Pointers

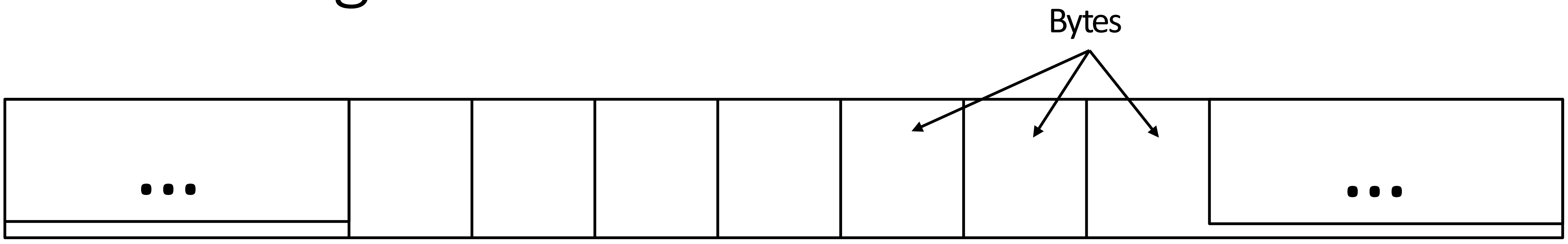
Computer Memory

- Random Access Memory (RAM)
  - The time it takes to access a given element in RAM is the same for any other random element in memory
- Store data for running programs
- All variables and arrays are stored in RAM
- Every byte (group of 8 bits) in memory has an address
  - Like one big array where each address is an index to a byte of storage space
- In C we can get the address of a variable using the & operator (address operator)

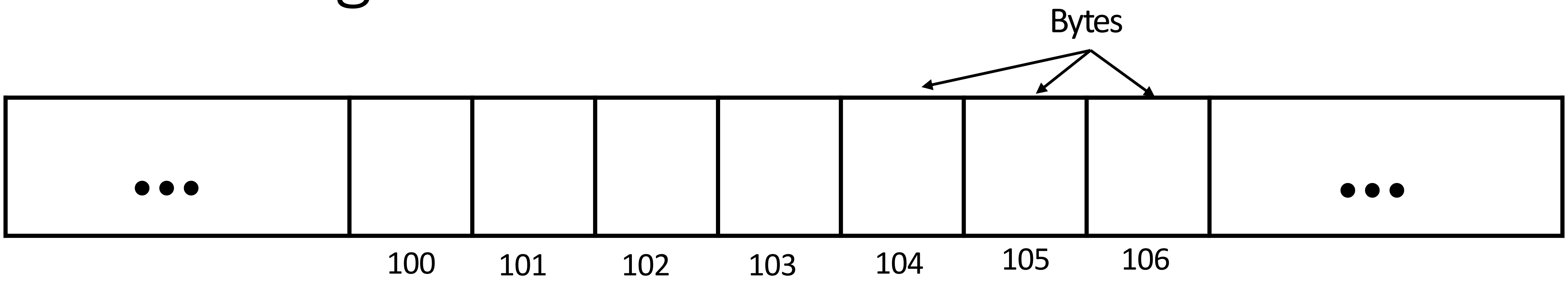
Addressing in RAM

…								…

Addressing in RAM



Addressing in RAM



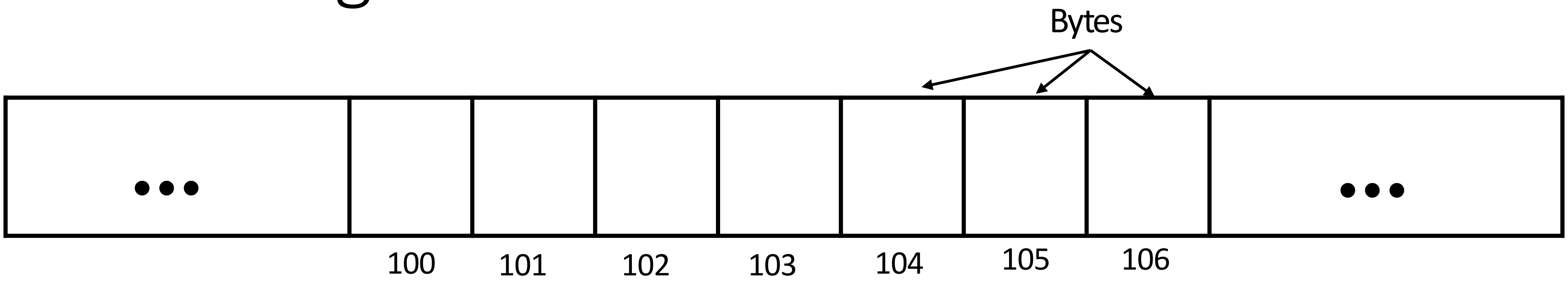
Bytes

…

100 101 102 103 104 105 106

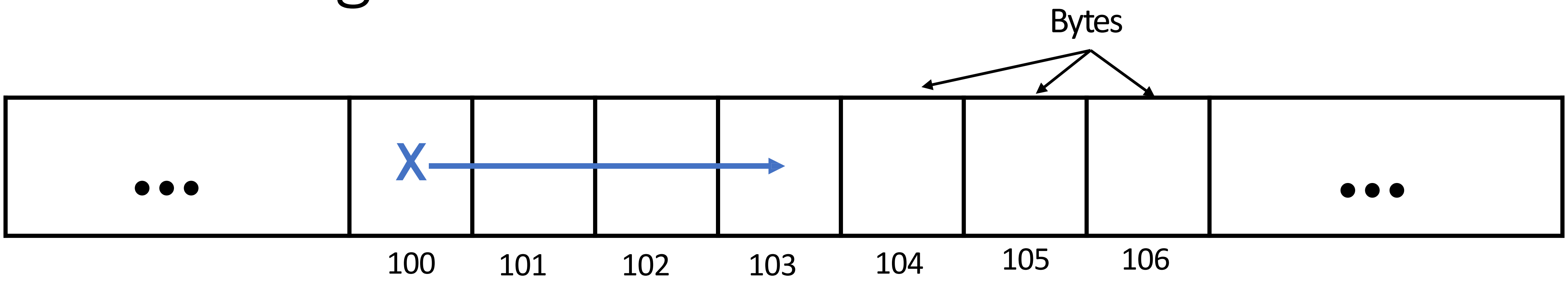
…

Addressing in RAM



```
int x;
```

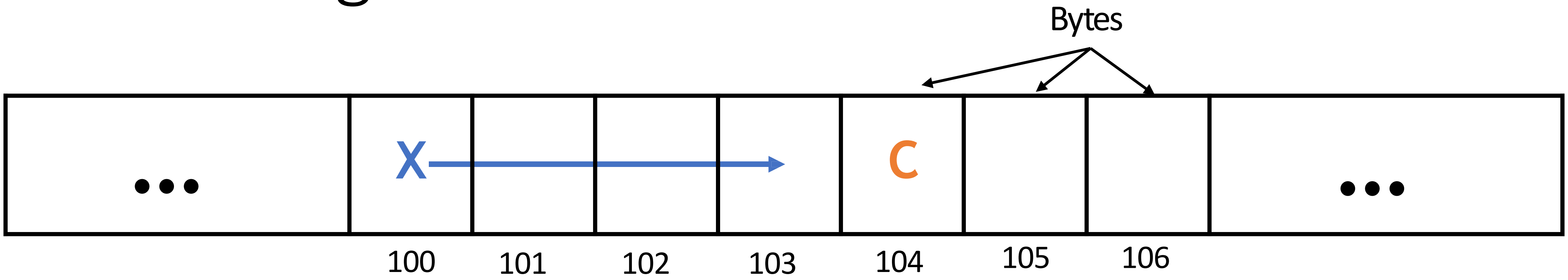
Addressing in RAM



int x; //32 bits or 4 bytes

Assuming x is stored at100.

Addressing in RAM



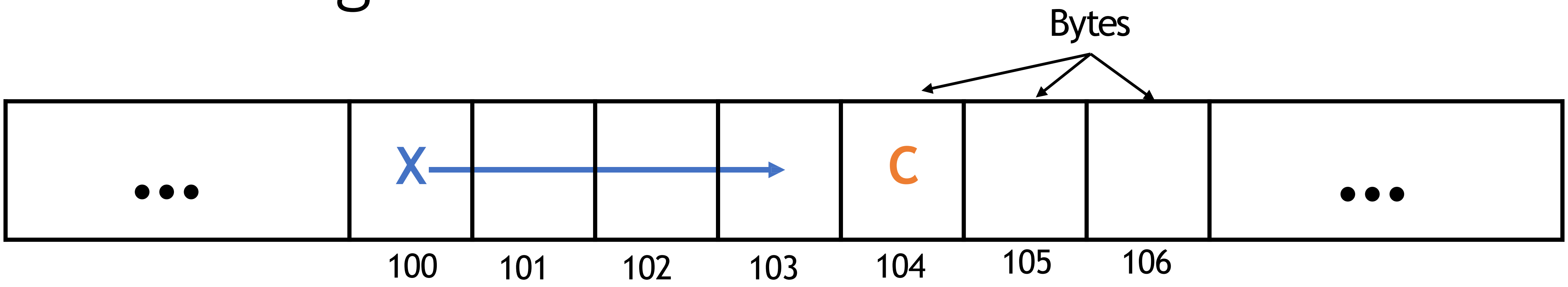
int x; //32 bits or 4 bytes

Assuming x is stored at 100.

char c; //8 bits or 1 byte

Assuming c is stored at 104.

Addressing in RAM



int x; //32 bits or 4 bytes

Assuming x is stored at 100.

char c; //8 bits or 1 byte

Assuming c is stored at 104.

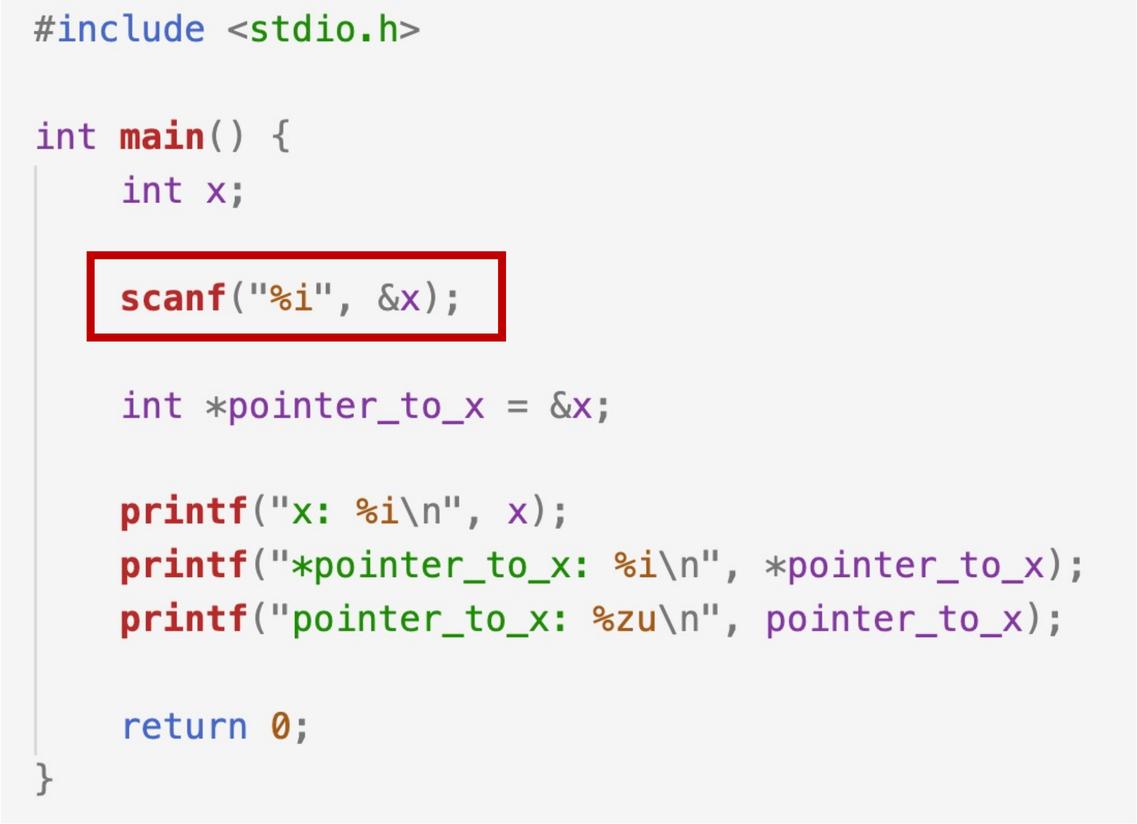
NOTE: Variables can be stored at any address, and in most cases, we do not have to worry about what specific address number isused.

Pointers

A pointer is a variable that stores an address (a number which is a location in memory, or RAM)

- Through a pointer, a value can be accessed indirectly by its address rather than by its original name

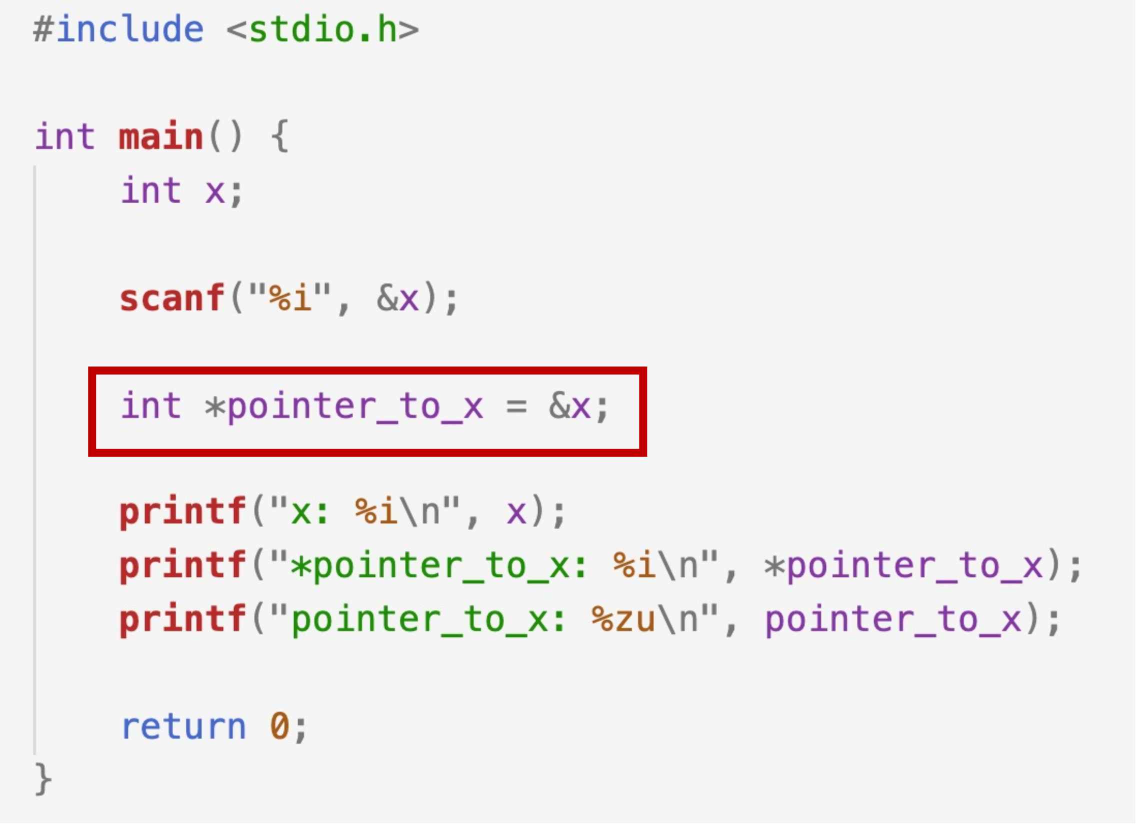
Pointer Intro



```
#include <stdio.h>

int main() {
    int x;

    scanf("%i", &x);

    int *pointer_to_x = &x;

    printf("x: %i\n", x);
    printf("*pointer_to_x: %i\n", *pointer_to_x);
    printf("pointer_to_x: %zu\n", pointer_to_x);

    return 0;
}
```

- Using the & operator gets the address of a variable.
- When we call scanf(), we are allowing scanf() to access x using its address

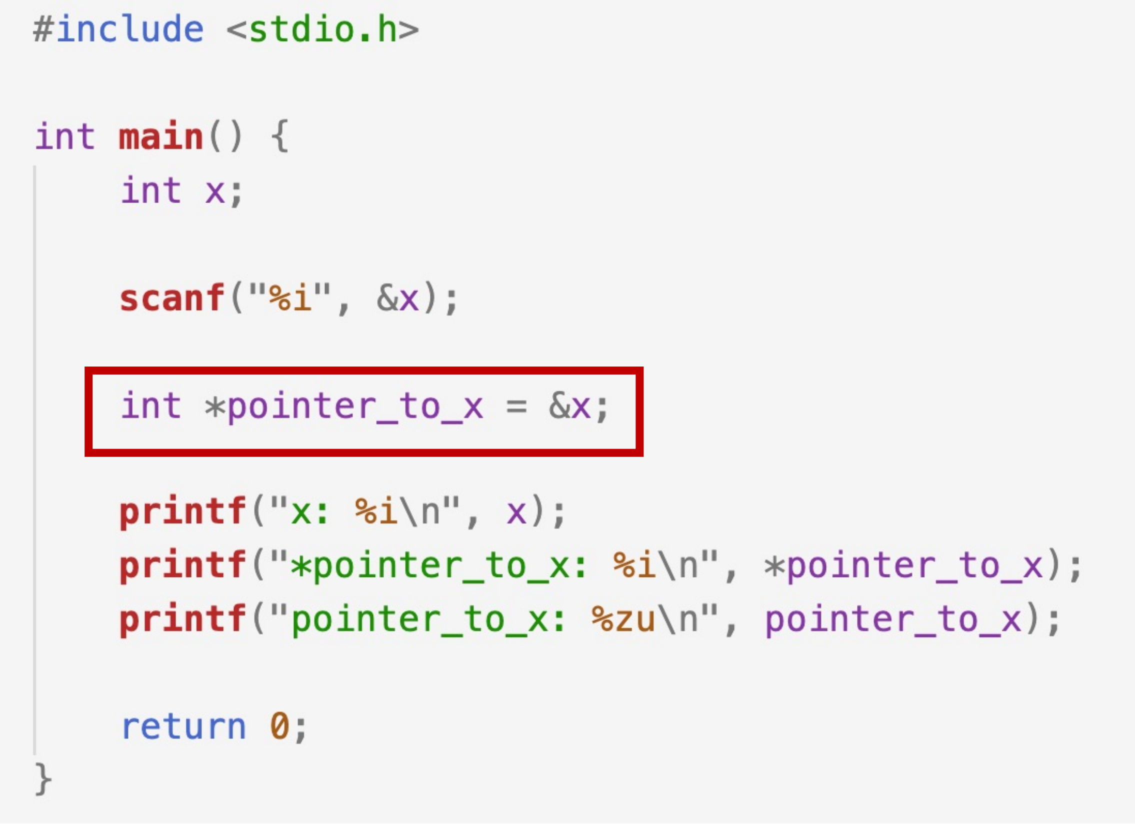
Pointer Intro



```c
#include <stdio.h>

int main() {
    int x;

    scanf("%i", &x);

    int *pointer_to_x = &x;

    printf("x: %i\n", x);
    printf("*pointer_to_x: %i\n", *pointer_to_x);
    printf("pointer_to_x: %zu\n", pointer_to_x);

    return 0;
}
```

- Declare pointers with the type of the data it will point to and an *
- Declare a pointer pointer_to_x and make it point to x:
  - int *pointer_to_x = &x;

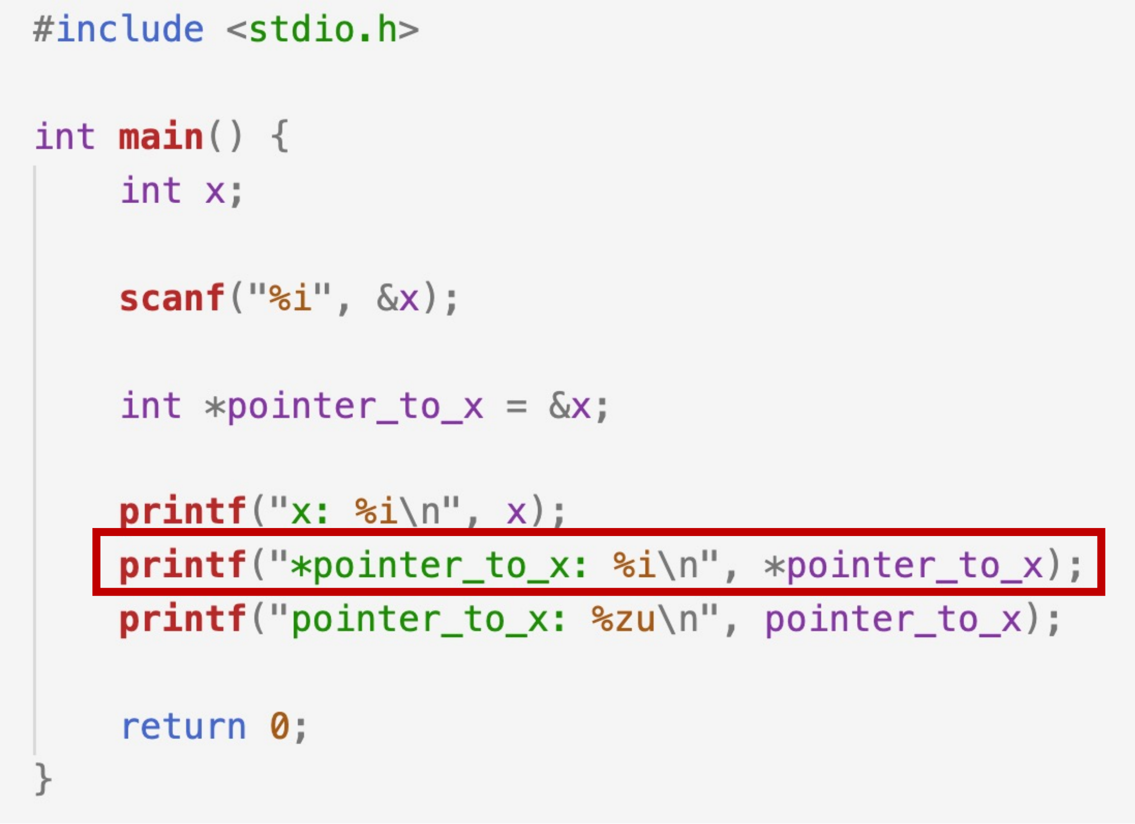
Pointer Intro



```
#include <stdio.h>

int main() {
    int x;

    scanf("%i", &x);

    int *pointer_to_x = &x;

    printf("x: %i\n", x);
    printf("*pointer_to_x: %i\n", *pointer_to_x);
    printf("pointer_to_x: %zu\n", pointer_to_x);

    return 0;
}
```

- Several syntax options:
  - int *pointer_to_x;
  - int* pointer_to_x;
  - int * pointer_to_x;
- Assign the address of variable x to pointer_to_x
  - pointer_to_x = &x;

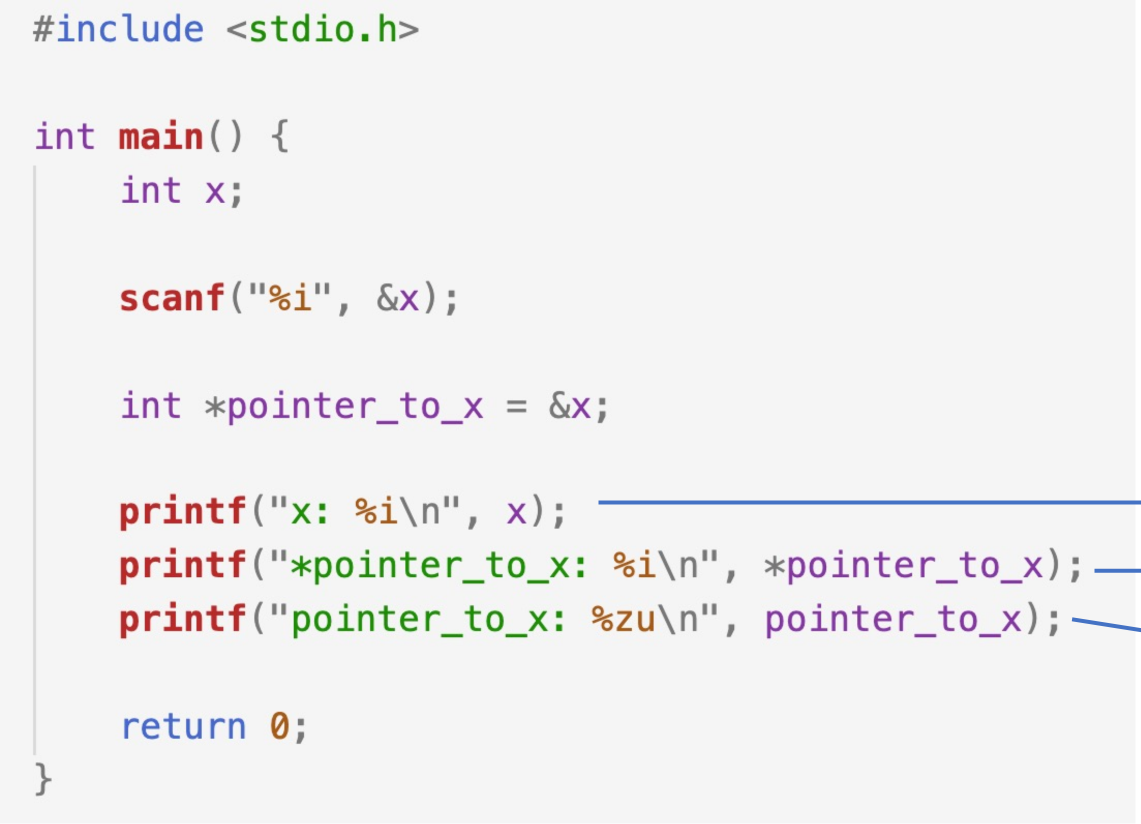
Pointer Intro



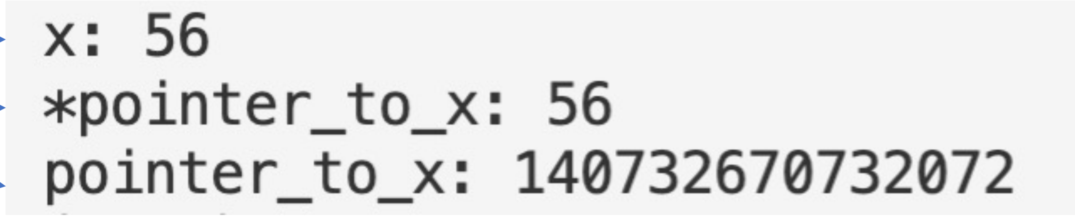
```c
#include <stdio.h>

int main() {
    int x;

    scanf("%i", &x);

    int *pointer_to_x = &x;

    printf("x: %i\n", x);
    printf("*pointer_to_x: %i\n", *pointer_to_x);
    printf("pointer_to_x: %zu\n", pointer_to_x);

    return 0;
}
```

- Dereferencing
  - Access the data located at the address stored by a pointer
  - * pointer_to_x = 5;
  - If pointer_to_x is a pointer to x, *pointer_to_x is the value of x

Pointer Intro

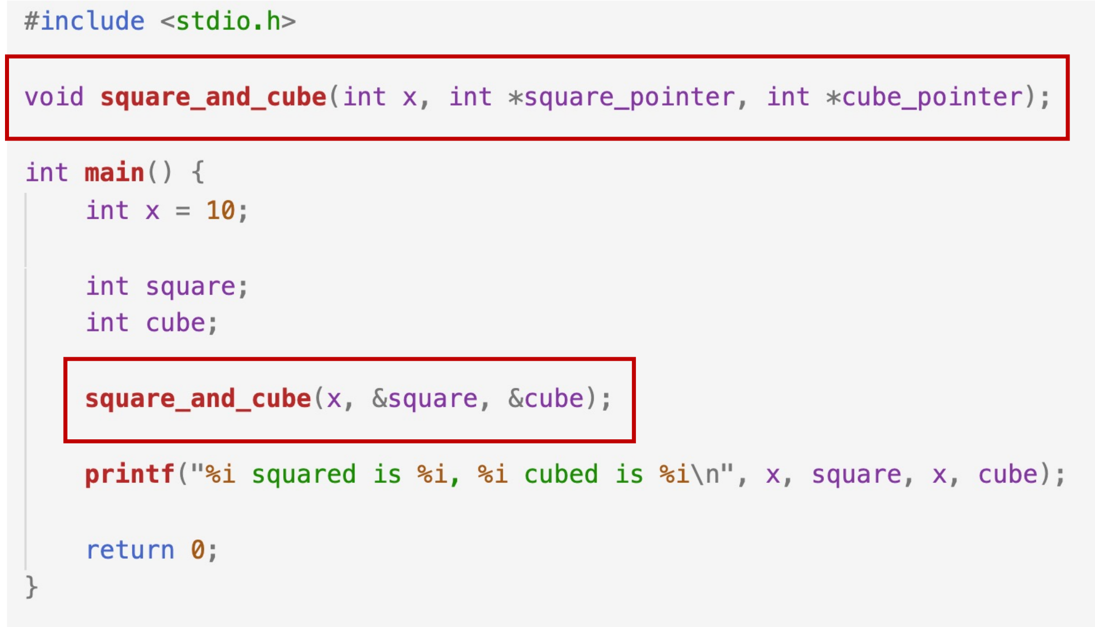


```
#include <stdio.h>

int main() {
    int x;

    scanf("%i", &x);

    int *pointer_to_x = &x;

    printf("x: %i\n", x);
    printf("*pointer_to_x: %i\n", *pointer_to_x);
    printf("pointer_to_x: %zu\n", pointer_to_x);

    return 0;
}
```



```
x: 56
*pointer_to_x: 56
pointer_to_x: 140732670732072
```

Pointer Parameters



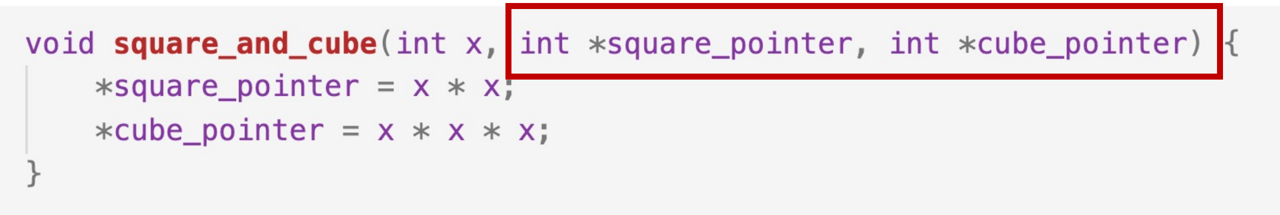
```
#include <stdio.h>

void square_and_cube(int x, int *square_pointer, int *cube_pointer);

int main() {
    int x = 10;

    int square;
    int cube;

    square_and_cube(x, &square, &cube);

    printf("%i squared is %i, %i cubed is %i\n", x, square, x, cube);

    return 0;
}
```

- void **square_and_cube**(int x,

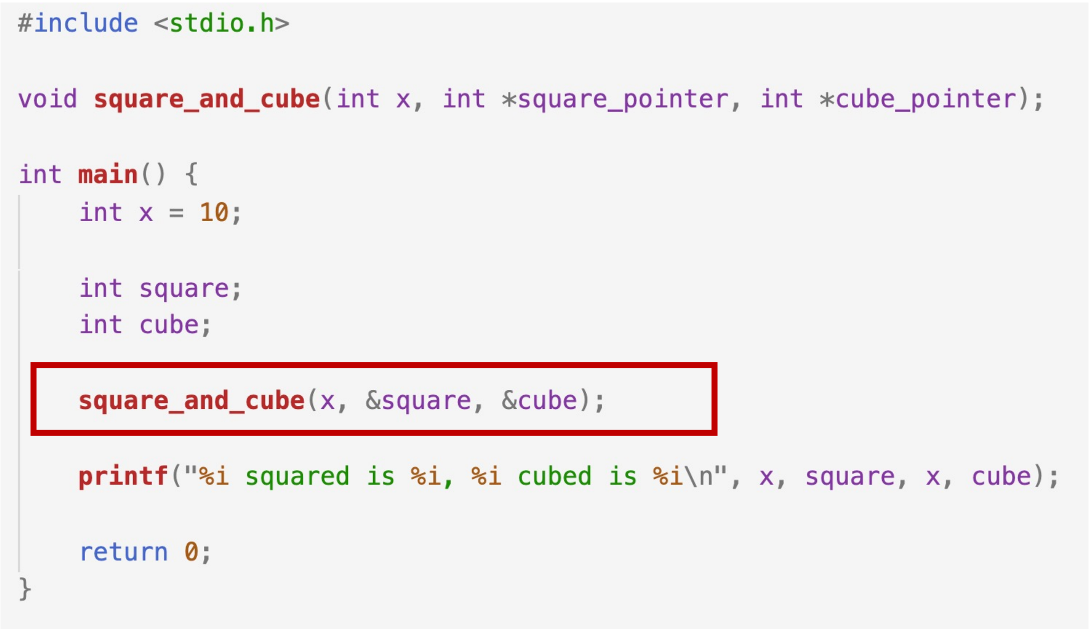
  int *square_pointer,

  int *cube_pointer)
  - Calculate both the square and cube of x, and return them via pointers

Pointer Parameters



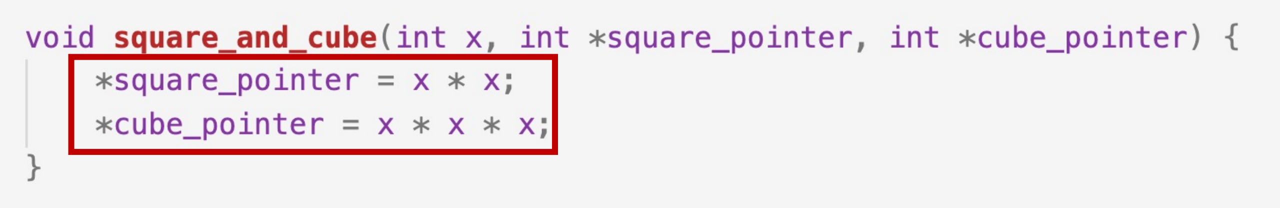
```c
void square_and_cube(int x, int *square_pointer, int *cube_pointer) {
    *square_pointer = x * x;
    *cube_pointer = x * x * x;
}
```



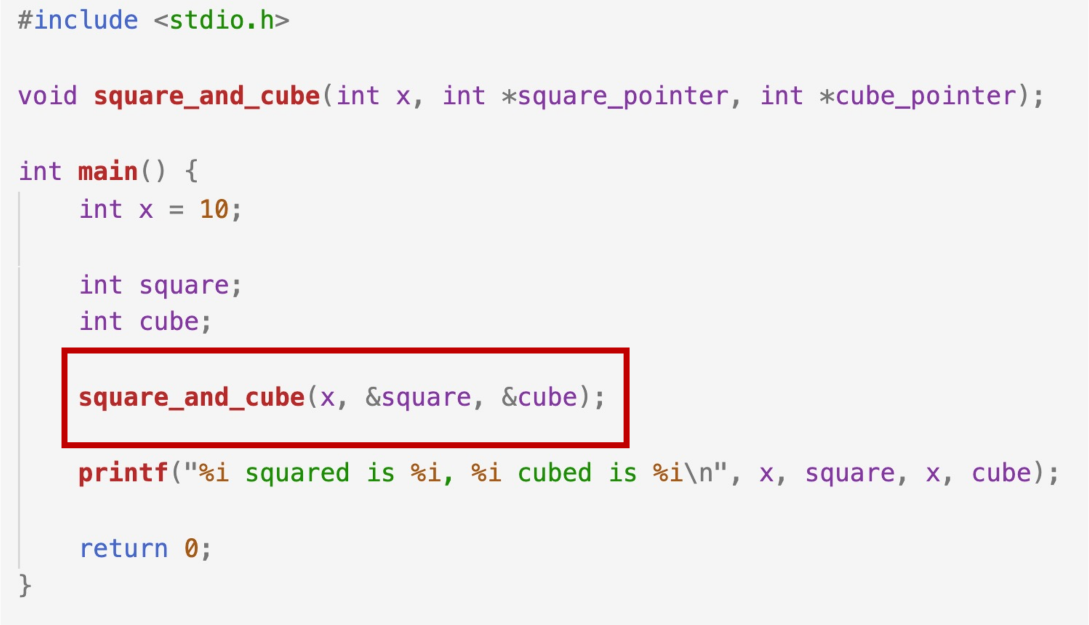
```c
#include <stdio.h>

void square_and_cube(int x, int *square_pointer, int *cube_pointer);

int main() {
    int x = 10;

    int square;
    int cube;

    square_and_cube(x, &square, &cube);

    printf("%i squared is %i, %i cubed is %i\n", x, square, x, cube);

    return 0;
}
```

- Pointer parameters
  - int *square_pointer and int *cube_pointer indicate that these parameters are pointers to integers

Pointer Parameters



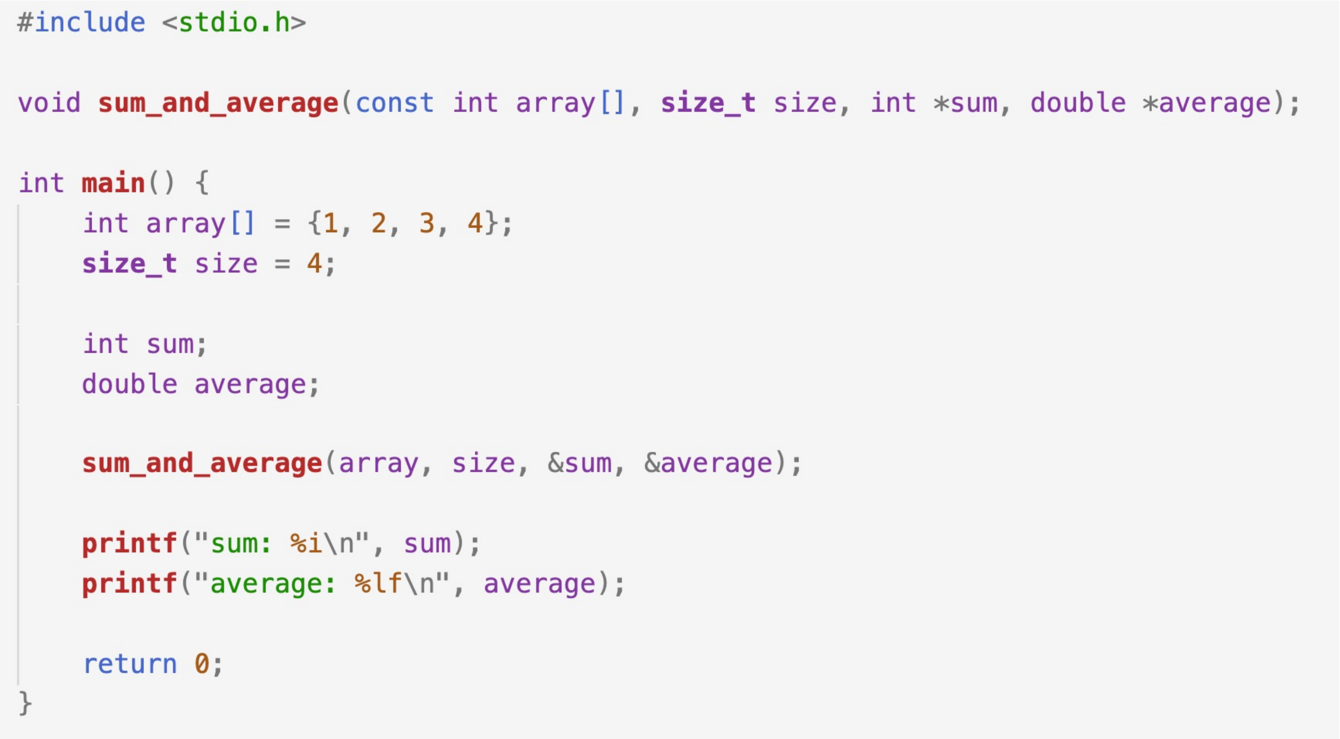
```c
void square_and_cube(int x, int *square_pointer, int *cube_pointer) {
    *square_pointer = x * x;
    *cube_pointer = x * x * x;
}
```



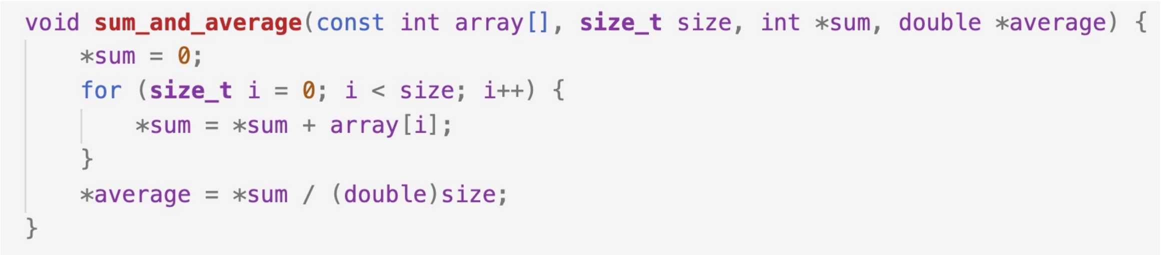
```c
#include <stdio.h>

void square_and_cube(int x, int *square_pointer, int *cube_pointer);

int main() {
    int x = 10;

    int square;
    int cube;

    square_and_cube(x, &square, &cube);

    printf("%i squared is %i, %i cubed is %i\n", x, square, x, cube);

    return 0;
}
```

- Dereferencing pointer variables
  - *square_pointer* points to the square variable
  - dereferencing the pointer variable will let us assign the values of *x * x* and *x * x * x* to *square* and *cube* respectively

Pointer Parameters



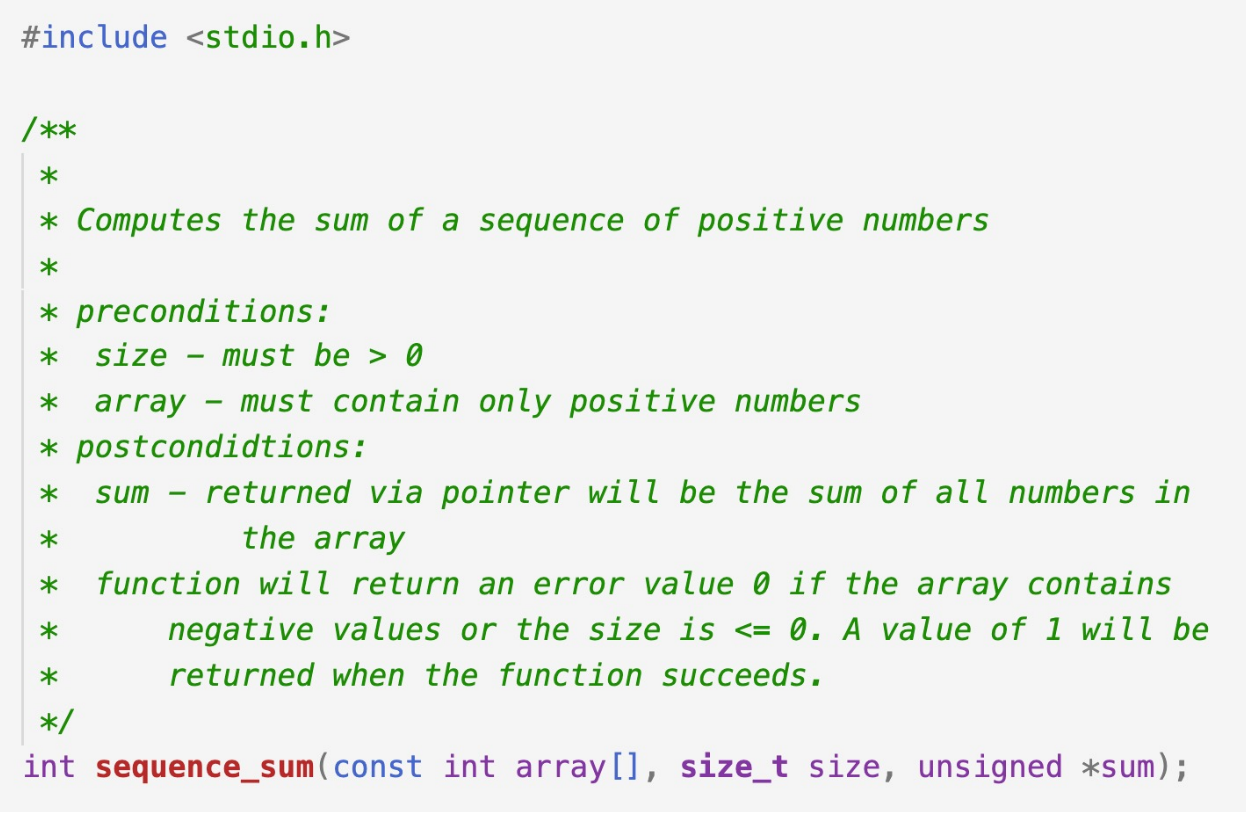
```c
#include <stdio.h>

void sum_and_average(const int array[], size_t size, int *sum, double *average);

int main() {
    int array[] = {1, 2, 3, 4};
    size_t size = 4;

    int sum;
    double average;

    sum_and_average(array, size, &sum, &average);

    printf("sum: %i\n", sum);
    printf("average: %lf\n", average);

    return 0;
}
```



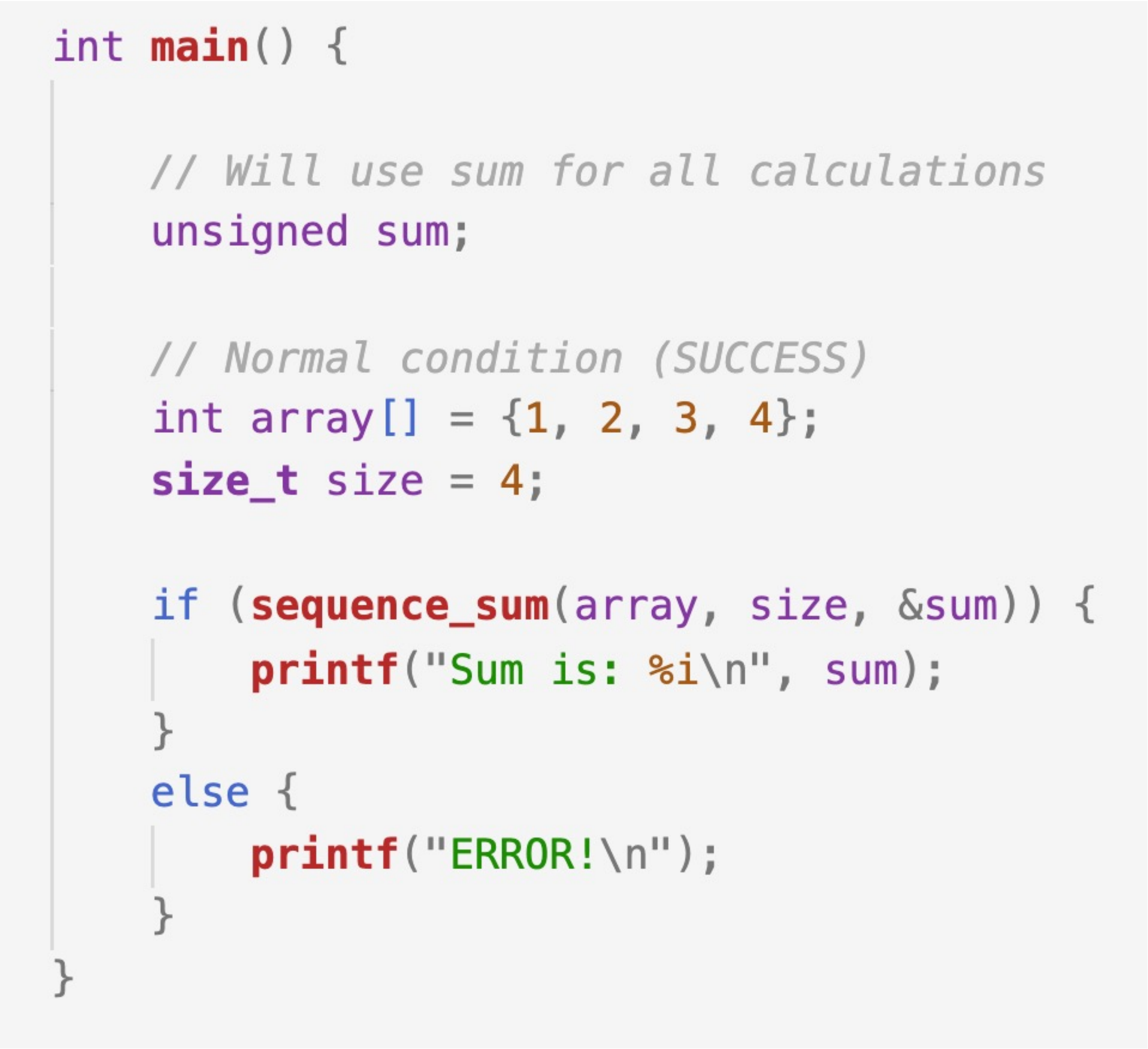
```c
void sum_and_average(const int array[], size_t size, int *sum, double *average) {
    *sum = 0;
    for (size_t i = 0; i < size; i++) {
        *sum = *sum + array[i];
    }
    *average = *sum / (double)size;
}
```

Why return values from functions at all?

sequence_sum()

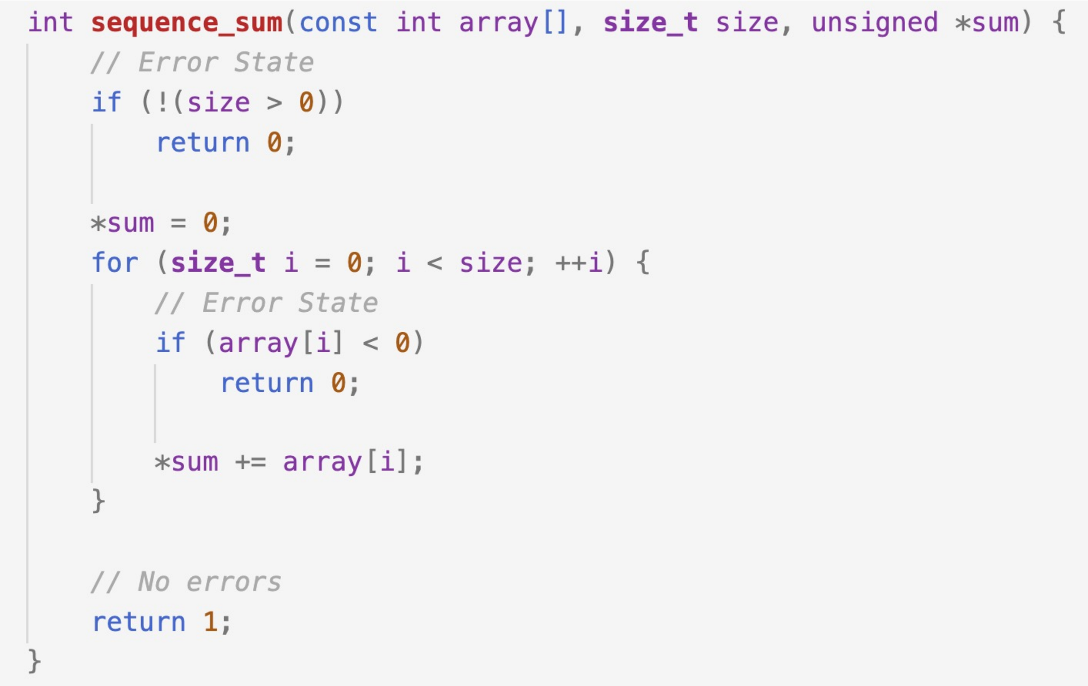


```c
#include <stdio.h>

/**
 *
 * Computes the sum of a sequence of positive numbers
 *
 * preconditions:
 *  size - must be > 0
 *  array - must contain only positive numbers
 * postcondidtions:
 *  sum - returned via pointer will be the sum of all numbers in
 *          the array
 *  function will return an error value 0 if the array contains
 *      negative values or the size is <= 0. A value of 1 will be
 *      returned when the function succeeds.
 */
int sequence_sum(const int array[], size_t size, unsigned *sum);
```

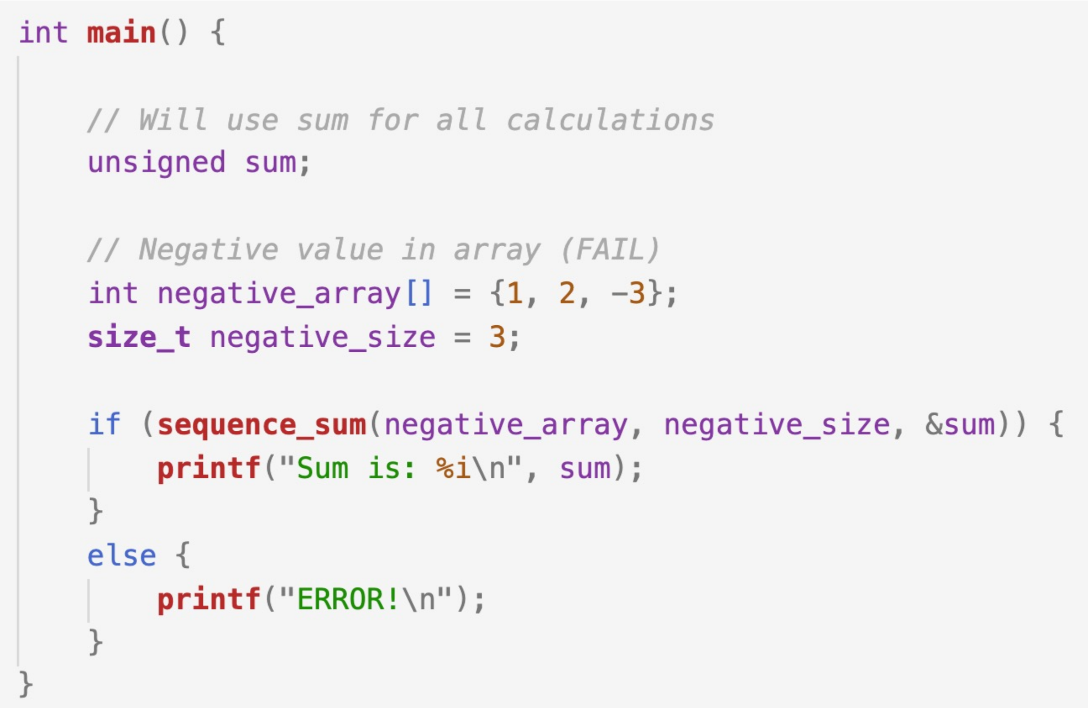


```c
int main() {

    // Will use sum for all calculations
    unsigned sum;

    // Normal condition (SUCCESS)
    int array[] = {1, 2, 3, 4};
    size_t size = 4;

    if (sequence_sum(array, size, &sum)) {
        printf("Sum is: %i\n", sum);
    }
    else {
        printf("ERROR!\n");
    }
}
```

sequence_sum()

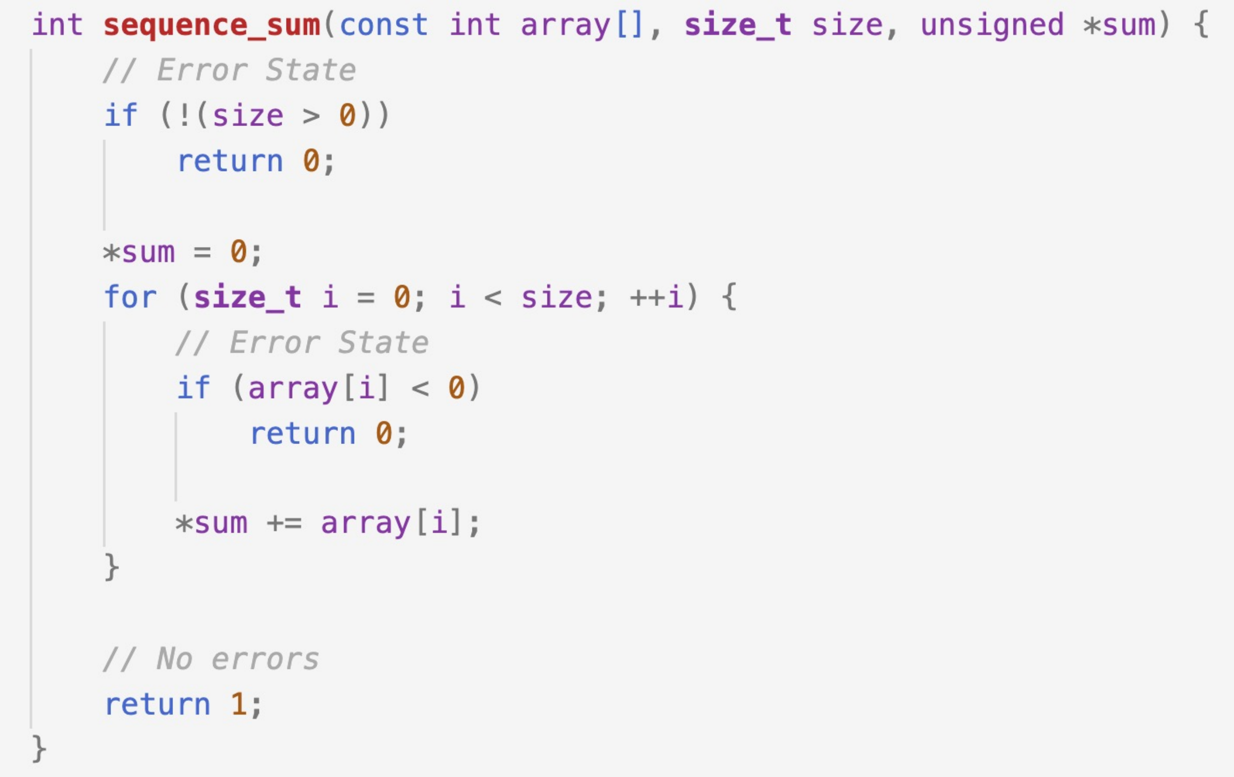


```c
int sequence_sum(const int array[], size_t size, unsigned *sum) {
    // Error State
    if (!(size > 0))
        return 0;

    *sum = 0;
    for (size_t i = 0; i < size; ++i) {
        // Error State
        if (array[i] < 0)
            return 0;

        *sum += array[i];
    }

    // No errors
    return 1;
}
```

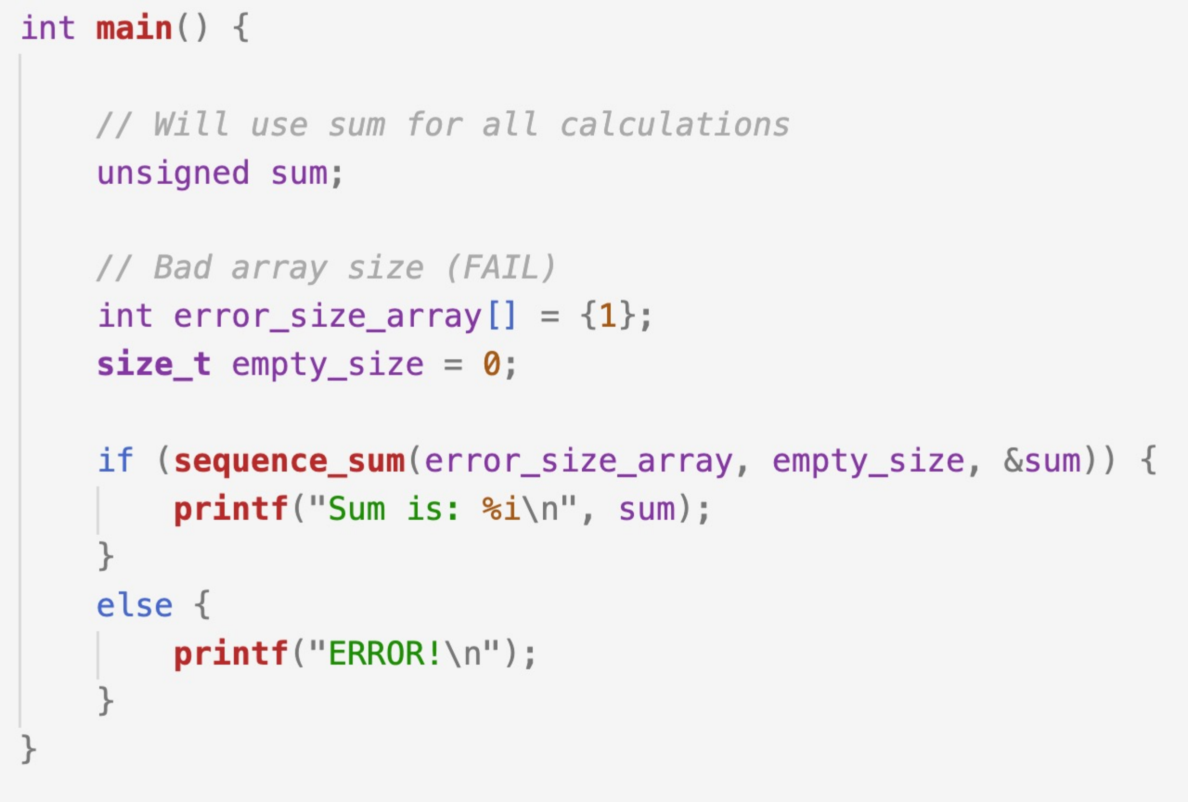


```c
int main() {

    // Will use sum for all calculations
    unsigned sum;

    // Negative value in array (FAIL)
    int negative_array[] = {1, 2, -3};
    size_t negative_size = 3;

    if (sequence_sum(negative_array, negative_size, &sum)) {
        printf("Sum is: %i\n", sum);
    }
    else {
        printf("ERROR!\n");
    }
}
```

sequence_sum()



```c
int sequence_sum(const int array[], size_t size, unsigned *sum) {
    // Error State
    if (!(size > 0))
        return 0;

    *sum = 0;
    for (size_t i = 0; i < size; ++i) {
        // Error State
        if (array[i] < 0)
            return 0;

        *sum += array[i];
    }

    // No errors
    return 1;
}
```



```c
int main() {

    // Will use sum for all calculations
    unsigned sum;

    // Bad array size (FAIL)
    int error_size_array[] = {1};
    size_t empty_size = 0;

    if (sequence_sum(error_size_array, empty_size, &sum)) {
        printf("Sum is: %i\n", sum);
    }
    else {
        printf("ERROR!\n");
    }
}
```