Lecture 34: Arrays in C

CS 62
Fall 2016
Kim Bruce & Peter Mawhorter

main.c & linked_list.c

- main.c
  - Build and manipulate list
  - Must destroy at end
- linked_list.c
  - Forward declaration of linked_list_node
  - Create functions: malloc, node->data *notation*
    - *Always check result of malloc!!*
  - Destroy: free
  - Notice functions end with return self

Compiling C Programs

- Start with
  - source code files (".c" files)
  - header files (".h" files)
- C preprocessor:
  - converts .c files into expanded source code file by processing pre-processor commands:
    - #define: Insert values into source code
    - #include: Insert header code into source code
  - Result is ".i" file (but erased quickly!)

Compiling C Programs

- Compiling:
  - clang -c myFile.c
  - generates myFile.o
  - Typically include other options:
    - -Wall // warn about legal, but dubious code
    - -pedantic // warn about non-portable constructs
- Linking:
  - clang myFile.o otherFile.o -o myProg
  - Our makefiles give myProg.san.out
- Running: myProg (or myProg.san.out)

Arrays in C

- int x[50] // declare array of size 50 *(uninitialized)*
- for(int i = 0; i < 50; ++i) {x[i] = i} // initializes array
- Arrays are pointers to 0’th element.
  - I.e., x[0] = *x
  - Can play games with pointer arithmetic:
    - *(x+1) = x[1]
    - *Resist the urge!*

Copying Arrays

- Suppose you’ve initialized x as on previous slide
- Declare int y[100]
- What is result of y = x?
  - Pointer copy!!! Shared array!
- How do you copy x into y?
  - For loop!

Arrays don’t know their size!

- If pass as a parameter, pass their size as well!
- If pass array as parameter, changes to components stick!!
  - You are passing a pointer!

C Strings

- Array of char:
  char my_string[]
- Equivalent type is *char
- char is 1 byte (ASCII), not unicode
- Strings are NULL-terminated:
  - “Hello” = {‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’}
  - Thus array length one more than string length
  - Can write type as: char*

String Operations

- Use string.h for functions (or third party library)
  - See https://www.tutorialspoint.com/c_standard_library/string_h.htm
- Examples:
  - strcpy, strcmp, strcat, strstr, strchr, etc.

Enumerated Types

- enum days {Sunday, Monday, …};
  - implicitly int's
  - Can assign int values explicitly:
  - enum suit {clubs = 1, diamonds = 12, hearts = 3, …}
- Declared like Java, but Java considers them distinct from ints (but can get using ordinal())

Const type qualifier

- const keyword used to make a variable read-only — i.e., a constant
- Examples:

```
const int x = 1;
x = 2;   // error!             int * const q = &y;
int y = 2;
const int* p = &y;             *q = 4; // OK
*p = 3;  // error!             q = p; // ERROR
p = &x; // OK
```

  - Use more often for parameters

#define

- Can use #define to specify constants that are manipulated, especially to debug
  - #define DEBUG 0
  - #define ARRAY_SIZE 100
  - if (DEBUG) {
    printf("Max length of list is %d.\n", ARRAY_SIZE);
- Saves space
  - Handled by pre-processor
  - Aside from debug issues, modern practice prefers const

Memory Errors

- Two Problems:
  - Forget to delete memory allocated on heap
    - Memory leak
  - Access something already recycled (segfault)
- Neither will likely cause immediate error
  - But will cause problems down the road!
  - Source of 50% of run-time errors!
- Advice:
  - Start w/out free's,
  - add when program works

Pointer Advice

- Coding advice:
  - if pointer not initialized at declaration, initialize it with NULL
  - before dereferencing pointer, check if value is null & print reasonable error message
  - When using malloc, ensure result not NULL.

More Pointers

- Dereference operator * has low precedence.
  - Can be an issue if we're not careful.
    - E.g., suffix ++ operator happens first.
- When increment the pointer, it increases ptr by the space taken to hold an item of that type.
  - Suppose the pointer points to an int.
    To increment the pointer is to point to the next int.
    If an int is 4 bytes in size, then the next int is 4 bytes away. Thus ((int)(p))+1 ≠ (int)(p+1)

Using Pointers

- Easy to leave out parens w/ (*v).push_back(i)
- C has alternate: v->push_back(i)

Type-Unsafe Generics

- Void pointers, void*, can point to anything!
- So they can be used to implement type unsafe generic data structures and algorithms
  - This is very dangerous, as types are not checked at all.
  - Use casts to ensure right types!
- Need to use functions pointer to pass functions as arguments to other functions

Example

- **quick sort from stdlib.h**

```
void qsort( void *ptr, size_t count, size_t size,
      int (*comp)(const void *, const void *));
```

- comp is the comparison function that takes two args and returns an int

Function Pointers

- Example:

```
int addInt(int n, int m) { return n+m; }
int (* functionPtr)(int,int);   // declares function pointer
functionPtr = &addInt;
int sum = (*functionPtr)(2,3);  // sum is 5
int add2to3(int (*fPtr)(int,int)) {
     return (*fPtr)(2,3);
}
int sum2 = add2to3(functionPtr)  // sum is 5 again
```

Doesn't capture outer scope correctly