CS 2124 Jumpstart

Workshop Guide

Mark Robinson

Me

- Dr. Mark Robinson : Mark.Robinson@utsa.edu
- Practicing software engineer since 1993
- Domain: speciality insurance market
- Platforms: service-oriented applications (web and server processes)
- Agile: small teams; fast, iterative development

Join ACM

- Very active, student-led, multi-discipline (but mostly CS) organization
- **Social**: meet like-minded people, study groups, D&D and gaming
- **Workshops**: lean cool, new, and very useful tech stuff
- **Hackathons**: excellent way to develop team skills, build stuff crazy fast, and meet interesting people
  - Hackathons also look great on a resume
- For more info: https://www.acm-utsa.org/

Join ICPC

- Competitive programming club (competition is not required though)
- Strengthen your algorithms and data structure knowledge
- Improve interviewing skills for the big companies like Google
- We meet Saturdays 10a starting in ???
- Discord server (where we meet): https://discord.gg/WNbHXVX2hC

Senior Design

- 2-semester software engineering course sequence
- Normally taken your senior year
- Build software for real clients like HEB, DOD, DHA, MITRE, and UT Health
- Looks great on your resume!!
  - And can be a foot in the door with the participating companies

A: Pointers

- **Definition**: a data type that stores **the memory address** of some other thing (e.g., variable, function)
- Declare: `<point to data type> * <identifier> [initializer] ;`
- Example: int * myPointer;

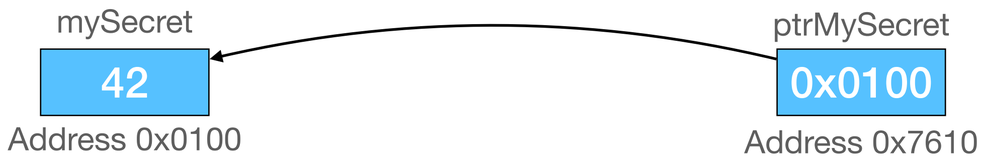
Logging a Pointer

- Handy to output pointer values to log (e.g., console) for debugging
- Use “%p” for the printf format specifier
  - E.g., `printf(“myPointer contains the address %p\n”, myPointer);`
- How is the below statement different?
  - `printf(“myPointer contains the address %p\n”, &myPointer);`
- And this?
  - `printf(“myPointer contains the address %p\n”, *myPointer);`

& Operator

- & is the “address of” unary operator
- Returns the address in memory of its given argument
- Example:

```
int mySecret = 42;
int * ptrMySecret = &mySecret;
```

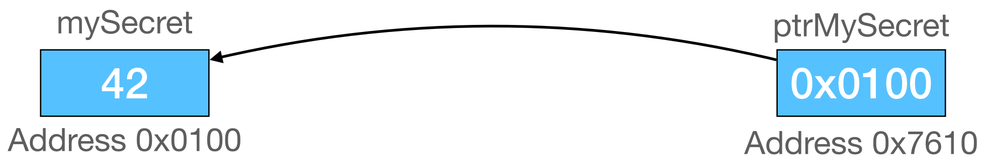


* Operator

- * is the dereferencing operator
- Returns the value of whatever the given pointer is pointing to
- Example:

```
int mySecret = 42;
int * ptrMySecret = &mySecret;

printf("pointer dereferenced is %d\n", *ptrMySecret);
// will print 42
```



Time for program A1

Declaration of a pointer
Printing pointer values as hex addresses
Address of operator

Benefits of Pointers

- Allows parameters to be READ and WRITE. Why do that?
  - YOU CANNOT WRITE A ONE FUNCTION PROGRAM!!!
  - Must break programs into small functions for good reasons and some functions must be able to change values of passed argument
- Keeps parameter size to constant value (size of a pointer)

Pointer Use Nutshell

- If you want to change the value of a variable in a function that you are passing it into…
  - Pass a POINTER to that variable into the function

```
void foo1(int a) {…

void foo2(int * a) {…

void foo3(int ** a) {…

int myVal = 42;
int myOtherVal = 100;
int * ptrMyVal = &42

foo1(myVal); // has no effect on myVal
foo2(ptrMyVal); // can change value of myVal in foo2
foo3(&ptrMyVal); // why do this?
```

Time for program A2

Passing by value in C
Passing a pointer into a function
Returning a pointer from a function

B: Arrays

- **Definition**: contiguous ordered collection of homogeneous elements
- **Element**: a single item or value that is stored in the array
- **Contiguous**: “next to”, or all elements are physically allocated together, one element after another
- **Ordered**: an element’s position is dependable (e.g., the first element in the array is the value that is placed in the first element’s allocated memory)
- **Collection**: there can be more than one element in the array
- **Homogeneous**: all of the elements are of the same data type

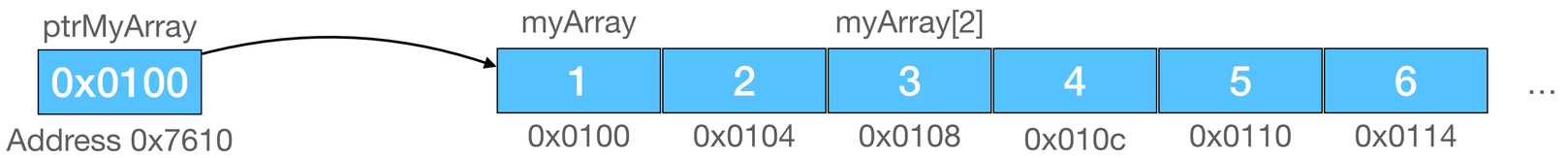
Declaring an Array

I.e., statically allocated

- Array variables CANNOT change once allocated
- AND C MUST know the size of the array when it is declared
  - E.g., int myArray[10]; // space allocated for 10 ints with values ???
- Can initialize the array at time of declaration IF values are known
  - E.g., int myArray[ ] = {1,2,3,4,5,6,7,8,9,10}; // now we know the values

Arrays and Pointers

- Arrays are similar to pointers
- Can index using subscripts (e.g., myArray[2] and ptrMyArray[2])
  - Or pointer arithmetic (e.g., *(ptrMyArray + 2) and *(myArray + 2))



Time for program B1

Declaring an array
Initializers
Compare it to pointer
Show subscripts and pointer arithmetic

Arrays and Functions

- C has no way to reliably keep track of array size
- Thus: functions MUST always be able to know the size of an array parameter
- E.g., `void foo(int [ ] myInts, int arraySize) {…`
- Notice that you do not specify the array size in the declaration of the myInts parameter
- REMEMBER: arrays are like pointers, so if you change the element values in a function, it affects the array OUTSIDE of the function

Returning an Array

- Since arrays are input and output parameters, no need to return the parameter
  - E.g.,

```
int [] foo(int [] myInts, int arraySize) {
    …
    return myInts;
}
```

- NEVER return a locally declared statically allocated array
  - The array's memory is on the call stack which gets reused
- Really doesn't make sense to return a statically allocated array parameter

Time for program B2

Pass array to a function
Return an array
Keeping track of array size

C: Dynamic Memory

- **Definition**: a chunk of general purpose memory that a program can use however it wishes (AKA the heap)
- Locally declared things are statically allocated on the program stack
  - Easy to do, fast, and automatically managed by the computer
  - But limited in size
- Heap memory
  - Much larger than program stack and can do what you want with it
  - Slower and manually managed by the programmer

When to use Dynamic Memory

- When you expect a variable's size to change during runtime
- When a variable's size is huge
- Need to use a pointer for new storage, not just pointing to an existing variable

Time for program C1

Using malloc and free, sizeof operator
Cast or no cast malloc return value
Dynamically allocating an array

Dynamically Allocated Arrays

- malloc allocates generic, contiguous bytes
- Those bytes can represent an array
- Thus, malloc allows dynamic allocation of arrays
  - malloc returns a pointer to the newly allocated array
- Since the array is dynamically allocated, the pointer can be changed to point to a new array
  - I.e., dynamic array variables are MUTABLE
  - Unlike statically allocated array variables

Arrays of Pointers

- Can have an array of pointers
  - Each element is a pointer, pointing to something dynamically allocated (e.g., an int, string, struct)
- The array itself is dynamically allocated
- BE CAREFUL WHEN FREEING!!!
  - First, free each element
  - THEN free the array

Time for program C2

Dynamically declare array of pointers
Write a function to initialize the array's pointer elements
Write function to change i^{th} element's value
Write function to print the elements

D: Strings

- An array of char elements
  - AND has a null char as its last element
- Commonly used to represent text in a program
- Data types for strings in C
  - char [ ] (statically allocated)
  - char * (usually dynamically allocated)

String Literals

- Enclosed in “” (remember ‘’ are for single chars)
- C stores them in special READ-ONLY program memory for efficiency
  - String literals are IMMUTABLE
- Initializing a char array to a string literal COPIES the literal values
- Initializing a char pointer to a string literal POINTS the pointer to the literal

Time for program D1

Declare strings as array and pointer
String initialization to literals
String addresses

String Functions

- C has lots of functions for working with strings
- #include <string.h>
- NEVER try to copy a string using pointer assignment
  - Use strcpy

Time for program D2

String functions
Passing and returning strings to functions
Iterating over a string

E: Structs

- **Definition**: contiguous collection of unordered heterogeneous members
- **member**: a single item of data in the struct (e.g., firstName, age)
- **heterogeneous**: each member can be of a different data type
- Structs are used to group together, and represent the pieces of data that form a specific concept
- The physical size of a struct is the sum of all of its immediate member sizes

Struct Declaration

```
struct person {
    int id;
    char ssn[12]; // include room for 2 hyphens and the null
char
};

typedef struct person person;

// without typedef, declare using tag
struct person bob;

// typedef is MUCH nicer
person sue;
```

. Operator to Access Members

- When you have a **struct variable**, use DOT notation to access members
  - E.g., bob.ssn
- When printing a struct member, use a format specifier appropriate for the member’s data type
  - printf(“id: %d, SSN %s\n”, bob.id, bob.ssn);

Initializing Structs

- Struct variables can use brace initializers (like arrays)
  - struct bob = {1, “111-22-3333”);
  - Remember string initialization for an array?
- What if you don’t initialize some of the members?

Time for program E1

Design a struct and a typedef
Declare and use a struct variable and show dot accessing
Print out struct members and size of the struct
Use a struct initializer

Struct Pointers

- Statically allocated struct variables are somewhat useful
- But usually you will want to dynamically allocate structs
- Use malloc and free with a struct pointer
- E.g.,

```
struct person * bob = malloc(sizeof(struct person));

// or even better and cleaner:
person * sue = malloc(sizeof(person));
```

-> Operator

- To access members of a struct via the struct's pointer
  - Use the -> operator
- You can also DEREFERENCE the struct pointer to give you the struct variable itself, and then use the . operator (but it's kind of ugly)

```
printf("bob's ssn: %s\n", bob->ssn);

printf("bob's ssn: %s\n", (*bob).ssn);
```

I frequently use parentheses to reinforce precedence AND I often forget the rules for C

Structs and Functions

- Never pass a struct by value!
  - E.g., void foo(struct person p) {…
- Pass and return struct pointers
  - E.g., struct person * foo(struct person * p) {…
- As with arrays, never return a reference to a new statically allocated struct
  - Should be dynamically allocated if you want to return a new struct

Initializing Struct Pointers

- No more automatic struct initializer when using a struct pointer :(
- Make your own struct initializer function that:
  - Dynamically allocates space for the struct
  - Assigns initial values to its members
    - You can pass the initial values in as parameters
  - And returns the pointer to the new struct
    - E.g., person * initPerson(int id, char * ssn) {…

Time for program E2

Dynamic allocation of a struct
Write a struct printing function
Write a struct initialization function

F: Complex Data Structures

- Structs, pointers, and arrays alone are not too complicated
- When you mix them together:
  - Your programs can represent realistic and useful things, making them more useful and powerful
  - BUT: the program itself becomes a lot more complicated
- Real programs are complicated. There are techniques to keep them as readable as possible (read Uncle Bob's Clean Coding book and/or take my OO class)
- Therefore: you must get used to working with complex data structures

What's Complex?

```
struct employee {
   int id;
    char * employeeName;
    char * title;
    char ssn[12];
    int salary;
    int withholding;
};

typedef struct employee employee;
```

Stop thinking of this as a struct!

This data structure represents the concept of an employee in our program.

From now on, just say "employee"

id, **ssn**, **salary**, and **withholding** are all pretty simple members

title will point to a string literal

employeeName will be dynamically allocated just enough space to store the given name's characters as a string (don't forget to include space for the null char)

Freeing a Complex Struct

- Can’t just free(bob) anymore
- **First**: free any dynamically allocated members!
  - **Then**: free the struct pointer
- Don’t free title (it’s just a pointer to a string literal. Not dynamically allocated)
- And don’t need to free the ssn array (it’s bytes are part of the employee struct)

```
free(bob->employeeName);
free(bob);
```

Time for program F1

Write a header file and C file for employee
Write an employee initialization function
Write an employee printing function
Write an employee free function

Arrays of Structs

- Dynamically allocate space for an array of employee pointers
  - Initialize all pointers to NULL
- Dynamically allocate space on demand for each employee
- Free each employee
- Free the employee array
- Note that functions that use the array of employee pointers will take a double pointer parameter, i.e., a pointer to a pointer (that’s ALL it is!)

Time for program F2

Write an employees initialization function
Write an employees printing function
Write an employees freeing function
Write an addEmployee function (much cleaner)

Advanced: Linked List of Employees

- Linked lists are collections like arrays
  - Arrays are fixed, linked lists are dynamic
    - Dynamic in this sense means can grow and shrink at runtime
- Don’t need the array of employees any more
- BUT:
  - Need a next pointer in the employee struct
  - And a pointer to the first employee in the list

The Linked List Pointer

- Need to keep track of the “head” or start of the employees list
  - I.e., a pointer to an employee pointer
- And all functions that “use” with the list will need to use this double pointer
  - Let’s just call it a list pointer

```
employee ** employees = initializeEmployees();
…
printEmployees(employees);
…
freeEmployees(employees);
```

Time for program F3

Write an employees initialization function
Write an employees printing function
Write an employees freeing function
Use as much of previous code as possible