

CSC 262

Lab Project 2

Due: Sep. 25, 2008

1 The C Programming Language

The Lab work in this course is going to be in the C language. C has reigned supreme as the systems programming language of choice for more than 3 decades, and with good reason. C has been described as a portable assembly language, and that's fairly true. Although technically a high-level language, C allows the programmer low-level access to machine details that make it very powerful for people who need to worry about what exactly the machine is doing.

Here's an example program, in C:

```
#include<stdio.h>

void main(int argc, char** argv){
    printf("Hello World\n");
}
```

And the equivalent Java code:

```
class Hello{
    public static void main(String[] argv){
        System.out.println("Hello World");
    }
}
```

So, this is the famous Hello World program. As you can see, there are some differences and similarities between the two languages. First there's the enigmatic `#include<stdio.h>`. Include directives fulfil the same role in C that `import` statements do in Java. `stdio.h` is a *header file*, which basically describes functionality available in a code library. `stdio` stands for standard I/O, and it's where the function `printf` is defined. Note that in java, all the code to do output to the console is part of `System` and so automatically included. The next difference you may notice is that in C THERE ARE NO CLASSES!. Java is an Object-Oriented language, and so lives and breathes classes. C is not, and has no concept of a class. In C, functions exist on their own, and don't need to be included in a class (a good way of thinking about it is that they are all basically public and static). If you want to aggregate data in C, you use something called a `struct` (which is like a class where everything is public and there are no functions). Structs will be described in detail later.

Further on down in the code, you start to see familiar things. A `main` function, for instance. In fact, the `main` function idea in Java was just appropriated from C. Again, because C doesn't have classes, keywords like `public` and `static` don't mean anything. The arguments to `main` also look different, but are basically the same thing. In both cases, they contain an array of the

command-line arguments handed to the program. However, C has NO String type! In C, strings are just considered to be an array of `chars`. C arrays are also different, in that they have no `length` attribute. The C programmer either has to know how long an array is, or check for a special element that signifies the end of the array (for strings, that is the character `'\0'`).

2 Building C code

First, in your home directory, create a directory for doing your work (traditionally this would be called `src` (short for source)). Do this with the command `mkdir src`. Now enter the directory (`cd src`). Fire up your favorite text editor, and type the code for the Hello World program and save it in a file named `hello.c`. To compile the code, simply type `cc hello.c`. (`cc` is the C compiler program, just like `javac` is the java compiler). Now, where did the compiled code go? If you type `ls` and get a directory listing, you'll see an enigmatic file named `a.out`. `a.out` is what `cc` names compiled code if the user didn't specify another name. Run `a.out`, and you'll see that it prints "Hello World". In order to tell `cc` what to name your compiled code (also called a binary), you have to tell it explicitly. Do this with `cc hello.c -o hello`, now the binary will be named `hello`.

2.1 man pages

UNIX (and therefore Linux) is a very C-centric environment. In fact, most C library functions will have their own man pages. Type `man printf` to get all the nitty-gritty details on the `printf` function. In fact, it's usually a good idea to use the man page for C functions in the same way you might use the Javadoc for Java code.

3 Variables and printf

Fortunately, Java appropriated much of C's syntax. So, C supports types like `int`, `char`, `float`, `short`, etc. However, C has no `byte` type, just use `char` instead. C also lacks a `boolean` type. In C, a boolean is any integer value. 0 means false, and anything else is true. That means that the value 1 is true, as is -147. Consider the following program:

```
#include<stdio.h>

void main(){
    int var = 14;

    printf("var = %d\n", var);
    printf("(float)var = %f\n", (float)var);
    printf("%d + 4 = %d\n", var, var + 4);
}
```

First, we include `stdio.h` because we're going to be using `printf`. Now notice that we create a local `int` variable named `var`. We initialize its value to 14, and then print it out in a variety of configurations. The first `printf` statement may be your first exposure to what's known as a format string. `printf("var = %d\n", var);` in Java would look like `System.out.println("var = " + var);`. In a format string the special sequences `%d` and `%f` stand for values that will be calculated from variables. `%d` means output the variable as a decimal integer, `%f` means output the variable as a floating point number.

Q1: Add another local `int` variable to the code (`var2`) and initialize it to 42. Add a `printf` statement that will perform what in Java would be `System.out.println("var = " + var + " var2 = " + var2 + ", var + var2 = " + (var + var2));`

Q2: Remove the initialization statement for `var` (that is, turn it into `int var;`). Recompile and run the code, what does the output look like? Why do you think this is the case? (Don't spend too much time wondering, I'll explain in class).

4 Conditionals and Loops

C has no boolean type, but it has all the conditional and looping constructs you're familiar with from Java. For example:

```
char c = 'a';

if(c == 'b'){
    //do something
}else{
    //do something else
}
```

Is a C conditional statment. `for` and `while` loops behave the same. A note of caution. Unlike in Java, you can have assignments in the condition for a loop. For instance, this code:

```
int foo = 14;

while(foo = 14){
    foo = 2;
}
```

is legal in C, but would generate an error in Java. In fact, this code would loop forever, because the result of `foo = 14` is the value 14, which in C is considered true!

5 Pointers

Java has references, in Java all objects and arrays are implicitly references, and the declared variables are references to the actual object or array data. C has pointers. A pointer is basically a variable that contains the location of data. For instance:

```
void main(){
    int num = 14;
    int* num_addr = &num;
}
```

`num_addr` is a pointer to an `int`. It is NOT an `int`, but it contains a reference to an `int`. In C, you use the `*` character to specify that a variable is a pointer to another kind of data (for instance `char*` is a pointer to a character and `float**` is a pointer to a pointer to a `float`). The line initializing `num_addr` sets it to the address of `num`. The `&` operator returns the address of a variable. For example, `&num` is the address of `num` (with type `int*`). And `&num_addr` is the address of `num_addr` (with type `int**`). `*` is used to dereference a pointer, which basically means reading out the value that it points to. For example:

```
void main(){
    int num = 14;
    int* num_addr = &num;

    printf("num = %d, *num_addr = %d\n", num, *num_addr);
}
```

This code will print out the value of `num` twice. The first time is simply because I passed it the `num` variable, the second time is because I dereferenced `num_addr`, which is pointing to `num`.

Q3: What values will the following code print out?

```
void main(){
    int num = 17, num2 = 42;
    int* ptr = &num;

    printf("%d\n", *ptr);
    ptr = &num2;
    printf("%d\n", *ptr);
    printf("%d\n", *(&num));
}
```

5.1 NULL and Seg faults

Pointers are basically addresses, and dereferencing an invalid pointer can cause a memory error that will make the OS kill your program. Usually, this error causes the OS to print out is either going to be "Bus error" or "Segmentation fault". The standard invalid address is `NULL` (which is usually the address 0). `NULL` is useful as a way to initialize unallocated pointers, or as a return value indicating an error. For the most part, `NULL` in C can be used similarly to `null` in Java. You can check pointers against it with `==` and `!=`, and you can assign it as a value to a pointer. Unfortunately, in C, not all invalid addresses are `NULL`. So, just because a pointer isn't `NULL` doesn't mean it's pointing at good data.

5.2 Arrays

Unlike in Java, arrays are not a first-class data type in C. The way that C handles arrays is by allocating enough space for all the elements and then using a pointer to the first element to reference the array. This is why C strings are just `char*`'s, the pointer just points to the first character in the string. Because C arrays are so threadbare, the responsibility for checking the length of an array falls on the programmer. There are two ways of doing this. The first is to have an additional variable (usually an `int`) that holds the length of the array. The second, which is used for strings in C, is to have a special value that indicates the end of the array. For instance, the string "Dr. Spaceman", is actually 13 characters long and is stored as "Dr. Spaceman\0" (where '\0' is the null character).

Arrays in C are dereferenced very similarly to Java. For instance `str[7]` will return the seventh character in the string `str`. However, arrays are declared slightly differently. In Java, an array of 5 integers is declared as: `int[5] array;`, in C it is declared as `int array[5];`.

Q4: The following code will put the first command line argument in a string named `str`

```
#include<stdio.h>

void main(int argc, char** argv){
    char* str;
    int len = 0;

    if(argc <= 1)
        return;

    str = argv[1];

    //put code here

    printf("length = %d\n", len);
}
```

Add code to calculate the length of `str` and store it in `len`. That is, output the number of characters occurring in `str` before the `'\0'` character.

6 scanf

Consider the following code:

```
#include<stdio.h>

void main(){
    int num;

    printf("Please enter a number:");
    scanf("%d", &num);
    printf("I think you just entered %d\n", num);
}
```

`scanf` is basically `printf` backwards. You use the format string to parse text input by the user. The `scanf` line in the preceding code takes input from the console and parses it as a decimal integer. Note that the second argument to `scanf` is `&num` NOT `num`. This is important. The `&` operator takes the address of the variable. `scanf` expects references to variables, not the variables themselves. `&` is a way to generate a variable reference from a variable.

Q5: Put the code in a loop that prompts the user with 'Do you want to continue [y/n]?' after printing out `num`. If the user answers 'y', then continue, otherwise exit the loop.

6.1 sscanf

`sscanf` is like `scanf` but it operates over a string rather than console input. For example:

```
#include<stdio.h>

void main(){
    char* str = "1234";
    int val;

    sscanf(str, "%d", &val);
}
```

This code will put the value 1234 into `val`.

Q6: In C, the command line arguments are passed in to `main` like so: `void main(int argc, char** argv)`. `argc` contains the number of arguments (i.e. the length of `argv`), and `argv` contains the actual text of the arguments. For instance `argv[1]` contains the first

argument passed into the program (`argv[0]` actually contains the name of the program itself). Write a program that will accept two integers as command line arguments and print out their sum. For example: `sum 6 4` should output 10.

7 Functions

Functions are important in C. Functions are how you organize your code and how libraries export functionality to other programs. So far, we've only been looking at the `main` function. Functions in C resemble Java functions, except they aren't contained in classes and don't need protection keywords. Lets look at an example program where the code to read an integer from the console has been moved into another function:

```
#include<stdio.h>

int readInt(){
    int val;

    scanf("%d", &val);
    return val;
}

void main(){
    printf("Enter a number: ");
    int val = readInt();
    printf("You entered: %d\n", val);
}
```

So, the `readInt` function looks similar to its Java equivalent. Functions are declared with their return type, name, argument list, and then code surrounded with braces. However, `readInt` was declared before `main` in the file. This is significant. In C, functions and data types have to be declared in the file before they are used. If you wanted to define `readInt` after `main`, you'd have to insert a forward declaration (or prototype), like so:

```
#include<stdio.h>

int readInt();

void main(){
    printf("Enter a number: ");
    int val = readInt();
    printf("You entered: %d\n", val);
}
```

```

int readInt() {
    int val;

    scanf("%d", &val);
    return val;
}

```

The third line (`int readInt();`) is the prototype. It's usually good practice to prototype all your functions before any actual definitions. If, when you're compiling, you get any error messages about a function not being defined, chances are you forgot the prototype.

Q7: Write a function named `stringLength` that takes a `char*` and returns an `int`. `stringLength` will return the number of characters in the string before `\0`. In `main`, call `stringLength` with `argv[0]`.

7.1 Function Arguments

In Java, primitives (`int`, `float`, etc) are passed by copy and Objects are passed by reference. In C, all function arguments are passed by copy. Therefore, if you want to modify data outside the function, it needs to be passed a pointer to that data.

8 Structs

C pre-dates Object Oriented design, and as such has no concept of objects or classes. However, it is still generally useful to be able to lump together a bunch of different data types into a big chunk. In C, these chunks are called `structs`. From a Java perspective, a `struct` is just a class with all public members and no methods. `Struct` members have names and are dereferenced just like members in classes. As an example, let's say you're sick of C-style strings and want to create something that looks more like a Java string:

```

#include<stdio.h>

typedef struct {
    char* str;
    int length;
} String;

...

```

This code defines a `struct` containing a pointer to an array of characters and an integer. The `struct` 'type' is `String`, however the special voodoo `typedef struct` has to occur first, otherwise the C compiler will not recognize future references to the `String` datatype. To reference parts of a `struct` you use the `.` operator (just like in Java). However, if you have a pointer to a

struct you either need to dereference it first (using `*`) or use the `->` operator. As an example, the following code grabs parts of a `String` struct:

```
int getLen(String* str){
    String my_str = *str;
    return my_str.length;
}

int getLenFromPointer(String* str){
    return str->length;
}
```

structs are usually allocated somewhere else and pointers are passed around. This is for design and efficiency reasons (because structs can be large, passing around copies can be really inefficient). Therefore, before you can really start using structs, you need to learn about memory management in C (which just happens to be the next section).

9 Memory

9.1 Stack and Heap

The C language uses the stack and heap similarly to Java, however the compiler assumes that you're conscious of stack discipline and so won't warn you about passing values around. In C, any variable declared inside a function will be allocated on the stack. This means that it's only legal to use it while the function is 'live'. Variables declared outside of functions are global, and are accessible to all functions in the program. As in Java, the heap is where you want to allocate data that you're going to be passing around between functions. However, you don't use `new` to allocate new objects. And C has no garbage collection, so the programmer (read: you) needs to determine when an object is dead and explicitly de-allocate it!

9.2 Malloc

In C, new memory is allocated with the function `malloc`. All the memory allocation functions are contained in the `stdlib.h` header file (so remember to include that if you're going to be allocating memory). `malloc` takes an `int`, allocates that many bytes and returns a pointer to the first byte. This is more intense than Java allocation, where the compiler automatically figures out how large objects are and does that calculation for you. In C, the special function `sizeof` will tell you how large a datatype is. For instance, if I wanted to allocate a new `String` I would type:

```
String* a_str = malloc(sizeof(String));
```

This will allocate space for a new `String`. Remember that the `str` field is NOT initialized to anything, so it is pointing at garbage and needs to be allocated before `a_str` should be used. Consider the following code to allocate and populate a new `String`:

```

#include<stdlib.h>

...

String* makeString(int len, char* chars){
    String* a_str = malloc(sizeof(String));
    a_str->length = len;

    //allocate enough characters for the string
    a_str->str = malloc(sizeof(char) * len);

    int i;
    for(i = 0 ; i < len ; i++){
        a_str->str[i] = chars[i];
    }

    return a_str;
}

```

This code had to first allocate the `String` and then, when the number of characters became known, the `str` field is allocated. Note the line to do that (namely: `a_str->str = malloc(sizeof(char) * len);`), we had to get the size of an individual `char` and then multiply that by the number of characters. This is pretty standard for allocating arrays.

Q8: Using the definition of `String` and the `makeString` function defined above, write a function named `String** argvToStrings(int argc, char** argv, int* len)` that will return an array of `String*` created from the command-line arguments passed to `main`, and will write out the number of strings into the integer pointed at by `len`. *HINT: you will have to write a function to compute the length of a normal `char*` string. HINT: remember not to forget the `'\0'` at the end, otherwise the string can't be printed by standard functions like `printf`.*

9.3 Free

Java has garbage collection. C does not. A garbage collector runs in the background and automatically de-allocates dead objects. For a variety of historical and technical reasons, garbage collection is not standard in C. Therefore, the programmer has to decide when objects are no longer live in the code and then de-allocate them explicitly. The function to do this is named `free`. `free` simply takes a pointer to the memory that you need to free up. So, a function that prints out a string and then de-allocates it would look like:

```

#include<stdio.h>
#include<stdlib.h>

```

...

```
void stringPrintln(String* dead_str){
    printf("%s\n", dead_str->str);
    free(dead_str->str);
    free(dead_str);
}
```

...

Note that first we had to deallocate the `char*` inside the `String` struct before we deallocate the `String`.

Q9: Consider the following code:

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

01: int* argsToInt(int argc, char** argv){
02:     int* intz = malloc(sizeof(int) * (argc - 1));
03:
04:     int i;
05:     int idx = 0;
06:     for(i = 1 ; i < argc ; i++){
07:         intz[idx++] = strtol(argv[i], NULL, 10);
08:     }
09:
10:     return intz;
11:}

12: int genSum(int argc, char** argv){
13:     int* vals = argsToInt(argc, argv);
14:     int len = argc - 1;
15:     int idx;
16:     int sum = 0;
17:
18:     for(idx = 0 ; idx < len ; idx++){
19:         sum = sum + vals[idx];
20:     }
21:
22:     return sum;
23: }
```

```
24: void main(int argc, char** argv){
25:     printf("I see you typed in %d numbers\n", (argc - 1));
26:     int sum = genSum(argc, argv);
27:     printf("The sum is = %d\n", sum);
28: }
```

An array of `ints` is allocated on line 02, when is the earliest in the code that it can be deallocated? Give the line number where you would insert the `free` call.