C Basics Continued

Functions: single file

- The file that contains main function can also contain other functions.
- Functions must be declared before they can be used.
  - This means they must appear in the source code before it is referenced, *this is different than Java*.
  - They can be declared or defined in the same file or another file. (More on that later.)
- See functions/single_file/function.c

Parameter passing with functions

- Like Java, C is *strictly* pass-by-value.
- That means every argument to a function has its value copied in to the stack frame of the called function.
- This can be easily be demonstrated by trying to write a swap function for two int parameters.
- See pass-by-value/broken-swap.c

+ Scoping

- The scope of a variable/function name is the part of the program within which the name can be used
- A global (external) variable or function's scope *lasts from where it is declared to end of the file*
- A local (automatic) variable's scope is within the function or block.
  - Scoping of automatic variables is the same as Java
- See scoping/scoping.c

Variable initialization

- In the absence of explicit *initialization*
  - global variables are guaranteed to be initialized to zero
  - local variables have undefined initial value. (!!!)
- Definition vs. declaration
  - Declaration specifies the type of a variable.
    - No value assigned, only a name and type.
  - Definition sets value at location for the variable
- See init/init.c

Static keyword

- Different than Java.
- Serves two purposes
  - limits a scope of a global variable to within the rest of the source file in which it is declared.
  - allows the variable (global or local) to preserve its value between function calls.
- *Note*: like globals, statics are guaranteed to be initialized to zero
- See static/main.c and function.c

+ Functions: multiple files

- In C, you can, of course, spread your code across multiple files.
- You will need to share functions or variables across those files.
- Again, functions must be *declared* before they can be used.
  - This means declarations must appear *lexically* in the source code before it is referenced, *this is different than Java*.
- Similarly, *variables* defined in other files must be declared before they can be used. The '**extern**' keyword is used for this purpose.
- See functions/multiple_files/function.c and main.c

Header files

- Header files contain declarations of functions and globals that can be included in other source code files.
- The header file's name usually matches the name of the file that provides the the definitions and ends with .h extension.
- This is nice because we don't have to clutter our source files with declarations for all the different variables and functions we want to use that exist in separate files.
- See headers/function.h, function.c and main.c

Header includes

- You might have noticed that the two includes in our main.c from the last example look a little different.
- **#include <*filename*>** indicates that this is a system header that can be found in some predetermined, system-dependent location.
- **#include "*filename*"** indicates that this is a header that exists in the same directory that the source code is in.

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

int main() {
  printf("counter is %d\n", counter);
  add_one();
  add_one();
  printf("counter is %d\n", counter);
  return 0;
}
```

Pointers

Introduction to memory

- In order to understand the feature of C that we will discuss next, pointers, we need to understand some basics about memory.
- So what is memory?
  - We can think of it as a big table of ordered slots.
  - Each slot can hold one byte.

Data layout in memory

The number of a slot is its **address**. One byte **value** can be stored in each slot.

Some "logical" data values span more than one slot, like the character string "Hello\n"

A **type** provides information about a span of memory. Ex.

char	a single character (1 slot)
char [7]	an array of 7 characters
int	signed 4 byte integer
float	4 byte floating point

Addr	Value
0	
1	
2	
3	
4	'H' (72)
5	'e' (101)
6	'l' (108)
7	'l' (108)
8	'o' (111)
9	'\n' (10)
10	'\0' (0)

Variables ‘under the hood’

A *variable* **names** an **address** in memory where you store a **value** of a certain *type*.

You first **define** a variable by giving it a name and specifying the type, and *optionally* an initial value

```
char x;
char y = ‘e’;
```

Initial value of x is undefined

Initial value

Name

Type (char)

The compiler puts them somewhere in memory.

Name	Addr	Value
	0	
	1	
	2	
	3	
x	4	?
y	5	‘e’ (101)
	6	
	7	
	8	
	9	
	10	

+ Name, address, value

- char x
  - The address of x is 4
  - The value at address 4 is 72 (or ‘H’)
- char y
  - The address of y is 5
  - The value at address 5 is 101 (or ‘e’)

Name	Addr	Value
	0	
	1	
	2	
	3	
x	4	‘H’ (72)
y	5	‘e’ (101)
	6	
	7	
	8	
	9	
	10	

Pointers

- In C, we can know the *address* of a given *name* by '*dereferencing*' that name.
- This gives us an *address as a value, aka a 'pointer'*
- Moreover, pointers are addresses in memory of some value of some type.
- ex

```
int main() {
    char x = 'H';
    char* ptr = &x;
    printf("address: %p\n", ptr);
    return 0;
}
```

address: 0x7fff5a13671b

- See pointers/simple/pointer.c

Name	Addr	Value
	0	
	1	
	2	
	3	
x	4	'H' (72)
y	5	'e' (101)
	6	
	7	
	8	
	9	
	10	

Pointer motivation

- Pointers are used in C for many other purposes:
  - Passing large objects without copying them
  - Having values that are not global that can be shared across many calls to functions.
  - Accessing dynamically allocated memory
  - Working with arrays
  - Passing functions as arguments to other functions
  - More…

Pointer operators

- There are two operators that are necessary when working with pointers.
  - The address-of operator ‘**&**’
  - The dereference and declaration operator ‘*’
- Ex.

```
void f(char* p) {
   *p = *p - 29;
}

int main() {
   char y = 101;      // ‘y’ is 101
   f(&y);             // i.e. 5
   // y is now 101- 29 = 72
}
```

- See pointers/functions/pointer.c

Name	Addr	Value
	0	
	1	
	2	
	3	
x	4	‘H’ (72)
y	5	‘H’ (72)
	6	
	7	
	8	
	9	
	10	

* Represents memory after function execution.

+

Pointer validity

- A valid pointer is one that points to memory that your program controls.
- Using invalid pointers will cause non-deterministic behavior (often ‘Segmentation Fault’)
- There are two general causes for these errors:
  - Program errors that set the pointer value to a invalid address
  - Use of a pointer that was at one time valid, but later became invalid.

+ Pointer validity

- Is this program correct?
  - Moreover, will ‘ptr’ be valid or invalid?

```
char* get_pointer() {
  char x=0;
  return &x;
}

int main() {
  char* ptr = get_pointer();
  *ptr = 12;  /* valid? */
}
```

Pointer validity

- Is this program correct?
  - Moreover, will ‘ptr’ be valid or invalid?
- A pointer to a variable allocated on the *stack* becomes invalid when that variable goes out of scope and the stack frame is “popped”.

```
char* get_pointer() {
   char x=0;
   return &x;
}

int main() {
   char* ptr = get_pointer();
   *ptr = 12;  /* valid? */
}
```

Pointer validity

- Is this program correct?
  - Moreover, will ‘ptr’ be valid or invalid?
- A pointer to a variable allocated on the stack becomes invalid when that variable goes out of scope and the stack frame is “popped”.
- The pointer will point to an area of the memory that may later get reused and rewritten. Could result in segfault.

```
char* get_pointer() {
  char x=0;
  return &x;
}

int main() {
  char* ptr = get_pointer();
  *ptr = 12;  /* valid? */
}
```

Pointers & Java references

- They behave in similar manners in some respects.
- Pointers have fewer constraints and are more explicit, but Java references have some of the same motivation.
- In fact, ‘under the hood’ a Java references really is a pointer.
- You can think of a Java reference as a ‘pointer on training wheels’.
- If you understand heap allocation and how reference types are passed to functions by address, you already have some intuition about pointers.

Arrays

Arrays

- Arrays are an aggregate data type, moreover, a collection of data that are of the same *type*.
  - int a[5];
- Array elements can be accessed and assigned by index/offset using subscript notation.
  - int i = a[3];
- Sounds pretty similar to Java. However, that's most of the similarities.

Arrays *con't*

- In Java, when we *declare* an array,
  - storage is not allocated.
- In Java, when we *define* an array.
  - each position is initialized to a default value.
- In C, when we *declare* an array,
  - storage is allocated.
  - each position is uninitialized.

+ Arrays *con't*

- Each element of the array is stored **contiguously** in memory.
- There is no bounds checking on arrays in C
  - C will let you write to the 5mm'th offset in a 2-element array.
- C arrays have no 'length' information!
  - We must pass that around along with the array.

+ Arrays sizes

- C arrays just refer to a raw region of memory.

- char foo[80];
  - An array of 80 characters
  - sizeof(foo) = $80 \times$ sizeof(char) = $80 \times 1 = 80$ bytes

- int bar[40];
  - An array of 40 integers
  - sizeof(bar) $40 \times$ sizeof(int) = $40 \times 4 = 160$ bytes

+ Arrays: subscript notation

- An array in C has a fixed size that cannot be changed after it is declared.
- The memory is allocated on declaration, ex.
  - int a[10] creates an array a of 10 integer ‘slots’.
- To populate the array with values, loop over the array and use subscript notation (just like Java).
- To access the array’s values, loop over the array and use subscript notation (just like Java).

```
const int size = 10;
int a[size];

int i;
for (i = 0; i < size; i++) {
	a[i] = i;
}

for (i = 0; i < size; i++) {
	printf("%d ", a[i]);
}
```

Arrays & pointers

- Arrays in C are simply syntactic sugar to access contiguous memory spaces, or - really - just a variant of a pointer notation.
- Any operation that can be done by subscripting ‘[]’ can also be done with pointer notation.
- This can be confusing at first, but if you think about what a pointer is and what arrays are, then it makes sense.

Arrays: pointer notation

- An alternative approach to accessing array locations is to set a pointer to point at the first element in the array.
- The array name itself is really a pointer to the first element. So we can set a pointer to be equal to an array.
- We can even continue to use the subscript notation on the pointer!

```
const int size = 10;
int a[size];

int* pa = a;

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

+ Arrays: pointer arithmetic

- It is possible to traverse an array by incrementing the *value of the pointer*!
- By incrementing the pointer, we 'jump' enough bytes to 'land' on the next element in the array.
- Moreover, by adding integral value N to a pointer, we advance 'N * sizeof(type)' in memory from the current address of the pointer.
- We can do this forward or backwards.
- See arrays/array.c

```
const int size = 10;
int a[size];

int* pa = a;

int i;

// forward
for (i = 0; i < size; i++) {
  printf("%d ", *pa);
  pa++;
}

// backward
pa = a + size-1;
for (i = 0; i < size; i++) {
 printf("%d ", *pa3);
 pa—;
}
```

+ Arrays: pointer arithmetic *cont*.

- Assume that we have an array of integers and a pointer to an array

```
int a[10]; //initialized with values somewhere
int *pa;
```

- An array name is a constant pointer to the first element in the array. The following two statements are equivalent:

```
pa = & a[0];
pa = a;
```

- An array name is a constant pointer so it cannot be modified. The following statements are all illegal:

```
a++;
a = a+5;
a = pa;
```

+ Arrays: pointer arithmetic *cont*.

- Adding a value N to a pointer advances the memory location that is 'N * sizeof(type)' away from the current pointer position.

```
pa=pa + 2;  //  advances pa by (2 * 4) bytes
```

- The following are equivalent, assuming pa was assigned value of a and i is an int between 0 and the size of the array a…

```
& a[i]    // fetching a pointer to an element at an offset
a + i
& pa[i]
pa + i
```

- As are these

```
a[i]      // fetching the value at an offset
*(a+i)
pa[i]
*(pa+i)
```

+

Arrays, pointers and functions

- You should be getting the idea by now, that pointers and arrays are very closely related.
  - Not only conceptually, but at the language level.
- When passing an array to a function, we are really passing the pointer to the first element of the array.
- These two function prototypes can be interpreted as equivalent:

```
int sumElements( int a[], int size );
int sumElements( int* a,  int size );
```

Multi-dimensional arrays

- In C multidimensional arrays are stored in consecutive memory locations.
- We can traverse them using nested loops as we do in Java.
- Ex.

```
const int ROWS = 3;
const int COLS = 2;

int matrix[ROWS][COLS] = {{ 1,2}, {3,4}, {5,6}};

 for (i = 0; i < ROWS; i ++) {
   for (j = 0; j < COLS; j++) {
      sum += matrix[i][j];
   }
}
```

- See arrays/multi_dimensional.c