csci 210: Data Structures

C warm-up

The history of C

- C is a general purpose language originally designed by Dennis Ritchie of Bell Labs in 1972.
- It was first used as a systems language for Unix

- Check out:
  - C on wiki: http://en.wikipedia.org/wiki/C_programming_language
  - The history of C: http://www.livinginternet.com/i/iw_unix_c.htm

Why C?

- C is a small language
  - few keywords
- C is (arguably) the most powerful language
  - includes the right control structures and data types allowing their uses to be unrestricted
- C is the native language of Unix
  - also, much of MS-Dos is written in C, many windowing packages, database programs, graphics libraries
- C is portable
  - code written on one machine can easily be moved to another
  - C provides a standard library of functions that work the same on any platform
  - C has a built-in pre-processor that helps the programmer isolate the system-dependent code
- C is terse
  - C has a powerful set of operators that allows the programmer to access the machine at bit level
  - indirection and address arithmetic can be combined to accomplish in one statement what would require many statements in another language
  - this makes it to many programmers both elegant and efficient

Why C?

- C is modular
  - C supports functions for which arguments are passed with call-by-value
  - nesting of functions not allowed
  - this supports modular programming
- C is the basis for C++ and Java
  - many constructs that are used in C are also used by C++ and Java
- C is efficient on most machines
  - compiled C code is very efficient
- But....
  - no automatic array bounds checking
  - multiple uses of operators such as * and =
  - need to deal with allocating and deallocating memory
  - pointer errors and memory leaks are hard to debug

Why C?

Still,

- C is an elegant language.
- It's simplicity make it beautiful to program in.
- It let's the programmer access the machine.
- Its imperfections are easier to live in than a perfected restrictiveness.
- A C programmer strives for functional modularity and effective minimalism.

hello world

```
#include <stdio.h>


int main() {
        printf(“Hello world!”);
        return 0;
}
```

- To compile:
  - suppose we have a file called first.c
  - gcc first.c
- If no errors, this creates the executable a.out
- An executable can be run
  - ./a.out
  - this will execute the program and print “hello world!”

Variables, assignment, expressions

```
 /*  The distance of a marathon in km */

#include    <stdio.h>

int    main(void)     {

    int miles, yards;

    float kilometers;



      miles = 26;

      yards = 385;

      kilometers = 1.609 * (miles + yards/1760.0);

        printf("\nA marathon is %f kilometers.\n\n", kilometers);

      return 0;

}
```

```
#define LIMIT   100
#define PI      3.1414

.....

printf("PI=%f\n", PI);
```

Functions

- A C program consists of one or more functions in one or more .c files
- Precisely one of the functions is called main(), where execution begins
- The other functions are called from within main() and from within each other
- Functions should be declared before they are used
    - a function declaration is called a prototype

```
#include <stdio.h>
float maximum(float x, float y);
float minimum(float x, float y);
void print_info();

int main() {
    int i,n;
    float max, min, x;

    print_info();
    printf("Input n: ");
    scanf("%d", &n);

    printf("\nInput %d real numbers: ", n);
    scanf("%f", &x);
    max = min = x;
    for (i=2; i<= n; i++) {
        scanf("%f", &x);
        max = maximum(max, x);
        min = minimum(min, x);
    }
    printf("\n%s%11.3f\n%s%11.3f\n\n",
            "maximum value", max, "minimum value: ", min);
    return 0;
}

float maximum(float x, float y) {
    if (x>y) return x;
    else return y;
}

float minimum(float x, float y) {
    if (x<y) return x;
    else return y;
}



void print_info() {

    printf("\n%s\n%s\n\n\n",

            "This program reads an integer value
    for n, and then ",

        "processes n real numbers to find max and
        min values");

}
```

Call-by-value

- In C, arguments to functions are always passed by value
  - This means that, when an expression is passed as an argument to a function, the expression is evaluated, and **this value** is passed to the function
  - the variables passed as arguments to functions do not change
    - that is, even if they do change inside the functions, they do not change outside
    - the functions changes its local copies of the arguments

```
void try_to_change_it(int x) {
    x = 777;
    printf("x=%d\n", x);
}
int main() {
  int a = 1;
  printf("a=%d\n", a);
  try_to_change_it(a);
  printf("a=%d\n", a);
  return 0;
}
```

Static arrays

- Size of the array is known at compile time
- Space is allocated on the stack

```
int a[10];
int i;
for (i=0; i< 10; i++) {
        a[i] = i;
}
```

Note: the array name a is the address of the memory chunk that holds the elements

Pointers

- A pointer is an address of an object in memory
- Operator &: the address operator
    - the value of &x is the address of variable x in memory
- Operator *: the dereference operator
    - if p is an address (a pointer), then the value of *p is the value at address p

```
int *p;
//*p is an int
int a;
//&a is a pointer to an int
```

Pointers

```
int main() {
        int a = 1;
        int *p;
        //p is a pointer to an int; but it has not been assigned a value yet, so we do not know what it points to

        printf("p=%d\n", p);

        p = &a;
        //now p holds the address of variable a

        printf("p=%d\n", p);
         *p = 10;
        printf("a=%d\n", a);
}
```

Pointers and Arrays

- An array is a pointer to the memory location that contains the data

Dynamic arrays

- Can have any size
- Space is allocated on the heap

```
int* a;

int i, n;

//ask the user to enter the desired size of the array, and read this value in n

.....

//allocate the array

a = (int*) malloc(n * sizeof(int));

if (a == null) {

      //the allocation dis not succeed --- handle this error

}

for (i=0; i< n; i++) {

       a[i] = i;

}
```

Further...

- passing pointers as arguments to functions
- 2D arrays
- struct
- files

Reading and References

- Chapter 0 and 1 in Al Kelley, Ira Pohl, “A book on C”