



CS 11 C track: lecture 3

- 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:

```
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:

```
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:

```
int my_array[10] = { 1, 2, 3, 4, 5 };
```

- rest of array initialized to 0

```
int my_array[10];
```

- entire array uninitialized
- contains garbage



One-dimensional arrays (4)

- Explicit initialization of arrays:

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

- This is the most flexible approach



One-dimensional arrays (5)

- Some bad things that can happen...

```
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!



One-dimensional arrays (6)

- NOTE! The following is illegal:

```
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

```
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)

```
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)

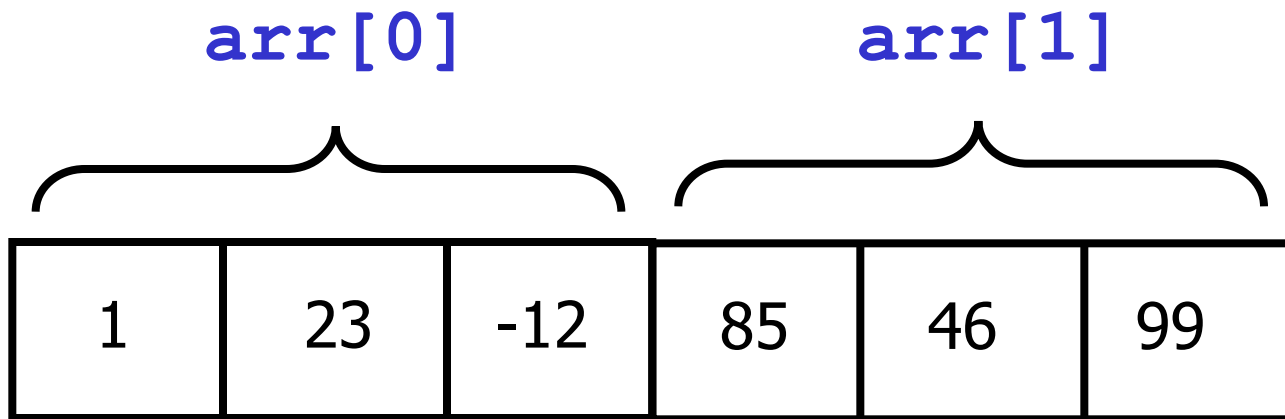
- Two-dimensional arrays can be split into component one-dimensional arrays:

```
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:





Two-dimensional arrays (6)

- Initializing two-dimensional arrays:

```
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)

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



Two-dimensional arrays (8)

```
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



Passing arrays to functions (1)

- What does this do?

```
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)



Passing arrays to functions (3)

- Arrays passed to functions *can* be modified:

```
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:

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

- Same as `foo()`



Passing 2D arrays to functions (1)

- 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



Passing 2D arrays to functions (2)

```
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;  
}
```



Passing 2D arrays to functions (3)

- Also OK to specify leftmost dimension:

```
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)



Command-line arguments (3)

- Recall: strings are arrays of characters (`char []`)
- Also written (`char *`) (see why later)
- Command line arguments are divided into
 - `int argc` (argument count)
 - `char *argv[]` (array of strings)
 - read as: `(char *) argv[]`
 - not allowed to write `char argv[][]`



Command-line arguments (4)

- To use command-line arguments, `main` function needs to have 2 new arguments: `argc` and `argv`

```
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 ... */  
}
```



Command-line arguments (5)

- 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"`



Command-line arguments (6)

- We usually process command-line arguments in `main()`:

```
#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 ... */
}
```



Command-line arguments (7)

- 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>`



Command-line arguments (8)

- 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



Assertions (2)

- 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)

```
#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 **DEBUG**ging"
- **-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