CS 449 – Lecture 3

Dr. Jack Lange

Variables in memory

- **`int x = 4;`**
- X is a handle to 4 bytes of memory assigned by the compiler
  - `&x;` → returns the memory location
- What if we want to modify the value of x in another function?
  - E.g. we want another function to modify x
- In C we cannot just pass x as an argument
  - C passes arguments by value, not by reference
  - Another big difference from Java

Passing by reference in C

- We have to send the location of x as an argument
  - But the standard types do not allow that
- We need a special data type that indicates it is an address
  - **These are pointers**
- What are pointers?
  - A primitive data type whose value is a memory location (address)
- How big are pointers?
  - Same size as the HW address space
    - 32 bits on i386
    - 64 bits on x86_64

Pointers

- A pointers is an alias, a renaming of a variable
  - Similar to a file shortcut/alias on a desktop
- `int * pi; -> '*' indicates a pointer`
- A pointer includes the type of data it is pointing to
  - In this case an int
  - Can be any C data type and a special `void` type

Using Pointers

```
int x = 4;
int * y = &x;
```

OR

```
int * y = NULL;
```

→ Why is this set to NULL?

How do we get the data value pointed to? **`Answer: *y;`**

A leading '*' **dereferences** a pointer

Pointers Examples

```
int   x =  4;
int * y = &x;
int   z = *y;
printf("z = %d\n", z);
```

➔ `z = 4`

- **Examples:**
  - **dereference.c**
    - Getting around pass by value
  - **ptrSwap.c**
    - Swapping two numbers in another function
  - **dangerous.c**
    - Be very careful about dereferencing
    - This is where a LOT of bugs happen
    - Pointers are very powerful, but can be very dangerous

Arrays

- Arrays are a natural data type in C
  - Memory is an array
  - Just a sequence of values back to back in memory
- Raw values of a single data type
  - This means that arrays cannot change size
  - Arrays must be initialized with the correct size
- Initialization
  - `int a[] = {1, 2, 3, 4, 5, 6};`
  - `int c[6];`
- Accessing arrays
  - `int x = c[2];`
  - The compiler knows where in memory `c` starts
  - … so it just goes to the correct offset in memory

Array examples

- arrays.c
- Swapping array values
  - arraySwap.c
- Aside:
  - In C the operator precedence is postfix before prefix
  - Any postfix operator will be applied before the prefix

What is the value of an Array?

- The variable of an array (its name) is a **pointer**
  - Its value is where in memory the array starts
- So an array is just a pointer!
  - `int * x;` is equivalent to `int x[];`
- What does a[x] mean?
  - An integer value the address:
    - `a + (x * sizeof(a[0]));`
  - The compiler automatically calculates the offset based on the size of the data type in the array

Arrays and pass by value

- C is pass by value
  - So what happens when we pass an array as an argument to a function?

```
int x[5] = {1, 3, 5, 7, 9};
foo(x);
```

- We pass the value of x
  - But, x is an array, and arrays are pointers
  - **So, we are just passing a pointer as the argument**

```
int foo(int * x) {
   printf("%d\n", x[0]);
}
```

Is this safe?

- **Example: sizeof-array.c**

Arrays and pointer arguments

- **`int * x;`** and **`int x[];`** are the same thing!
  - **`int foo(int * x) == int foo(int x[]);`**
- x[1] is a base/offset calculation
  - Address of array 'x' **+** offset of index '1'
- Offsets
  - C is smart about offset calculations
    - Scales offset by the size of the ***type***

Array/pointer offsets

- **`int x[10];`**
- Accessing elements (2 equivalent syntaxes)
  - `X[4];`
  - `*(x + 4);`
- We really added ***16*** to the address of x!
  - `x + (4 * sizeof(x[0]));`
- The compiler is smart enough to scale the offset values.
  - How does it do this?
  - The is the only help the compiler gives us.
  - **No bounds checking!**

Strings

- What are strings in C?
  - Already saw them defined as (`char *`)'s.
  - … so they are just arrays of characters (bytes)…
  - … with a NULL terminator ('\0') at the end
- String format
  - Valid strings must be NULL terminated
  - Assumed by the standard library string functions
    - Many functions that look like: `str*(…);`

Inputting Strings

- We've already seen this

```
char my_name[32];
scanf("%s", my_name);
```

- But what if a name is longer than 31 characters?
  - scanf will still copy the entire name into my_name.
  - This will overrun the buffer!
    - Overwrite memory that belongs to something else
  - Why will it overrun?
  - Why only when a name is more than 31 characters?

Common string operations

- `size_t strlen(char * str);`
  - Returns the length (int) of the string
- `char * strcpy(char * dst, char * src);`
  - Copies src into dst
- `int strcmp(char * s1, char * s2);`
  - Compares 2 strings, returns 0 if they match
- `char * strcat(char * s1, char * s2);`
  - Appends s2 to the end of s1

- **Programmer's tip:** Never use any of these functions
  - Why Not?

Better string functions

- Earlier functions were **VERY DANGEROUS**
  - Did not have any bounds restrictions
- Better to use explicit bounds versions
  - `size_t strnlen(char * s, size_t maxlen);`
  - `char * strncpy(char * dst, char * src, size_t n);`
  - `int strncmp(char * s1, char * s2, size_t n);`
  - `char * strncat(char * s1, char * s2, size_t n);`

- All(?) str*() functions have strn*() versions
  - But you still need to be careful
  - strn*() functions do not guarantee the output will be NULL terminated

Best option (if available)

- str/strn are portable (every system has them)
  - There is a 3rd better option
    - **But it is not as portable**
  - Enforces bounds restrictions and guarantees NULL termination
- strl*(); functions
  - `size_t strlcpy(char * dst, char * src, size_t size);`
  - `size_t strlcat(char * dst, char * src, size_t size);`
  - … etc …