This week:

- Arrays
  - one-dimensional
  - multidimensional
- Command-line arguments
- Assertions
Arrays

- What is an "array"?
- A way to collect together data of a single type in a single object
- A linear sequence of data objects e.g.
  - array of `ints`
  - array of `chars` (string)
Creating and using arrays

- One-dimensional array of three ints:

```c
int arr[3];
int sum;
arr[0] = 1;
arr[1] = 22;
arr[2] = -35;
sum = arr[0] + arr[1] + arr[2];
```
One-dimensional arrays (1)

- Arrays can be
  - initialized
  - partially initialized
  - not initialized

- Uninitialized space contains?
  - "garbage"
One-dimensional arrays (2)

Examples:

```c
int my_array[10];
    /* not initialized */
int my_array[5] = { 1, 2, 3, 4, 5 };
    /* initialized */
int my_array[] = { 1, 2, 3, 4, 5 };
    /* OK, initialized */
int my_array[4] = { 1, 2, 3, 4, 5 };
    /* warning */
int my_array[10] = { 1, 2, 3, 4, 5 };
    /* OK, partially initialized */
```
One-dimensional arrays (3)

- Note on partial initialization:
  
  ```c
  int my_array[10] = { 1, 2, 3, 4, 5 };  
  ```
  
  - rest of array initialized to 0
  
  ```c
  int my_array[10];  
  ```
  
  - entire array uninitialized
  
  - contains garbage
Explicit initialization of arrays:

```c
int i;
int my_array[10];
for (i = 0; i < 10; i++) {
    my_array[i] = 2 * i;
}
```

This is the most flexible approach
Some bad things that can happen...

```c
int my_array[10];
/* What happens here? */
printf("%d\n", my_array[0]);
/* What happens here? */
printf("%d\n", my_array[1000]);
```

- No checking!
- C is an UNSAFE language!
NOTE! The following is illegal:

```c
int my_array[5];
my_array = { 1, 2, 3, 4, 5 }; /* WRONG */
```

The `{ 1, 2, 3, 4, 5 }` syntax is *only* usable when declaring a new array, and not for reassigning the contents of the array

```c
int my_array[5] = { 1, 2, 3, 4, 5 }; /* OK */
int my_array[] = { 1, 2, 3, 4, 5 }; /* OK */
```
Two-dimensional arrays (1)

int arr[2][3]; /* NOT arr[2, 3] */
int i, j;
int sum = 0;
arr[0][0] = 1;
arr[0][1] = 23;
arr[0][2] = -12;
arr[1][0] = 85;
arr[1][1] = 46;
arr[1][2] = 99;
/* continued on next slide */
Two-dimensional arrays (2)

```c
for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++) {
        sum += arr[i][j];
    }
}
printf("sum = %d\n", sum);
```
Two-dimensional arrays (3)

- Two-dimensional arrays can be split into component one-dimensional arrays:

```c
int arr[2][3];
/* initialize... */
/* arr[0] is array of 3 ints */
/* arr[1] is another array of 3 ints */
```
Two-dimensional arrays (5)

- How `arr` is laid out in memory:

```plaintext
arr[0]    arr[1]
  1  23  -12   85  46  99
```
Two-dimensional arrays (6)

- Initializing two-dimensional arrays:

```c
int my_array[2][3];
    /* not initialized */
int my_array[2][3]
    = { { 1, 2, 3 }, { 4, 5, 6 } };
    /* OK */
int my_array[2][3]
    = { 1, 2, 3, 4, 5, 6 };
    /* warning with -Wall */
```
Two-dimensional arrays (7)

```c
int arr[2][2]
    = {{ 1, 2, 3 }, { 4, 5, 6 }};
/* invalid */

int arr[][3]
    = {{ 1, 2, 3 }, { 4, 5, 6 }};
/* invalid */

int arr[][3]
    = {{ 1, 2, 3 }, { 4, 5, 6 }};
/* OK */
```
Two-dimensional arrays (8)

```c
int my_array[][3]
    = { 1, 2, 3, 4, 5, 6 };
    /* warning with -Wall */
int my_array[][3]
    = { { 1, 2, 3 }, { 4, 5 } };
    /* OK; missing value = 0 */
```

- Rule: all but leftmost dimension must be specified
- Compiler can compute leftmost dimension
- OK to specify leftmost dimension as well
What does this do?

```c
void foo(int i) {
    i = 42;
}
/* later... */
int i = 10;
foo(i);  /* What is i now? */
```
Passing arrays to functions (2)

- Current value of \( i \) is copied into function argument \( i \)
- Passing a value to a function as an argument doesn't change the value
- We say that C is a "call-by-value" language
- But arrays are "different"!
  - (actually, not really, but it seems like they are; need pointers for full explanation)
Arrays passed to functions *can be modified:*

```c
void foo(int arr[]) {
    arr[0] = 42; /* modifies array */
}

/* later... */
int my_array[5] = { 1, 2, 3, 4, 5 };
foo(my_array);
printf("%d\n", my_array[0]);
```
Passing arrays to functions (4)

- Last array dimension in declaration is ignored for one-dimensional arrays:

```c
void foo2(int arr[5]) /* same as arr[] */
{
    arr[0] = 42;
}
```

- Same as `foo()`
Two-dimensional (or higher-dimensional) arrays can also be passed to functions.

However, must specify all array dimensions except for the leftmost one (which is optional).

- same rule as for initializing 2d arrays
int sum_2d_array(int arr[][3], int nrows) {
    int i, j;
    int sum = 0;
    for (i = 0; i < nrows; i++) {
        for (j = 0; j < 3; j++) {
            sum += arr[i][j];
        }
    }
    return sum;
}
Also OK to specify leftmost dimension:

```c
int sum_2d_array(int arr[2][3], int nrows){
    /* same as before */
}
```

Compiler still ignores leftmost dimension
- May need to pass it in as an extra argument e.g. as `nrows` here
Command-line arguments (1)

- http://courses.cms.caltech.edu/cs11/material/c/mike/misc/cmdline_args.html
- When you type this at the unix prompt:
  
  \% myprog inputfile outputfile

- This is a command line
- First word is the program name (myprog)
- Other words are the program arguments
- Here: inputfile, outputfile
Command-line arguments (2)

Arguments give program information it needs
- e.g. names of files to read from/write to
- or data the program needs

Can also have *optional* arguments

- sorter 5 1 3 2 4
- sorter \(-b\) 5 1 3 2 4
  - \(-b\) is optional
  - changes the way the *sorter* program works
  - convention: all arguments starting with "-" are optional (unless they're e.g. negative numbers)
Recall: strings are arrays of characters (\texttt{char []})
Also written (\texttt{char *}) (see why later)
Command line arguments are divided into
\begin{itemize}
  \item \texttt{int argc} (argument count)
  \item \texttt{char *argv[]} (array of strings)
  \item read as: \texttt{(char *) argv[]}
  \item not allowed to write \texttt{char argv[][]}
\end{itemize}
To use command-line arguments, `main` function needs to have 2 new arguments: `argc` and `argv`

```c
int main(int argc, char *argv[]) {
    /* argc is the number of arguments
    * argv is the arguments,
    * represented as an array of strings.
    */

    /* ... code goes here ... */
}
```
Cmdline args are `argv[0], argv[1], ...`

`argv[0]` is name of program

In previous example:

- `argv[0]` → "myprog" (program name)
- `argv[1]` → "inputfile"
- `argv[2]` → "outputfile"
We usually process command-line arguments in `main()`:

```c
#include <string.h>

int main(int argc, char *argv[]) {
    int i;
    /* process command-line arguments */
    for (i = 1, i < argc; i++) {
        if (strcmp(argv[i], "-b") == 0) {
            /* process optional argument */
        }
        /* process non-optional arguments */
    }
    /* ... rest of program ... */
}
```
Useful functions for command-line argument processing:

- `atoi()` – converts string to `int`
  - `atoi("123")` → 123
  - in `<stdlib.h>`

- `strcmp()` – compares strings
  - `strcmp("foo", "foo")` → 0
  - in `<string.h>`
Notes on `strcmp()`:

- `strcmp()` returns 0 if strings are the same, nonzero otherwise.
- Do not use `==` to compare strings!
  - You can use it, but it won't do what you expect.
  - Always use `strcmp()` instead.
Assertions (1)

- Sometimes expect code to behave in a certain way
- *e.g.* `sort()` function should sort its input
- Would like to make programs self-checking
- An assertion is a "sanity check" on code
- "If there are no bugs in this code, this must be true at this point in the code."
  - This is the kind of thing assertions check
Example:

Assume have a function called `sorted()` that returns 1 if array sorted, else 0

Can use `assert()` in conjunction with `sorted()` to check arrays for sortedness every time they're sorted
Assertions (3)

```c
#include <assert.h>
void sort(int arr[], int nelems) {
    /* ...sort the array somehow... */
    assert(sorted(arr));
    /* "sorted" defined somewhere else;
    * returns 1 if array is sorted;
    * otherwise returns 0. */
}
```

- If assertion fails, program terminates
  - file and line number of failure is printed
Assertions (4)

- Assertions make program slower
  - but usually not much
- Use only to check *logical correctness* of code
  - "What *must be true* at this point in the code?"
- Don't try to use them to check e.g. user input
  - Example: user should enter a number between 1 and 10
  - Don't use `assert()` to check this!
Assertions (5)

- After debugging, may not need them anymore (you know code is correct)
- Might not want the slowdown
- Might want to turn off assertions
Assertions (6)

- Command-line argument to `gcc` that turns off assertions:

  % gcc -DNDEBUG program.c -o program

- `NDEBUG` means "Not DEBUGging"
- `-D` means "define" (don't worry for now)
- Now assertions are just ignored
- Program will run faster
  - but if assertion is violated, you won't know!
Next week

- Pointers!
  - The one hard topic in C programming
  - Will take several weeks to cover thoroughly