CS032 2008 Lab 09
Intro to C

1 Intro to C

C is one of the most widely used programming languages. It has become such a popular choice due to its efficiency and the power it gives the programmer over the machine. Almost any electronic device that uses a microprocessor can be programmed in C ? whether its a typical computer, a specialized gaming console, or a onboard real-time system controlling a military jet, C gives you the power to tell a machine what to do, and how to do it.

C is also the pre-cursor to C++. C++ is a superset of C more suited for Object Oriented Programming. While some people have learned C++ before learning C, most people find that learning C first and gaining a solid understanding of how it works makes learning C++ little more than learning its syntactical differences, and the new libraries it offers. This lab however will focus solely on C.

2 Hello, World

It's the example you've all been waiting for ? here's "Hello, World" written in pure C.

```
#include <stdio.h>

int main()
{
printf("Hello, World\n");

return 0;
}
```

compare to Java:

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

The first line `include <stdio.h>` tells the compiler's preprocessor (1) to include the header file named "stdio.h" (standard input output) into the code. Why does it do this? So that the compiler knows what to do with the function `printf()`. In C (as in most languages) you need to declare any functions you use before using them. Unlike in Java, where you get java.lang for free essentially, in C the only function that is accepted without any includes is the `main()` function ? everything else must be declared somewhere.

Aside from the include, everything else should look fairly familiar. We have `int main()` which is our C equivalent of "public static void main()" in Java, it prints out our message with a new-line character appended and then returns 0. Why return 0? Returning 0 is a bit of a convention in systems programming that indicates "exited normally". When a function or a program exits and returns a non-zero, that usually indicates some kind of error code that the caller should be aware of.

3 Variables

Variables in C are declared similarly to variables in Java. For example, to declare an int you can say:

```
int a;
```

And to declare three ints you could say:

```
int a, b, c;
```

Initializing them is also similar:

```
a = 4;
b = 6;
c = 32;
```

In fact, declaring variables in C is identical to declaring primitive data types in Java ? as long as you're declaring variables on the stack (we'll explain what this means later).

4 Arrays

Arrays (again, when declared on the stack) are similar to arrays of primitive data types in Java. To declare an array of 100 ints in C nothing more than:

```
int array[100];
```

In C, when declaring this or any other variable, there is no guarantee as to what its initial contents are. Its up to you to initialize each index of your array, because it will most likely have garbage left over from whatever happened to last be in that spot in memory previously.

If you remember anything about arrays in C, remember this ? there is no bounds checking. You will NOT get an array index out of bounds error, even if you do go out of bounds. Your program will either seg fault, or even worse, keep going, giving you whatever data it happens to find in the memory addresses beyond your array. You may not even tell it's doing anything wrong if the values happen to be values that make sense in your program. Don't let this happen to you. Bound check your arrays.

5 Branching and Looping

Assuming you know Java, you already know how to perform if statements and for loops in C, because they are virtually identical. In the case of if statements and while loops, they are literally the same. for loops are also essentially the same, except that you cannot declare your iterating variable inside of the for statement.

For example, in Java you can say

```
for(int i = 0; i < 50; i++)
{
//Do something
}
```

But this will throw a compiler error in C. Instead, you need to declare your "int i" outside of the loop like this:

```
int i;

for(i = 0; i < 50; i++)
```

CHECKPOINT 1: Write a program that says "Hello, World!" 50 times in a for loop. Make sure to include the newline character at the end of each sentence. You can just put all the code into the `main()` function.

6 printf() / scanf()

`printf()` and `scanf()` are the two basic ways of printing output and getting input to and from the command line. You've already seen `printf()` a little bit in the Hello, World example, but it has a few more tricks up its sleeve. `printf()` (as well as `scanf()`) make use of what are known as "formatting characters". For example:

```
int age = 900; int kids = 12;
```

```
printf("Yoda is %d years old and has %d kids
n", age, kids);
```

All those funny % symbols are part of the formatting characters. %d lets printf() know that an integer number should go there, and it looks for which number to use immediately after the string. Since there are two formatting characters, there are two variables added at the end. This is a feature special to C, because functions like these can have an infinite amount of arguments.

`scanf()` functions similarly, except that it waits for the user to type something into the command line in the format specified by the formatting characters, and then reads them into those variables.

For example:

```
int a;
scanf("%d", &a);
printf("%d\n", a);
```

This program declares an int called "a", reads in an integer from the command line into the address of a (more on that later), and then prints it back out.

CHECKPOINT 2: Write a program that prompts the user for their age, and then prompts the user for the number of siblings they have, and then prints it back out.

7 Stacks, Heaps, and a little bit of Pointers

a. Scope You might have noticed that C has no concept of public, private, protected, and a lot of them other funny keywords you see in languages like Java (and C++). That's largely due to the fact that there are no classes in C. There is scope however. Scope can be thought of as the lifetime of a variable. You may not have thought about this much when programming Java (it being "magic" (2) and all), but in C variables have a very explicit lifetime and you need to be aware of your variables lifetime lest they "die" (go out of scope) before you're done using them.

Any time you declare a variable on the stack (that is, any time you declare a variable simply by stating it it like `int age = 4`) you're declaring it on the stack. You can do this in Java with the primitives like ints, doubles, and floats. Declaring on the stack means that this variable will only last within its current stack frame. That could be a for loop or a function, or generally (though not always!) whenever the curly braces end.

In Java, when you use "new", you're no longer declaring on the stack ? that's the heap now. C has a way of declaring things on the heap too ? malloc().

b. Heaps & Pointers `malloc()` ? C's memory allocating function ? is how you get memory from the heap. You tell `malloc()` how many bytes you want, it returns to you a pointer (i.e. a memory address) that contains those n bytes that you requested. If it returns a null pointer (i.e. a memory address of "0" or "0x0") that means it couldn't allocate any more memory and something bad will probably happen soon.

For example, if I wanted to declare memory for 100 integers that would exist until I explicitly got rid of it (regardless of scope), I'd simply say:

```
int* myints = (int*)malloc(sizeof(int) * 100);
```

There's a lot of stuff going on here so let's break it down. First off, I declared myints as int* ("int-star"). This means that this variable (myints) is not an int, its a memory address that is referring to one or multiple ints (in this case, 100). Memory addresses on most CS machines (and most of your own, most likely) are actually 32-bit longs, which are actually quite bigger than ints. To initialize this pointer we've declared, we call `malloc()` and we pass it the amount of bytes we want. Unless you're really good at remembering the sizes of every data type, and want to change your code for every architecture it runs on, its best to just use `sizeof()` and have it tell you exactly how many bytes you need. In this case we said that we need the amount of bytes that is the size of an int, but times 100 because we need space for 100 of them. Then, we cast the pointer `malloc()` returns to an int* since `malloc()` is a generic memory allocator and returns generic (void*) pointers.

CHECKPOINT 3: Use `malloc()` to get space for 100 doubles on the heap. Print this memory address out to the screen using `printf()` and the special formatting character that prints out pointers/memory addresses in there hex representation. Look this up if you do not already know it.

8 Arrays on the Heap

The cool thing about pointers is that they can work almost exactly like an array. In fact, you may not have realized it, but in Checkpoint 3, you made an array ? just on the heap instead of the stack. Let's take our ints example again.

```
int* myints = (int*)malloc(sizeof(int) * 100);
```

Since we know myints is a pointer to space for 100 ints, that means that we can references up to 100 ints before we go out of bounds of the memory. That means we can do this:

```
int i;

for(i = 0; i < 100; i++)
{
myints[i] = 0;
}
```

This would initialize each of the ints to 0. But wait, did you notice something weird? You should have. myints was an `int*` not an `int[]`, yet we indexed it anyway. That works because in C, anytime you try to "index" into a pointer, it automatically dereferences it and takes you to the value that would be at the nth data type. Therefore in myints is a pointer to 100 ints, myints[45] will give me the 46th int. Anytime you change a pointer from its address to the value it actually represents, this is known as dereferencing. This can be done manually by placing an asterisk (*) before the identifier, or implicitly by referencing it like an array. You can go the other way around (data value to address of data value) by placing a "&" before the identifier (as we did in `scanf()`).

CHECKPOINT 4: Change the code you wrote for checkpoint 3 to also initialize each of the 100 doubles with a random double. You will have to look up a function that does this for you, and include the necessary header file. After you've initialized them, write a loop that prints each of their values AND their memory address on a new line. Your output should look like this:

```
Double 1: 0.74, Addr: 0xbffff95c
```

9 Checkpoints

1. 25 points

   for loop TA Initials: __________

2. 25 points

   I/O TA Initials: __________

3. 25 points

   malloc TA Intiials: __________

4. 25 points

   initialize double TA Initials: __________

FOOTNOTES 1. When you compile C or C++ code, a preprocessor first passes over the source code before the actual compiling begins. This preprocessor checks for preprocessor directives (i.e. instructions for it to do, such as including other source files or replacing certain constants with actual numbers) before the compiling begins.

2. And by magic we mean ”having a garbage collector”