Pointer arithmetic – arrays only

- Can add or subtract an integer – as long as result is still within the bounds of the array
- Can subtract a pointer from another pointer – iff both point to elements of the same array
  ```c
  char word[] = "cat";
  char *p = word;
  while (*p++ != '\0');
  printf("word length: %d", p-word-1);
  ```
- But – no pointer multiplication or division, and cannot add two pointers

Multi-dimensional and pointer arrays, and pointers to arrays

- Multi-dimensional arrays – arrays of arrays
  ```c
  int x[5][3]; /* allocates memory for 15 ints */
  ```
- Actually, 5 arrays, each able to store 3 integers
- Arrays of pointers
  ```c
  int *p[5]; /* allocates memory for 5 pointers */
  ```
- Now p can be used as an alias for x
- Pointers to arrays – require pointers to pointers
  ```c
  int **px = x; /* points to first array in x */
  ```

Command line arguments

- Declare main with two parameters
  ```c
  int main(int argc, char *argv[]) {
      ...}
  ```
- argc = 1 plus the number of tokens typed by the user at the command line after the program name
- argv[0] is the program name
- argv[1]...[argc-1] are the other tokens
- Each one points to an array of characters (i.e., a C string)

sizeof

- A unary operator – computes the size, in bytes, of any object or type
  ```c
  int size = sizeof x / sizeof x[i];
  ```
- Works for arrays too – total bytes in whole array
  ```c
  int size = sizeof x / sizeof x[i];
  ```
- Actual type of result is size_t
  ```c
  ```
- An unsigned integer defined in <stddef.h>
- Similarly, diff_t is result type of pointer subtraction
- Especially useful to find the sizes of structures

C structures

- Structures are variables with multiple data fields
  ```c
  struct example{ int x; double d; }
  ```
- Can still use array notation, but also can argv++ and so on

/\ copy t to s /*/ 

```c
void stringcopy(char *s, char *t)
{
    while (*s++ = *t++) /* Actually works! */
    { ...
    }
}
```
typedef and macros

- Can precede any declaration with `typedef`
  - Defines a name for the given type:
    ```c
    typedef struct example ExampleType;
    ExampleType e, *ep; /* e, ep same as prior slide */
    ```
- Very handy for pointer types too:
  ```c
  typedef ExampleType *ETPointer;
  ETPointer ep; /* ep same as above */
  ```
- Macros can simplify code too
  ```c
  #define X(p) (p)->x
  X(ep) = 8; /* preprocessor substitutes correct code */
  ```

Macros can simplify code too

And see:
`~mikec/cs12/demo01/*.c`

Unions

- Can hold different data types/sizes (at different times)
  ```c
  union myValue{
    int x;
    double d;
  } u, *up; /* u is a union, up can point to one */
  ```
- Access `x` or `d` by `u`, or `up` just like structures
- `sizeof u` is size of largest field in union
  ```c
  - Equals `sizeof(double)` in this case
  ```
- Often store inside a structure, with a key to identify type