Arrays, Pointers, File I/O

CSC209H5: Software Tools & Systems Programming

Robert (Rupert) Wu
rupert.wu@utoronto.ca

Department of Computer Science
University of Toronto

January 23, 2023

Lesson Plan

1. Memory & Arrays
2. Pointers
3. File I/O
4. Housekeeping

Acknowledgements

Part of the slides are borrowed from Karen Reid, Andi Bergen and Manos Papagelis, with assistance from Bahar Aameri.

Section 1

Memory & Arrays

Memory

- The operating system manages the real memory based on hardware.
- From our perspective we're working with virtual memory on top.
- Bytes are typically the smallest unit of memory.
  - Each unit has an address, which is an integer-like numeric that can be operated on with integers.
- The address of a variable/struct is the address of its first byte.
- Local-scope variables are typically allocated memory on the *stack*.
- Dynamic allocation to the *heap* is explicitly handled (seen later).

Expressions: Arnold's Examples

`mcs.utm.utoronto.ca/~209/23s/lectures/src/c/expressions.c`

Arrays: Declaration & Allocation

Arrays are sequence of uniformly-sized elements stored in a contiguous region of memory. They're declared to contain types (such as characters or numerics) and an array size between brackets [size].

```
float A[65];
A[0] = 6.0;
A[1] = 3.141592654;
printf("%f\n", A[1]); // bad approx of pi
printf("%f\n", A[2]); // undefined
```

A normal declaration assigns a region of memory to the array, but doesn't normally re-initialize the values.

Static Arrays

However, static arrays do initialize values as 0.

```
static long B[4];
printf("%ld\n", B[3]); // 0
```

Arrays: Initialization

Alternatively, you can directly initialize arrays with values.

- Values in the array beyond the initializer are initialized as 0.

```c
int csc209[4] = {2, 0, 9}; // csc209[3] == 0
```

- Due to type inference, the size of such declarations is optional.

```c
int csc369[] = {3, 6, 9}; // size inferred
```

Arrays: Bounds in Memory (ρ)

- C doesn't require that subscript bounds be checked; if a subscript goes out of range, the program's behavior is undefined.
- No run-time check of array bounds: behaviour exceeding bounds is undefined. If lucky, it might (appear to) work with no side effects.
    - Sometimes it'll do something random, harmless or not.
    - Worst-case, it might crash the program or OS.

```
int csc469 = {2, 2, 0, 8};
csc469[4] = 1; // will likely crash with stack smashing
```

- **Warning:** It is the programmer's responsibility to keep track of the size of an array! Take care not to violate the bounds of the array.

Arrays: Arnold's Examples

`mcs.utm.utoronto.ca/~209/23s/lectures/src/c/arraysVarLength.c`

Variable-Length Arrays

In C99 it's possible to use a non-constant expression in declare length of arrays. As of C11, it is an optional feature that implementations aren't required to support.

```
#include <stdio.h>
int main(){
    int i, n;
    printf("How many numbers do you want to read?");
    scanf("%d", &n);
    int a[n]; /* C99 only - length of array depends on n */
    printf("Enter %d numbers: ", n);
    for (i = 0; i < n; i++) scanf("%d", &a[i]);
    return 0;
}
```

The array a in this program is an example of a variable-length array (VLA).

- The length of a VLA is determined at runtime, not compile time.
- One big advantage of a VLA that belongs to a function f is that it can have a different length each time f is called.

Section 2

Pointers

Pointers: An Introduction

A *pointer* wrapper of an address; operating on a pointer means operating on the address. A pointer variable can only point to objects of particular type, and such a variable must be declared with an asterisk * preceded by the type. Pointer `ptr` points to object `obj` if the address of `obj` is stored in `ptr`.

```
int obj = 7; int *ptr; // ptr starts as a NULL pointer
ptr = &obj;
int same_obj = *ptr;
```

Pointers can point to anything, including other pointers.

Operators:

- addressing (&): `&obj` takes the address of `obj`, to store in `ptr`.
- indirection (*): `*ptr` dereferences the `ptr` to get `obj`.
    - Dereferencing a `NULL` pointer results in errors.

Pointers: Operators

They're formatted into strings with %p.

```
#include <stdio.h>
int main() {
    char ch = 'Y'; char *ch_pt = &ch;
    printf("ch_pt points to %c\n", *ch_pt);

    int i = 5;
    printf("Value and address of i: %d, %p\n", i, &i);
    int *pt = &i;
    printf("Value and address of pt: %p, %p\n", pt, &pt);
    printf("Value pointed to by pt: %d\n", *pt);

    return 0;
}
```

Source: PCRS (University of Toronto)

Pointers: Operators

Dereferenced values can be operated on, or incremented/decremented.

```c
#include <stdio.h>
int main() {
    int i = 7, j = i;
    int *pt = &i;
    *pt = 9;
    printf("Value of i: %d\n", i); // 9
    printf("Value of j: %d\n", j); // 7
    printf("Value of j | (i+1): %d\n", j ^ *pt++); // 14
    printf("pt points to %d\n", ++*pt); // 8
    return 0;
}
```

Source: PCRS (University of Toronto)

Pointers: Why Pointers?

- Easy access to and abstraction of complex structures.
- Allows reference to the same data when desired.
- Pointers consume less memory than deep copies.
- Convenient null values for initialization/error-checking.

Pointers: Function Arguments by Value

C passes arguments by value. Implicit casting is performed on numerical function arguments; beware of truncation!

```c
#include "math.h"
#include "stdio.h"
int as_long(long l) { return l; }
float as_float(float d) { return d; }

int main() {
    int nine_plus_ten = 21;
    long massive = __LONG_MAX__ - nine_plus_ten;
    printf("%ld -> %d\n", massive, as_int(massive));
    double pi = M_PI;  // approximate the approximation
    printf("%1.32f -> %1.32f\n", pi / 2, as_float(pi) / 2);
    return 0;
}
```

Full code

github.com/rhubarbwu/csc209/blob/master/lectures/lec04/arg_cast.c

Pointers: Function Arguments by Value

What does this do to `mass`? It's being passed by value.

```c
#include <stdio.h>
#define half_life 12
#define time 100

void decay(double mass) {
    mass /= 2;
}
int main() {
    double mass = 244817;
    for (int i = 1; i < time; i++)
        if (i % half_life == 1)
            decay(mass);
    printf("After %d, %lf remains.\n", time, mass);
    return 0;
}
```

Pointers: Why Pointers?

- Easy access to and abstraction of complex structures.
- Allows reference to the same data when desired.
- Pointers consume less memory than deep copies.
- Convenient null values for initialization/error-checking.

As Function Arguments

Pointers allow you to pass primitives or structures by reference, rather than value. Instead of copying and passing the entire structure, copy/pass the pointer(s) in constant time.

Pointers: Function Arguments by Reference

What about this? It's passed by reference.

```c
#include <stdio.h>
#define half_life 12
#define time 100

void approx_decay(double *mass_ptr) {
    *mass_ptr /= 1.4142857;
}
int main() {
    double mass = 244817;
    for (int i = 1; i < time; i++)
        if (i % (half_life / 2) == 1)
            approx_decay(&mass);
    printf("After %d, %lf remains.\n", time, mass);
    return 0;
}
```

Pointers: Pointers to Pointers (to Pointers …)

Because pointers can point to anything, you also have pointers to pointers.

```
int main() {
    int i = 81; int *pt = &i; int **pt_ptr = &pt;

    int *r= *pt_ptr; // intermediate dereference
    int k = *r; // complete the dereference

    int k1 = **pt_ptr; // direct double dereference

    int ***pt_ptr_ptr = &pt_ptr; // triple pointer
    int k2 = ***pt_ptr_ptr;
    return 0;
}
```

Source: PCRS (University of Toronto)

Pointers: Relationship with Arrays

The relationship between pointers and arrays in C is a close one. Understanding this relationship is critical for mastering C.

- C allows to perform addition and subtraction on pointers to array elements. This leads to an alternative way of processing arrays in which pointers take the place of array subscripts.
- Pointers can point to array elements. Here's an example:

```
int a[10], *p;
p = &a[0];
*p = 5; // stores 4 in a[0]
```

- A pointer is not an array but it can contain the address of an array. An array is not a pointer either but the compiler interprets the name of an array as the address of its 0th element.

```
int *x = &a[0];
int *y = a;
```

Multidimensional Arrays

Arrays can be multi-dimensional (N-D) to represent higher dimensional tensors.

```
const int Y = 0, R = 1, B = 2, G = 3, O = 4, W = 5;
int rubiks_face[3][3] = {
    {Y, Y, R},
    {W, G, B},
    {Y, Y, R}};
```

The name of a two-dimensional array is a pointer to a pointer – a double pointer. What's the type of the name of a three-dimensional array like `rubiks_cube`?

```
int rubiks_cube[6][3][3]; // 6 faces, int ***
```

For any two-dimensional array `A`, the expression `A[k]` is a pointer to the first element in row `k` of the array.

```
int k = 2, *p = rubiks_cube[k];
for (; p < rubiks_face[k] + 2; p++) *p = Y;
```

Multidimensional Arrays: Row-Major Order

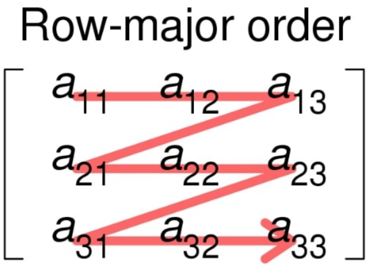
Row-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}

Figure 1: row-major order of a 3x3 matrix

Although we visualize two-dimensional arrays as tables, that's not the way they're actually stored in computer memory. C stores arrays in *row-major order*, with row 0 first, then row 1, and so forth.

```c
int *row_ptr = rubiks_face[0];
int *square_ptr = &row_ptr[2]; // pointing to rubiks_face[0][2]
square_ptr++; // pointing to rubiks_face[1][0]
```

Pointers: Arrays as Arguments/Parameters

When passed to a function, an array name is treated as a pointer. That is, what is passed to the function decays to a pointer to the first element.

```c
int find_largest(int a[], int n){
    int i, max = a[0];
    for (i = 1; i < n; i++)
        if (a[i] > max) max = a[i];
    return max;
}
int main() {
    ...
    find_largest(A, N); // A is not copied;
    ...                 // rather, a points to A[0]
    return 0;
}
```

Pointers: Arrays as Arguments/Parameters

The size of an array is not inherently stored in the array itself; the only way to know/pass on how large the array is is to pass the length of the array alongside.

- Remember `argv`? It's an array of "strings" of length `argc`.

```c
int main(int argc, char **argv) { return 0; }
```

Strings

"Strings" are actually `char`-arrays, i.e. `char *`. They are *null-terminated*: their last values are the `\0` to indicate the end of the string; more about this when we discuss strings...

Pointers: Arrays as Arguments/Parameters

1. An array used as an argument isn't protected against change.
2. Just like with all structures, latency and bandwidth of passing an array are not affected by the size of the array.
3. An array parameter can be declared as a pointer if desired.
   - Although declaring a parameter to be an array is the same as declaring it to be a pointer, the same isn't true for a variable.

   ```
   int A[10]; // allocates memory for 10 integers
   int *a;    // allocates memory for a pointer, not an array
   ```
4. A function with an array parameter can be passed an array "slice":

   ```
   find_largest(&b[5], 10);
   ```

Pointers: Arithmetic

Pointers are technically numbers, so you can add integers to them. Then, you can access other values with relative pointers.

- If `p` points to `A[i]`, other `A[j]` can be accessed by performing arithmetic on `p`.
- C supports exactly these three forms of pointer arithmetic.
  - pointer + integer; pointer - integer; or pointer - pointer
- Adding an integer `j` to a pointer `p` yields a pointer to the element `j` places after the one that `p` points to. That is, if `p` points to the array element `A[i]`, then `p+j` points to `A[i+j]`.
  - In other words, `A + i` is the same as `&A[i]` because both represent a pointer to element `i` of `A`.
  - Similarly, `*(A+i)` is equivalent to `A[i]` because both represent `i`'th element of `A`.
  - Assuming 32-bit integers, each increment on a pointer will move 4 bytes down, giving us the pointer to the next element.

Pointers: Arithmetic (Virtual Memory)

```c
#include <stdio.h>
int main () {
    int A[] = {1, 2, 4, 8, 16, 32, 64};
    for (int i=0; i<6; i++)
        printf("A[%d]: addr %x; val %d\n", i, &A[i], A[i]);
    return 0;
}
```

Note the 4-byte intervals of consecutive addresses in contiguous memory.

```
A[0]: addr f4ec9730; val 1
A[1]: addr f4ec9734; val 2
A[2]: addr f4ec9738; val 4
A[3]: addr f4ec973c; val 8
A[4]: addr f4ec9740; val 16
A[5]: addr f4ec9744; val 32
```

Pointers: Arithmetic (Virtual Memory)

```
#include <stdio.h>
int main () {
    int A[] = {1, 2, 4, 8, 16, 32, 64};
    for (int i=0; i<6; i++)
        printf("A[%d]: addr %x; val %d\n", i, &A[i], A[i]);
    return 0;
}
```

Another run… the addresses (or rather, base addresses) always change, depends on memory.

```
A[0]: addr b49d4720; val 1
A[1]: addr b49d4724; val 2
A[2]: addr b49d4728; val 4
A[3]: addr b49d472c; val 8
A[4]: addr b49d4730; val 16
A[5]: addr b49d4734; val 32
```

Pointers & Memory

Arnold's Examples: `mcs.utm.utoronto.ca/~209/23s/lectures/src/c/memory.c`

Section 3

File Input/Output (I/O)

File I/O: File Descriptors

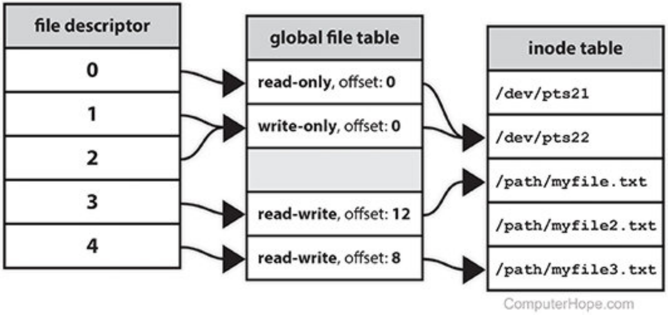

Figure 2: Example of FDs pointing to inodes.

A file descriptor (FD) is a number (non-negative integer) that uniquely identifies an open file in a computer's operating system. It describes a data resource, and how that resource may be accessed.

`www.computerhope.com/jargon/f/file-descriptor.htm`

File I/O: Streams, File Descriptors & Pointers

Streams are files to which data is written or from which data is read. They're accessed through file pointers (`FILE *` from `<stdio.h>`) that wrap around FDs. The following default streams (FDs) are provided by `<stdio.h>`.

- `stdin` (0): default input; typically from user keyboard or pipes.
- `stdout` (1): default output; usually to terminal screen or pipes.
- `stderr` (2): default error; also to terminal screen.
- Use > to *redirect* `stdout`, and 2> to redirect `stderr`
  - > overwrites the output file, >> appends to it.

File I/O: Opening a File with Modes

```
FILE *fopen(const char *filename, const char * mode);
```

A file `filename` is opened with `fopen()` in a `mode` `{r|w|a}{|+}` to perform the following operations. Returns a file pointer that wraps the FD. The pointer is `NULL` if we fail to open the file (often because the file doesn't exist or your process doesn't have permission).

action\mode	`r`	`w`	`a`	`r+`	`w+`	`a+`
read	yes	no	no	yes	yes	yes
write	no	yes	no	yes	yes	yes
append	no	no	yes	no	no	no
file exists	ok	ok	ok	ok	truncate	append
doesn't exist	fail	create	create	fail	ok	create

File I/O: Closing a File: `fclose()`

```
int fclose(FILE *stream);
```

- `stream`: a `FILE *` opened by `fopen()`/`freopen()`.
- returns: `0` if closed properly, `EOF` otherwise.

You should always close files (as soon as possible) when you're done with them.

```
if ((fp = fopen("doesnt_exist.txt", "a")) == NULL) {
    fclose(fp);
    return 1;
}
```

File I/O: Reading & Writing

Reading

1. `getchar()`: read a character from `stdin`.
2. `fgetc()`: read a single character from the file.
3. `fgets()`: read strings from files.
4. `fscanf()`: formatted input from a file.
5. `fread()`: block of raw bytes from files; useful for binary files.

Examples: `e1.c`, `e2.c`.

Writing

You can use `putchar()` to write a character to `stdout`.

```
size_t fwrite(const void *ptr, size_t size,
    size_t nmemb, FILE *stream)
```

Alternatively, use `fwrite()` to write `nmemb` elements (each `size` large) from `*ptr` to `stream`.

Section 4

Housekeeping Items

Housekeeping: Upcoming Assessments

Lab 3

- Due 10pm on Friday, January 27th

Assignment 1

- Due 11:59pm on Sunday, February 5th.
- Covers shell scripting and C basics.

Housekeeping: Don't ask to ask, just ask

When someone asks...

Any [thing] experts around?

What they really mean is...

Any [thing] experts around who are willing to commit into looking into my problem, whatever that may turn out to be, even if it's not actually related to [thing] or if someone who doesn't know anything about [thing] could actually answer my question?

In other words...

I have a question about [thing] but I'm too lazy to actually formalize it in words unless there's someone on the channel who might be able to answer it

Housekeeping: Don't ask to ask, just ask

Instead, just ask...

How do I do [problem] with [thing]?

- *I want [this kind of result]...*
- *I tried [this other thing]...*
- *[something] happened...*

`https://dontasktoask.com/`

Housekeeping: Using Piazza

How to ask questions and get better/faster answers:

- Make your question public unless it's sensitive.
- Search for duplicates before posting.
- Search on the internet first.*
- When asking a question, include as much information as you can.
  - Show us code (if it isn't sensitive) or shell output.
  - Describe what you tried and why it doesn't work.
  - If a potential answerer has to ask you for basic information about your question, they'll be inclined not to answer at all.

* but please don't commit an academic offence...