Darrell Long

CSE 13S

Pointers and Dynamic Memory

6 May 2021

What is a pointer?

- A variable that holds a memory address.
  - The variable points to the location of an object in memory.
- Not all pointers contain an address.
  - Pointers that don't contain an address are set to the `NULL` pointer.
  - Value of the `NULL` pointer is 0.
  - `NULL` is a macro for either:
    - `((void *)0)`
    - `0`
    - `0L`
      - The definition depends completely on the compiler.

Review: Memory Addresses

- Memory is stored in registers that can be accessed by a specific number (address).
- Usually, each byte has a unique address.
- Bytes are grouped into words (timeshare uses word size 4).

```c
int main(void) {
  int fib[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
  return 0;
}
```

Address	00	04	08	0C
A000_7FA0	00000000	00000000	00000000	00000000
A000_7FB0	00000000	00000001	00000001	00000002
A000_7FC0	00000003	00000005	00000008	0000000D
A000_7FD0	00000015	00000022	00000000	9D000350
A000_7FE0	00000000	00000000	00000000	00000000

Pointers and addresses

- Pointers are said to point at the address they are assigned.
- Can assign a pointer the address of a variable using the address-of operator (&).
- Multiple pointers can point to the same address.

```c
#include <stdio.h>

int main(void) {
  int a = 42;
  int *ptr_a = &a;
  int *ptr_b = &a;

  // Two pointers can point to the same address.
  printf("The address of pointer A is %p\n", ptr_a);
  printf("The address of pointer B is %p\n", ptr_b);
  return 0;
}
```

Address	00	04	08	0C
A000_7FA0	00000000	00000000	00000000	00000000
A000_7FB0	00000000	00000000	00000000	00000000
A000_7FC0	00000000	00000000	A0007FD0	A0007FD0
A000_7FD0	0000002A	00000000	00000000	00000000
A000_7FE0	00000000	00000000	00000000	00000000

How to use the & operator

Example of using the & operator to access the address of a variable:

```c
#include <stdio.h>

int main(void) {
  int foo = 0;

  // Print out address of foo.
  printf("Address of foo: %p\n", &foo);
  return 0;
}
```

Dereferencing a Pointer

- The object a pointer points to can be accessed through dereference, or indirection.
- A pointer can be dereferenced using the dereferencing operator (*).
- Useful for manipulating the values of several variables through call-by-reference.

```c
#include <stdio.h>

void increment_two_ints(int *a, int *b) {
  *a += 1;
  *b += 1;
  return;
}

int main(void) {
  int x = 3;
  int y = 4;
  increment_two_ints(&x, &y);

  // Now, x is 4 and y is 5.
  printf("The value of x is now: %d\n", x);
  printf("The value of y is now: %d\n", y);
  return 0;
}
```

How to use the * operator

Example of using * to instantiate a pointer variable and using it to dereference a pointer:

```c
#include <stdio.h>

int main(void) {
  int foo = 13;

  // The * denotes that bar is a pointer variable.
  // The address of foo is stored in bar.
  int *bar = &foo;

  // Dereference bar using * to print out value at its stored address.
  printf("The value in the address stored in bar: %d\n", *bar);
  return 0;
}
```

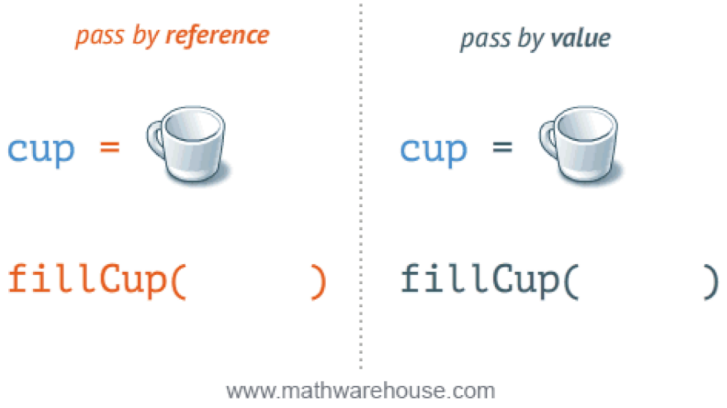
Benefits of Pointers

- Can be used when passing actual values is difficult.
- Can “return” more than one value from a function.
- Building dynamic data structures.
- Useful for passing large data structures around.
  - Pointers are efficient for this since copies of data structures don’t need to be pushed into the stack.

```c
void inc_by_ref(int *x) {
  *x = (*x) + 1;
  return;
}

int inc_by_val(int x) {
  return x + 1;
}

int main(void) {
  int x = 5;
  inc_by_ref(&x);
  x = inc_by_val(x);
  return 0;
}
```



Passing by value versus Passing by reference

- "Passing by value" duplicates passed values onto stack.
- "Passing by reference" duplicates a pointer onto the stack.

```c
#include <stdio.h>

int main(void) {
  int age;
  double gpa;

  // Pass age and gpa variables by reference.
  printf("Please enter your age and gpa: ");
  scanf("%d %lf", &age, &gpa);
  return 0;
}
```

```c
int main(void) {
  int a[2][2];
  int x = matrix_determinant_by_val(a[0][0], a[0][1], a[1][0], a[1][1]);
  int x = matrix_determinant_by_ref(a);
  return 0;
}
```

Passing by reference

- Allows “returning” multiple values.
- Allows passing large amounts of data quickly.
  - You’re not copying the data, just telling where it is stored.

Pointer Arithmetic

- Since pointers in **C** are just an addresses, numeric values, you can perform arithmetic on them.
- ++ : increments to next address (increment by 4 bytes assuming 32-bit integers.
- –– : decrements to previous address.
- + : can only add a numeric value to a pointer (no pointer and pointer addition).
- – : if a pointer is subtracted from another pointer, the distance between both addresses is calculated.
- Pointers can also be compared using relational operators (e.g. ==, <=, <, etc.).

Pointer Arithmetic

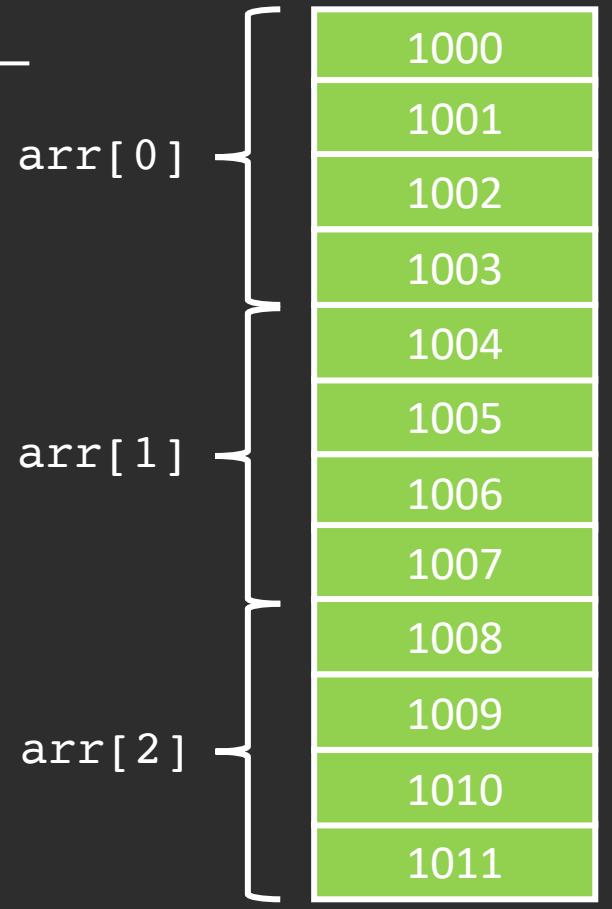
- Can offset a pointer by adding/subtracting an integer.
- Can get the number of elements between two pointers by getting their difference.
- Cannot sum, divide, or multiply two pointers.

```c
#include <stdio.h>

int main(void) {
  char *str = "Hello World!";
  printf("str[0]: %c\n", *str);
  printf("str[1]: %c\n", *(str + 1));
  return 0;
}
```

Arrays And Pointer Arithmetic

- For `int arr[i]`:
  - `arr[i] == *(a + i)`
  - If `arr` starts at address 1000, then `arr[i]` is at 1000 + (i * 4)
    - Why? An `int` is typically 4 bytes
  - If `arr` was an array of `uint64_t`'s, then `arr[i]` would be at 1000 + (i * 8)
  - You can use `sizeof()` to get the size in bytes of scalar types



But does it make sense?

- Adding an `int` to a pointer makes sense.
- Subtracting an `int` from a pointer makes sense.
- Subtracting two pointers makes sense.
- Adding two pointers makes no sense.
- Multiplying or dividing with pointers never makes sense.

Pointer arithmetic example: Iterating over an array

```c
int main(void) {
  int fib[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };

  int *fib_ptr = fib;
  while (fib_ptr < (fib + 10)) {
    *(fib_ptr)++;    // Increment array element.
    fib_ptr++;       // Increment pointer.
  }

  return 0;
}
```

fib_ptr

Address	00	04	08	0C
A000_7FA0	00000000	00000000	00000000	00000000
A000_7FB0	00000000	00000000	A0007FBC	00000001
A000_7FC0	00000002	00000002	00000003	00000004
A000_7FD0	00000005	00000008	0000000D	00000000
A000_7FE0	00000022	00000000	00000000	9D000390

Pointers and arrays

- Array subscripting can also be done with pointers.
  - Using pointer arithmetic in general is faster, but harder to understand.
  - Assuming some array `int arr[10]`:
    - `arr[i]` is equivalent to `*(arr + i)`, where `0 <= i < 10`
- Arrays can always be written using pointers.
  - Declaring an array in a function allocates it on the *stack*.
  - A global array is in the *data area*.
  - Dynamically declaring an array (to get a pointer) allocates it on the heap.

Strings as arrays

- In **C**, strings are handled as arrays
  - With some special syntax, for convenience
- A string is a pointer to an array of chars
- Strings can be indexed, passed by reference, *etc*.

```c
int main(void) {
  char *hello = "Hello World!";
  char *bye = { 'G', 'o', 'o', 'd', 'b', 'y', 'e', 0 };

  // Strings can be indexed.
  printf("hello[3]: %c\n", hello[3]);
  return 0;
}
```

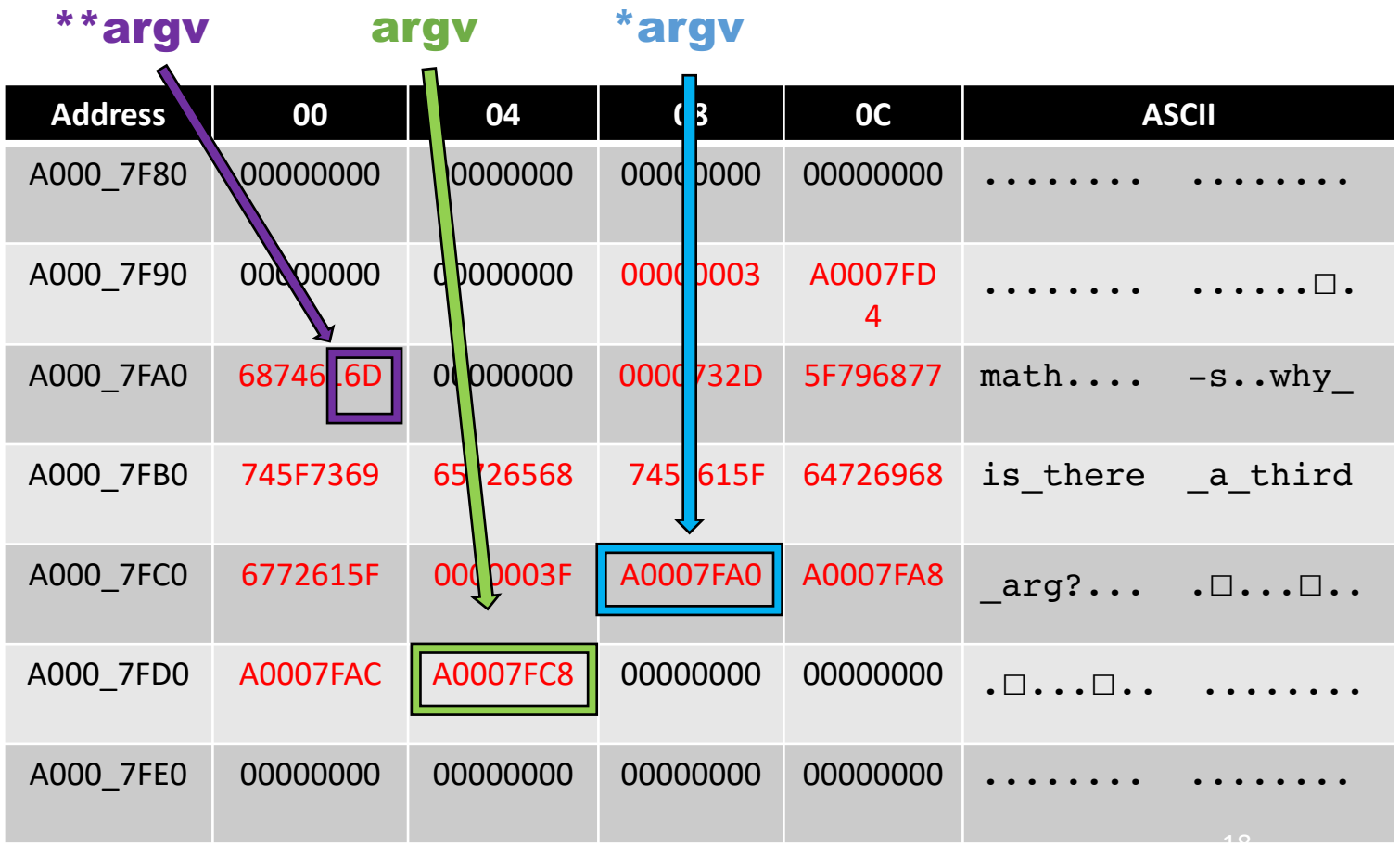
Address	00	04	08	0C	ASCII
0000_7FB0	00000000	00000000	00000000	00000000	....
0000_7FC0	6C6C6548	6F57206F	21646C72	00000000	Hello Wo rld!....
0000_7FD0	646F6F47	00657962	00000000	9D0007B8	Goodbye. ....
0000_7FE0	00000000	00000000	00000000	00000000	....

Pointers to pointers

- Pointers can point to other pointers
  - or to pointers to other pointers, or to pointers to pointers to pointers, etc
- Can be used to pass arrays of arrays, such as a list of strings
- For example, char **argv

```
bash-5.0$ math -s why_is_there_a_third_arg?
```

```
int main(int argc, char **argv);
```



Address	00	04	08	0C	ASCII	
A000_7F80	00000000	00000000	00000000	00000000	....	....
A000_7F90	00000000	00000000	00000003	A0007FD4	....	......□.
A000_7FA0	68746D	00000000	0000732D	5F796877	math....	-s..why_
A000_7FB0	745F7369	65726568	745F615F	64726968	is_there	_a_third
A000_7FC0	6772615F	0000003F	A0007FA0	A0007FA8	_arg?...	.□...□..
A000_7FD0	A0007FAC	A0007FC8	00000000	00000000	.□...□..	....
A000_7FE0	00000000	00000000	00000000	00000000	....	....

Pointers to pointers

- Pointers can point to other pointers
  - or to pointers to other pointers, or to pointers to pointers to pointers, etc
- Can be used to pass arrays of arrays, such as a list of strings
- For example, char **argv

```
bash-5.0$ math -s why_is_there_a_third_arg?
```

```
int main(int argc, char **argv);
```

argv **(argv + 1) *argv

Address	00	04	08	0C	ASCII	
A000_7F80	00000000	00000000	00000000	00000000	....	....
A000_7F90	00000000	00000000	00000008	A0007FD4	....	......□.
A000_7FA0	6874616D	00000000	00007F2D	5F796877	math....	-s..why_
A000_7FB0	745F7369	65726568	745F615F	64726968	is_there	_a_third
A000_7FC0	6772615F	0000003F	A0007FA0	A0007FA8	_arg?...	.□...□..
A000_7FD0	A0007FAC	A0007FC8	00000000	00000000	.□...□..	....
A000_7FE0	00000000	00000000	00000000	00000000	....	....

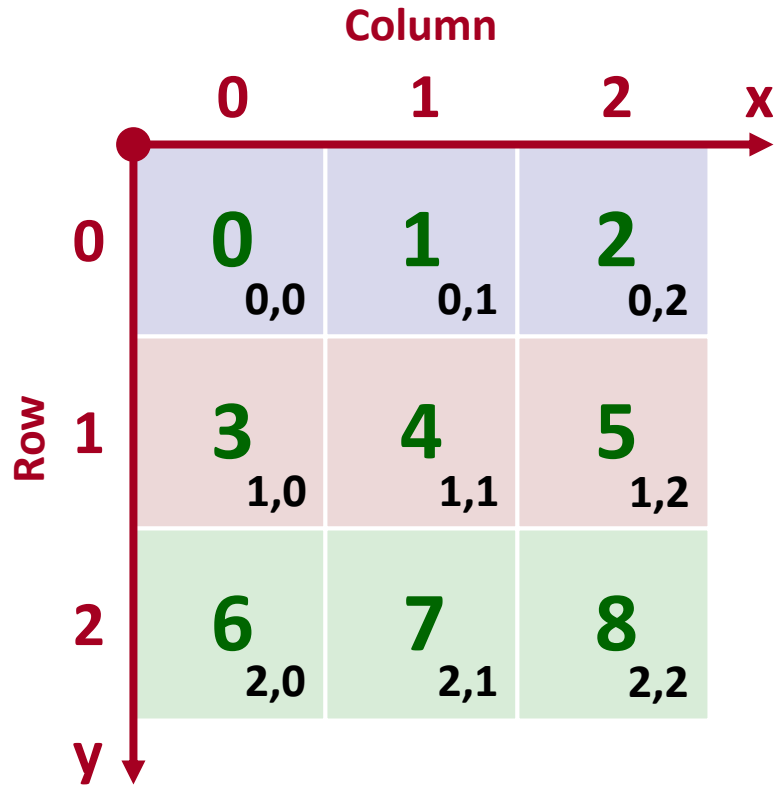
Multidimensional Arrays

- In some cases, arrays of arrays support a more compact data structure:
  - If data is of same type, and each sub-array is of same length, a multidimensional array is appropriate.

```
int   [3][3] = {{0, 1, 2},
                {3, 4, 5},
                {6, 7, 8}};
```

[y][x]

```
a[0][0] = 0;
a[0][1] = 1;
a[0][2] = 2;
a[1][0] = 3;
a[1][1] = 4;
a[1][2] = 5;
a[2][0] = 6;
a[2][1] = 7;
a[2][2] = 8;
```



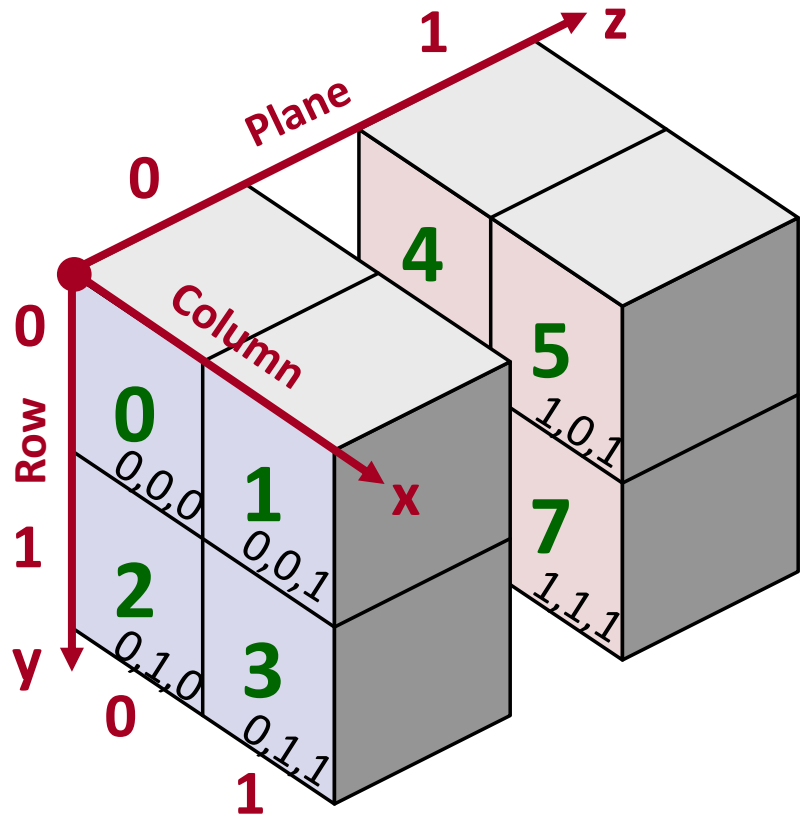
Multidimensional Arrays

- Multidimensional arrays can be of any dimensionality.
- Note: these grow in size *very* rapidly!

```
int[2][2][2] = {{{0, 1},{2, 3}},
                {{4, 5},{6, 7}}};
```

```
[z][y][x]
```

```
a[0][0][0] = 0;
a[0][0][1] = 1;
a[0][1][0] = 2;
a[0][1][1] = 3;
a[1][0][0] = 4;
a[1][0][1] = 5;
a[1][1][0] = 6;
a[1][1][1] = 7;
```



Function Pointers

- Points to executable code in memory instead of a data value.
- Dereferencing a function pointer yields the referenced function.
- Notice how there are parentheses around the function pointer.
  - Without them, the function pointer would be parsed as a function declaration.

```c
#include <stdio.h>

void increment(int *a) {
  *a += 1;
  return;
}

int main(void) {
  // Make func_ptr point at increment().
  void (*func_ptr)(int *) = increment;

  int x = 5;

  // Increment x by invoking increment().
  func_ptr(&x);
  return 0;
}
```

Function Pointer Example

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

//
// Numerically computes the derivative of a function f about the point x0.
// Takes a double as an argument and returns a double.
//
double diff(double (*f)(double), double x0) {
  double epsilon = 0.00001;
  double delta_x = f(x0 + epsilon) - f(x0 - epsilon);
  return delta_x / (2 * epsilon);
}

int main(void) {
  for (double x = 0.0; x < (2 * M_PI); x += (M_PI / 16)) {
    printf("x = %6.3f, cos(x) = %6.3f, d/dx(cos(x)) = %6.3f\n"
      i, cos(x), diff(sin, x));
  }

  return 0;
}
```

Function Tables (Jump Tables)

```c
#include <stdio.h>

enum { INCREMENT, DECREMENT };

void increment(int *a) {
  *a += 1;
  return;
}

void decrement(int *a) {
  *a -= 1;
  return;
}

int main(void) {
  // func_table is an array of function pointers.
  // func_table[0] is increment().
  // func_table[1] is decrement().
  void (*func_table[])(int *) = { INCREMENT, DECREMENT };

  int x = 5;

  func_table[INCREMENT](&x);  // Increments x to 6.
  func_table[DECREMENT](&x);  // Decrements x back to 5.
  return 0;
}
```