CMPT 125: Introduction to Computing Science and Programming II

Spring 2023

Week 1: Introduction to C (Cont'd)

Instructor: Victor Cheung, PhD

School of Computing Science, Simon Fraser University

Joke of The Day

https://xkcd.com/138/

Recap from Last Lecture

- Logistics, Expectations, Objectives, Academic Integrity
- Concepts in Computing Science
  - Algorithms, programming, data structures
  - Interpreted and compiled programming languages

Something about C

- C was developed by Dennis Ritchie between 1972 & 1973 at AT&T Bell Labs
- Designed to **map to typical machine instruction** – so the source code can easily be compiled & optimized (but as a result sometimes hard to understand)
- **Widely used** to implement operating systems, computationally intensive programs, computer graphics
- Provides **low-level access to system memory** via pointers – highly flexible but can cause strange and sometimes harmful impacts to the system

Let's Write a C Program

This allows you to use external code (library)

Lines 3-7 is what we call the "main function", which is the entry point of a C program

```
1 #include <stdio.h>
2
3 int main() {
4     printf("Hello World!");
5
6     return 0;
7 }
```

What would you do if you want the program to print a different sentence?

What happens if you want to print more sentences?

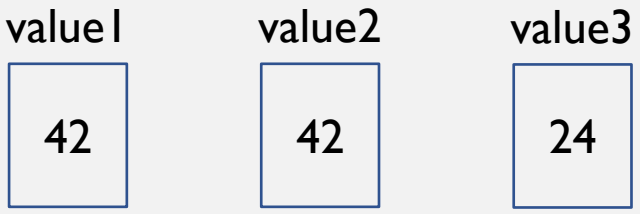
*Refer to the "Accessing CSIL machines remotely" page on Canvas to learn about different ways to connect to CSIL

Today's Topics

- Built-in (aka primitive) types in C
- Pointer variables
  - Their uses and how to use them
- Arrays
  - Their uses and how to use them

Variables

- **Variables** are containers ("boxes") that hold values when the program is running
  - The program can use these values by examining what they represent or changing them
  - Each variable typically can hold one value
  - Each variable is identified by a unique name (there are some rules about what can be a name)
- In C all variables must be declared before use



Declaration And Initialization

- Declaration of a variable in C includes specifying:
  - its type (what kind of value it is storing)
  - its name (how is it going to be referred to in the remainder of the program)
- Initialization of a variable is to give (i.e., assign) the variable a value **when it is declared**
  - usually a default value like 0, an empty string
  - you should always do that (some compilers will do that automatically, but don't count on that!)

```
int x; //this is a declaration
long y = 15; //this is an initialization
```

Type

- A **type** is a set of values together with a set of operations that can be applied to those values
  - Example: int specifies all 32-bit integers (negative/zero/positive) and the arithmetic they support (+-*/)
  - When a variable is declared in C, its type is known and the required memory is reserved, for example:
    - int has 4 bytes (sometimes 2 bytes, depends on the compiler and the system the program is running on)
    - long has 8 bytes
    - double has 8 bytes
    - char has 1 byte
  - As each type has a pre-defined memory allocated, their possible values are limited to the amount of bits used to store the value. E.g., a 4-byte int can only store 2^{8*4} different values → between -2147483648 and 2147483647

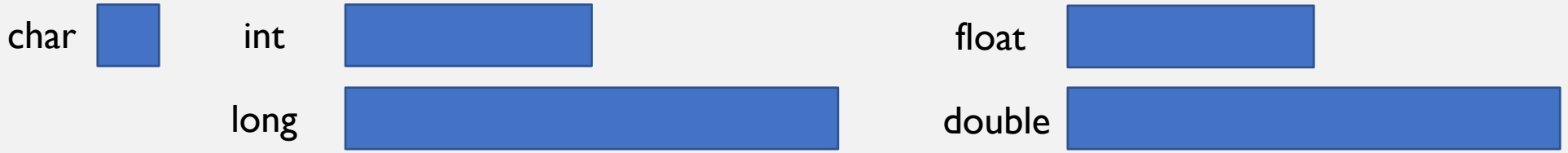
You should have learned what "byte" means previously

Strongly-Typed VS Weakly-Typed

- Strongly-typed language – all variables need to declare a type and stay at that type
  - Compiler & run-time system will check to make sure no unsupported operations are applied (some allow explicit casting – changing a type to the other)
  - C is a strongly-typed language
  - Other examples: C++, Java, C#, Pascal
- Weakly/loosely-typed language – variable types are converted implicitly
  - Examples: Javascript, Perl

Built-in (Primitive) Types in C

- Every variable has a type and has to be declared in C before use
  - Different types might require different storage spaces, e.g., 4 bytes for an int, and thus have their own possible values (range)
- C pre-defines some types so they are ready for use, we call them "built-in" or "primitive" types
  - E.g., char (character), int (integer), long (integer with larger range), float (decimal), double (decimal with larger range)
  - For numbers there is also an "unsigned" version, essentially doubling the max possible value by giving up the negatives



Using Variables

- Depending on the type, you can perform different operations on one or more variables, for example:
  - **add** 2 int variables together
  - **change the polarity** of a double variable
  - **concatenate** 2 string (sequence of characters) variables together
- You can **assign** the result of the operation to a variable using the **=** operator

```
int sum = 0; //initalize the sum to 0
int a = 2, b = 3; //you can do this to initalize variables of the same type
sum = a + b; //assign the result of a + b (2 + 3) to the variable sum
```

Printing Variables

- C provides a way to display (print) the value of a variable **in a certain format** (e.g., just a number for int, a decimal point for double, the sequence of characters for string)

```
int magicNumber = 42;
printf("The magic number is %d!\n", magicNumber);
```

an external function printf made available by including stdio.h

Write the sentence you want to print, specify how you want the variable to be printed (possible to have more)

Specify which variable you want to print (possible to have more)

Printing Built-in (Primitive) Types in C

- Using **printf**, we can display the values in their respective format according to their type
  - %d – for printing integers
  - %f – for printing decimals (can also control the number of decimal places)
  - %e – for printing numbers in scientific notation
  - %c – for printing characters
  - %s – for printing strings
  - %p – for printing pointer address (we'll cover this later in this lecture)

Int VS Long

- Try the following code (supply your own main), what will be printed out for the variable w?

```
int w = 9876543210; // assigned value is too large for int
printf("printing w = %d\n", w);

long z = 9876543210;
printf("printing z = %ld\n", z);
```

- The long type allows the program to use larger numbers than what can be stored by the int type, as it uses more memory space (there are other numeric types doing similar things too)

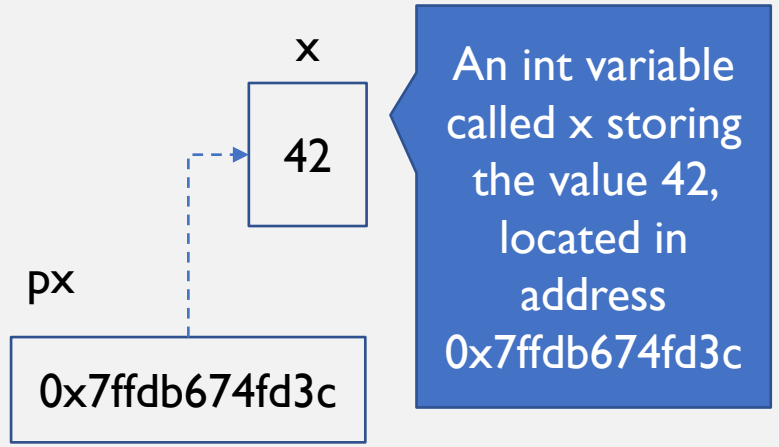
Pointer Variables (Pointers)

- One unique property of C is its ability to access memory directly via code, as achieved by **pointer variables**
- Main idea: each variable is stored in a unique location in the memory, as represented by an address
  - Pointer variables are variables that can **store those addresses**

```
int x = 42;
int* px = &x;
prin f("The address of x is %p\n", px); //same as &x
prin "The value of x is %d\n", *px); //same as x
```

the "&" gets the address of the variable

the * indicates that the variable is a pointer to an int




```
The address of x is 0x7ffdb674fd3c
The value of x is 42
```

Dereferencing Pointers

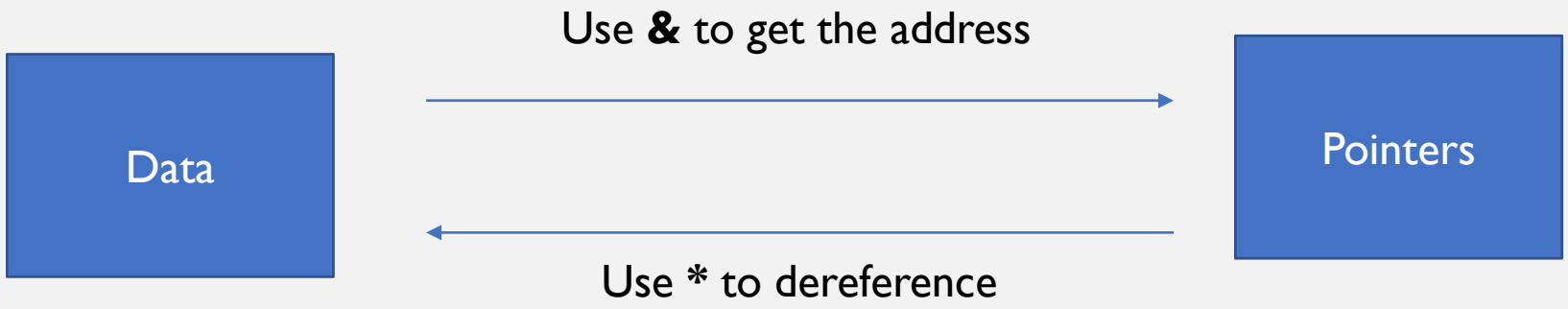
- The * symbol means different things when used in different places:
  - When it is used with the type in the declaration/initialization, it means the variable is a pointer type (e.g., int* is a pointer to an int, double* is a pointer to a double)
  - When it is used in the rest of the code, it means "dereference" the pointer, that is, "*go to the memory address the pointer is pointing to and use that instead*"

```
int x = 42;
int* px = &x;
printf("The address of x is %p\n", px); //same as &x
printf("The value of x is %d\n", *px); //same as x

*px = 56;
printf("The value of x is %d\n", x);
```

```
The address of x is 0x7ffc168d831c
The value of x is 42
The value of x is 56
```

Pointers VS Data Variables

- Pointer variables are special variables that **store addresses**
  - they are values too, just not used in the same way as normal data, as they change every time the program runs
- Data variables are variables that store values



Why Are Pointers Useful?

- They allow functions to modify parameters
- They allow passing larger data types to a function using just the address (takes less space and the function can modify it)
- Optimize code as it allows reference to the data to reduce duplication of the data
- Let programmers to use arrays and strings

Pointers Allow Functions to Modify Parameters (1)

- The **swap** function is supposed to exchange the values stored in a & b (a very useful function in sorting)

```c
void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```

This is a function, we'll cover the exact syntax later

```c
int main() {

    int a = 2;
    int b = 3;
    printf("a is %d and b is %d\n", a, b);
    swap(a,b); // the values will not change!!
    printf("a is %d and b is %d\n", a, b);

    return 0;
}
```

```
a is 2 and b is 3
a is 2 and b is 3
```

It doesn't work!

Pointers Allow Functions to Modify Parameters (2)

- The reason why it doesn't work is because when a function is called it creates its **own** set of variables which are separated from main's variables

```c
void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```

swap's a | swap's b | swap's tmp

2 | 3 | 2

```c
int main() {

    int a = 2;
    int b = 3;
    printf("a is %d and b is %d\n", a, b);
    swap(a,b); // the values will not change!!
    printf("a is %d and b is %d\n", a, b);

    return 0;
}
```

main's a | main's b

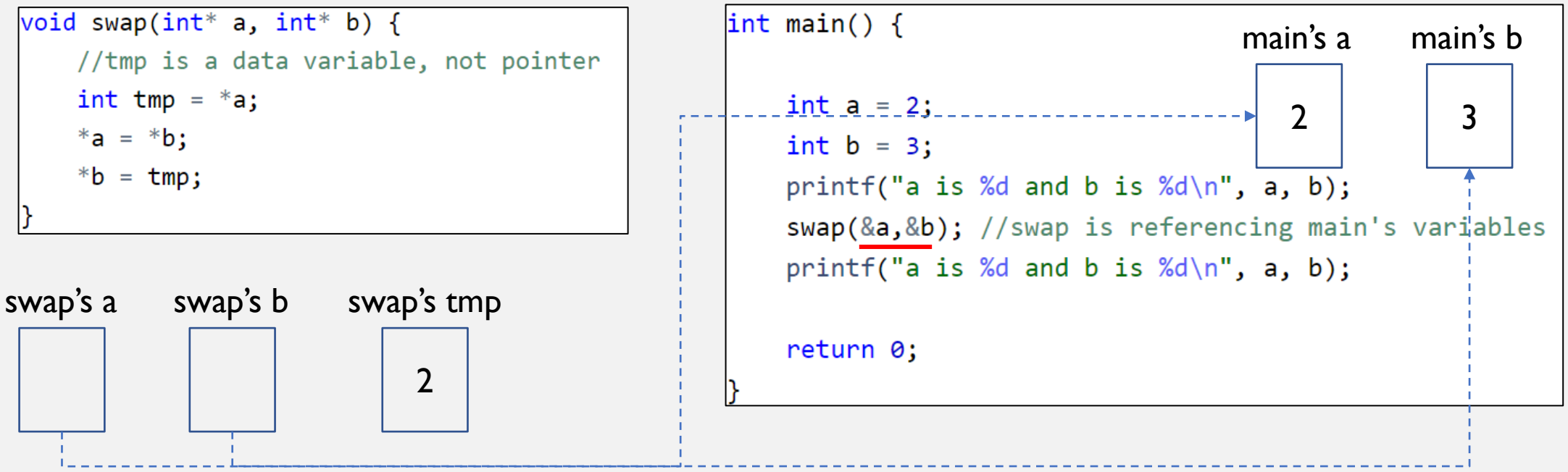
2 | 3

Pointers Allow Functions to Modify Parameters (3)

- Using pointers, we can make the variables of the swap function reference the variables of the main function, so the variables can be modified (we call this mechanism "pass-by-reference")

```
void swap(int* a, int* b) {
    //tmp is a data variable, not pointer
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

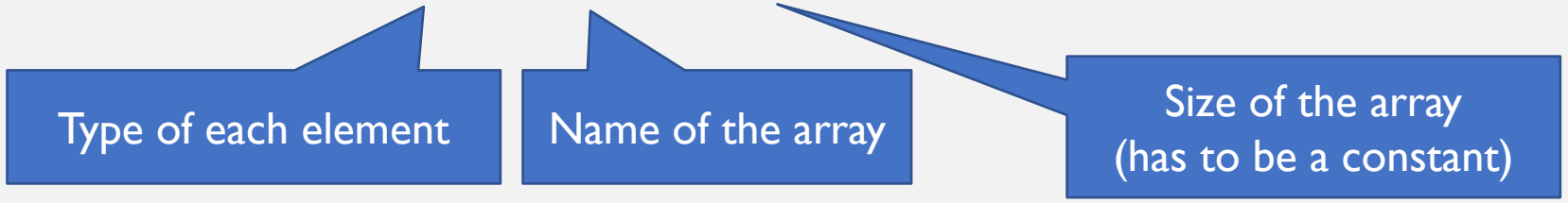
```
int main() {

    int a = 2;
    int b = 3;
    printf("a is %d and b is %d\n", a, b);
    swap(&a,&b); //swap is referencing main's variables
    printf("a is %d and b is %d\n", a, b);

    return 0;
}
```



Arrays

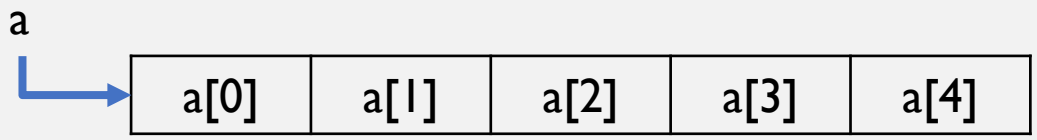
- We started using 1 variable to store 1 value at a time, works ok when the program is small, but:
  - Gets overwhelming when we have lots of values to store
  - In many cases a bunch of variables are related (e.g., height of the class, names of songs in an album)
- Arrays allows us to use **1 name to store a bunch of related values**, called elements (hence the same type)

```
int anInt;              // a single integer
int intArray[100];      // an array of 100 integers

char aChar;             // a single character
char charArray[100]; // an array of 100 characters
```



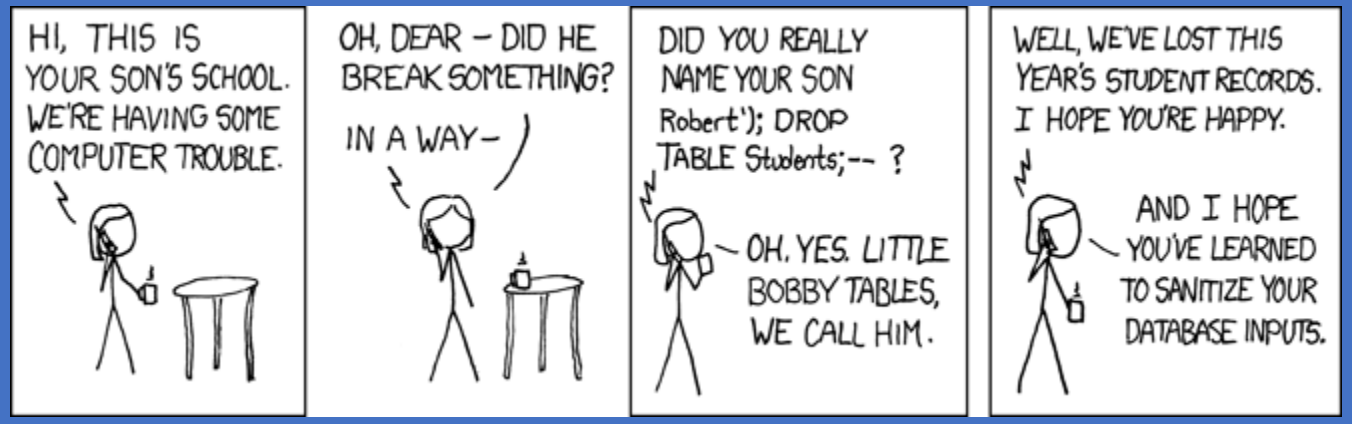
Properties of Arrays

- Once created, the array cannot be resized (exception if it is a "dynamic array", we'll cover that later)
- All elements in the array have the **same type**
- Array elements occupy **a contiguous block of memory**
- Array elements are identified using name[index], can then use them just like any other variables
  - The index begins with 0 (**not 1**), and thus only go till length-1
  - In fact, the name is just a pointer to the 0-th element of the array (though you cannot change its value)



An array of size 5 with the name **a**

Let's Take a 5min Break



https://xkcd.com/327

Using Arrays

- Arrays should be initialized with values
  - For short arrays you can use a direct initialization, e.g., `int shortIntArray[] = {0, 0, 0, 0, 0};`
  - For long arrays use a for-loop, e.g., `int longIntArray[256];`
    `for(int i=0; i<256; i++) { longIntArray[i] = 0; }`

This is a shortcut, you can also put 5 here

- Functions also accept arrays as arguments (but need to know their sizes), and will change their content
  - Function definition:
    `void addOne(int arr[], unsigned int len) { … }`
  - Calling functions (only needs the array's name)
    `addOne(longIntArray, 256);`

Array Example

- **Think**: which element in the array is array[3] referring to? How do you replace the direct initialization with a for-loop?

```
int array[7] = {5, 6, 7, 8, 9, 10, 11}; //initialization only works during declaration

printf("array[3] = %d\n", array[3]); // array[3] = 8
array[3] = 66;
printf("array[3] = %d\n", array[3]); // array[3] = 66
```

Iterating through an Array

- We can use the same for-loop structure for initialization array elements to access the array (especially useful when the array has a lot of elements and we want to access most or all of them)
  - Pay attention to how the iterator (i) of the for-loop is used as an index

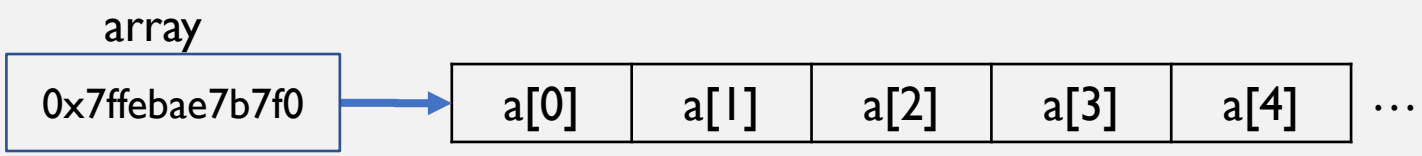
```
int array[10] = {0, 1, 8, 2, 18, 3, 6, 2, 2, -4};

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

for (int j = 0; j < 10; j+=2) {
    printf( "array[%d] = %d\n", j, array[j] );
}
```

Accessing Array with Pointer Arithmetics

- You can actually increment or decrement a pointer variable, but the amount of increment/decrement depends on the type that it is pointing to (e.g., adding 1 to an int pointer increases the address value by size of int)
  - What it really does is convert array+i to array+(i*sizeof(int))
- An array in C is simply a pointer to the first element of the array

```
int i;
int array[10] = {0, 1, 8, 2, 18, 3, 6, 2, 2, -4};
for (i = 0; i < 10; i++) {
    printf("array[%d] at %p = %d\n", i, (array+i), *(array+i));
}
```

```
array[0] at 0x7ffebae7b7f0 = 0
array[1] at 0x7ffebae7b7f4 = 1
array[2] at 0x7ffebae7b7f8 = 8
array[3] at 0x7ffebae7b7fc = 2
array[4] at 0x7ffebae7b800 = 18
array[5] at 0x7ffebae7b804 = 3
array[6] at 0x7ffebae7b808 = 6
array[7] at 0x7ffebae7b80c = 2
array[8] at 0x7ffebae7b810 = 2
array[9] at 0x7ffebae7b814 = -4
```



Arrays in C are Pointers (Constant Pointers to be Exact)

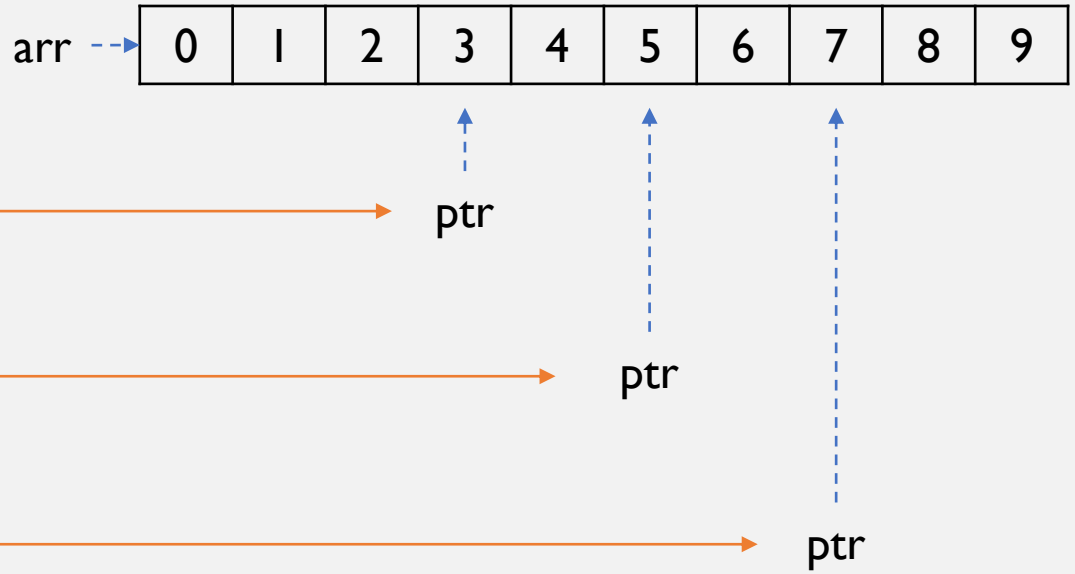
- Pay attention to the relationship between the value in the pointer arithmetic and the index of the item

No need to use & as arr is storing the address

Can't do arr = &arr[7]

```
int arr[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int* ptr;

ptr = arr+3;
printf("*ptr  = %d\n",  *ptr);

ptr = ptr+2;
printf("*ptr  = %d\n",  *ptr);

ptr = &(arr[7]); // same as arr+7
printf("*ptr  = %d\n",  *ptr);
```

```
*ptr  = 3
*ptr  = 5
*ptr  = 7
```



Arrays And Pointers

- Arrays being pointers means we can have a different pointer pointing to the same array (this is actually how arrays are passed to functions), but you cannot change the value of the original pointer

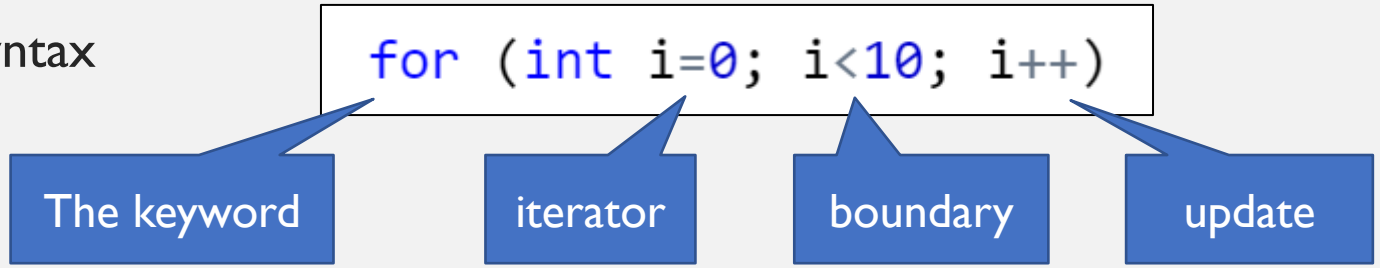
```
int array[10] = {0, 1, 8, 2, 18, 3, 6, 2, 2, -4};
int* first = array;
int* last = array + 9;
int* iter;
for (iter = first; iter <= last; iter++) {
    printf("%d is at the address %p \n",  *iter, iter);
}

*(array+3) = 99;
printf("array[%d] is updated to %d\n",  3, array[3]);
```

Just getting the result for array+3, not changing the value of **array**

```
0 is at the address 0x7ffd640090b0
1 is at the address 0x7ffd640090b4
8 is at the address 0x7ffd640090b8
2 is at the address 0x7ffd640090bc
18 is at the address 0x7ffd640090c0
3 is at the address 0x7ffd640090c4
6 is at the address 0x7ffd640090c8
2 is at the address 0x7ffd640090cc
2 is at the address 0x7ffd640090d0
-4 is at the address 0x7ffd640090d4
array[3] is updated to 99
```

Accessing Arrays Using For-Loops

- The for-loop syntax



- Iterator is used as index to access the array (bounded by size, updated after each iteration)

```
int arr[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
for (int i=0; i<10; i++) {
    printf("Array element at index i = %d\n", arr[i]);
}
```

Side Note – While-Loops

- Another loop structure, while-loop can also be used to access the array

iterator

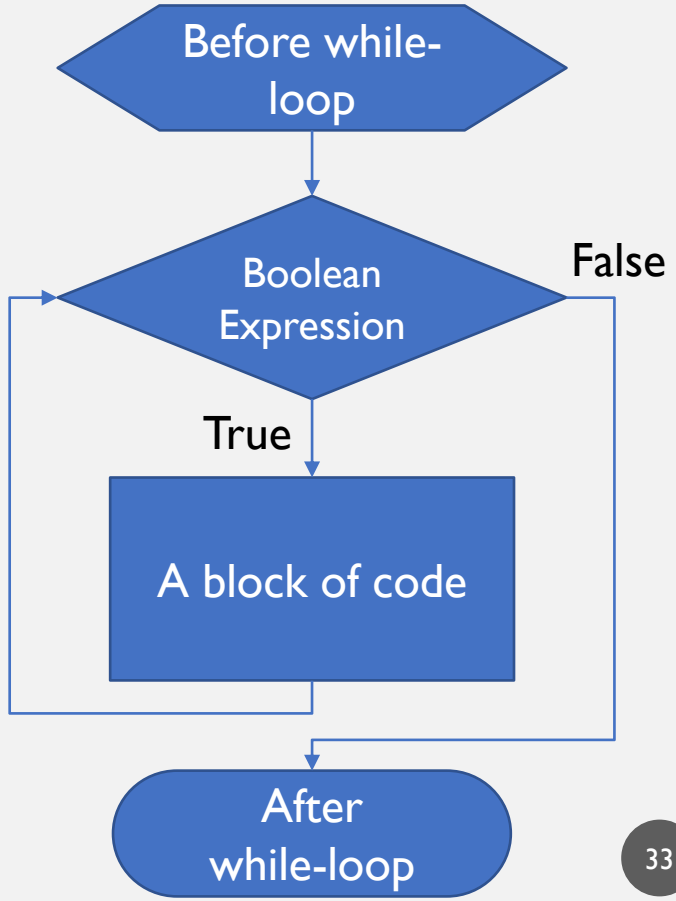
boundary

update

```
int arr[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int i = 0;
while (i<10) {
    printf("Array element at index i = %d\n", arr[i]);
    i++;
}
```

=

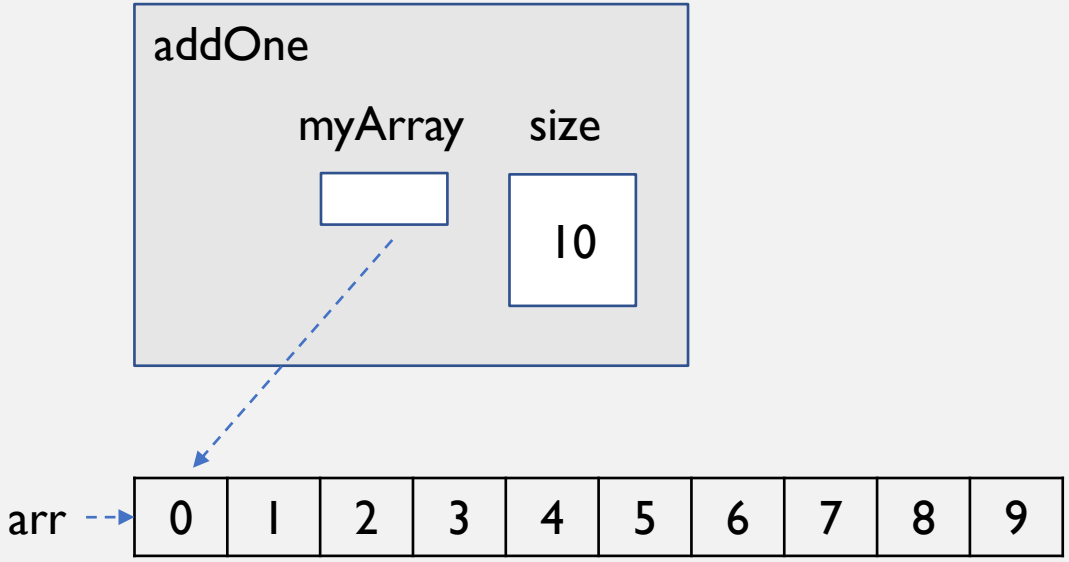
```
int arr[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
for (int i=0; i<10; i++) {
    printf("Array element at index i = %d\n", arr[i]);
}
```



Passing Arrays as Parameters to Functions

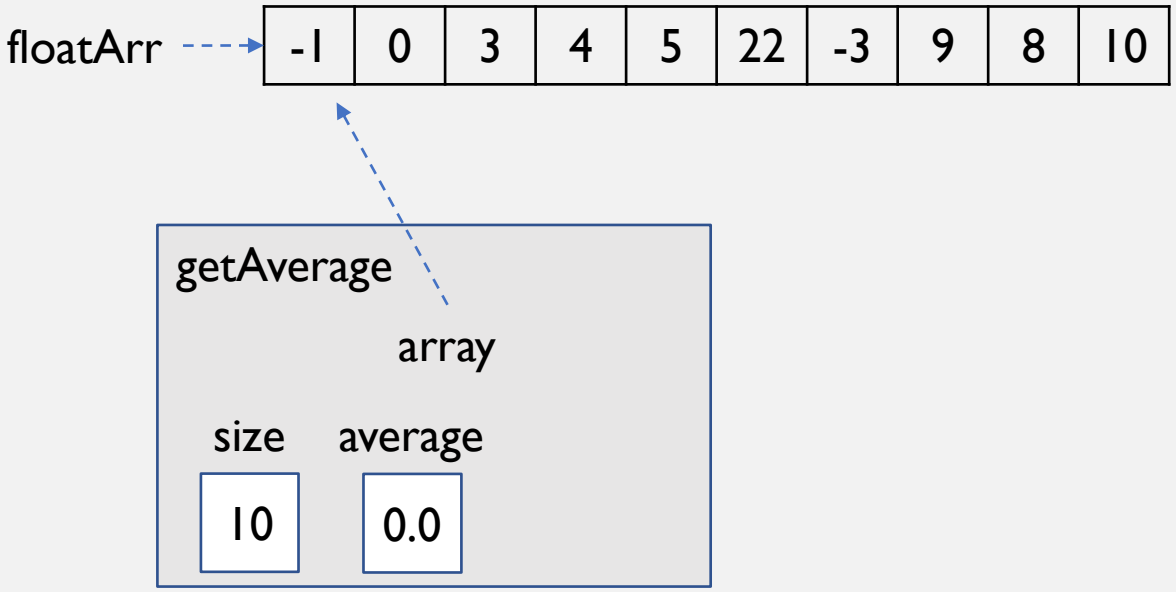
- When passing an array as a parameter to a function, the address of the array is being passed

```
void addOne(int myArray[], unsigned int size) {
    for (int i=0; i<size; i++) {
        myArray[i] = myArray[i] + 1;
    }
}

int main()
{
    int arr[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    addOne(arr, 10);

    return 0;
}
```



Array as Parameter Example

- Write a function that calculates the average of the numbers in an array of floats of length size




```c
float getAverage(float array[], unsigned int size) {
    float average = 0.0;
    for (int i=0; i<size; i++) {
        average += array[i];
    }
    return average / size;
}

int main() {

    float floatArr[10] = {-1, 0, 3, 4, 5, 22, -3, 9, 8, 10};
    printf("The average is %.2f\n", getAverage(floatArr, 10));

    return 0;
}
```

Arrays as Parameters Example

- Write a function that copies all values from one array to another (both of length size)
- Another way to pass the array as parameter is to treat it like a pointer by replacing int dest[] with int* dest
  - Though the [] way is better because it clearly shows that the function is handling an array
- This code works fine, but what if we want to be extra certain that the function won't accidentally change the values in the array? (see next slide)

```
void copyArray(int* dest, int* src, unsigned int size) {
    for (int i=0; i<size; i++) {
        dest[i] = src[i];
    }
}

int main() {

    int intArr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
    int intArr_copy[8] = {0, 0, 0, 0, 0, 0, 0, 0};
    copyArray(intArr_copy, intArr, 8);
    for (int i=0; i<8; i++) {
        printf("%d ", intArr_copy[i]);
    }

    return 0;
}
```

Using the const Keyword

- This code works fine, but what if we want to be extra certain that the function won't accidentally change the values in the array?
- In C, the const keyword is used to indicate the variable cannot change its value once it is set

```
float getAverage(const float array[], unsigned int size) {
    float average = 0.0;
    for (int i=0; i<size; i++) {
        average += array[i];
    }

    array[0] = 100; //compilation error!

    return average / size;
}
```

```
float getAverage(const float* array, unsigned int size) {
    float average = 0.0;
    for (int i=0; i<size; i++) {
        average += array[i];
    }

    array[0] = 100; //compilation error!

    return average / size;
}
```

Constant Variables

- Constant variables are often used to declare (and initialize) values that doesn't change throughout the lifetime of the program, for example, width & height of the screen and size of a static array
  - By convention, we use all-caps for the variable name to indicate that it is a constant variable

```
const int ONE = 1;
int const TWO = 2;

ONE = 4; //modifying the value is not allowed
TWO = 5; //modifying the value is not allowed
```

Constant Pointer VS Pointer to A Constant

- The position of the * when using the const keyword for a pointer is important!
- int* const means a constant pointer

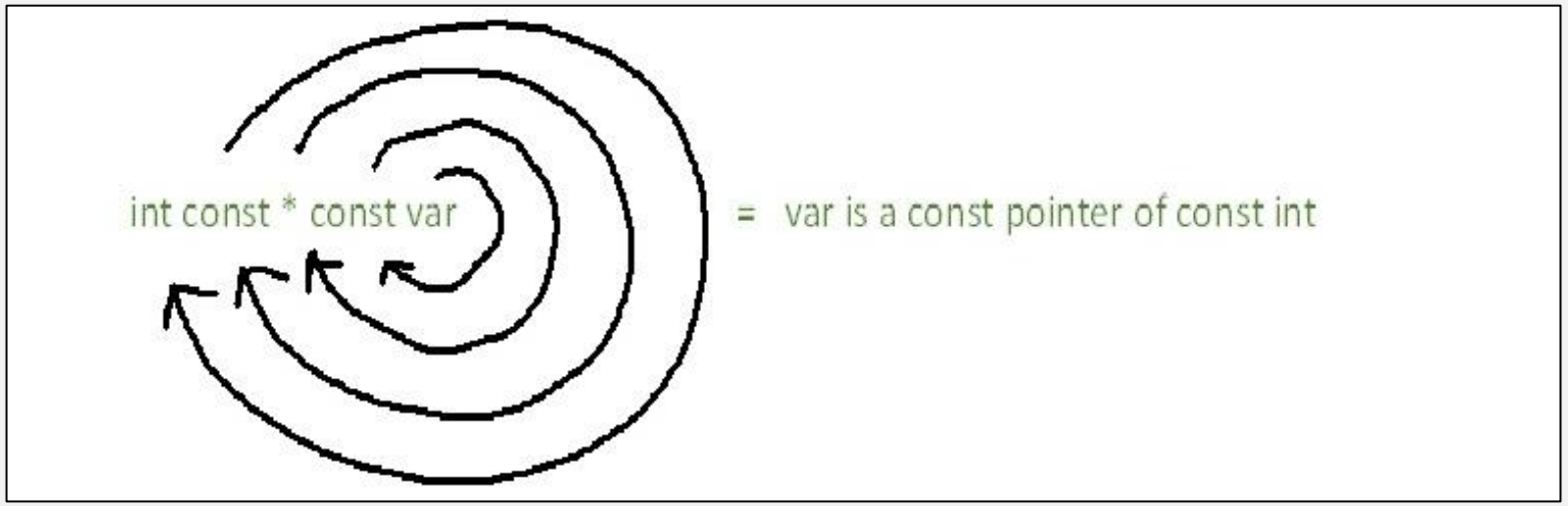
```
int x = 1, y = 2;
int* const const_ptr = &x;

*const_ptr = y; //modifying the value pointed to by the pointer is OK
const_ptr = &y; //modifying the pointer is not allowed
```

- int const* means a pointer to a constant

```
const int x = 1, y = 2;
int const* ptr_const = &x;

*ptr_const = y; //modifying the value pointed to by the pointer is not allowed
ptr_const = &y; //modifying the pointer is OK
```

The Spiral Rule

- There is a way to tell what it means based on where the keyword const is placed



Source: https://www.geeksforgeeks.org/difference-between-const-int-const-int-const-and-int-const/

Char

- Besides numbers C also has a built-in type storing characters (not just English alphabets, but some other symbols such as punctuation marks too)
  - In fact, numerical digits can also be considered as characters
- The type char represents one symbol (letter / digit / punctuation mark / special symbol)

```
char c1 = 'a', c2 = 'B', c3 = ';', c4 = '6';
printf("c1 = %c, c2 = %c, c3 = %c, c4 = %c\n",  c1, c2, c3, c4);
```

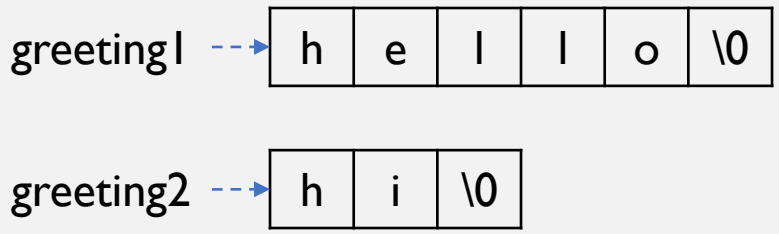
- In the computer, char is stored as a number (1 byte), and C allows some kind of arithmetic
  - Essentially moving the value within the ASCII table

```
char theCharA = 'A';
theCharA = theCharA + 2; //moves theCharA up by 2 in the ASCII table
printf("%c\n", theCharA); //what is the output?
```

C Strings

- It is often important for the program to be able to process words/sentences, C provides a built-in mechanism:
  - A sequence of characters stored in a char array, ended by a special character **\0** (the null character)
  - This means the char array must have at least the size of number of characters + 1
- This mechanism is triggered by assigning a "string literal" to a char array
  - Technically, you can initialize the char array like this: char greeting[] = {'h', 'e', 'y', '\0'}; but no one does that…

```
char greeting1[] = "hello";
char* greeting2 = "hi";
```



Today's Review (1)

- How many variables are here? For each line: what is the name of the variable? what is the type of the variable? Is it a declaration or an initialization?

```
int x;
long y;
int z = 15;
```

- Write a program that prints out the first 10 prime numbers in one printf function call

Today's Review (2)

- What will be the output of these code pieces?

```
int i;
int array[5] = {0, 1, 8, 2, 18};
printf("array[5] = %d\n", array[5]);
```

```
int x = 5;
int* ptr1 = &x;
x = 8;

printf("*ptr1 = %d\n", *ptr1);
```

```
int x = 5;
int y = 7;
int* ptr1 = &x;

y = *ptr1;
*ptr1 = 10;

printf("x = %d\n", x);
printf("y = %d\n", y);
```

- Suppose ptr1 and ptr2 are both pointers, what is the difference between
  - ptr1 = ptr2;
  - *ptr1 = *ptr2;

Today's Review (3)

- Describe what each of these expressions mean, if array is an int array of size 10, or explain if it is incorrect:
  - array[3], array[10], *(array+9), &array[4], array++
- What's wrong with these code pieces?

```
void foo(int* a) {
    //do something
}

int main() {
    const int x = 5;
    foo(&x);
}
```

```
const int x = 1;
int* const const_ptr = &x;
```

Homework! (1)

- Read Ch. 1 of the Effective C book
  - In particular, *The Swapping Values, Pointer Types* and *Arrays* sections
- Do the [Self-Test] Week 01 on Canvas
- Find out what for-loop is and how to use it
  - *The for Statement* section in Effective Ch. 5
  - *The for loop* section in http://www.cplusplus.com/doc/tutorial/control/
- Try using the format specifiers to control how values are being printed using printf
  Reference: https://www.cplusplus.com/reference/cstdio/printf/

Homework! (2)

- Read Ch. 2 & Ch. 5 of the Effective C book
  - Pay particular attention to sections in Ch. 2: Pointer Types, Arrays, const
  - Pay particular attention to sections in Ch. 5: Iteration Statements
    - In practice, don't use "goto" and avoid using "break" and "continue", as they often lead to hard-to-trace code
- Learn different ways write your code in a CSIL machine (check out our Lab01)
- Look up the full list of functions provided by string.h
  http://www.cplusplus.com/reference/cstring/