C Programming Tutorial – Part I

CS 537 - Introduction to Operating Systems

Java and C Similarities

- C language syntax is very similar to Java
- These structures are identical in Java and C
  - *if* statements
  - *switch/case* statements
  - *while*, *do/while* loops
  - *for* loops
  - standard operators
    - arithmetic: +, -, *, /, %, ++, --, +=, etc.
    - logical: || , && , ! , == , != , >= , <=
    - bitwise: | , & , ^ , ~

Java and C Similarities

- The following similarities also exist
  - both have functions
    - Java calls them methods
  - both have variables
    - local and global only in C
  - very similar data types in C
    - short, int, long
    - float, double
    - unsigned short, unsigned int, unsigned long

Java and C Differences

- C has no classes
- All work in C is done in functions
- Variables may exist outside of any functions
  - global variables
  - seen by all functions declared after variable declaration
- First function to execute is *main*

Simple C Program

```
#include <stdio.h> // file including function declarations for standard I/O

int main() {
    printf("Hello World!\n");   // prints a message with a carriage return
    return 0;   // return value of function - end of program
}
```

I/O in C

- There are many functions that retrieve information or place information
  - either to standard I/O or to files
- Introducing 2 standard functions
  - printf: writes to standard output
  - scanf: reads from the standard input
- Both of these functions require formatting within the string using special characters

Simple I/O Example

```
#include <stdio.h>

int main() {
    char ch;

    printf("Enter a character: ");
    scanf("%c", &ch);   // read a char from std input (pass-by-reference)
    printf("Character read is: %c\n", ch); // prints character to std output
                                           // pass-by-value
    return 0;
}
```

Common Codes for printf/scanf

- character and strings
  - %c - character
  - %s - string (must pass a pointer to array of characters)
- integers and long integers
  - %d - integer
  - %ld - long integer
  - %x - hexidecimal integer
  - %lx - hexidecimal long integer
  - %u - unsigned integer
  - %lu - unsigned long integer
- floating point or double
  - %f - floating point in m.nnnnn
  - %e - floating point in m.nnnnne±xx
- there are more but you can look those up if needed

Global & Local Variables and Constants

- Variables declared outside any scope are called global
  - they can be used by any function declared after them
- Local variables only exist within their scope
  - must be declared at the very beginning of the scope
  - stored on the stack
  - destroyed when scope ends
- Prefer not to use global variables if possible
  - too many naming conflicts
  - can be confusing to follow in large programs
- Constants are usually declared globally
  - use the *const* key word

Variable Example

```
#include <stdio.h>

const float PI = 3.14;   // declaring a constant
float radius;   // declaring a global variable - should be done locally

int main() {
    float area;   // declaring local variable

    printf("Enter radius of a circle: ");
    scanf("%f", &radius);
    area = PI * radius * radius;
    printf("Area of circle with radius %f is: %f\n", radius, area);

    return 0;
}
```

#define

- Many programmers using *#define* instead of declaring variables as constants
- The entity being defined is called a "macro"
- *#define* is a precompile directive
  - it replaces each instance of the macro in the static code with its definition at compile time

#define Example

```
#include <stdio.h>

#define   PI   3.14
#define   perror(x)   printf("ERROR: %s\n", x)

int main() {
    float radius, area;

    printf("Enter radius of a circle: ");
    scanf("%f", &radius);
    if(radius <= 0)
        perror("non-positive radius");   // expand to macro at compile time
    else {
        area = PI * radius * radius;   // change PI to 3.14 at compile time
        printf("Area of circle with radius %f is: %f\n", radius, area);
    }
    return 0;
}
```

Functions

- Any non-trivial program will have multiple functions
- C functions look like methods in Java
- Functions have return types
  - int, float, void, etc.
- Functions have unique names
- Functions have parameters passed into them
- Before a function can be used, it must be declared and/or defined
  - a function declaration alone is called a prototype
  - prototypes can be in a separate header file or included in the file their definition appears in

Function Example

```
#include <stdio.h>

#define  PI  3.14

float calcArea(float);   // prototype for function to be defined later

int main() {
    float radius, area;

    printf("Enter radius of a circle: ");
    scanf("%f", &radius);
    area = calcArea(radius);   // call function
    printf("Area of circle with radius %f is: %f\n", radius, area);
    return 0;
}

float calcArea(float radius) {
    return PI * radius * radius;
}
```

Arrays

- Like Java, C has arrays
  - they are declared slightly different
  - indexes still go from 0 to size-1
- C arrays have some major differences from Java
  - if you try to access an index outside of the array, C will probably let you
  - C arrays are kept on the stack
    - this limits the maximum size of an array
  - size of a C array must be statically declared
    - no using variables for the size

Declaring Arrays

- Legal array declarations

```
int scores[20];
#define   MAX_LINE   80
char line[MAX_LINE];   // place 80 inside [ ] at compile time
```

- Illegal array declaration

```
int x = 10;
float nums[x];   // using variable for array size
```

Initializing Arrays

- Legal initializations

```
int scores[5] = { 2, -3, 10, 0, 4 };

char name[20] = { "Jane Doe" };

int totals[5];
int i;
for(i=0; i<5; i++)
    totals[i] = 0;

char line[MAX_LINE];
scanf("%s", line);
```

- Illegal initialization

```
int scores[5];
scores = { 2, -3, 10, 0, 4 };
```

More on Arrays

- Accessing arrays
  - exactly like Java except:
    - no *.length* parameter in array
    - remember, no bounds checking
- Using arrays in functions
  - arrays can be passed as parameters to functions
  - arrays are always passed-by-reference
    - the address of the first element is passed
    - any changes made to array in the called function are seen in the calling function
      - this is the difference from pass-by-value

Array Example

```
#include <stdio.h>

#define   NUM_STUDENTS   70

void setNums(int nums[], int size) {
    int i;

    for(i=0; i<size; i++) {
        printf("Enter grade for student %d: ", i);
        scanf("%d", &nums[i]);
    }
}

int main() {
    int grades[NUM_STUDENTS];

    setNums(grades, NUM_STUDENTS);
    return 0;
}
```

Strings

- In C, strings are just an array of characters
- Because strings are so common, C provides a standard library for dealing with them
  - to use this library, include the following:
    - #include <string,h>
- This library provides means of copying strings, counting characters in string, concatenate strings, compare strings, etc.
- By convention, all strings are terminated by the null character (\0)
  - regardless of the size of the character array holding the string

Common String Mistakes

- C does not allow standard operators to be used on strings
  - *str1* < *str2* does not compare the two strings
    - it does compare the starting address of each string
  - *str1* == *str2* does not return true if the two strings are equal
    - it only returns true if the starting address of each string is the same
  - *str3* = *str1* + *str2* does not combine the two strings and store them in the third
    - it adds the starting addresses of each string

Common String Functions

- *int strlen(char str[]);*
  - counts the number of characters up to (but not counting) the null character and returns this number
- *int strcpy(char strTo[], char strFrom[]);*
  - copies the string in strFrom to the string in strTo
  - make sure strTo is at least as big as strFrom
- *int strcat(char strTo[], char strFrom);*
  - copies the string in strFrom to the end of strTo
  - again, make sure strTo is large enough to hold additional chars
- *int strcmp(char str1[], char str2[]);*
  - compares string 1 to string 2
  - return values are as follows
    - less than 0 if str1 is lexicographically less than str2
    - 0 if str1 is identical to str2
    - greater than 0 if str1 is lexicographically greater than str2

Structures

- C does not have classes
- However, C programmers can create their own data types
  - called *structures*
- Structures allow a programmer to place a group of related variables into one place

Creating a Structure

- Use the keyword *struct* to create a structure
- Example of creating a structure

```
struct foo {
    char student[30];
    int grades[7];
    float endingGrade;
};
```

- Variables can now be created of the type *struct foo*
- Example of creating a structure variable

```
int main() {
    struct foo myStruct;
    …
```

- Notice that the *struct* keyword is part of the new data type name

Using Structures

- To access any of the member variables inside the structure:
  - use the structure variable name, a period, and the member variable name
- When passed to a function, a structure is passed by value
  - just like any other data type

Example Using Structures

```
int main() {
   struct foo myStruct;

   strcpy(myStruct.student, "John Doe");
   for(i=0; i<7; i++)
     myStruct.grades[i] = 0;
   myStruct.endGrade = 0;
}
```

typedef

- It can be hassle to always type *struct foo*
- C provides a way for you to give "nicknames"
  - it is the keyword *typedef*
- Simply put *typedef* in front of the data type and then follow it with the "nickname"

Examples of typedef

- Using *typedef* with a standard data type

```
typdef unsigned long ulong_t
```

- Using *typedef* with a structure declaration

```
typdef struct foo {
  char student[30];
  int grades[7];
  float endingGrade;
} Foo;
```

- Now whenever an *unsigned long* is needed, just type *ulong_t*
- Whenever a *struct foo* is needed, just type *Foo*