Pointers and Arrays

Class 21

Whole-Array Assignment

- I would like to copy an entire array's values to another array

```
#define SIZE 4
int array1[] = {-2, -1, 0, 1};
int array2[SIZE];

array2 = array1;
```

Whole-Array Assignment

- I would like to copy an entire array's values to another array

```
#define SIZE 4
int array1[] = {-2, -1, 0, 1};
int array2[SIZE];

array2 = array1;
```

- this will not work!
- remember: the name array1 refers to the address of the first byte of the first element of array1
- `array2 = array1;` is interpreted as, "change the place where array2's first byte is to the same place where array1's first byte is"
- but you cannot move the place where a variable is located in memory to a different place

Whole-Array Assignment

- the only way to copy an array's values to another array is element-by-element

```
size_t index;
for (index = 0; index < SIZE; index++)
{
  array2[index] = array1[index];
}
```

Whole-Array Comparison

- I would like to see if two arrays have the same elements

```
int array1[] = {-2, -1, 0, 1};
int array2[] = {-2, -1, 0, 1};

array1 == array2 /* should be true? */
```

Whole-Array Comparison

- I would like to see if two arrays have the same elements

```
int array1[] = {-2, -1, 0, 1};
int array2[] = {-2, -1, 0, 1};

array1 == array2 /* should be true? */
```

- this will not work!
- remember, array1 is just an address (say, 0x861a)
- and array2 is a different address (say, 0x862c)
- array1 == array2 really means 0x861a == 0x862c, which is clearly false

Whole-Array Comparison

- the only way to compare an array's values to another array is element-by-element
- what would the code for this look like?

Whole-Array Comparison

- the only way to compare an array's values to another array is element-by-element
- what would the code for this look like?

```
unsigned same = 1;
size_t index = 0;
while (same && index < SIZE)
{
  same = array1[index] == array2[index];
  index++;
}
return same;
```

Array Elements as Function Parameters

- array elements are simple variables
- they can be used anywhere "normal" variables can

```
unsigned values[] {10, 15, 20};
unsigned remainder;
...
remainder = compute_modulus(values[2], values[0]);
```

- the two parameters are pass by value
- any changes made to them in the function do not affect the array, because the values are copied into the function

Pointer Parameters

- a pointer can easily be a parameter to a function
- let's look at pass-by-pointer functions

```
1  void get_number(int* input);
2  void double_value(int* value);
3
4  int main(void)
5  {
6    int number;
7
8    get_number(&number);
9    double_value(&number);
10   printf("the value you entered, doubled, is %d\n", number);
11   return 0;
12 }
13 void get_number(int* input)
14 {
15   printf("enter an integer: ");
16   scanf("%d", input);
17 }
18 void double_value(int* value)
19 {
20   *value *= 2;
21 }
```

Arrays as Function Parameters

- an array name is the starting address of the array in memory
- an array cannot be copied in one step, but only one element at a time

Arrays as Function Parameters

- an array name is the starting address of the array in memory
- an array cannot be copied in one step, but only one element at a time

- thus an array cannot be passed by value into a function
- arrays can only be passed by pointer into a function

- because an array's name is simply the address of the array, it already is a pointer

```c
#define SIZE 6
void show_values(int values[], size_t size);

int main(void)
{
  int numbers[] = {5, 10, 15, 20, 25, 30};

  show_values(numbers, SIZE);
  return 0;
}

void show_values(int values[], size_t size)
{
  size_t index;
  for (index = 0; index < size; index++)
  {
    printf("%d ", values[index]);
  }
  printf("\n");
}
```

Arrays as Function Parameters

- several important things to note
  - lines 2 and 12: empty square brackets denote this is an array parameter
  - an array parameter is a pointer parameter
  - lines 2 and 12: we must pass the size of the array to the function because otherwise the function cannot determine the array size
- since an array passed to a function is a pointer, a change made to the array inside a function does affect the array in the calling scope

`const` Array Parameters

- the fact that you cannot pass an array by value is problematic
- for "normal" variables, if you do not want a function to change their value, you use pass by value
- this prevents any change from affecting the calling scope

- but you cannot pass an array by value
- how do you prevent the function from being able to alter the array?

- you declare a const array parameter
- always declare an array parameter const if the function will not change the array

```
1  void show_values(const int values[], size_t size);
2
3  int main()
4  {
5    int numbers[] = {5, 10, 15, 20, 25, 30};
6
7    show_values(numbers, SIZE);
8    return 0;
9  }
10
11 void show_values(const int values[], size_t size)
12 {
13   size_t index;
14   for (index = 0; index < size; index++)
15   {
16     printf("%d ",values[index]);
17   }
18   printf("\n");
19 }
```

Syntactic Sugar

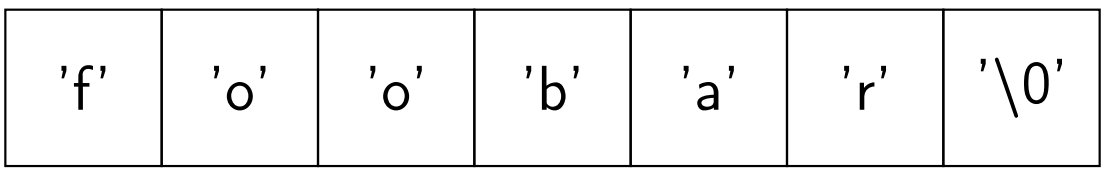
- `int array[]` and `int* array` are identical
- so the previous prototype could just have easily been written:

```
void show_values(const int* values, size_t size);
```

C-Strings

- a C-string is an array of characters
- after the "real" characters, the final character is the null character, the character with ASCII code 0
- the null character is written '\0'
- '\0' is just 0, but emphasizes it's a character

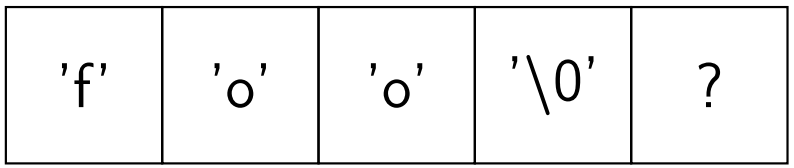
- a string literal enclosed in double quotes is stored in memory as a null-terminated C-string: `"foobar"` is



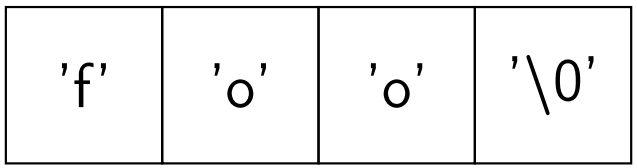
C-String Variables

- you can declare C-string variables

```
char name[5] = "foo";
```




```
char name[] = "foo";
```



- a three-character string occupies four array elements
- one element is required for the null character

C-string Output

- once a C-string (an array of characters) exists, it can be used for output

```
char name[10] = "Fred";
printf("%s\n", name);
```

- printf stops outputting characters when the null character is encountered
- regardless of the array size

'F'	'r'	'e'	'd'	\0	?	?	?	?	?

The Null Character

- everything about a C-string depends on the null character
- if something happens to that null character, everything goes south

```
char stuff[] = "foobar";
char name[] = "Ann"; /* name[3] is \0 */
name[3] = 'x'; /* replace \0 with x */
printf("%s\n", name);
```

```
output: Annxfoobar
```

- once the null character is gone, printf doesn't know where to stop
- printf keeps going until it reaches a byte equal to zero

A Subtle Error

```
1  int main(void)
2  {
3    char foo[] = "foobar";
4    char name[5] = "Becky";
5    printf("%s\n", name);
6    return 0;
7  }
```

output: Beckyfoobar

gets

- K&R blithely talk about the gets routine for string input

- NEVER use gets
- show test_gets.c

- however, fgets is safe and is used extensively
- show echo_fgets.c
- fgets notes:
  - will read at most MAX - 1 characters
  - reads the newline character and stores it in the string also

Array of Strings

- a C-string is an array of chars
- we can have an array of C-strings, which is an array of an array of char

```
char* poem[] = {"Roses are red", "Violets are blue",
                "Sugar is sweet", "Wish you were, too."};
```

- when you define an array of strings, the index before the last must be given, explicitly or implicitly

Legal:

```
char poem[4][] = {"Roses are red", "Violets are blue",
                  "Sugar is sweet", "Wish you were, too."};
```

Illegal:

```
char poem[][] = {"Roses are red", "Violets are blue",
                 "Sugar is sweet", "Wish you were, too."};
```

```
char** poem = {"Roses are red", "Violets are blue",
               "Sugar is sweet", "Wish you were, too."};
```

Declaration

- but a declaration (e.g., function prototype) does not contain size information
- all these are ok:
  `char** poem` ← this is typically preferred
  `char* poem[]` ← this is ok also
  `char poem[][]` ← rarely used for an array of strings

Command Line Arguments

- main can call any other function in your program
- but what calls main?

- the operating system calls main for you when you invoke the program, by entering the program's name at the terminal prompt, or by double-clicking on the program on a GUI

- when the operating system calls main, it passes the command line arguments to main as actual parameters

main Parameters

- so far, our main function has had no parameters
- but two formal parameters are in fact defined for it
  1. argc: an integer that is the count of arguments
  2. argv: an array of strings, i.e., an array of arrays of characters, which are the arguments themselves as strings
     argv stands for "vector of arguments"

```
int main(int argc, char** argv)
{ ...
```

main Parameters

```
int main(int argc, char** argv)
{ ...
```

- argc tells how many command line arguments were passed to main (just like BASH $#)
- there is always one argument (argv[0]), which is the name of the program itself (just like BASH $0)
- the other arguments (if any) are strings that can be addressed as array elements argv[1], argv[2], etc (just like BASH $1 $2 etc)

main Parameters

a simple program to echo command line arguments to screen

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

run echo_command_line

A More Substantial Program

```
$ cp -av /tmp/toy_wc.c .

$ rsync -vutz username@sand.truman.edu:/tmp/toy_wc.c .
```