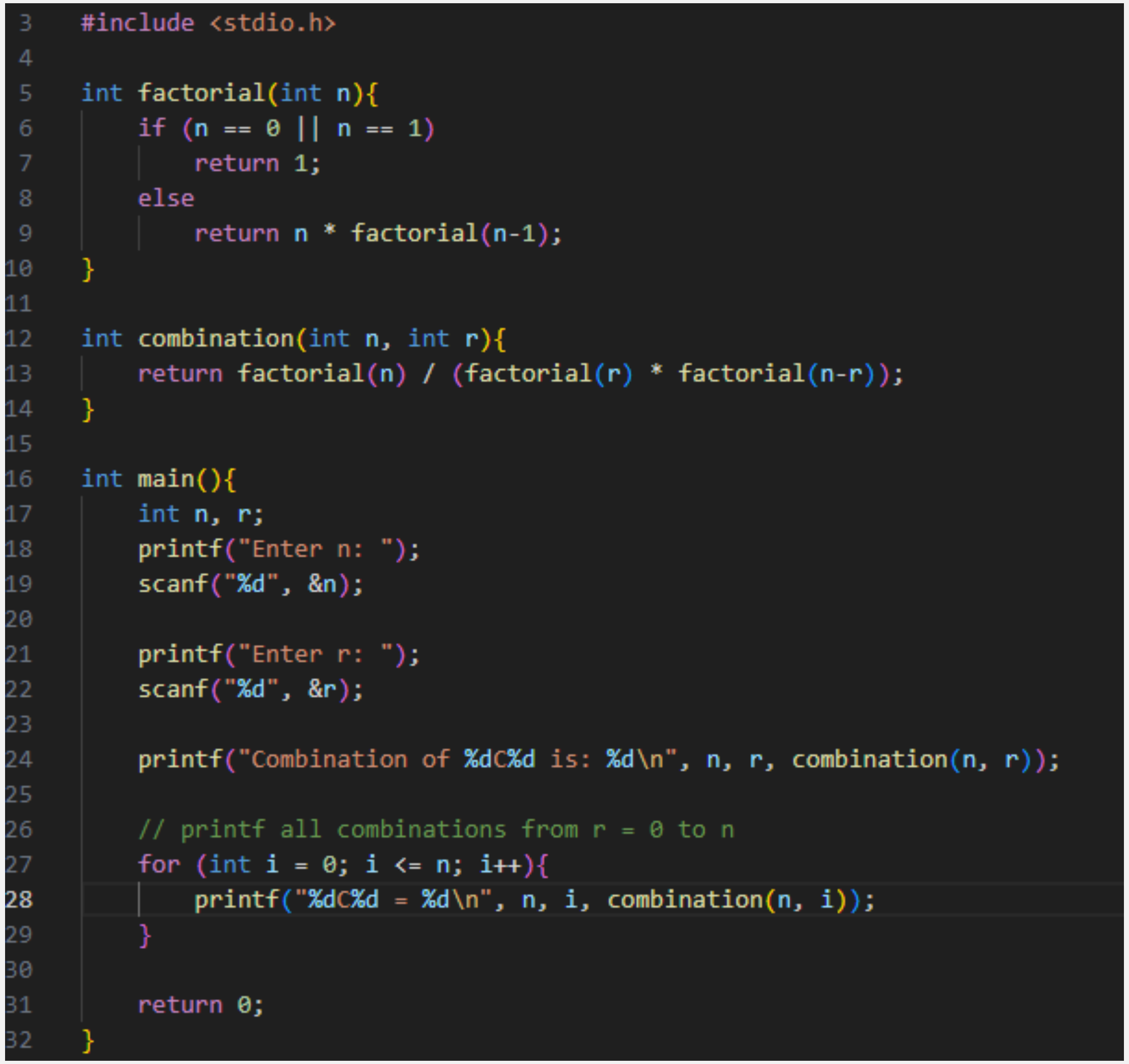
Recursion function

- Combination: $^{n}C_{r} = \frac{n!}{r!(n-r)!}$

Output



```
Enter n: 5
Enter r: 3
Combination of 5C3 is: 10
5C0 = 1
5C1 = 5
5C2 = 10
5C3 = 10
5C4 = 5
5C5 = 1
```



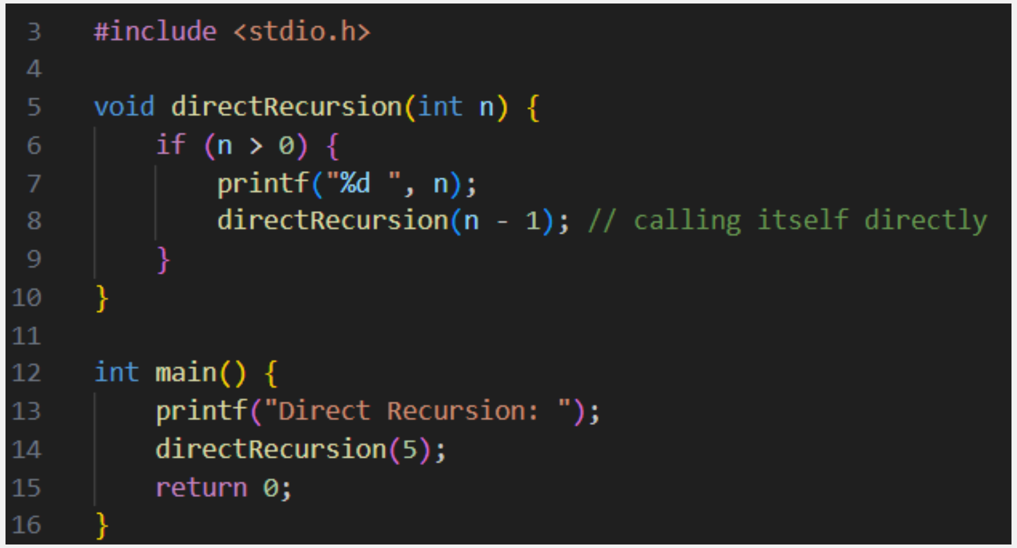
```c
#include <stdio.h>

int factorial(int n){
    if (n == 0 || n == 1)
        return 1;
    else
        return n * factorial(n-1);
}

int combination(int n, int r){
    return factorial(n) / (factorial(r) * factorial(n-r));
}

int main(){
    int n, r;
    printf("Enter n: ");
    scanf("%d", &n);

    printf("Enter r: ");
    scanf("%d", &r);

    printf("Combination of %dC%d is: %d\n", n, r, combination(n, r));

    // printf all combinations from r = 0 to n
    for (int i = 0; i <= n; i++){
        printf("%dC%d = %d\n", n, i, combination(n, i));
    }

    return 0;
}
```

Recursion vs Iteration

Recursion	Iteration
Function calls itself directly or indirectly	Uses a repetition structure with loops
Stops when the base case is reached	Terminates when the loop condition fails
Stores all the steps in a memory stack	Doesnot use stack memory
Continues to generate simplified versions of the original problem until the base case is reached	Continues to modify the counter until the loop continuation condition fails
Code is comparatively smaller	Code tends to be bigger in size
Stack overflow error occurs if the base case is not defined	Infinite loop if the control variable doesnot reach the termination value
Slower in execution	Comparatively faster
Best to use if a problem can be divided into smaller subproblems similar to the original problem	Best to use if a problem can be divided into smaller, repeated steps

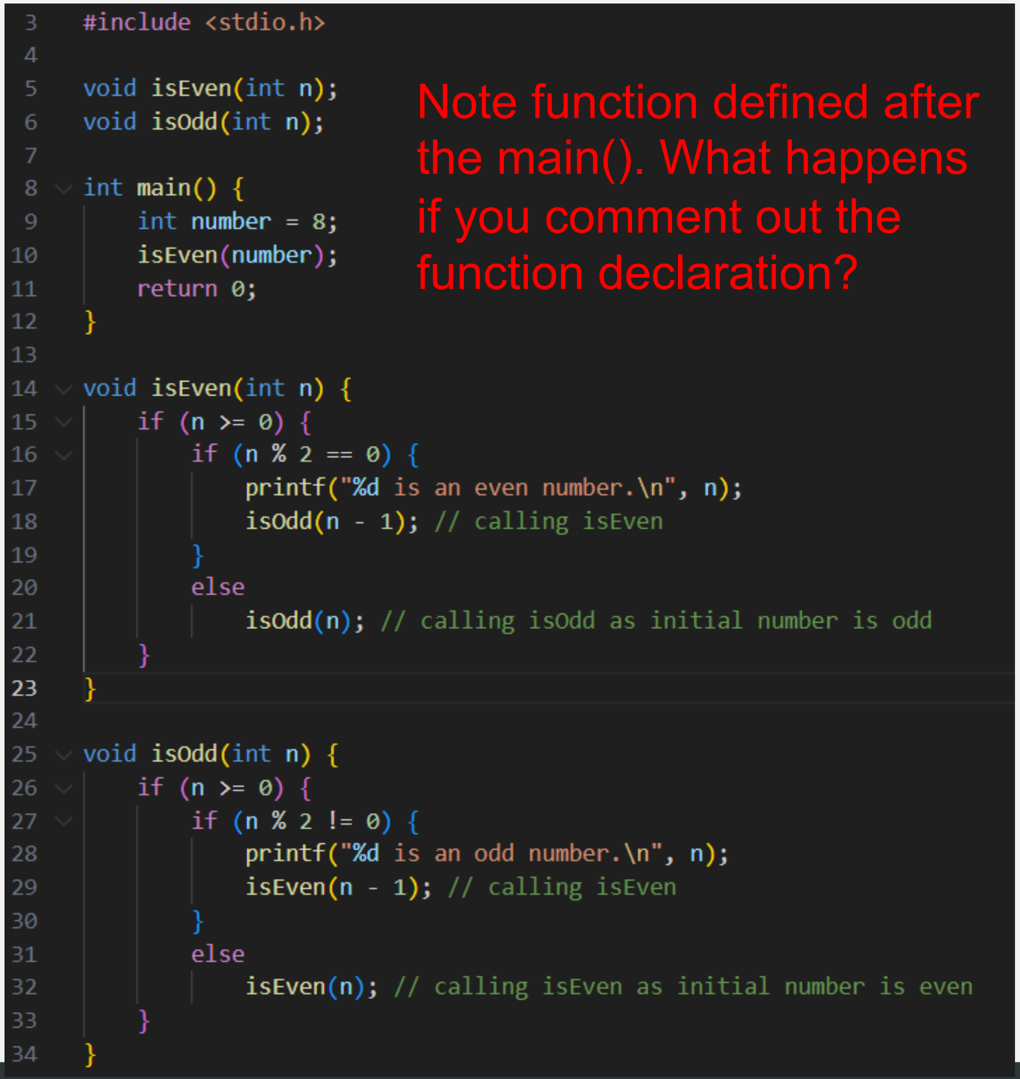
Direct vs Indirect Recursion

- Function calls itself



```c
#include <stdio.h>

void directRecursion(int n) {
    if (n > 0) {
        printf("%d ", n);
        directRecursion(n - 1); // calling itself directly
    }
}

int main() {
    printf("Direct Recursion: ");
    directRecursion(5);
    return 0;
}
```

- One function calls another, which then calls the first function or another function, creating a cycle.



```c
#include <stdio.h>

void isEven(int n);
void isOdd(int n);

int main() {
    int number = 8;
    isEven(number);
    return 0;
}

void isEven(int n) {
    if (n >= 0) {
        if (n % 2 == 0) {
            printf("%d is an even number.\n", n);
            isOdd(n - 1); // calling isEven
        }
        else
            isOdd(n); // calling isOdd as initial number is odd
    }
}

void isOdd(int n) {
    if (n >= 0) {
        if (n % 2 != 0) {
            printf("%d is an odd number.\n", n);
            isEven(n - 1); // calling isEven
        }
        else
            isEven(n); // calling isEven as initial number is even
    }
}
```

Note function defined after the main(). What happens if you comment out the function declaration?

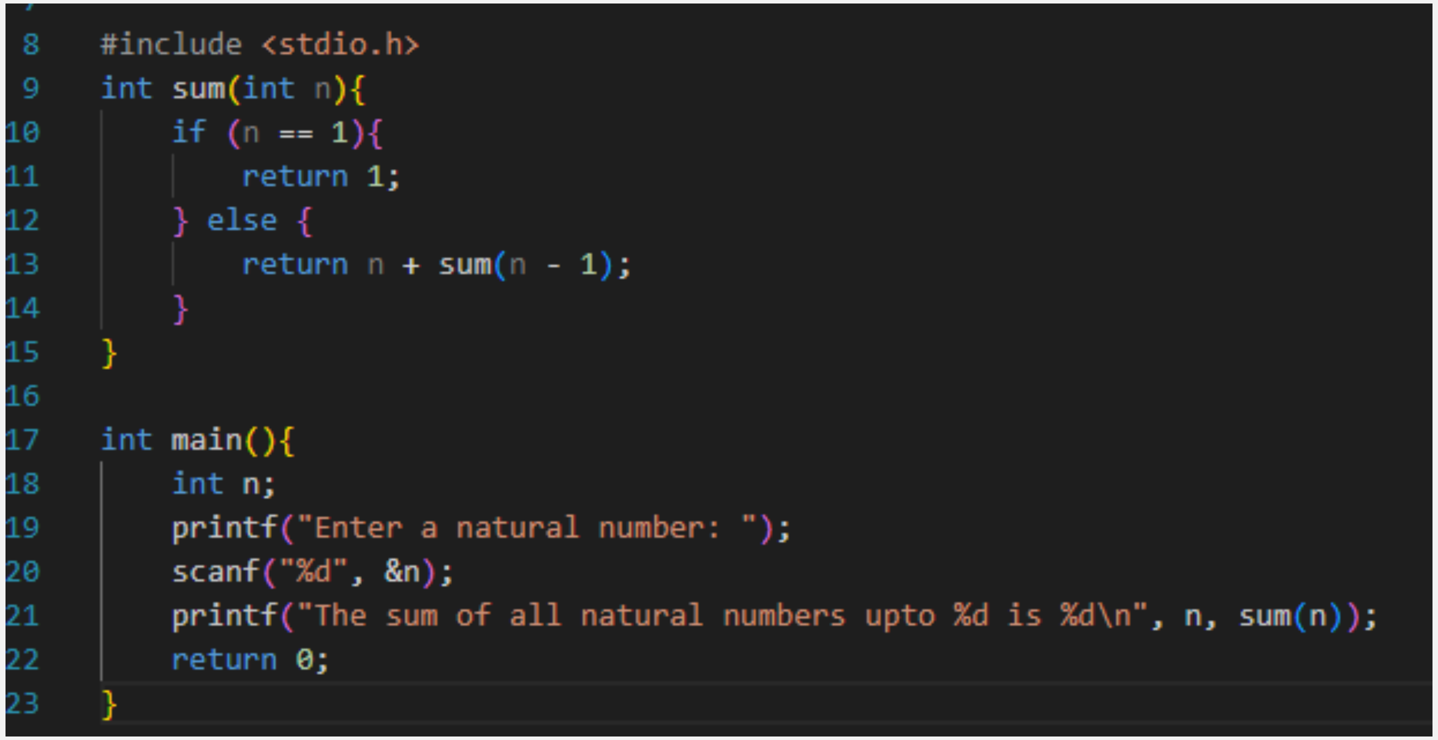
Recursion – practice exercise

- Write a C program using recursion function that take a natural number *n* as input, and calculate the sum of all natural numbers upto *n*.

Hint: $n \rightarrow 1 + 2 + 3 + \ldots (n-1) + n$

Recursion – practice exercise

- **Solution**



```c
#include <stdio.h>
int sum(int n){
    if (n == 1){
        return 1;
    } else {
        return n + sum(n - 1);
    }
}

int main(){
    int n;
    printf("Enter a natural number: ");
    scanf("%d", &n);
    printf("The sum of all natural numbers upto %d is %d\n", n, sum(n));
    return 0;
}
```

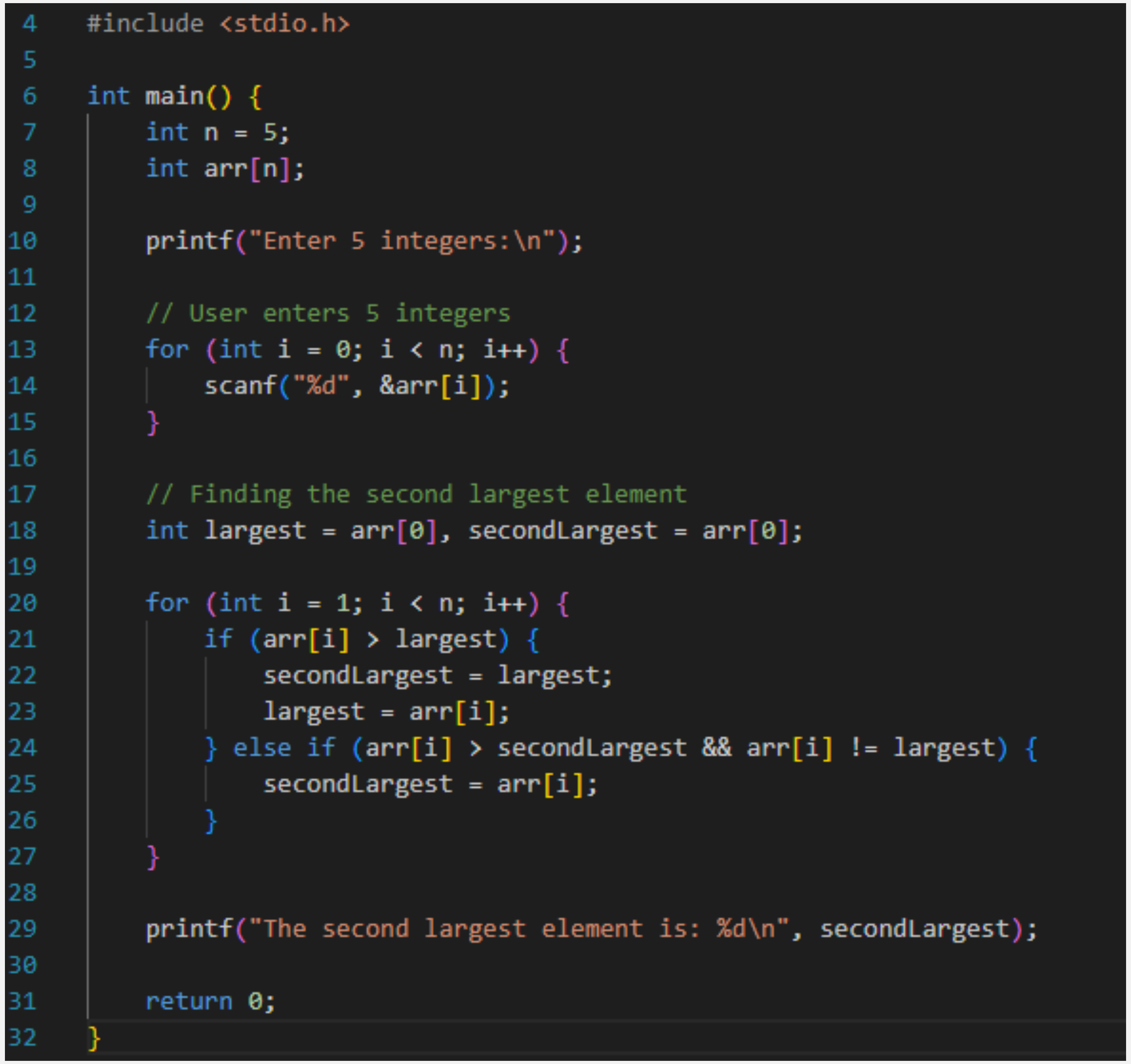
Practice exercise: Array

Exercise

Write a C program that takes 5 integers as input from the user and stores them in an array. Find and print the second largest element in the array.

Practice exercise: Array

- Solution



```c
#include <stdio.h>

int main() {
    int n = 5;
    int arr[n];

    printf("Enter 5 integers:\n");

    // User enters 5 integers
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Finding the second largest element
    int largest = arr[0], secondLargest = arr[0];

    for (int i = 1; i < n; i++) {
        if (arr[i] > largest) {
            secondLargest = largest;
            largest = arr[i];
        } else if (arr[i] > secondLargest && arr[i] != largest) {
            secondLargest = arr[i];
        }
    }

    printf("The second largest element is: %d\n", secondLargest);

    return 0;
}
```

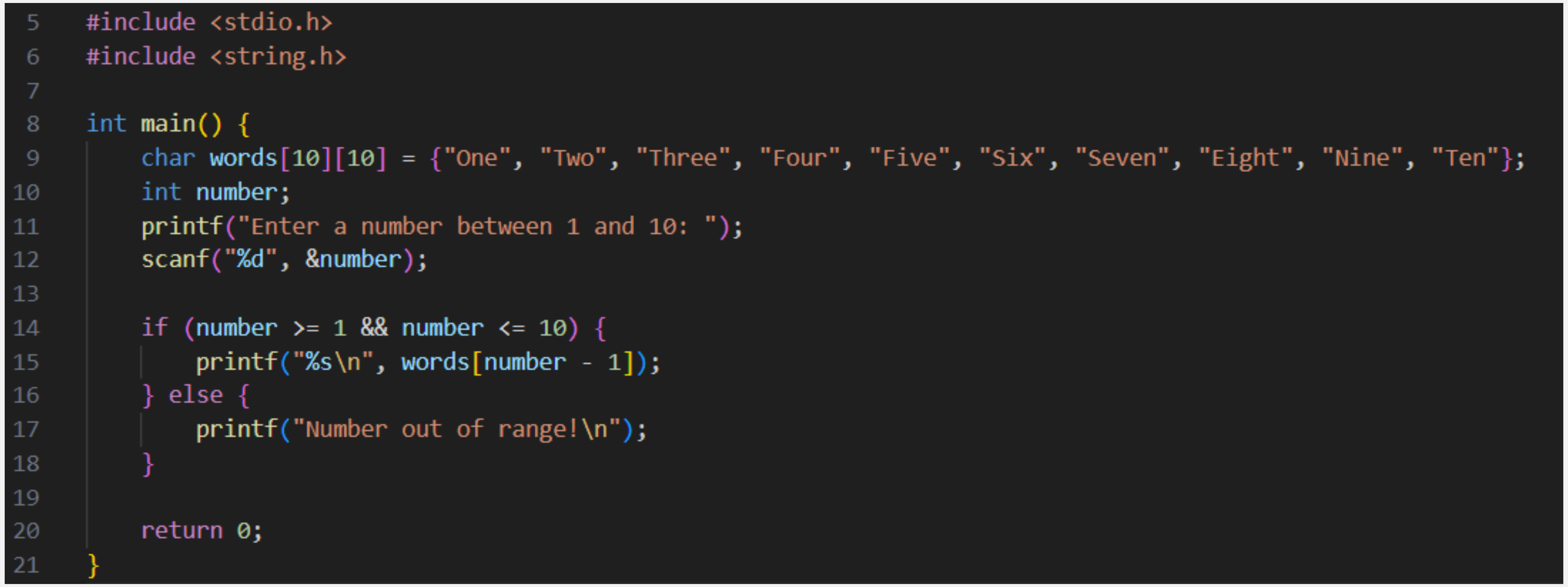
Practice exercise: Array and String

Exercise

Write a C program that takes an integer input between 1 and 10 from the user and converts it into its corresponding word form. For example, if the user inputs 5, the program should print "Five". Ensure that the program displays an appropriate message if the user enters a number outside the specified range.

Practice exercise: Array and String

Solution



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

int main() {
    char words[10][10] = {"One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine", "Ten"};
    int number;
    printf("Enter a number between 1 and 10: ");
    scanf("%d", &number);

    if (number >= 1 && number <= 10) {
        printf("%s\n", words[number - 1]);
    } else {
        printf("Number out of range!\n");
    }

    return 0;
}
```

Online resources – practice

- C programming essentials (Linkedin Learn)
- https://www.codingame.com/home (practice conditions, loops, arrays) (select C programming)

Today’s lecture

- Arrays in more details
  - String handling functions
  - Multi-dimensional arrays
  - Passing arrays to function
- Performance (time) of algorithm
  - Sequential vs Binary search
- Pointers are coming!

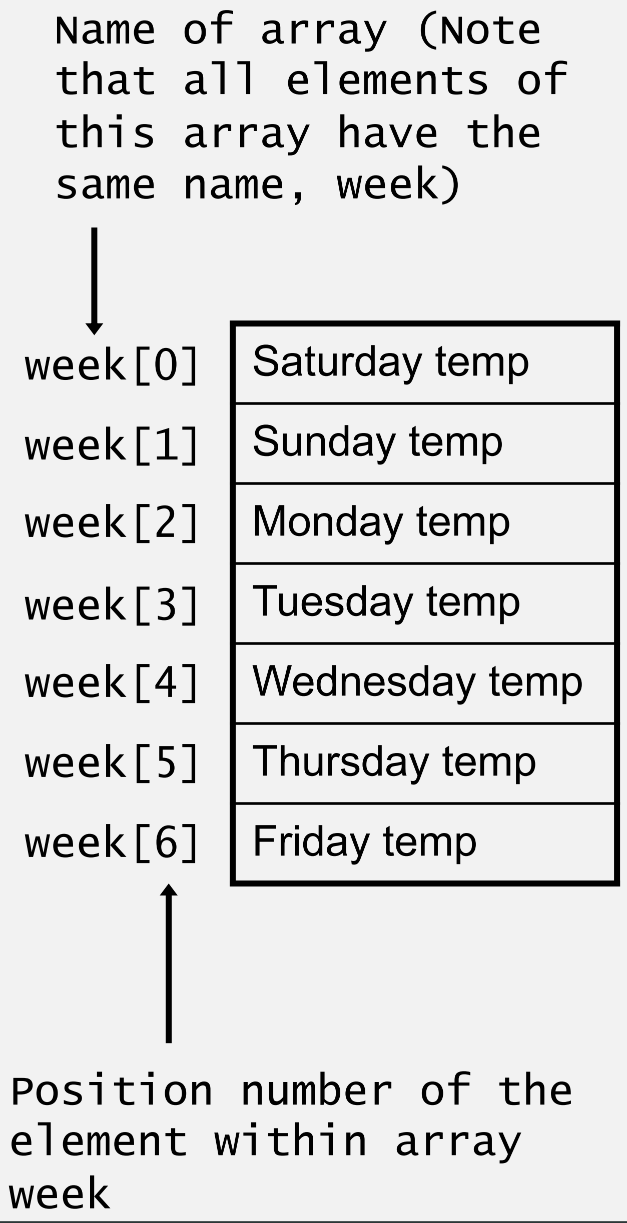
Arrays in more detail

Arrays

- Collection of similar data types stored at contiguous memory locations.
- Provides a convenient way to store multiple values of the same data type under a single identifier.

Declaration:

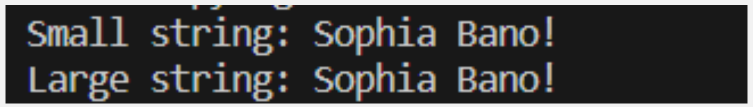
```
datatype arrayName[arraySize];

int week; // defines a non-array variable
int week[7]; // defines an array

char name[7] = “Sophia”; // leave room for null
```



Arrays – String operations

```
char name[7] = “Sophia”; // leave room for null
char fullname[20] = “Sophia Bano!";

strcpy(name, fullname); //copy fullname to name
```

Output



```
Small string: Sophia Bano!
Large string: Sophia Bano!
```

But this result would be disastrous because other data in memory could be overwritten unintentionally

Important:
If the array needs to be larger than its initial value, specify larger array size in the definition

```
char name[20] = “Sophia”;
```

Common String Handling Functions in C

Function	Description
strlen()	Can compute the length of the string
strcpy()	Can copy the content of a string to another
strcat()	Is used to concatenate or join two strings
strcmp()	Can compare two strings
strlwr()	Can convert the string to lowercase
strupr()	Is used to convert the letters of string to uppercase
strrev()	Is used to reverse the string

- **Exercise:** Write a C program to experiment with the above functions and observe the output.

Multi-dimensional Arrays

- Multi-dimensional arrays are arrays that contain arrays as their elements. They are useful for representing data in multiple dimensions, such as matrices or tables.
- Declaration:

```
        datatype arrayName[size1][size2]; // 2D array
```

Example:

```
float x[2][4];
// x can hold 8 elements
```

	Column 1	**Column 2**	**Column 3**	**Column 4**
Row 1	`x[0][0]`	`x[0][1]`	`x[0][2]`	`x[0][3]`
Row 2	`x[1][0]`	`x[1][1]`	`x[1][2]`	`x[1][2]`

Initialization of a 2D Array

Different ways of initialising 2D arrays

```
int c[2][3] = {{1, 2, 3}, {-1, -2, -3}};

int c[][3] = {{1, 2, 3}, {-1, -2, -3}};

int c[2][3] = {1, 2, 3, -1, -2, -3};
```

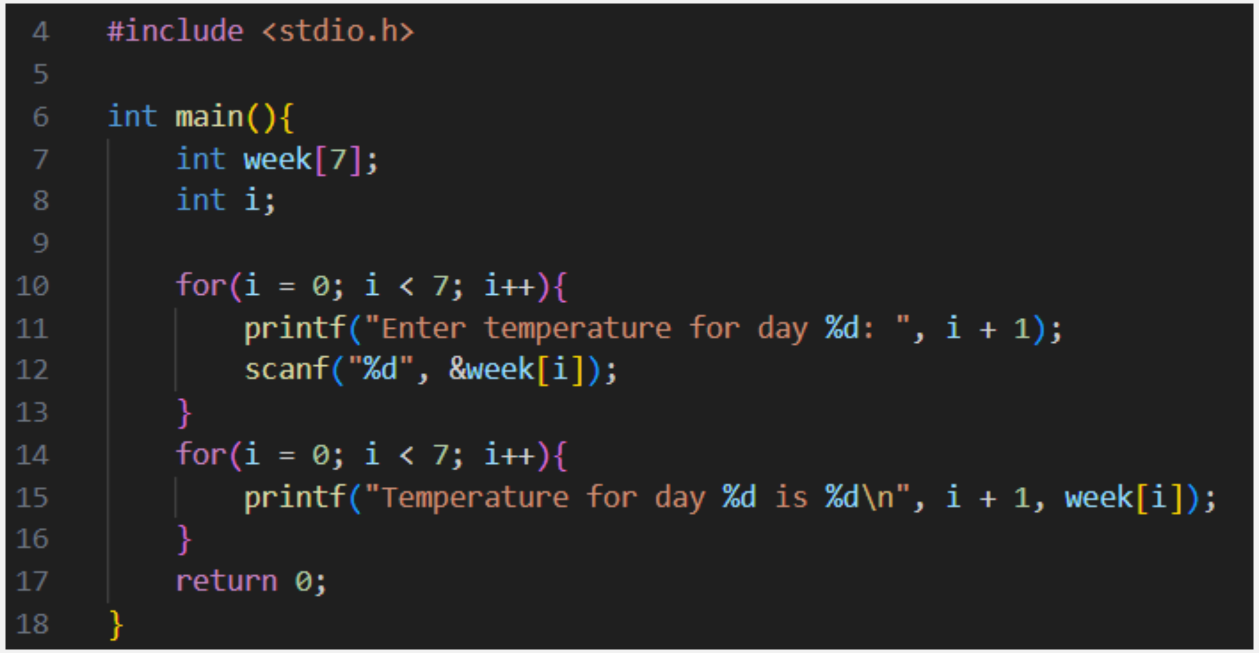
	Column 1	Column 2	Column 3
Row 1	1	2	3
Row 2	-1	-2	-3

Similarity, we can define 3D arrays:

```
float y[2][4][3];
```

One Dimension Array - example

- We saw last week how to read data and process 1D array.
- Imagine we donot know the size of the array. How do we compute its length?

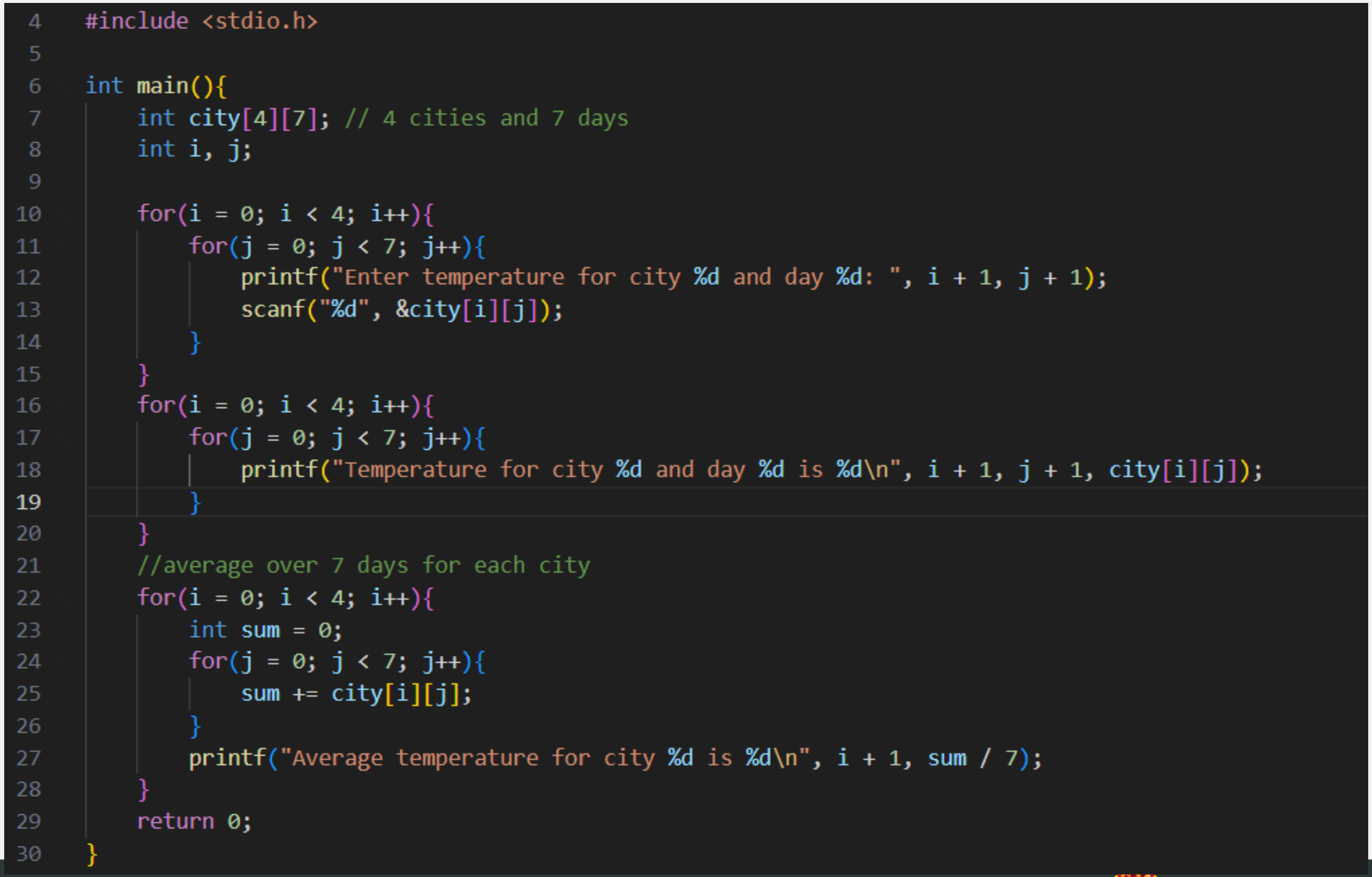


```
#include <stdio.h>

int main(){
    int week[7];
    int i;

    for(i = 0; i < 7; i++){
        printf("Enter temperature for day %d: ", i + 1);
        scanf("%d", &week[i]);
    }
    for(i = 0; i < 7; i++){
        printf("Temperature for day %d is %d\n", i + 1, week[i]);
    }
    return 0;
}
```

```
int size = sizeof(week)/sizeof(week[0]);
```

Two Dimension Array - Example

- Temperature of 4 cities over 7 days, and average temperature of each city over 7 days.
- **Note:** How the array elements are indexed and used



```c
#include <stdio.h>

int main(){
    int city[4][7]; // 4 cities and 7 days
    int i, j;

    for(i = 0; i < 4; i++){
        for(j = 0; j < 7; j++){
            printf("Enter temperature for city %d and day %d: ", i + 1, j + 1);
            scanf("%d", &city[i][j]);
        }
    }
    for(i = 0; i < 4; i++){
        for(j = 0; j < 7; j++){
            printf("Temperature for city %d and day %d is %d\n", i + 1, j + 1, city[i][j]);
        }
    }
    //average over 7 days for each city
    for(i = 0; i < 4; i++){
        int sum = 0;
        for(j = 0; j < 7; j++){
            sum += city[i][j];
        }
        printf("Average temperature for city %d is %d\n", i + 1, sum / 7);
    }
    return 0;
}
```

Practice exercise: 2D Array

Exercise

Write a C program that output the sum of two given n x n matrices.

Passing Arrays to Function

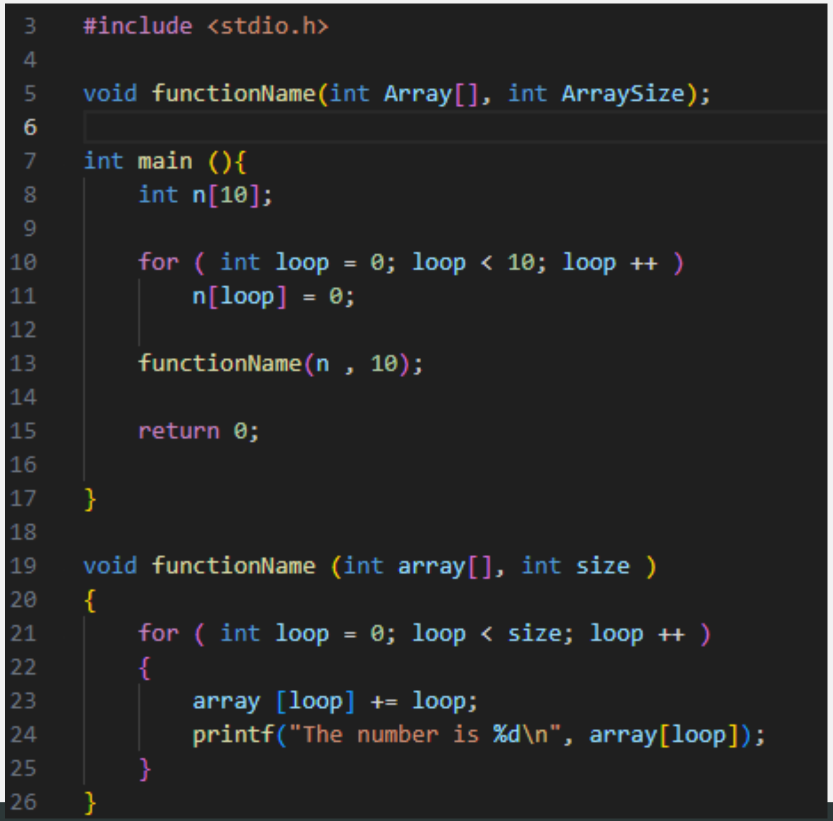
- Arrays can also be passed to Functions Like normal variables.

- To pass an Array to Function two things are specified.
  - Array variable name
  - Array size

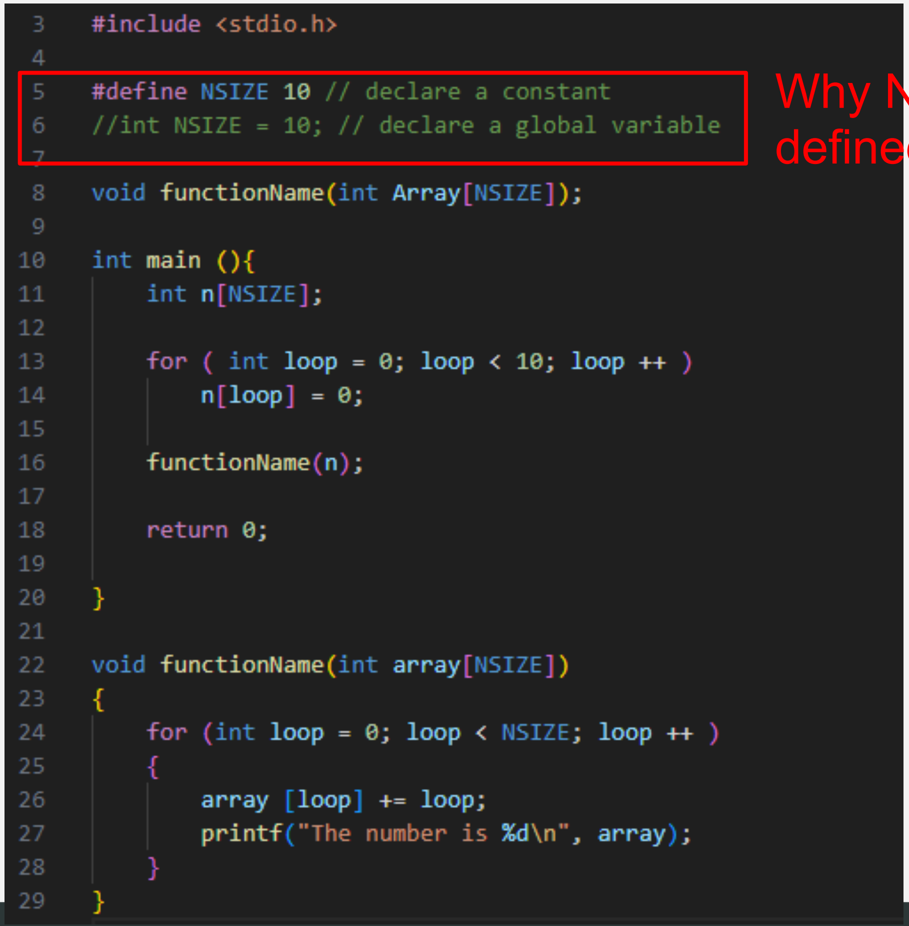
- Example:-
  - `functionName( ArrayName, Array_size );`
  - `functionName(Array1, Array1_size, Array2, Array2_size);`

Passing Arrays to Function

- Note the two different ways for passing array size



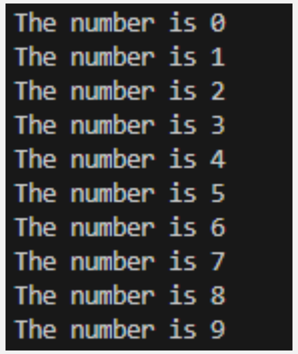
```c
#include <stdio.h>

void functionName(int Array[], int ArraySize);

int main (){
    int n[10];

    for ( int loop = 0; loop < 10; loop ++ )
        n[loop] = 0;

    functionName(n , 10);

    return 0;

}

void functionName (int array[], int size )
{
    for ( int loop = 0; loop < size; loop ++ )
    {
        array [loop] += loop;
        printf("The number is %d\n", array[loop]);
    }
}
```



```c
#include <stdio.h>

#define NSIZE 10 // declare a constant
//int NSIZE = 10; // declare a global variable

void functionName(int Array[NSIZE]);

int main (){
    int n[NSIZE];

    for ( int loop = 0; loop < 10; loop ++ )
        n[loop] = 0;

    functionName(n);

    return 0;

}

void functionName(int array[NSIZE])
{
    for (int loop = 0; loop < NSIZE; loop ++ )
    {
        array [loop] += loop;
        printf("The number is %d\n", array);
    }
}
```

Why NSIZE is defined like this?

Output



```
The number is 0
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5
The number is 6
The number is 7
The number is 8
The number is 9
```

Analysis of Algorithms

Analysis of Algorithms

- The study of the efficiency of various algorithms is called the analysis of algorithms.
- If the algorithms are efficient then obviously the execution will be also efficient and fast.
- How can we compare the performance (time) of algorithm?

Comparing the Performance of Algorithm

- We can compare the performance (time) of algorithm by the number of comparisons and iterations that it performs.

- Consider the following code

```
for ( int i=0; i<n; i++)
{
    int value = info[i];
}
```

The algorithm perform n iterations.

So the algorithm time will be n.

This is denoted by **O(n)**. ← the 'big O' notation

Three Different Algorithm Times

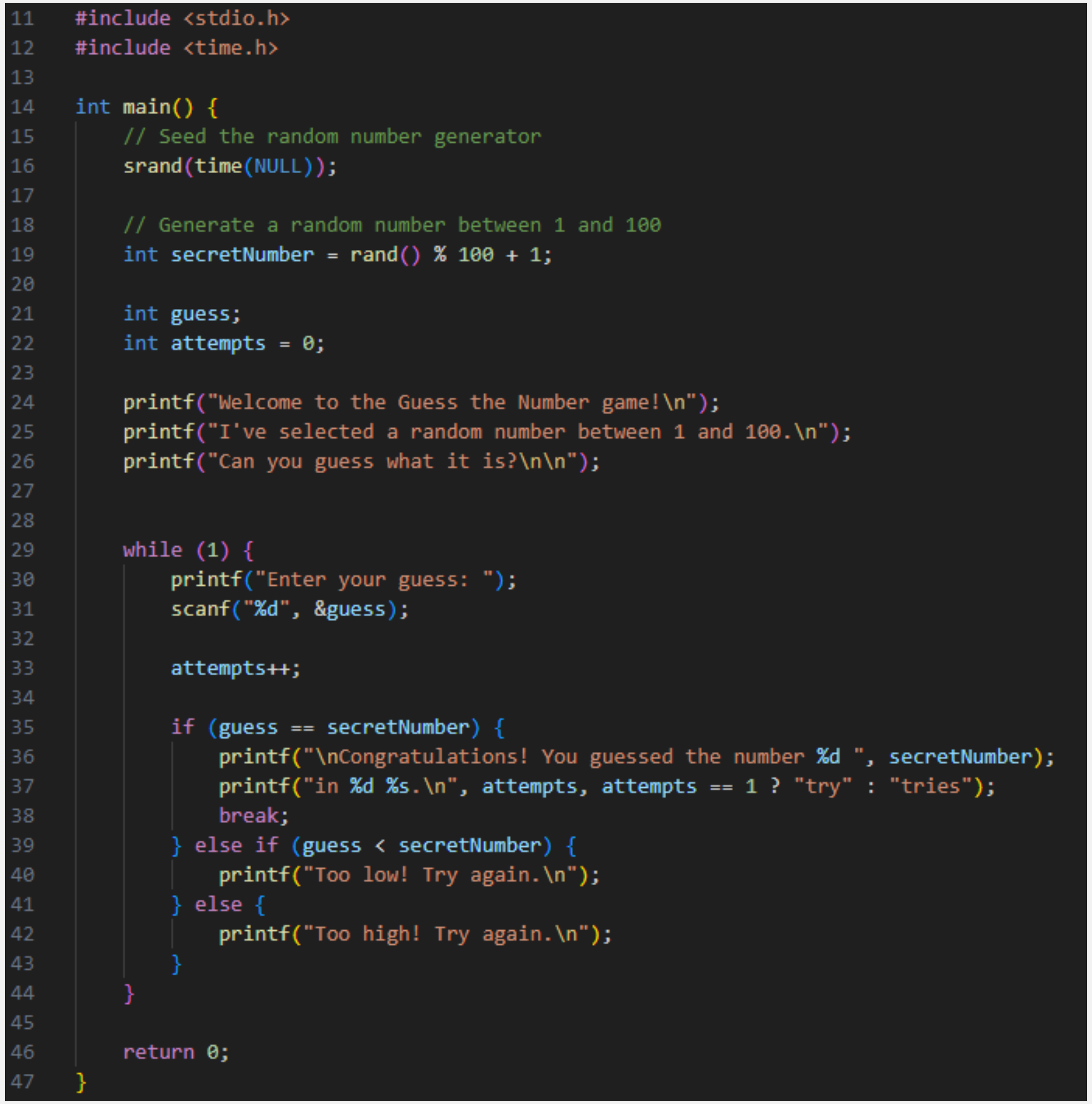
- Suppose if you have to find a file name (image_050.png) in your computer picture folder.
  - And this folder contains 100 different types of files.
- **Best case:**
  - You check the first file, and luckily its name was (image_050.png). This means you have found it in only one iteration.
- **Worst case:**
  - You check all the 100 files, and the last name was (image_050.png).
- **Average case:**
  - What will be the average time that you will find (image_050.png).
  - Probability (50%)

- But remember, we **always consider worst case in algorithm time.**

Revisit – Lab 3

Guess the number game

https://onlinegdb.com/n-RZc4F4r



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

int main() {
    // Seed the random number generator
    srand(time(NULL));

    // Generate a random number between 1 and 100
    int secretNumber = rand() % 100 + 1;

    int guess;
    int attempts = 0;

    printf("Welcome to the Guess the Number game!\n");
    printf("I've selected a random number between 1 and 100.\n");
    printf("Can you guess what it is?\n\n");


    while (1) {
        printf("Enter your guess: ");
        scanf("%d", &guess);

        attempts++;

        if (guess == secretNumber) {
            printf("\nCongratulations! You guessed the number %d ", secretNumber);
            printf("in %d %s.\n", attempts, attempts == 1 ? "try" : "tries");
            break;
        } else if (guess < secretNumber) {
            printf("Too low! Try again.\n");
        } else {
            printf("Too high! Try again.\n");
        }
    }

    return 0;
}
```

Sequential Search

- The Sequential search algorithms finds a target in a list or array.
- This algorithm finds the target by comparing each element of list. Each individual checking is called iteration.
- If the target is equal to element, then the algorithm return “found”.
- If the target not equal to element, then the algorithm check the next iteration.

Sequential Search

Element	0	1	2	3	4	5	6	7	8
Info	76	34	235	66	430	373	23	64	120

Target = 23

Sequential Search

Iteration = 1

Element	0	1	2	3	4	5	6	7	8
Info	76	34	235	66	430	373	23	64	120

Target = 23

Sequential Search

Iteration = 2

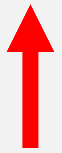
Element	0	1	2	3	4	5	6	7	8
Info	76	34	235	66	430	373	23	64	120

Target = 23

Sequential Search

Iteration = 3

Element	0	1	2	3	4	5	6	7	8
Info	76	34	235	66	430	373	23	64	120



Target = 23

Sequential Search

Iteration = 4

Element	0	1	2	3	4	5	6	7	8
Info	76	34	235	66	430	373	23	64	120

Target = 23

Sequential Search

Iteration = 5

Element	0	1	2	3	4	5	6	7	8
Info	76	34	235	66	430	373	23	64	120

$\uparrow$

Target = 23

Sequential Search

Iteration = 6

Element	0	1	2	3	4	5	6	7	8
Info	76	34	235	66	430	373	23	64	120

↑

Target = 23

Sequential Search

Iteration = 7

Element	0	1	2	3	4	5	6	7	8
Info	76	34	235	66	430	373	23	64	120

Element[6] is equal to Target

Target = 23

Sequential Search (C Function)

```
int sequentialSearch(int list[ ], int listSize, int target){
        for ( int loop=0; loop<listSize; loop++ ){
                if ( list [loop] == target )
                        return 1;
        }
        return 0;
}
```

The algorithm perform listSize iterations.

Where the listSize = n

So the algorithm time will be n.

Binary Search

- Our second searching algorithm is Binary search.
- Binary search is a **divide and conquer strategy**.
- In each iteration of the algorithm, the searching list is divided into two partitions, and only one partition is checked, the other one is ignored.
- Binary search algorithm requires the list elements to be in a sorted order.

Binary Search

Iteration = 1 **Target = 23**

Element	0	1	2	3	4	5	6	7	8
Info	23	34	64	66	76	120	235	373	430

- First the list is divided into two partitions.
- The partitions are determined on the basis of middle size.

- Here middle is 4 = (listSize/2)

Binary Search

Iteration = 1 **Target = 23**

Element	0	1	2	3	4	5	6	7	8
Info	23	34	64	66	76	120	235	373	430

- Here middle is 4 = ((first element + last element) /2)

Element	0	1	2	3
Info	23	34	64	66

Element	5	6	7	8
Info	120	235	373	430

- If the middle **element = target** then **return found element**
- If the **target > middle element** then **search right partition1**
- If the **target < middle element** then **search left partition2**

Binary Search

Iteration = 2 **Target = 23**

Element	0	1	2	3
Info	23	34	64	66

- Here middle is 1 = ((first element + last element) /2)

Element	0
Info	23

Element	0	1
Info	23	34

- If the middle **element = target** then **return found element**
- If the **target > middle element** then **search right partition1**
- If the **target < middle element** then **search left partition2**

Binary Search

Iteration = 3 **Target = 23**

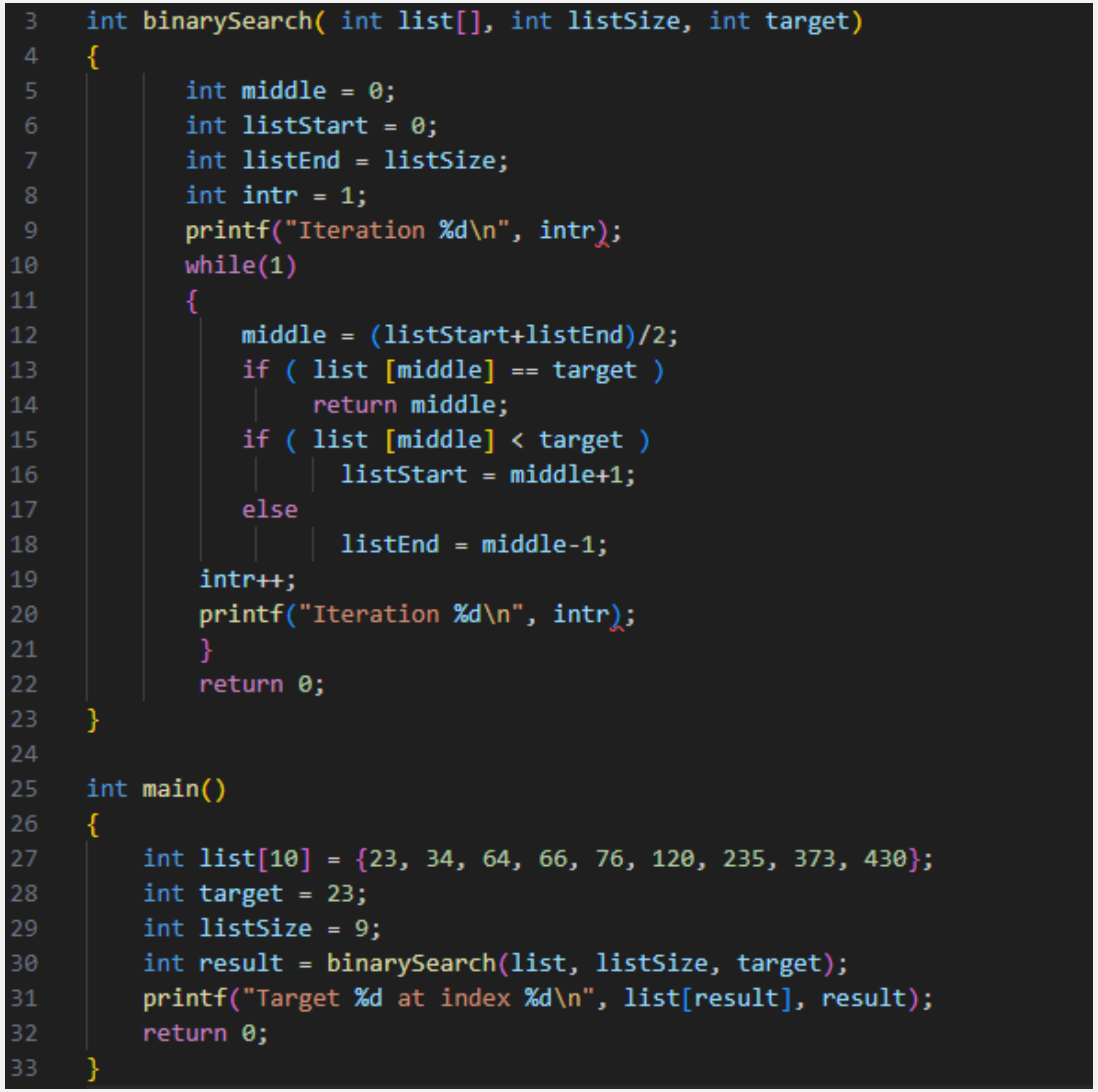
Element	0
Info	23

- Here middle is 0 = ((first element + last element) /2)

- If the **middle element = target** then **return found element**

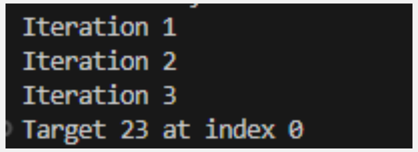
Binary Search

- Solution



```
int binarySearch( int list[], int listSize, int target)
{
        int middle = 0;
        int listStart = 0;
        int listEnd = listSize;
        int intr = 1;
        printf("Iteration %d\n", intr);
        while(1)
        {
            middle = (listStart+listEnd)/2;
            if ( list [middle] == target )
                return middle;
            if ( list [middle] < target )
                listStart = middle+1;
            else
                listEnd = middle-1;
         intr++;
         printf("Iteration %d\n", intr);
         }
         return 0;
}

int main()
{
    int list[10] = {23, 34, 64, 66, 76, 120, 235, 373, 430};
    int target = 23;
    int listSize = 9;
    int result = binarySearch(list, listSize, target);
    printf("Target %d at index %d\n", list[result], result);
    return 0;
}
```

Output



```
Iteration 1
Iteration 2
Iteration 3
Target 23 at index 0
```

Exercise

Write a C program to perform binary search with recursion.

You can use arrays (but not pointers)

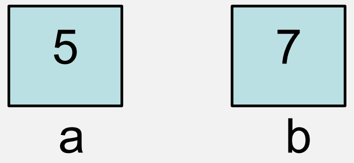
Comparison between Sequential and Binary search

- For same problem the Sequential search find element in 7 iterations.
- While Binary search find it only in 3 iterations.
- This shows that Binary search is most efficient than Sequential search.

Diving into pointers (eventually)

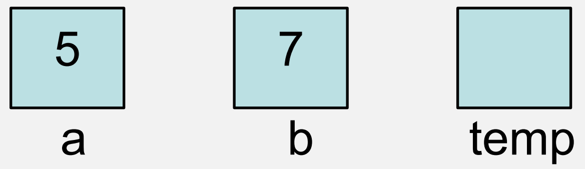
Swap two numbers

```
int a = 5;
int b = 7;
```



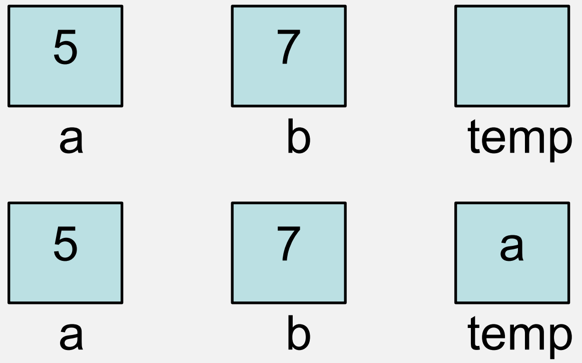
Swap two numbers

```
int a = 5;
int b = 7;
int temp;
```



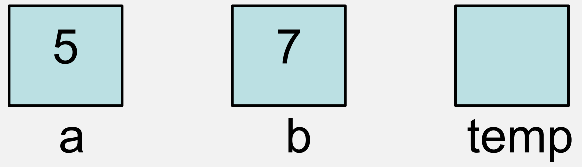
Swap two numbers

```
int a = 5;
int b = 7;
int temp;

temp = a;
```

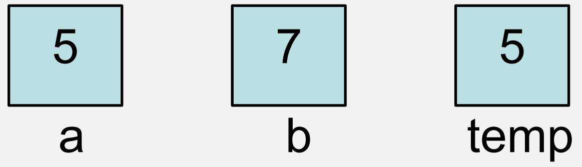


Swap two numbers

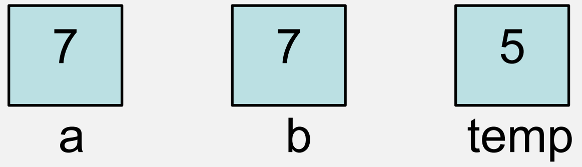
```
int a = 5;
int b = 7;
int temp;
```



```
temp = a;
```

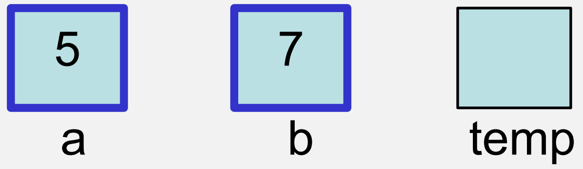


```
a = b;
```

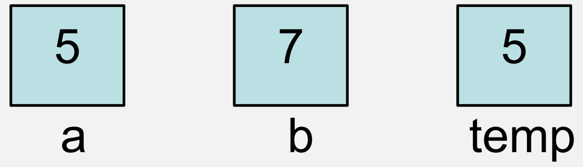


Swap two numbers

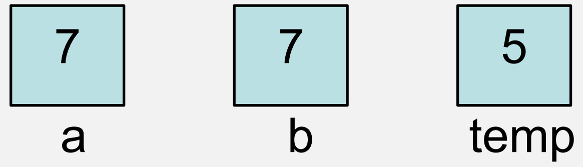
```
int a = 5;
int b = 7;
int temp;
```



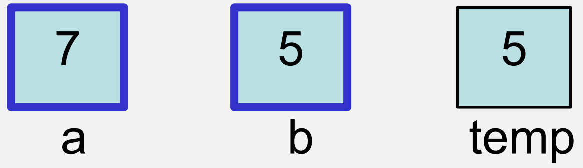
```
temp = a;
```



```
a = b;
```

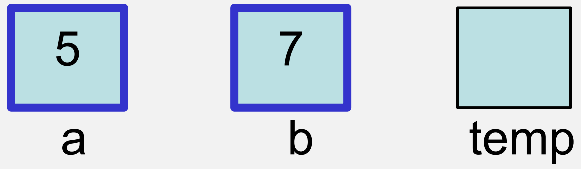


```
b = temp;
```

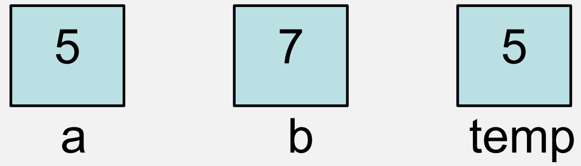


Swap two numbers

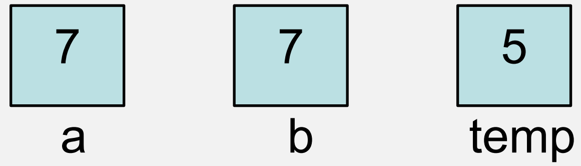
```
int a = 5;
int b = 7;
int temp;
```



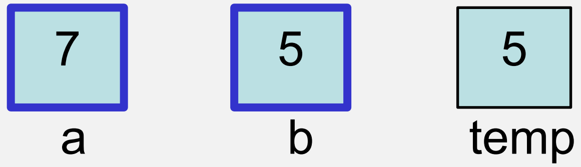
```
temp = a;
```



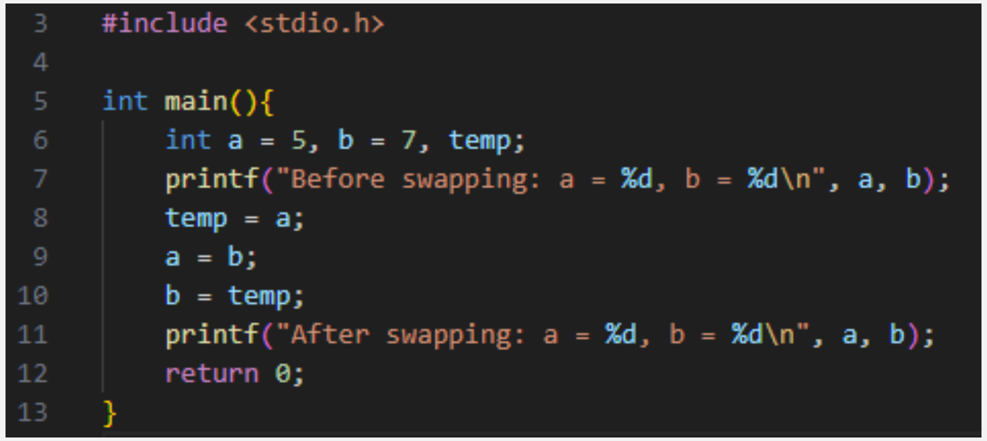
```
a = b;
```



```
b = temp;
```

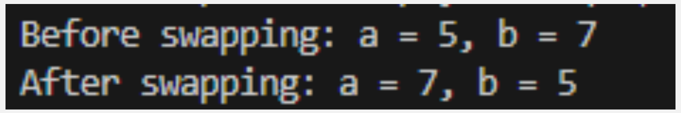


Code



```
#include <stdio.h>

int main(){
    int a = 5, b = 7, temp;
    printf("Before swapping: a = %d, b = %d\n", a, b);
    temp = a;
    a = b;
    b = temp;
    printf("After swapping: a = %d, b = %d\n", a, b);
    return 0;
}
```

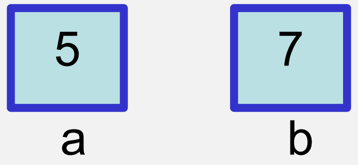
Output



```
Before swapping: a = 5, b = 7
After swapping: a = 7, b = 5
```

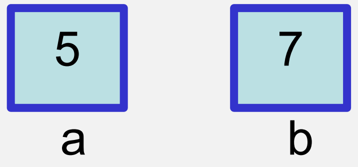
Swap two numbers without using a 3rd variable

```
int a = 5;
int b = 7;
```

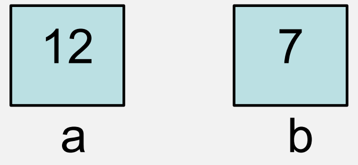


Swap two numbers without using a 3rd variable

```
int a = 5;
int b = 7;
```



```
a = a + b;
```

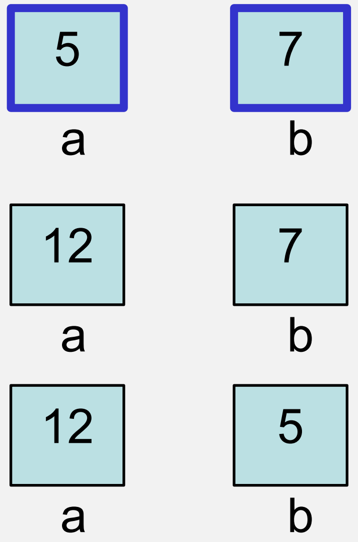


Swap two numbers without using a 3rd variable

```
int a = 5;
int b = 7;
```

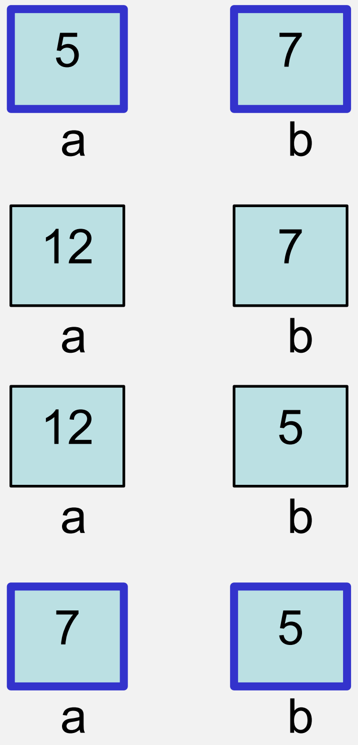
```
a = a + b;
```

```
b = a - b;
```

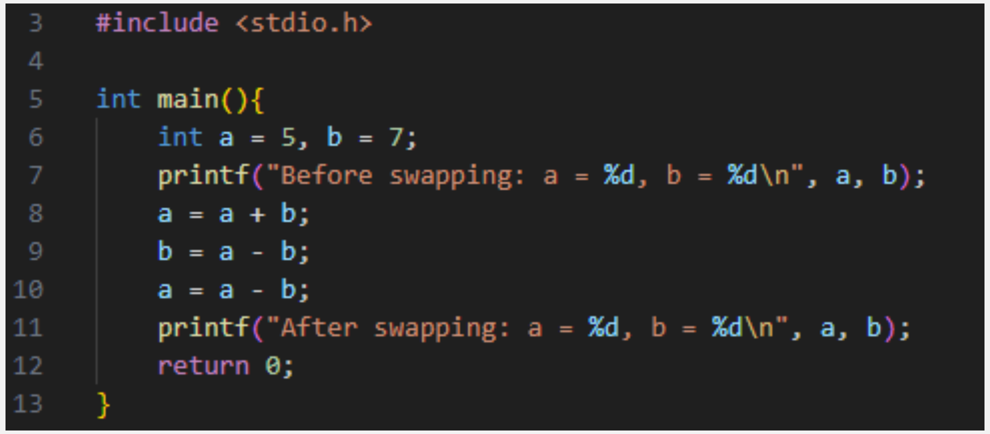


Swap two numbers without using a 3rd variable

```
int a = 5;
int b = 7;

a = a + b;

b = a - b;

a = a - b;
```

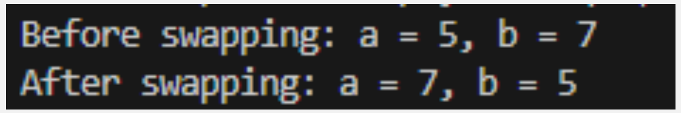


Code



```c
#include <stdio.h>

int main(){
    int a = 5, b = 7;
    printf("Before swapping: a = %d, b = %d\n", a, b);
    a = a + b;
    b = a - b;
    a = a - b;
    printf("After swapping: a = %d, b = %d\n", a, b);
    return 0;
}
```

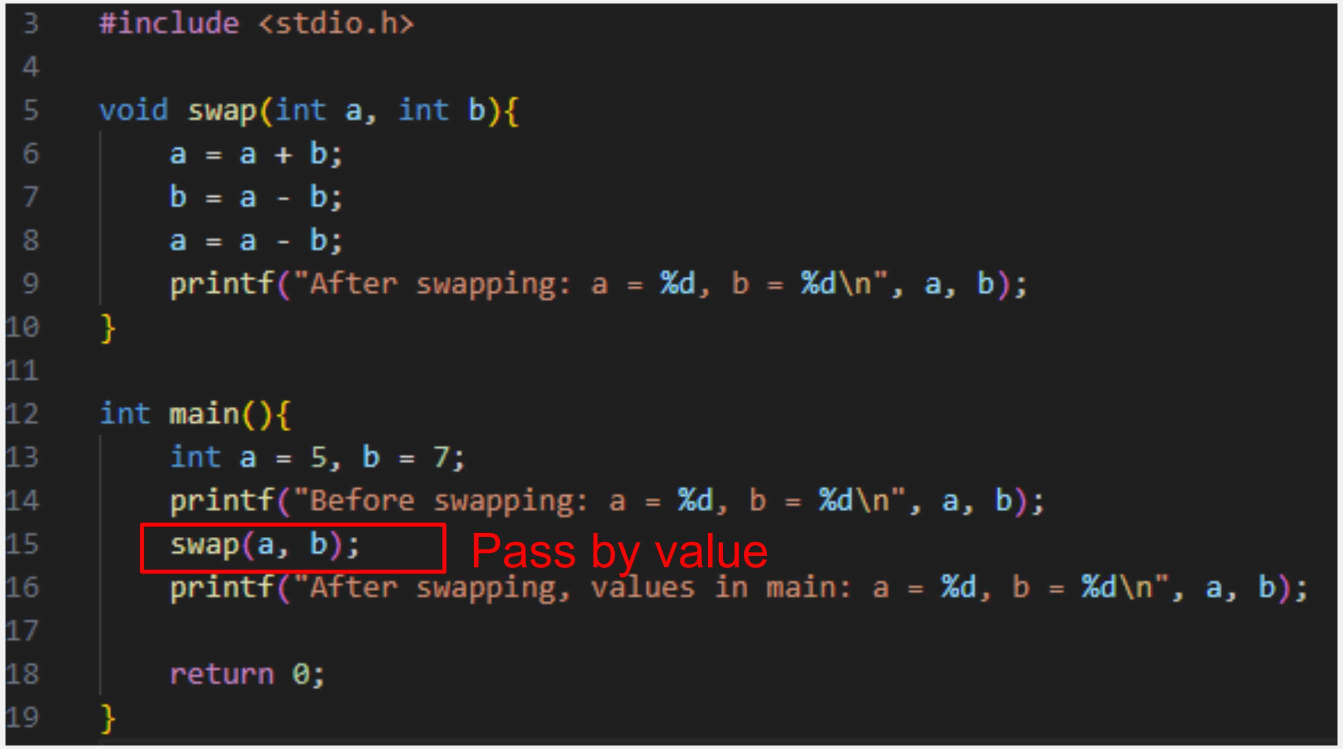
Output



```
Before swapping: a = 5, b = 7
After swapping: a = 7, b = 5
```

Pass by value and Pass by reference

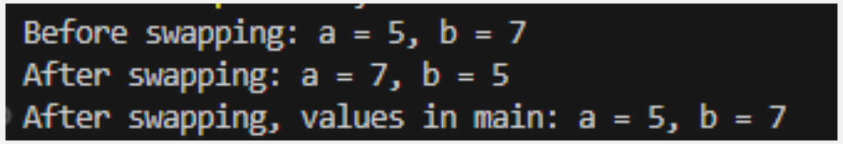
- Swap function



```
#include <stdio.h>

void swap(int a, int b){
    a = a + b;
    b = a - b;
    a = a - b;
    printf("After swapping: a = %d, b = %d\n", a, b);
}

int main(){
    int a = 5, b = 7;
    printf("Before swapping: a = %d, b = %d\n", a, b);
    swap(a, b);
    printf("After swapping, values in main: a = %d, b = %d\n", a, b);

    return 0;
}
```

Pass by value

Output



```
Before swapping: a = 5, b = 7
After swapping: a = 7, b = 5
After swapping, values in main: a = 5, b = 7
```

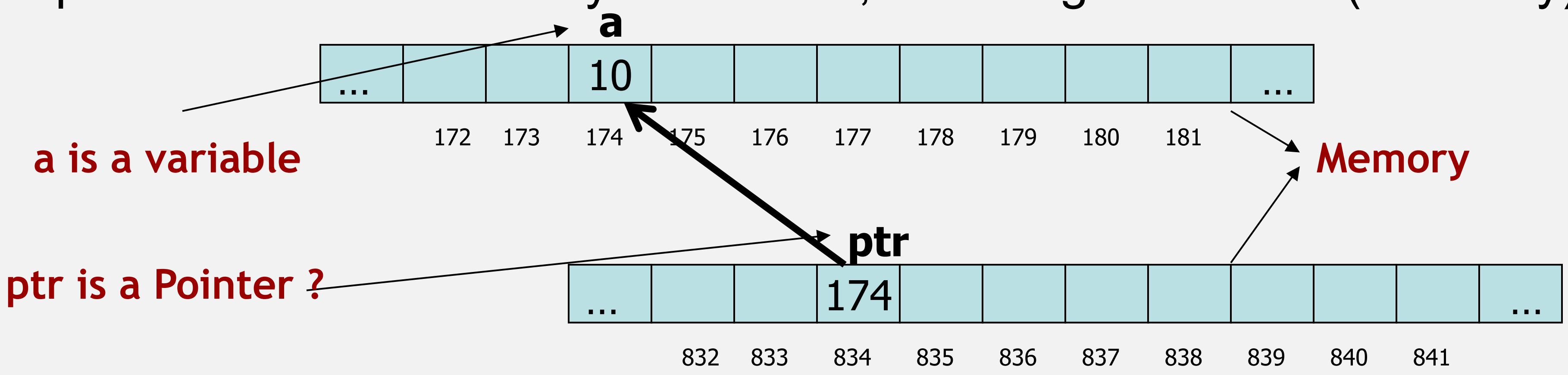
Pass by value and Pass by reference

- Instead of value, we pass the address

- How?

Pointer Variable Declaration and Initialization

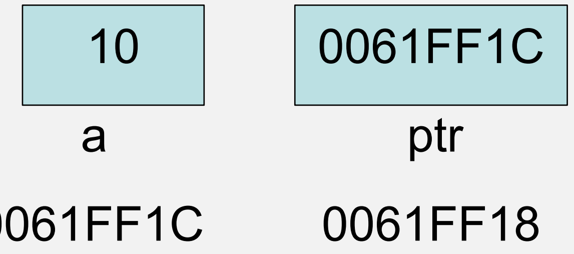
- **Why we need Pointers?**
  - Pointers enable programs to create dynamic memory, which can grow at runtime.
- The main difference between normal variables and pointers is that **variables contain a value**, while **pointers contain a memory address**.
- **Memory address:**
  - We know each memory address contain some value.
  - Thus, if pointers contain memory addresses, we can get its value (indirectly).

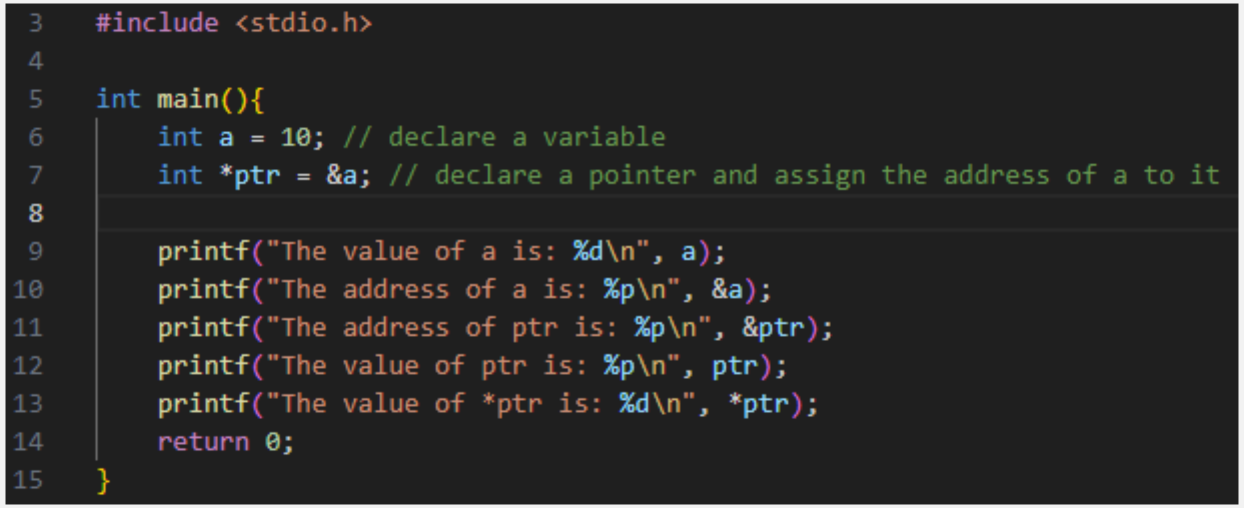


Pointers

- Pointers store address of a variable

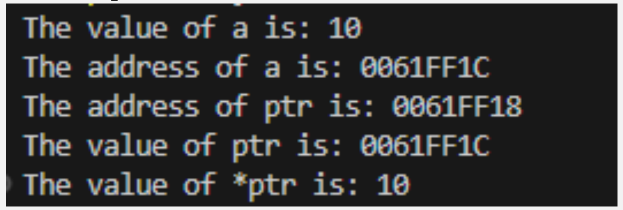
```
int a = 10;
int *ptr = &a;
```





```c
#include <stdio.h>

int main(){
    int a = 10; // declare a variable
    int *ptr = &a; // declare a pointer and assign the address of a to it

    printf("The value of a is: %d\n", a);
    printf("The address of a is: %p\n", &a);
    printf("The address of ptr is: %p\n", &ptr);
    printf("The value of ptr is: %p\n", ptr);
    printf("The value of *ptr is: %d\n", *ptr);
    return 0;
}
```

Output



```
The value of a is: 10
The address of a is: 0061FF1C
The address of ptr is: 0061FF18
The value of ptr is: 0061FF1C
The value of *ptr is: 10
```

Pointer Declaration

- A pointer variable must be declared before it can be used.
- Examples of pointer declarations:

```
int  *a;
float *b;
char *c;
```

- The asterisk, when used as above in the declaration, tells the compiler that the variable is to be a pointer, and the type of data (the value that we get indirectly) that the pointer points to.

Pointer Referencing

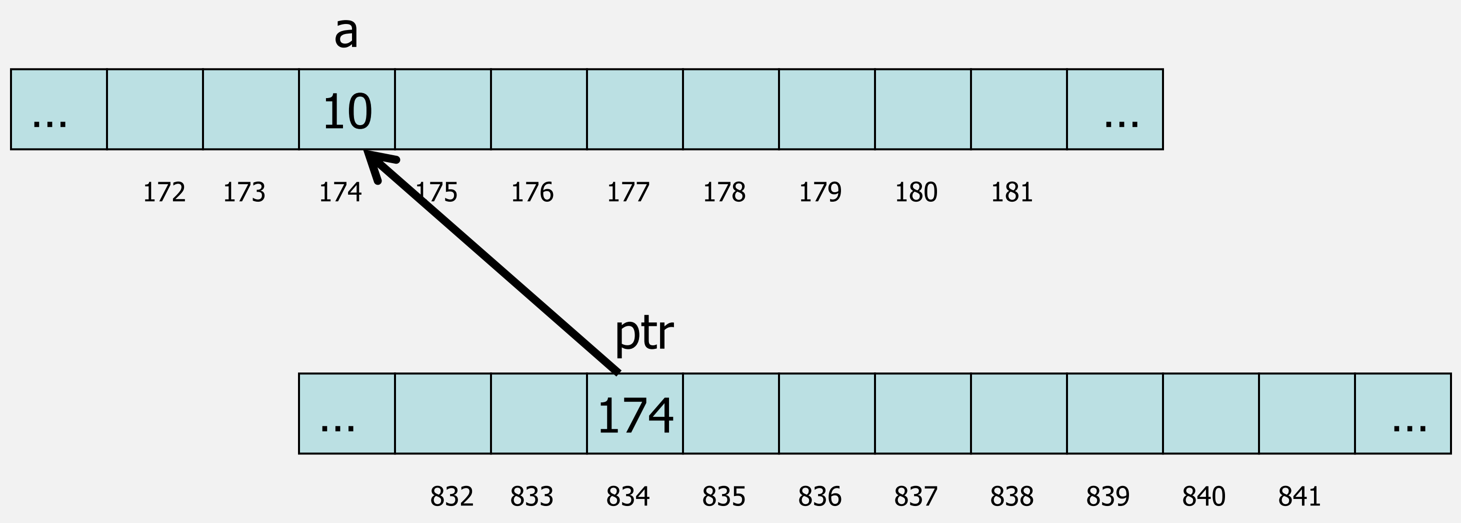
- The unary operator & gives the address of a variable
- The statement

```
ptr = &a;
```

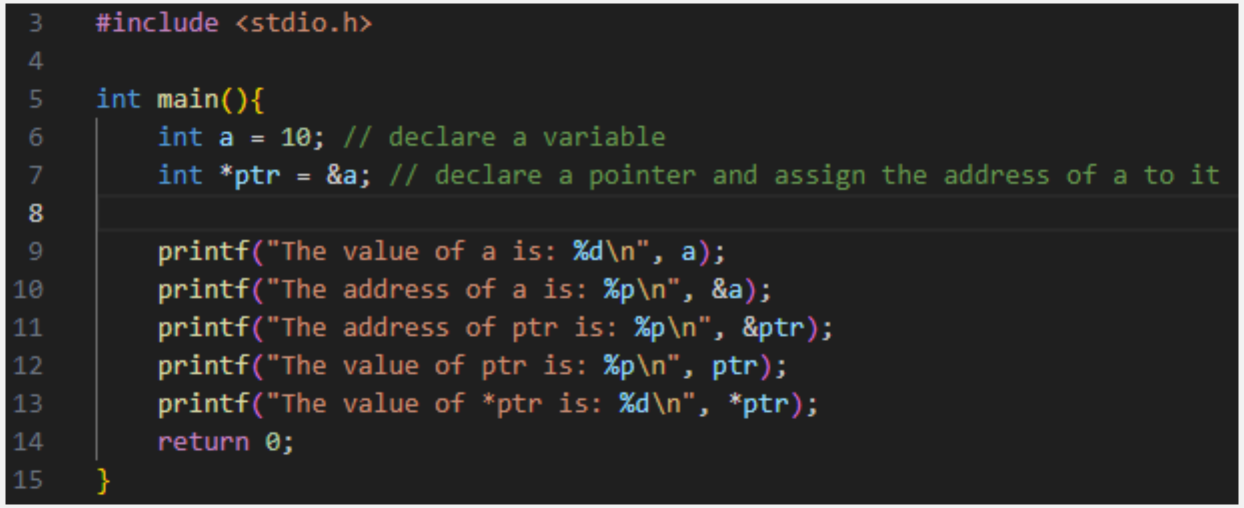
- assigns the address of **a** to the variable **ptr**, and now **ptr** points to **a**
- To print a pointer, use %p format.

Pointer Referencing

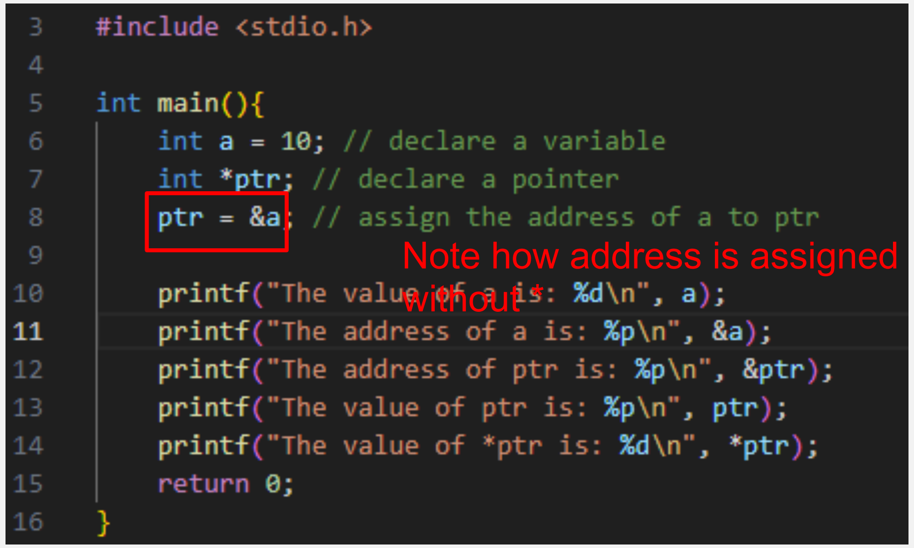
```
int a;
int *ptr; /* Declare ptr as a pointer to int */
a = 10;
ptr = &a;
```



Pointer Referencing



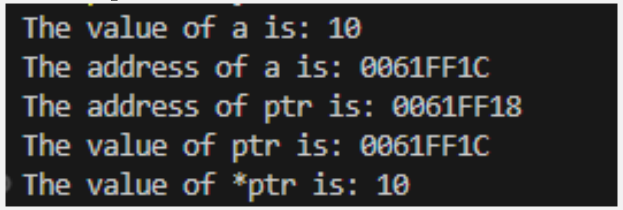
```c
#include <stdio.h>

int main(){
    int a = 10; // declare a variable
    int *ptr = &a; // declare a pointer and assign the address of a to it

    printf("The value of a is: %d\n", a);
    printf("The address of a is: %p\n", &a);
    printf("The address of ptr is: %p\n", &ptr);
    printf("The value of ptr is: %p\n", ptr);
    printf("The value of *ptr is: %d\n", *ptr);
    return 0;
}
```



```c
#include <stdio.h>

int main(){
    int a = 10; // declare a variable
    int *ptr; // declare a pointer
    ptr = &a; // assign the address of a to ptr

    printf("The value of a is: %d\n", a);
    printf("The address of a is: %p\n", &a);
    printf("The address of ptr is: %p\n", &ptr);
    printf("The value of ptr is: %p\n", ptr);
    printf("The value of *ptr is: %d\n", *ptr);
    return 0;
}
```

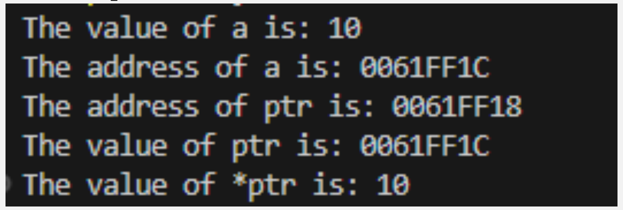
Note how address is assigned without *

Output



```
The value of a is: 10
The address of a is: 0061FF1C
The address of ptr is: 0061FF18
The value of ptr is: 0061FF1C
The value of *ptr is: 10
```

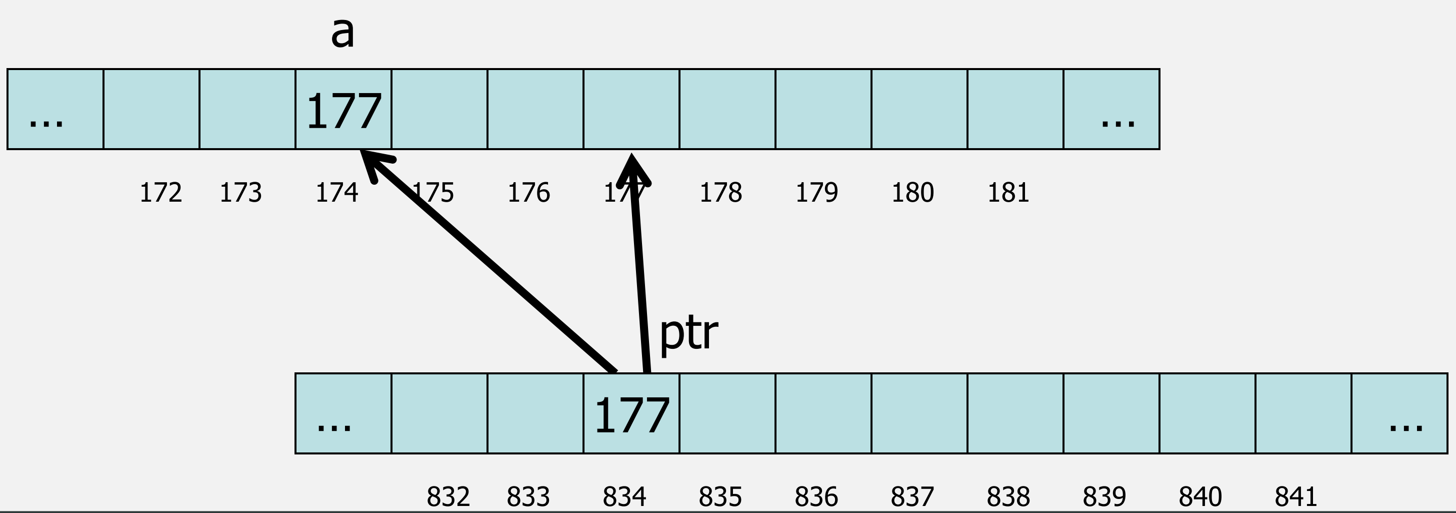
Output



```
The value of a is: 10
The address of a is: 0061FF1C
The address of ptr is: 0061FF18
The value of ptr is: 0061FF1C
The value of *ptr is: 10
```

Dereferencing

```
printf(“%d”, *ptr); /* Prints out ‘10’ */
*ptr = 177;
printf(“%d”, a); /* Prints out ‘177’ */
ptr = 177; /* This is unadvisable!! */
```



Pointer Dereferencing

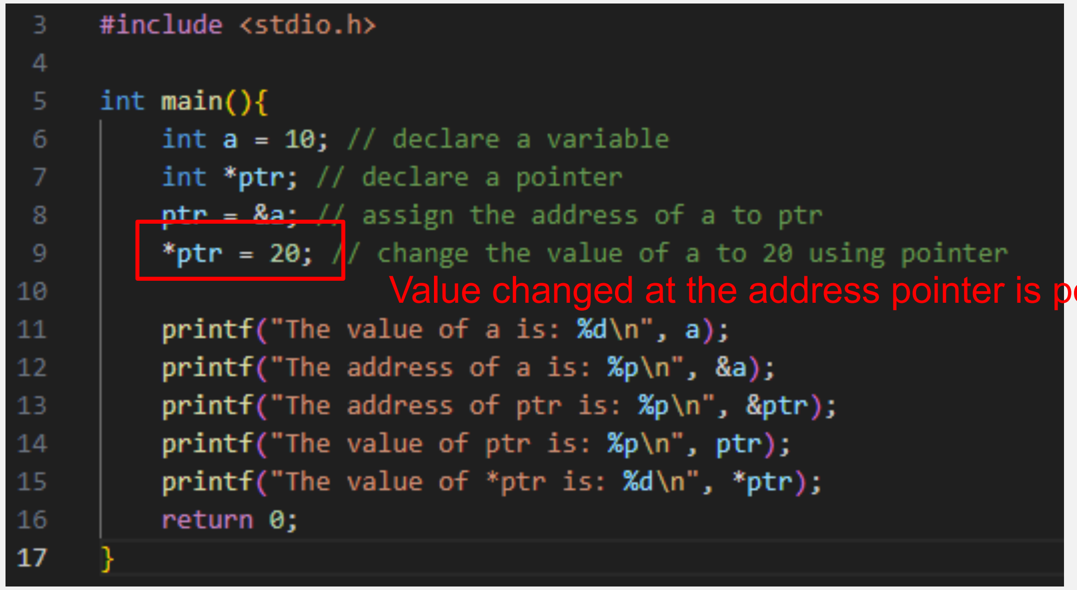
- The unary operator `*` is the dereferencing operator
  - Applied on pointers
  - Access the value of object the pointer points to

- The statement

```
*ptr = 20;
```

puts in **a** (the variable pointed by **ptr**) the value 20

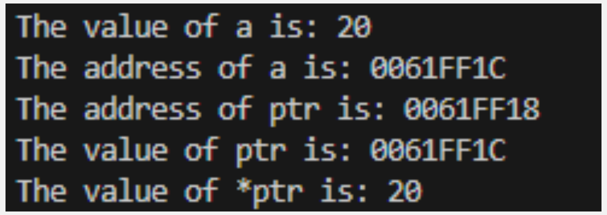
Pointer Dereferencing



```c
#include <stdio.h>

int main(){
    int a = 10; // declare a variable
    int *ptr; // declare a pointer
    ptr = &a; // assign the address of a to ptr
    *ptr = 20; // change the value of a to 20 using pointer

    printf("The value of a is: %d\n", a);
    printf("The address of a is: %p\n", &a);
    printf("The address of ptr is: %p\n", &ptr);
    printf("The value of ptr is: %p\n", ptr);
    printf("The value of *ptr is: %d\n", *ptr);
    return 0;
}
```

Value changed at the address pointer is pointing to

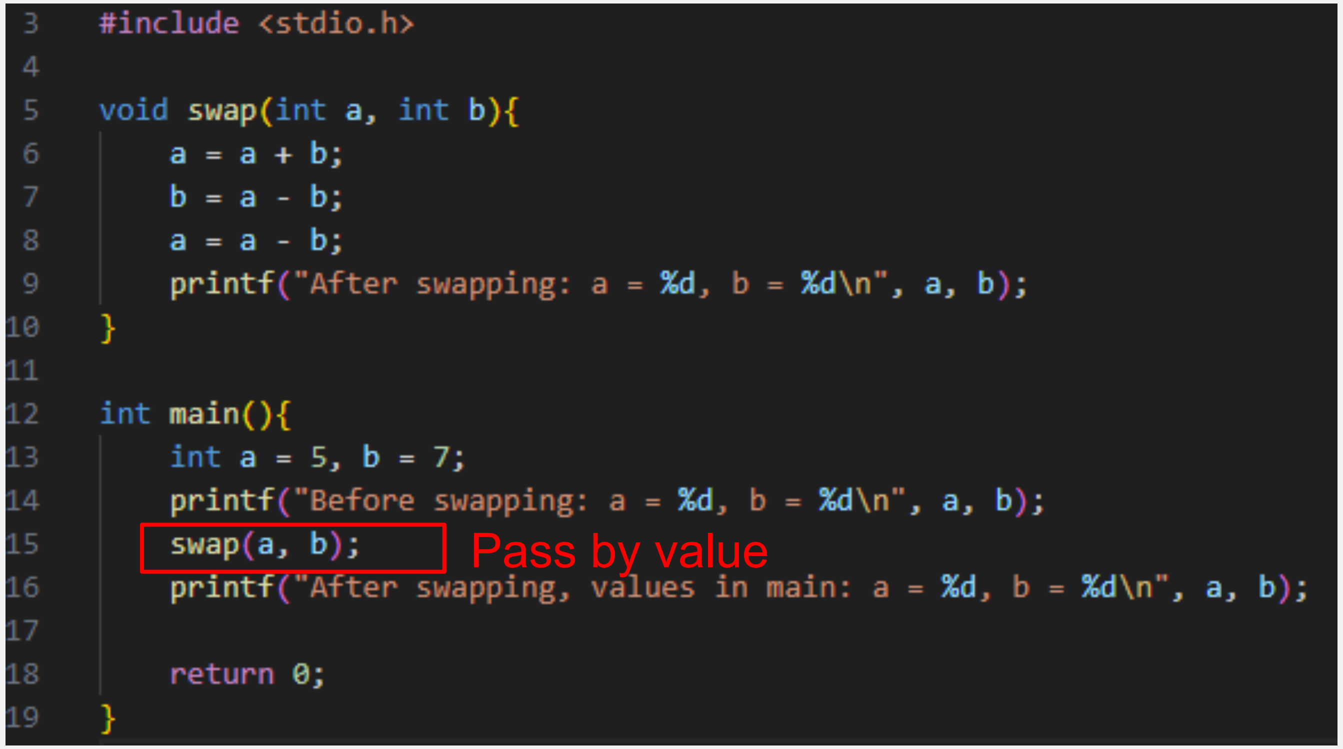
Output



```
The value of a is: 20
The address of a is: 0061FF1C
The address of ptr is: 0061FF18
The value of ptr is: 0061FF1C
The value of *ptr is: 20
```

Pass by value and Pass by reference

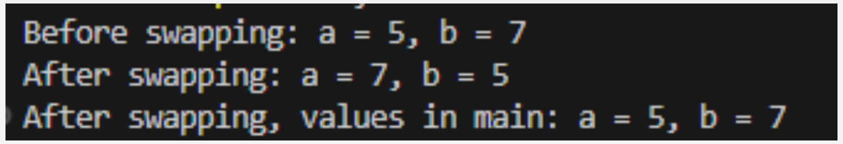
- Swap function



```
#include <stdio.h>

void swap(int a, int b){
    a = a + b;
    b = a - b;
    a = a - b;
    printf("After swapping: a = %d, b = %d\n", a, b);
}

int main(){
    int a = 5, b = 7;
    printf("Before swapping: a = %d, b = %d\n", a, b);
    swap(a, b);
    printf("After swapping, values in main: a = %d, b = %d\n", a, b);

    return 0;
}
```

Pass by value

Output



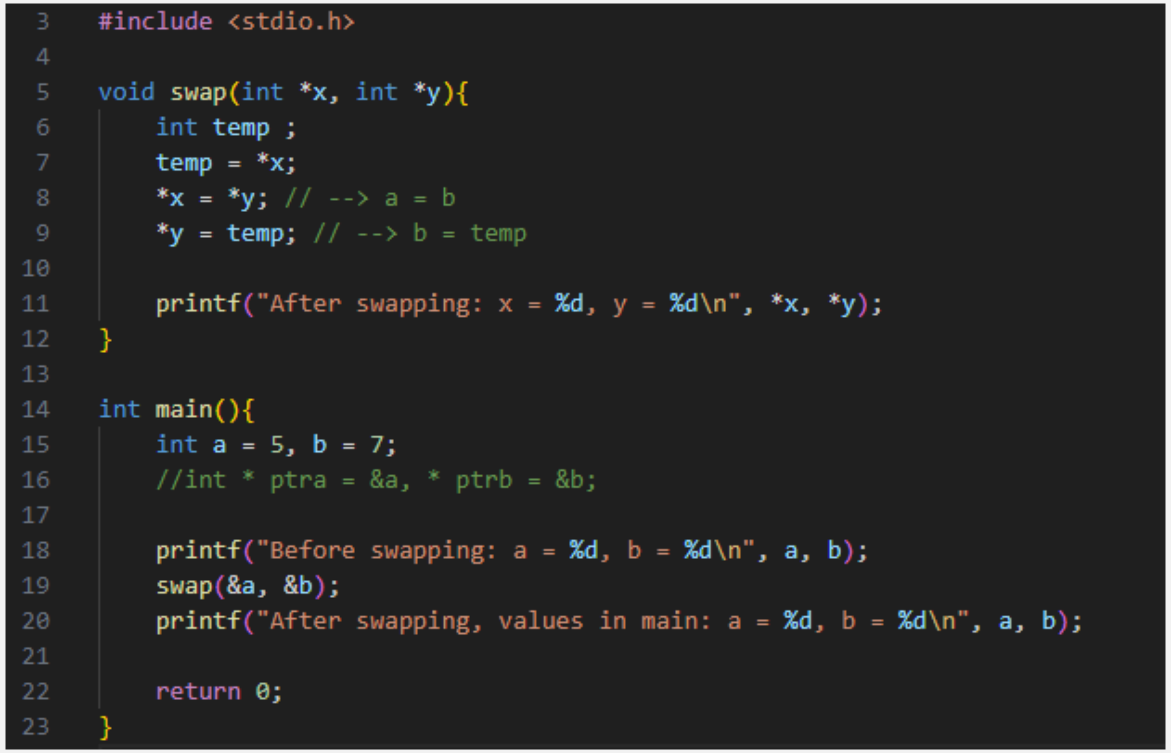
```
Before swapping: a = 5, b = 7
After swapping: a = 7, b = 5
After swapping, values in main: a = 5, b = 7
```

Pointers (Swap two numbers)

Pass by Reference

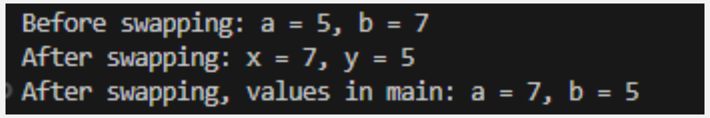
Pointers (Swap two numbers)

Pass by Reference



```c
#include <stdio.h>

void swap(int *x, int *y){
    int temp ;
    temp = *x;
    *x = *y; // --> a = b
    *y = temp; // --> b = temp

    printf("After swapping: x = %d, y = %d\n", *x, *y);
}

int main(){
    int a = 5, b = 7;
    //int * ptra = &a, * ptrb = &b;

    printf("Before swapping: a = %d, b = %d\n", a, b);
    swap(&a, &b);
    printf("After swapping, values in main: a = %d, b = %d\n", a, b);

    return 0;
}
```

Output



```
Before swapping: a = 5, b = 7
After swapping: x = 7, y = 5
After swapping, values in main: a = 7, b = 5
```

Exercise

- Perform number swab with pointers without the use of a 3rd variable.

Assessment 3:

- 30th Oct 2023
- Timed 30 minutes exercise
- Two problems to solve

Important to revise and practice all topics covered upto arrays. Make sure you understand and can solve practice and lab exercises.