C Programming (Aside)

CSC 343, Operating Systems

C Basics

- Summary:
    - pointers / arrays / structs / casting
    - Memory management
    - Function pointers / generic types
    - Strings
    - Miscellaneous

Pointers

- A pointer stores the address of a value in memory
  - For example, `int*`, `char*`, `int**`, etc.
  - Access the value by dereferencing (`*a`); can be used to read value or write value to given address
  - Dereferencing `NULL` causes a runtime error
- A pointer to type `a` references a block of `sizeof(a)` bytes
- Get the address of a value in memory with the `&` operator.
- Can alias pointers to the same address.

Call-by-Value versus Call-by-Reference

- Call-by-value: changes made to arguments passed to a function are not reflected in the calling function.
- Call-by-reference: changes made to arguments passed to a function are reflected in the calling function
- C is a call-by-value language
- To cause changes to values outside the function, use pointers.

Example

```
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 42;
    int y = 54;
    swap(&x, &y);
    printf("%d\n", x);
    printf("%d\n", y);
}
```

Pointer Arithmetic

- Can add/subtract from an address to get a new address
  - Only perform when absolutely necessary (that is, `malloc`)
  - Results depends on the pointer type
- Examples:
  - `int* a; a+i` → `a = &a + sizeof(int) * i`
  - `char* a; a+i` → `a = &a + sizeof(char) * i`
  - `int** a; a+i` → `a = &a + sizeof(int*) * i`
- Rule of thumb: cast pointer explicitly to avoid confustion
  - prefer `(char*)(a) + i` versus `a + i`
  - absolutely do this in macros

Structs

- Collection of values placed under one name in a single block of memory
- Given a struct instance, access the fields using the `.` (dot) operator
- Given a stuct pointer, access the fields using the `->` operator

Struct Example

```
struct foo_s {
    int a;
    char b;
}

struct bar_s }
    char arr[10];
    foo_s baz;
}

bar_s biz;
biz.arr[0] = 'a';
biz.baz.a = 42;
bar_s* boz = &biz;
boz->baz.b = 'b';
```

Arrays/Strings

- Arrays: fixed-size collection of elements of the same type
  - Can allocate on the stack or on the heap
  - `int a[10]; // array of 10 ints on the stack`
  - `int* a = calloc(10, sizeof(int)); // array of 10 ints on the heap`
- Strings: null-terminated character arrays
  - null-character (\0) tells us where the string ends
  - all standard C library functions on strings assume null-termination

Casting

- Can cast a variable to a different type
- Integer type casting:
  - signed *leftrightarrow* unsigned: change interpretation of the most significant bit
  - smaller signed → larger signed: sign-extend (duplicate the sign bit)
  - smaller unsigned → larger unsigned: zero-extend (duplicate 0)
- Cautions:
  - cast explicitly; C will cast operations involving different types implicitly, often leading to errors
  - never cast to a smaller type; will truncate (lose) data
  - never cast a pointer to a larger type and dereference it; this accesses memory with undefined contents

malloc, free, calloc

- Handle dynamic (heap) memory
- void* malloc (size_t size)
  - allocate block of memory of size bytes
  - does not initialize memory
- void* calloc (size_t num, size_t size)
  - allocate block of memory for array of num elements, each size bytes long
  - initializes memory to zero values
- void free(void* ptr)
  - frees a previously allocated block pointed to by ptr
  - use exactly once for each pointer you allocate
- Note: the size argument should be computed with the sizeof operator

Memory Management Rules

- `malloc` what you `free`, `free` what you `malloc`
  - client should free memory allocated by client code
  - library should free memory allocated by library code
- number of `malloc`s $=$ number of `free`s
  - number of `malloc`s $>$ number of `free`s: definitely a memory leak
  - number of `malloc`s $<$ number of `free`s: definitely a double free
- Free a `malloc`ed block exactly once
  - should not dereference a freed memory block

Stack versus Heap Allocation

- Local variables and function arguments are placed on the stack
  - deallocated after the variable leaves scope
  - do not return a pointer to a stack-allocated variable
  - do not reference the address of a variable outside its scope
- Memory blocks allocated by calls to `malloc`/`calloc` are placed on the heap
- Globals, constants are placed elsewhere
- Example:
  - `int* a = malloc(sizeof(int))`
  - `// a is a pointer on the stack to a memory block on the heap`

typedef

- Creates an alias type name for a different type
- Useful to simply the names of complex data types

```
struct list_node {
    int x;
}

typedef int pixel;
typedef struct list_node* node;
typedef int (*cmp)(int e1, int e2);

pixel x; // int type
node foo; // struct list_nod type
cmp int_cmp; // int (*cmp)(int e1, int e2);
```

Macros

- Fragment of code given a name; replace occurrence of name with contents of macro
- Uses:
  - defining constants
  - defining simple operations
- Warnings:
  - use parentheses around arguments/expressions to avoid problems after substitution
  - do not pass expressions with side effects as arguments to macros

```
#define INT_MAX 0x7FFFFFFF
#define MAX(A, B) ((A) > (B) ? (A) : (B))
```

Generic Types

- `void*` type is C's provision for generic types
  - raw pointer to some memory location (unknown type)
  - cannot dereference a `void*`
  - must cast `void*` to another type in order to dereference it
- Can cast back and forth between `void*` and other pointer types

Generic Types Example

```
// stack implementation
typedef void* elem;

stack stack_new();
void push(stack S, elem e);
elem pop(stack S);

// stack usage
int x = 42; int y = 54;
stack S = stack_new();
push(S, &x);
push(S, &y);
int a = *(int*)pop(S);
int b = *(int*)pop(S);
```

Header Files

- Includes C declarations and macro definitions to be shared across multiple files
  - only include function prototypes/macros; no implementation code
- Usage: `#include <header.h>`
  - `#include <lib>` for standard libraries (for example, `#include <string.h>`
  - `#include "file"` for your source files (for example, `#include "header.h"`
  - never include `.c` files (bad practice)

Header Guards

- Double-inclusion problem: include the same header file twice
- Solution: header guard ensures single inclusion
- Syntax Example:

```
#ifndef FILENAME_H
#define FILENAME_H


#endif
```

Odds and Ends

- Prefix versus postfix increment/decrement
  - a++: use a in the expression, then increment a
  - ++a: increment a, then use a in the expression
- Switch Statements:
  - remember break statements after every case, unless you want fall through
  - should probably use a default case
- Variable/function modifiers
  - global variables: defined outside functions, seen by all files
  - static variables/functions: seen only in the file it is declared in

`string.h`

- One of the most useful libraries
- Important usage details regarding arguments:
  - prefixes: `str` → strings, `mem` → arbitrary
  - ensure that all strings are null-terminated
  - ensure that `dest` is large enough to store `src`
  - ensure that `src` actually contains `n` bytes
  - ensure that `src`/`dest` do not overlap

`string.h` Common String/Array Functions

- Copy
  - `void* memcopy (void* dest, void* src, size_t n)`: copy `n` bytes of `src` into `dest`
  - `char* strcopy (char* dest, char* src)`: copy `src` string into `dest`, return `dest`
- Concatenation
  - `char* strcat (char* dest, char* src)`: append copy of `src` to end of `dest`, return `dest`
- Comparison
  - `int strcmp (char* str1, char* str2)`: compare `str1` to `str` by character (based on ASCII value), return comparison result

`string.h` Common String/Array Functions

- Searching
  - `char* strstr (char* str1, char* str2)`: return pointer to first occurrence of `str2` in `str1`, else `NULL`
  - `char* strtok (char* str, char* delimiters)`; tokenize `str` according to delimiter characters provided in `delimiters`, return next token per successive `strtok` call, using `str = NULL`
- Other
  - `size_t strlen (const char* str)`: returns length of the string
  - `void* memset (voide* ptr, int val, size_t n)`: set first `n` bytes of memory block addressed by `ptr` to `val`

stdlib.h: General Purpose Functions

- Dynamic memory allocation:
  - malloc, free, calloc
- String conversion:
  - int atoi (char* str): parse string into integral value (return 0 if not parsed)
- System calls:
  - void exit (int status): terminate calling process, return status to parent process
  - void abort(): aborts process abnormally
- Searching/Sorting:
  - provide array, array size, element size, comparator (function pointer)
  - bsearch: returns pointer to matching element in the array
  - qsort: sorts the array destructively
- Integer arithmetic:
  - int abs (int n): returns absolute value of n
- Types:
  - size_t: unsigned integral type

stdio.h

- Used for:
  - argument parsing
  - file handling
  - input/output

Note about Library Functions

- These functions can return error codes
  - `malloc` could fail
  - a file could not be opened
  - a string may be incorrectly parsed
- Remember to check for the error cases and handle the errors accordingly
  - may have to terminate the program
  - may be able to recover

Tools

- GCC: compiler
- GDB: stepping debugger
- Valgrind: find memory errors, detect memory leaks
  - Common errors:
    - illegal read/write
    - use of uninitialized values
    - illegal frees
    - overlapping source/destination addresses
  - `--leak-check=full` details each definitely/possibly lost memory block

GCC

- Used to compile C projects
  - list the files that will be compiled to form an executable
  - specify options via flags
- Important flags:
  - `-g`: produce debug information
  - `-Werror`: treat all warnings as errors
  - `-Wall/-Wextra`: enable all construction warnings
  - `-pedantic`: indicate all mandatory diagnostics listed in C standard
  - `-O0/-O1/-O2`: optimization levels
  - `-o <filename>`: name of output binary filename