CSE 374: Programming Concepts and Tools

Spring 2026
Instructor: Megan Hazen

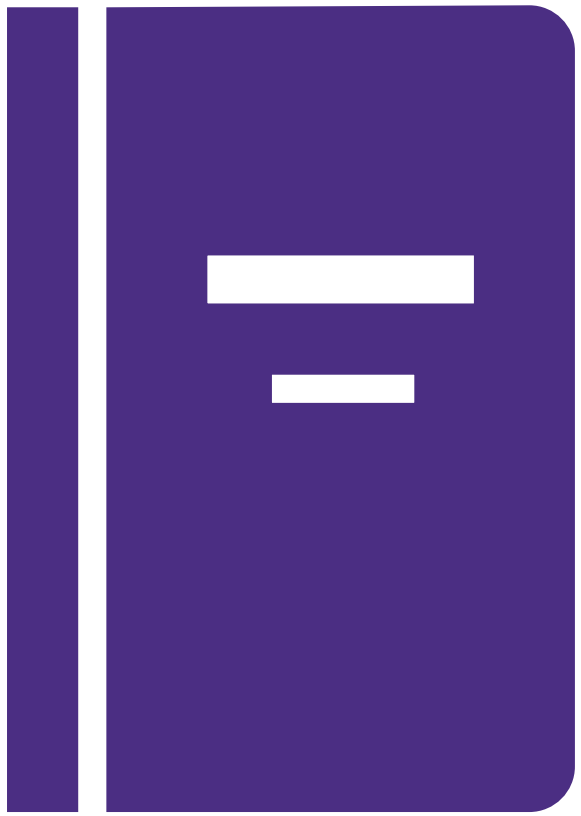
Today's Goals

Subject Matter

- More about C and memory management

Your Goals

- Relax – you got this



Aside: Assessment 1

- 50 minute written assessment
- Worth 25 points
- MONDAY, starts at 9:30am
- NO notes/books/phones
- But BRING A PEN.*
- Covers Lectures 1-7
- Study by doing practice exercises and homework
- Let us know ASAP if there is an issue – a missed assessment earns zero points!

Hello World

- Compile at a bash command line with `gcc hello.c`
  - Creates an executable `a.out`
- `gcc –Wall –std=c11 –o hello hello.c`
  - `Wall` turns on all warnings
  - `C11` specifies C11 standard
  - Creates executable hello
- Run: `./a.out` or `./hello`
- Exits with `0` (`return 0;`)

```
echo ‘alias ccomp=“gcc –Wall –std=c11 –o”’ >> .bashrc
```

```c
#include <stdio.h>
#define REPS 5


/**
 * Compile this file with:
 *      gcc -o hello hello.c
 * Run the program:
 *      ./hello
 */
int main(int argc, char **argv) {
  for (int i=0; i<REPS; i++) {
    printf("Hello, World!\n");
  }
  return 0;
}
```

Working Memory

Address space: list of bytes addressed in order

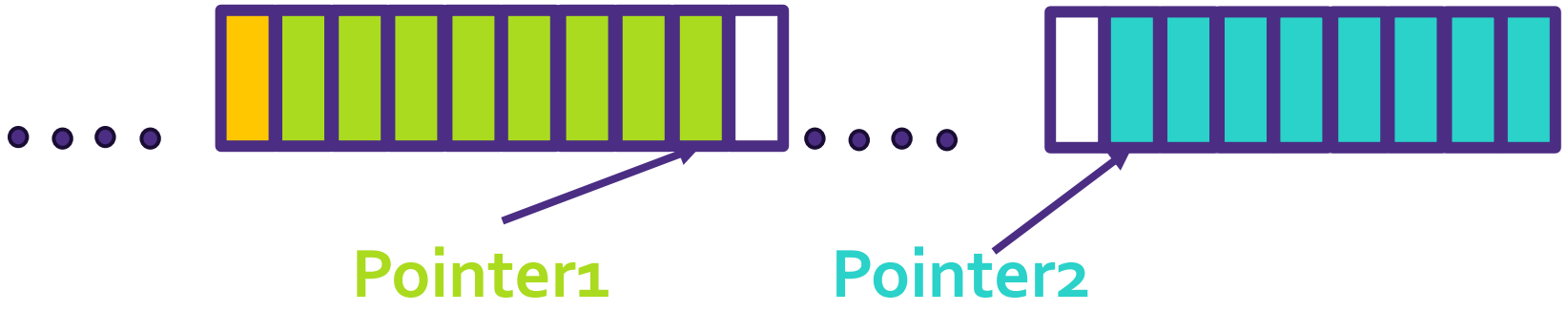
- Programs are said to have access to this 2^{64} byte space
  - '64 bit' system refers to needing 64 bits to index the space
  - But really don't - many other things are also using this space
  - Addressing is also usually 'virtual', but, this is a good model for us
- Location in array is the 'address' of a byte
- Programs keep track of addresses of each of their pieces of memory
- Accessing unused address causes a 'segmentation fault'

Working Memory - zones

Code Globals Heap-> <-Stack

- Lowest memory stores program instructions, then global variables (static constants, string literals)
- 'Heap' holds dynamically allocated variables ('new' or 'malloc' variables)
- 'Stack' holds current instructions, each function in a frame
  - 'Stack' memory implies that a frame is added, and then the last frame added is removed first
- The heap and stack grow dynamically. Meet in the middle ?= 'out of memory' error

So, what is a pointer?



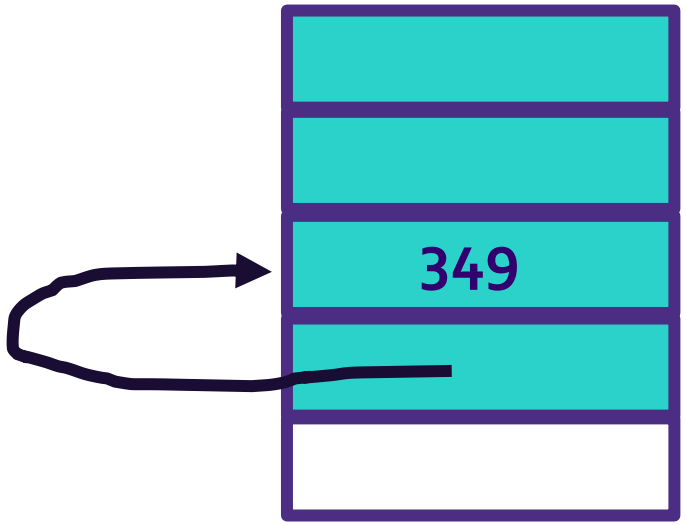
- Point to an address in memory

Code	Meaning
`int x = 4;`	Variable called 'x' of type 'int' storing value of '4'
`int *xPtr = &x;`	Variable called 'xPtr' ot type 'pointer to int', given value of 'the location of x'
`int xCopy = *xPtr;`	Variable called 'xCopy' of type 'int' storing value at the location pointed to by xPtr
`int *noPtr = NULL;`	Variable called 'noPtr' of type 'pointer to int' correctly set to 'no location'

Pointers

- Pointers hold an address in memory
- Pointers can have a type (T) (int/float)
  - Specifies how much memory is at that address
  - Specifies how to interpret bits at that address

- Declare: `T* x;` or `T *x;` or `T*x;`
- Get value at address: `*x`
  - dereference
- Get address of variable: `&y`



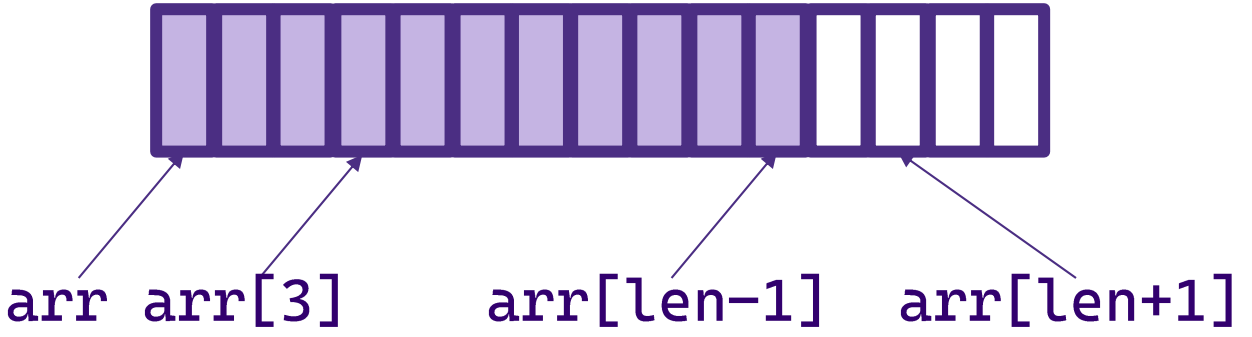

```
int var = 349;
int *varptr = & var;
```

Pointer Sizes / Types

- Data types (T) take up varying bytes of memory – an int uses 4 bytes.
- How big is a pointer?
  - On a 64 bit machine, 64 bits, or 8 bytes

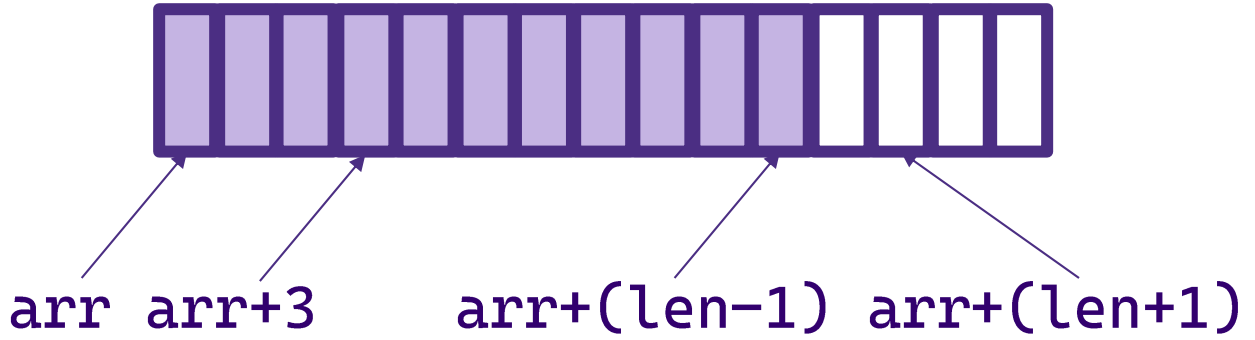
```
int var = 349;
int *varptr = & var;
```

Arrays



- Arrays are contiguous blocks of memory
  - `Datatype arr[len]`
  - Has type: `Datatype*`
- Why?
  - Array types actually store the address at the beginning of the block of memory
- What happens with arr[len+1]?
- **Best case scenario: you crash.**

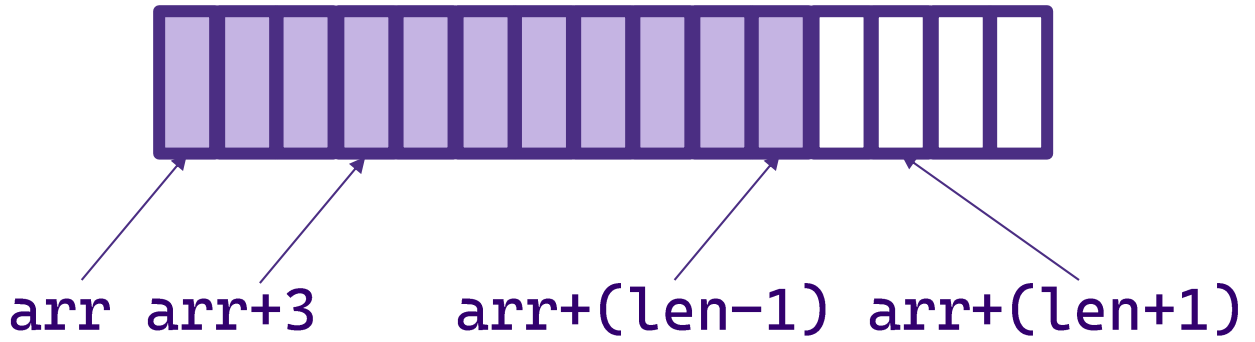
Pointer arithmetic



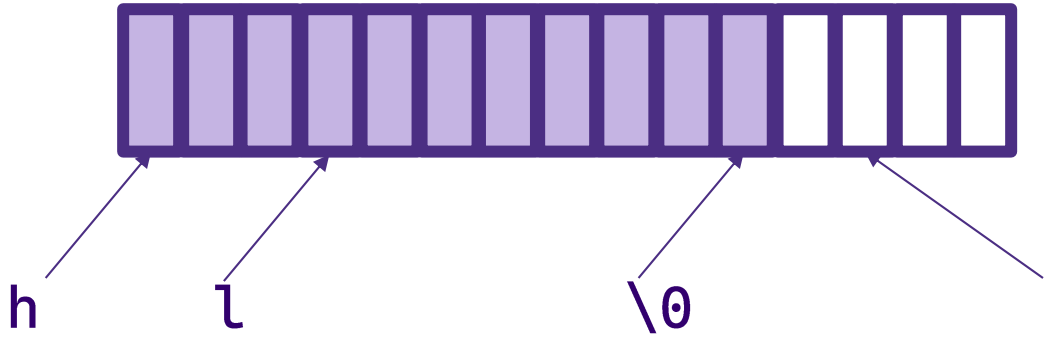
- If `p` has type `T*` or `T[]`
  - `*p` has type `T`
- If `p` points to one item of type `T`, `p+1` points to a place in memory for the next item of type `T`
  - So, `p[0]` is one item of type `T`, `p+i = p[i]`
- `T[]` always has type `T*`, even if it is declared as `T[]`
  - Implicit array promotion
    Result: Arrays are always passed by reference (the address of the data), not by value.

Pointer arithmetic Fun Fact!

- *You can do this outside arrays.*
- *It always works because pointers are just numbers*



Strings



- Strings are just arrays of characters
  - `char string[len]`
  - Has type: `char*`

```
[ "h", "e", "l", "l", "o", " ", "w", "o", "r", "l", "d", "!", \0 ]
```

Strings terminate with `\0` so their length can be determined

```
char str[] = "hello";  // array syntax
char *str2 = "hello";   // pointer syntax
char *arrStr[] = {"ant", "bee"};  // array containing char*'s
```

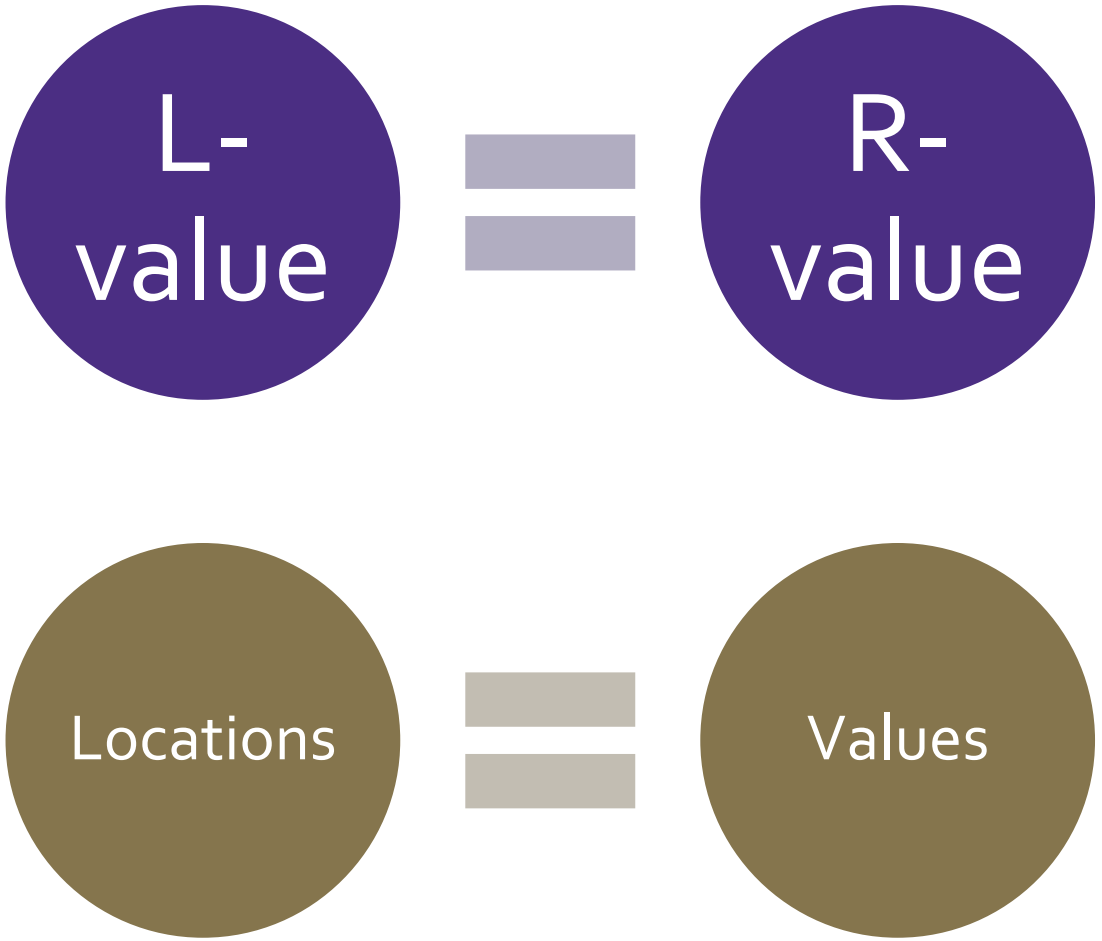
Pointers to pointers

- Levels of pointers make sense:
  - argv, *argv, **argv
  - argv, argv[0], argv[0][0]
- But an address of an address doesn't
  - &(&x) NOPE

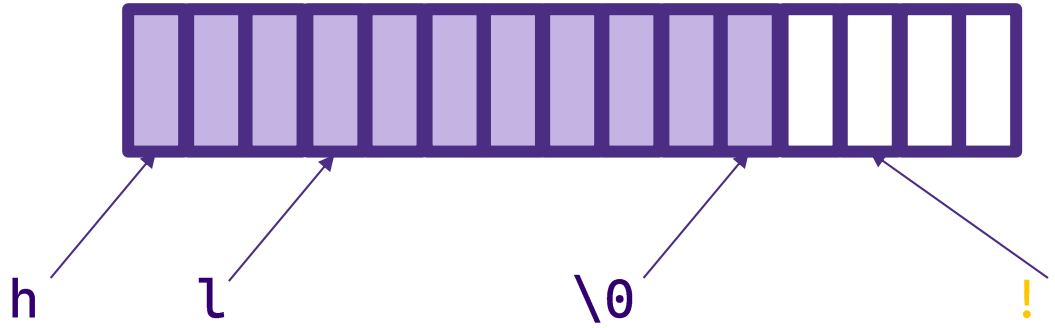
```
void f(int x) {
int *p = &x;
int **q = &p;  //ok
// x, p, *p, q, *q, **q
}
```

L. Values v R. Values

```
9 = x;          // 9 isn't a LOCATION
int x = 1;
    // VALUE 1 at a LOCATION which has the LABEL x.
x = 2;          // VALUE 2 at the LOCATION x.
int* xPtr = &x;
    // VALUE of address of x at LOCATION xPtr.
*xPtr = 3;
    // VALUE 3 at LOCATION at address in xPtr.
int** xx = &(&x);
    // r-value doesn't resolve to a value.
```



String Arrays



- Strings are just arrays of characters
  - `char string[len]`
  - Has type: `char*`

```
[ "h", "e", "l", "l", "o", " ", "w", "o", "r", "l", "d", "!", \0 ]
```

Strings terminate with `\0` so their length can be determined

```
char str[] = "hello";  // array syntax
char *str2 = "hello";   // pointer syntax
char *arrStr[] = {"ant", "bee"};  // array containing char*'s


char **arrStrPtr = arrStr;  // pointer to an array containing char*'s
arrStr[0] = "cat";
```

So, now we can talk about `argv`

- The C main can have arguments:
  - `int argc, char **argv`
  - Any guesses?
- `char` - datatype
- `char*` - pointer to a place in memory that stores a char
- `char**` - pointer to a place in memory that stores pointers to chars
- The variable `argv` holds `argc` pointers to `char* ptrs`
  - In c array lengths must be sent as separate arguments, as is done here
- Also access values with `argv[0]`, `argv[1]`, …. `argv[argc-1]`

```c
#include <stdio.h>
#define REPS 5


/**
 * Compile this file with:
 *      gcc -o hello hello.c
 * Run the program:
 *      ./hello
 */
int main(int argc, char **argv) {
  for (int i=0; i<REPS; i++) {
    printf("Hello, World!\n");
  }
  return 0;
}
```

Getting arguments

BASH

```
#!/bin/bash
echo "0: $0"
for file in "$@"; do
    echo "$file"
done


while [ $# -gt 0 ]; do
  echo "$1"
  shift
done
```

C (printargs.c)

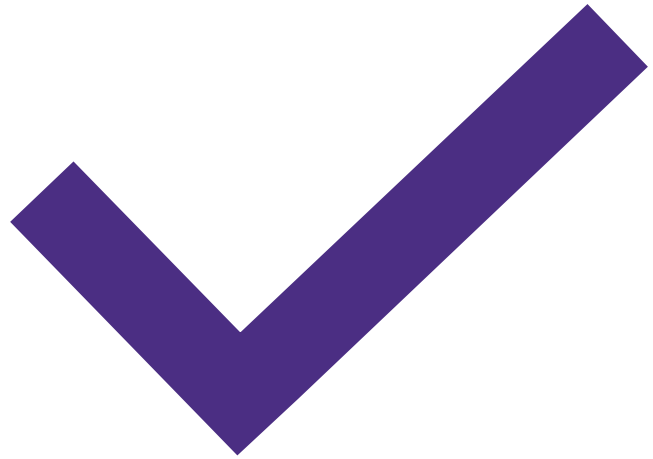
```
#include <stdio.h>


int main(int argc, char ** argv) {
  int k;
  printf("argc = %d\n", argc);
  for (k = 0; k < argc; k++)
   printf("argv[%d] = %s\n", k, argv[k]);
  return 0;
}
```

Declaring variables

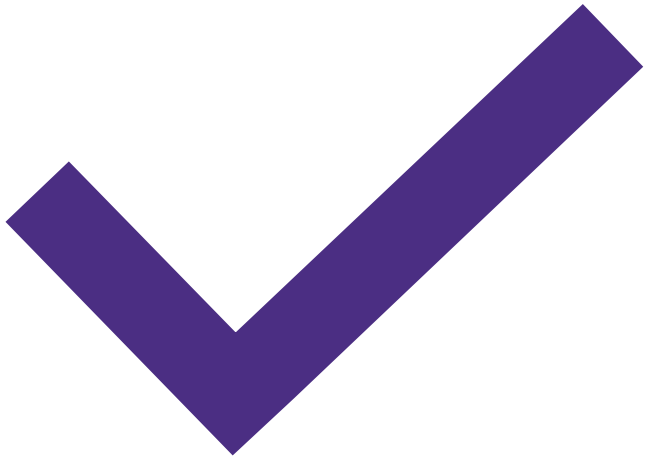
- You can put multiple declarations on one line
  - `int x, y;` or `int x=0, y;` or `int x, y=0;`, or …
  - But `int *x, y;` means `int *x; int y;`
    - you usually mean (want) `int *x, *y;`
- Common style rule: one declaration per line (clarity, safety, easier to place comments)
- Arguments to functions are comma separated, with their own datatype
- Array types in function arguments are pointers(!)
  - `void func (int *integerarray, int arraylength)`

Defining variables

- You can declare things more than once
- But you **define** it ONLY once.
  - Defining allocates memory
- If you try to use an *undefined* item an error will occur.
- To use something before it is defined, you must declare it first
  - Forward declaration
  - Function prototypes
  - Shared global variables

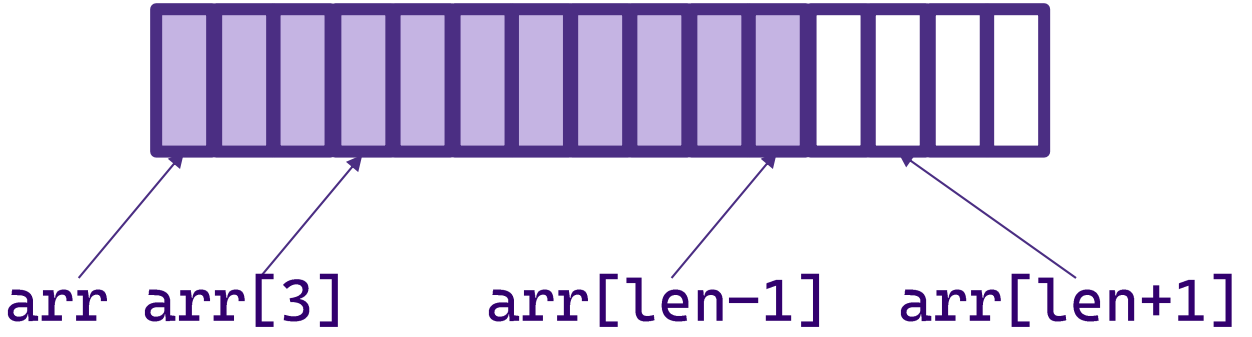


Initializing Variables



- Defining something allocates memory
  - But it doesn't initialize that memory
  - Doesn't put useful values in it….
- Unlike Java, you MUST provide a value to initialize memory
- If you access un-initialized bits, you may get weird and random values back
- **Best case scenario: you crash.**

Array definition



- Arrays are contiguous blocks of memory
  - `Datatype arr[len]`
  - Has type: `Datatype*`
- Definition means allocating memory
  - You MUST tell the array how many elements it has
  - How much memory to allocate
- If you need to have an array for which you don't know the size
  - Declare a pointer, set it to NULL
  - Dynamically allocate