What do you think?

Work with a partner(s):

Write a command that creates an *alias* called *compile*, which maps to the compile command for 374 C assignments.

What command do you need?
What flags do you need?
Run `compile outfile infile`

CSE 374 Lecture 9

Declarations, control, printf

Spot check: What is stored by the variable

```
int *ptrint;
```

How is it different than what is stored by the variable

```
int intarry[5];
```

Hello World in C

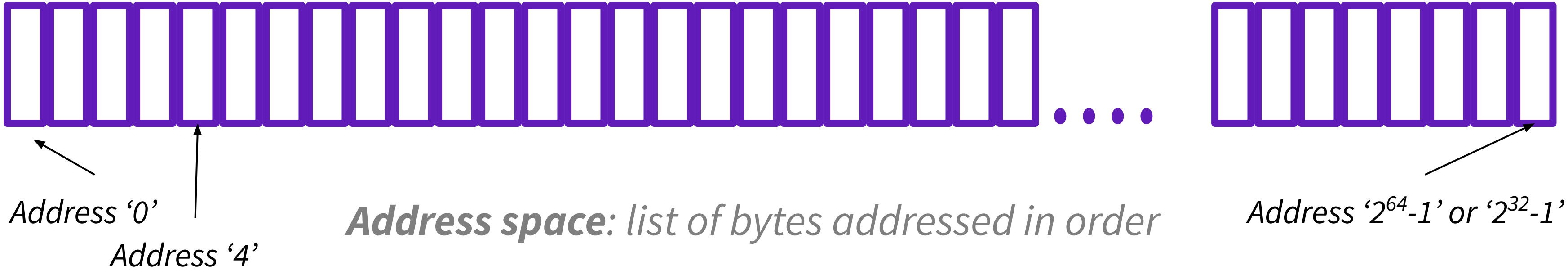
```
#include <stdio.h>

/**
 * Compile this file with:
 *     gcc -o hello hello.c
 */
int main(int argc, char **argv)
{
  printf("Hello, World!\n");
  return 0;
}
```

- Compile: `gcc hello.c`
  - creates executable `a.out`
- Or: `gcc -Wall -std=c11 -o hello hello.c`
  - Wall - turns all warnings on
  - C11 - specifies using C11 standard libraries
  - Creates executable `hello`
- Run: `./a.out` or `./hello`
  - Exits with ‘0’ (`return 0;`)

```
alias compile='gcc -Wall -std=c11 -o'
```

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
- 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'

Pointers

“Point to memory location”

`int x = 4;`	Variable called ‘x’ of type ‘int’ given value of ‘4’
`int *xPtr = &x;`	Variable called ‘xPtr’ of type ‘pointer to an integer’, given value of the location of ‘x’
`int xCopy = *xPtr;`	Variable called xCopy given the value stored at the location pointed to by xPtr
`int* noPtr = NULL;`	Variable ‘noPtr’ correctly set when location is not yet known

Pointer Review

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

Pointers point to an address in memory

`&x` returns the address

Declare a pointer to a pointer type and it has a specific type/size of memory:

`T *x;` or `T* x`; or `T * x`; or T*x

(T is a type, x is a variable)

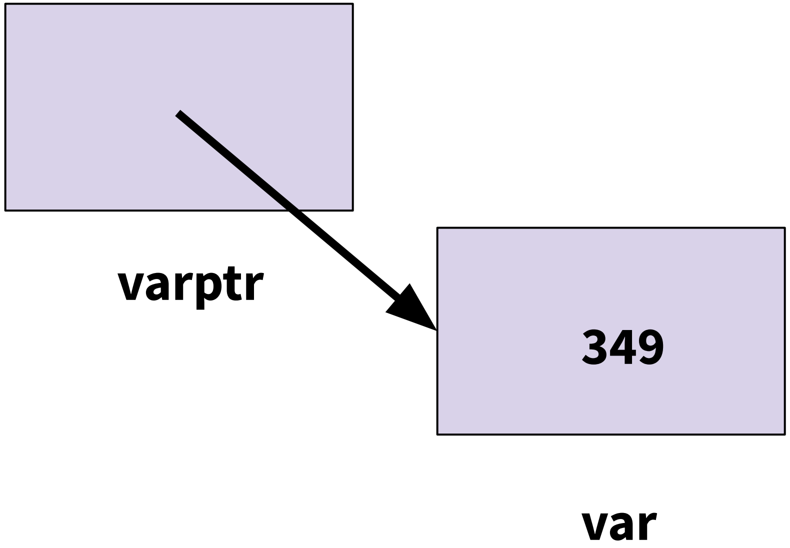
An expression to dereference a pointer

`*x` (more generally *expression)

Dereference - get the value at the address

Arrays have an implicit pointer type

`T = x[n]` implies x is of type T*



Pointer Review

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

Pointers point to an address in memory

`&x` returns the address

Declare a pointer to a poin

specific type/size of memo

`T *x;` or `T* x;` or `T * x`

(T is a type, x is a variable)

An expression to dereferen

`*x` (more generally *expression)

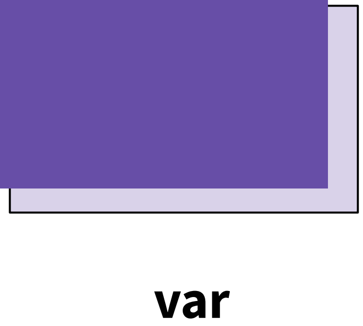
Dereference - get the value at the address

Arrays have an implicit pointer type

`T = x[n]` implies x is of type T*

How big (how many bytes) is an address?

Why?



Pointer Review

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

Pointers point to an address in memory

`&x` returns the address

Declare a pointer to a pointer t
specific type/size of memory:

`T *x;` or `T* x;` or `T * x;` or
(T is a type, x is a variable)

An expression to dereference a
`*x` (more generally *expressio

Dereference - get the valu

Why do pointers need a type if they are just addresses?

What can we do with that type?

var

Arrays have an implicit pointer type

`T = x[n]` implies x is of type T*

Arrays

Contiguous blocks in memory

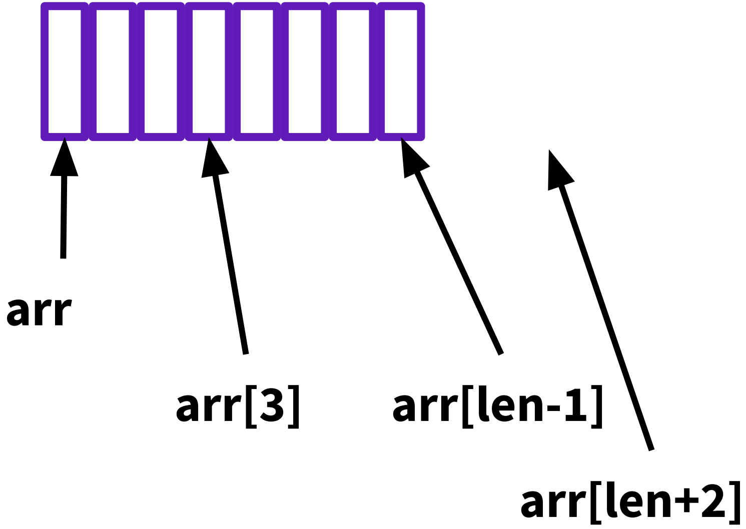
Declare as

```
Datatype arr[len]
```

Has type

```
Datatype*
```

Stores the location in memory of the first value; when arrays are passed passes this memory location



arr

arr[3]

arr[len-1]

arr[len+2]

Danger, Will Robinson!!

Pointers to pointers

Levels of pointers make sense:

I.e.: `argv, *argv, **argv`

Or: `argv, argv[0], argv[0][0]`

But

`&(&p)` doesn't make sense

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

Integer, pointer to integer, pointer to pointer to integer

`&p` is the address of '`p`',

`&(&p)` would be the address of the address of `p`, but that value isn't stored separately anywhere and doesn't have an address

Try using `printf ("The address of x is %p\n", &x);`

Strings

No real strings - just arrays of characters.

```
[ "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";
```

Pointer arithmetic

- If p has type T* or T[] and   *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, not by value. (The information passed is the address of where the values are stored.)

Arrays

Contiguous blocks in memory

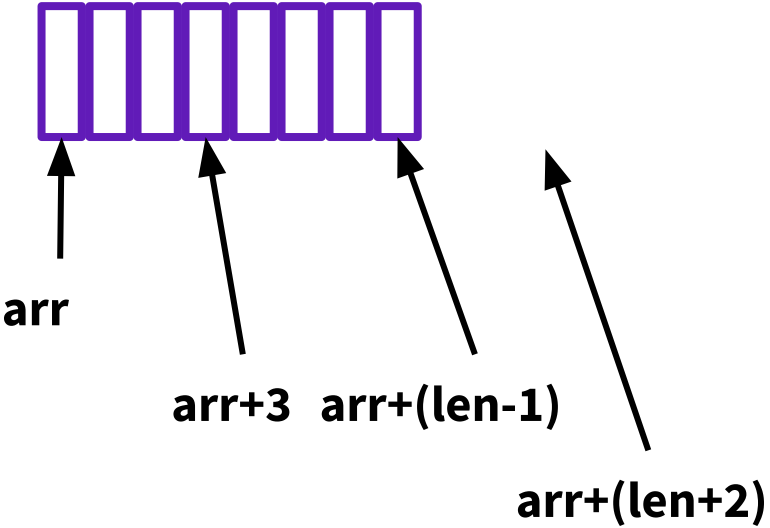
Declare as

```
Datatype arr[len]
```

Has type

```
Datatype*
```

Stores the location in memory of the first value; when arrays are passed passes this memory location



arr

arr+3

arr+(len-1)

arr+(len+2)

Danger, Will Robinson!!

Hello World in C

```c
#include <stdio.h>

/**
 * Compile this file with:
 *     gcc -o hello hello.c
 */
int main(int argc, char **argv)
{
  printf("Hello, World!\n");
  return 0;
}
```

- Compile: `gcc hello.c`
  - creates executable `a.out`
- Or: `gcc -Wall -std=c11 -o hello hello.c`
  - Wall - turns all warnings on
  - C11 - specifies using C11 standard libraries
  - Creates executable `hello`
- Run: `./a.out` or `./hello`
  - Exits with ‘0’ (`return 0;`)

```
alias compile='gcc -Wall -std=c11 -o'
```

What is char **argv ??

- 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 variables `argv` hold `argc` points 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]`

Okay, so, argv[i] ?

- Any argv[i] points to a char* (pointer to characters)
- char* - pointer to a place in memory that stores a char or multiple chars
- If char* points to an array of characters ending in \0 (a zero byte)
- Aka a string!!
- Argv are usually has arguments coded into strings

Arguments

Bash

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

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

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;
}
```

Printargs .c

Source File Structures

```
// includes for functions & types
defined elsewhere
#include <stdio.h>
#include “localstuff.h“
// symbolic constants
#define MAGIC 42
// global variables (if any)
static int days_per_month[ ] = { 31,
28, 31, 30, …};
// function prototypes
// (to handle “declare before use”)
 void some_later_function(char, int);
// function definitions
void do_this( ) { … }
char *return_that(char s[ ], int n)
{ … }
int main(int argc, char ** argv) { … }
```

Preprocessor

Pre-processes your C code before the compiler gets to it.

- ➔ Follows commands prefaced by ‘#’
- ➔ Includes content of header files
- ➔ Defines constants and macros
- ➔ Conditional compilation (not covered right now)

File inclusion

- ➔ `#include <foo.h>`
  - ◆ Searches for foo.h in “system include” directories (/usr/include, etc)
- ➔ `#include “foo.h”`
  - ◆ Starts by searching in current directory (allows coder to break project into smaller files)
- ➔ Include include file’s preprocessed contents
- ➔ Recursively include all the includes from original file
- ➔ Use `gcc -l dir1` to tell gcc to look for include in dir1

Demo magic.c

Preprocessor Cont.

Define constants

```
#define PI 3.14
#define NULL 0 // in stdlib

#define TRUE 1
#define FALSE 0
```

And macros

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

Constants are ALL_CAPS to differentiate them from other variables.

Defined constants will override variables of the same name used in the code.

Shadow with another #define, or, #undef

```
gcc -e control.c > controlpp
```

Declarations Cont.

You can put multiple declarations on one line, e.g., 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)

Array types in function arguments are pointers(!)

Definitions

Defines properties of item; this happens only ONCE, even if the item is declared more than once.

Linker-error will occur if an item is used but not defined.

To use something before it is defined, you must declare it before you use it (forward declaration).

```
int count=4

countptr = &count;

int count[3] = {1,2,3};

int adding(int a, int b) {
    return (a+b);
}


void printing (char *str){
    printf("%s\n", str);
}
```

L-values v. R-values

Left Side

Evaluated to locations (addresses) =

Right Side

Evaluated to values (the contents at the address)

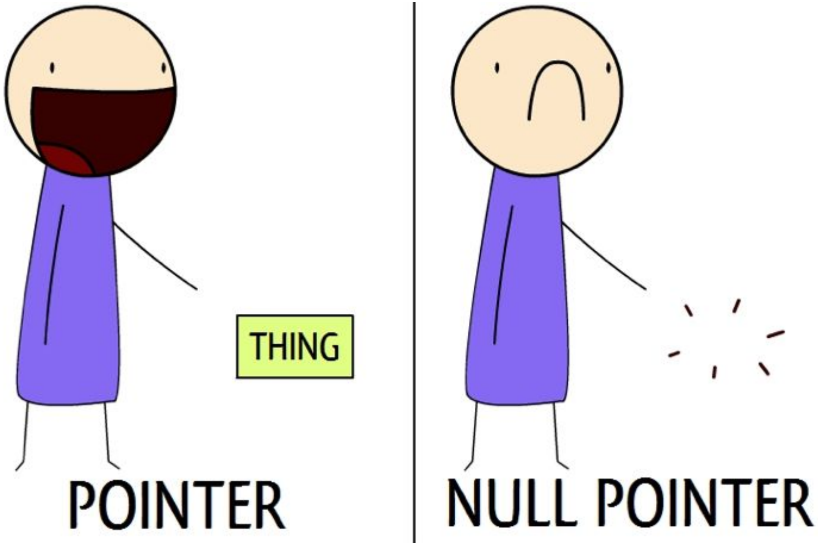
Values may be numbers (or characters) OR addresses

```
9 = x;     // Nonsense, because 9 isn’t a LOCATION
int x = 1; // Stores the VALUE 1 at a LOCATION which has the LABEL x.
x = 2;     // Stores the VALUE 2 at the LOCATION x.
int* xPtr = &x;  // Stores VALUE of address of x at a LOCATION labelled xPtr.
*xPtr = 3;  // Stores VALUE 3 at a LOCATION defined by address stored in xPtr.
int** xx = &(&x);// Nonsense, the r-value needs to resolve to a value.
                 // &x does indeed represent a value (the address x), but
                 // &(&x) refers to the address of the address of x -
                 // which is just a number and not stored anywhere
```

Definitions

- Int *arrspace = myArr;
- Arrays that rely on run-time info to determine size are dynamically allocated to the heap (and declared *array syntax)
- Define as NULL until otherwise defined.

https://www.codewithc.com/understanding-c-pointers-beginners-guide/



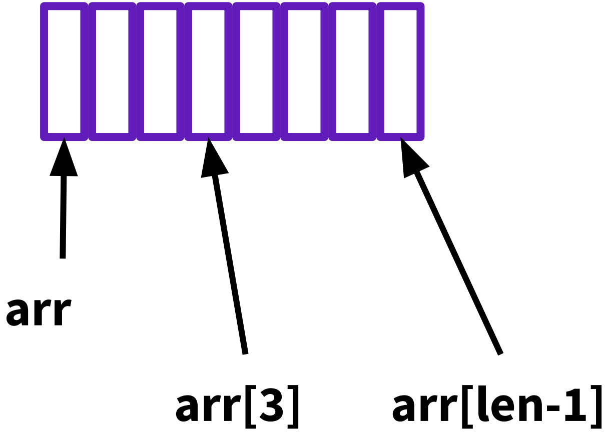
Initialization

Memory allocation and initialization are not the same thing

Unlike Java, you MUST provide a value to initialize a bit of memory

It is possible to access un-initialized bits
unlike Java which sets defaults and checks for initialization
best case scenario: you crash

Arrays



- int myArr[10];
  - User must store length (10).
- Int *arrspace = myArr;
  - Implicit conversion
- myArr[3] is ??
  - (Not automatically initialized to any value.)
- Arrays MUST be declared with a constant length (the compiler needs to allocate space)
- Arrays that rely on run-time info to determine size are dynamically allocated to the heap (and declared *array syntax)

Is your answer more nuanced?

Spot check: What is stored by the variable

```
int *ptrint;
```

How is it different than what is stored by the variable

```
int intarry[5];
```