CSE 333 – SECTION 4

Midterm Review

Types of Questions

- Given spec. - write/complete code.
- Given code - Give output.
- Given code - Find bugs - Fix bugs.

Type 1 example

- Given spec. - write/complete code.

Question 1. (20 points) A little C programming. A *palindrome* is a string that reads the same forwards or backwards. For instance, "madam", "abba", and "x" are palindromes, while "ab", and "foo" are not. You are to complete a function to determine if a string is a palindrome. For this question, a string must be exactly the same forward and backward to be a palindrome, including whitespace (so the string "nurses run" is not a palindrome here). We will also consider an empty string (length 0) to be a palindrome.

Complete the definition of function `IsPalindrome` below so it returns 1 (true) if its string argument is a palindrome and returns 0 (false) if it is not. You may assume that the function argument is a properly \0-terminated C string. You may use any of the C string library functions in <string.h>. You may not copy or modify the string – only examine it.

```
#include <string.h>

// Return 1 if s is a palindrome, otherwise return 0.
// If the string has length 0, return 1 (true).
int IsPalindrome(char *s) {
```

Type 2 example

- Given code - Give output.
- Tips
  - Draw pictures!
  - Box and arrow diagrams.

Write the output of the following C++ code.

```
#include <stdlib.h>
#include <iostream>

int mystery1(int &a, int *b, int c) {
  a++;
  (*b)--;
  c = a + *b;
  return c;
}

int main(int argc, char **argv) {
  int w = 0, x = 1;
  int &y = x;
  int *z = &x;

  *z = mystery1(w, &x,
                mystery1(*z, &w, x));

  std::cout << w << " " << x << " " << y << " ";
  std::cout << *z << std::endl;
  return 0;
}
```

Type 3 example

- Given code - Find bugs - Fix bugs.

```
#include <stdlib.h>
#include <iostream>
#include <string>       // needed for std::string

using namespace std;    // to use "cout" instead of "std::cout", etc.
// A class that stores a pair of things of type T.

template <class T> class Pair {
 public:
  Pair() { }  // need default constructor for new Pair<string>[2];
  Pair(T a, T b): first_(a), second_(b) { }

  void Print() {
    cout << "(" << first_ << "," << second_ << ")" << endl;
  }

  void Set(T a, T b) { first_ = a; second_ = b; }

 private:

  T first_, second_;
};  // <-- end a class Foo { ... }; with a semicolon
```

Things to watch for

- Memory Leaks
- Invalid reads/writes
- Uninitialized variables
- Pointers and references
- Arguments and parameters
- Return types and return values
- Syntax errors

General program organization and where C fits in the ecosystem

- System layers: C language, libraries, and operating system
- General workflow needed to build a program – preprocessor, compile, link
- Preprocessor – how #include, #define, #ifndef and other basic commands rewrite the program
- Structure of C/C++ programs: header files, source files
  - Declarations vs definitions
  - Organization and use of header files, including #ifndef guards
  - Faking modularity in C – headers, implementations
  - Internal vs external linkage; use of static for internal linkage
  - Dependencies – what needs to be recompiled when something changes (dependency graph behind make and similar tools)
  - Make and makefile basics – how build dependencies are encoded in makefile rules

C language and program execution

- **Review**: standard types, operators, functions, scope, parameters, strings, etc.
- Extended integer types (int32_t, uint64_t)
- **Standard I/O** library and streams: stdin, stdout, fopen, fread, scanf, printf, etc.
- **POSIX libraries** – wrappers for system calls
  - POSIX-layer I/O: open, read, write, etc.
  - Relationship between C standard library, POSIX library functions, and system calls
- **Error handling** - error codes and errno
- **Process address space and memory map** (code, static data, heap, stack)
  - Object lifetimes: static, automatic, dynamic (heap)
  - Stack and function calls – what happens during function call, return
- **Function parameters**
  - Call by value semantics (including structs, pointers)
  - Arrays as parameters - pointers
  - Using pointers for call-by-reference semantics
  - Function pointers as parameters

More C

- Pointers, pointers, pointers - &, *, and all that
  - Typing rules and pointer arithmetic (what does p+1 mean?)
  - Relationship between pointers and arrays, a[i] and pointer arithmetic
  - String constants, arrays of characters, C string library
  - Using void* as a “generic” pointer type
  - Casting
  - Dynamic allocation (malloc, free)
  - Potential bugs – memory leaks, dangling pointers (including returning pointers to local data), etc.
  - Be able to draw and read diagrams showing storage and pointers, and be able to trace code that manipulates these things.
- Structs – how to define and use, meaning of p->x (= (*p).x), structs as local variables, parameters, and return values (value semantics) vs. heap-allocated structs, struct values vs pointers to structs
- Typedef – how to define and use
- Linked data structures in C – linked lists, hash tables, etc.

C++

- Classes and modularity, namespaces
  - Be able to read simple class definitions and add to them, implement functions, trace code, etc.
  - Know the difference between constructors, copy constructors, and assignment and when these are called
  - Know what a destructor is and when it gets called
- Other basic differences from C
  - Simpler, type-safe stream I/O (cout, cin, << and >>)
  - Type-safe memory management (new, delete, delete[])
  - References – particularly reference parameters
  - More pervasive use of const (const data and parameters, const member functions)

Questions (?)