C Basics

This document is designed as a quick reference guide to C programming. For more in-depth coverage, please consult the recommended textbook.

Hello, World

The first program students write in any new language is the "Hello, World" program. Here's how it looks in C:

```
#include <stdio.h>

int main() {
     printf("Hello, World!\n");
     return 0;
}
```

To write this program, edit the file `hello.c` in Pico (see the Unix guide for more information). Exit the file, and type:

```
gcc hello.c
```

to compile your program. Correct any errors that are listed. When your program compiles correctly, the executable `a.out` will be generated. To run your program:

```
./a.out
```

It should print "`Hello, World!`" to the console.

Variables

Variables in C are declared exactly like variables in Java. Just say:

```
type name;
```

where `type` is the type of the variable, and `name` is its name. The most common types in C are:

```
int
double
float
char
```

Notice that C does not have a boolean type or a string type.

Some examples:

```
int num;
char c;
double val;
```

Initializing Variables
To initialize:

```
name = value;
```

Where `name` is the name of the variable, and `value` is the value you want to give it. You can also do:

```
type name = value;
```

to declare and initialize all on one line.

Some examples:

```
int num;
char c = 'A';
double val;

num = 2;
val = 4.7;
```

Variables in C are not assigned an initial value. However, they will hold whatever garbage value was left in the memory spot reserved for the variable. This is usually a really big or really negative integer. In java, if you try to use a variable that has not been initialized, you will get a compiler error. The C compiler will not complain – it will just use the garbage value left in the memory spot. For example:

```
int num;          //holds some garbage value, say 13145687
num = num+1;      //compiler error in Java – num not initialized
                  //in C, num now has the value 13145688
```

You will find that C is far more lenient in compilation than Java. Remember that just because your program compiles doesn't mean that it's right!

Where to Declare
There are probably a hundred different versions of C compilers. Consequently, a program may compile with one compiler (say, on cislinux) but not compile on another (such as when using Visual Studio .NET).

Certain compilers require that variables only be declared at the beginning of a block (a block begins when a brace { is opened). Because some compilers have this requirement, please try to uphold this custom in your programs. For example, the following is fine:

```
int main() {
     double a;
     char b = 'T';   //all declarations are together
                     // at the beginning of the block
     a = 7.2;
}
```

But this is not:

```
int main() {
     double a;

     a = 7.2;        //code begins – this is not a declaration
     char b = 'T';   //b is declared AFTER the code begins
}
```

Operations

Mathematical operations in C work exactly like mathematical operations in Java. You can use +, -, *, /, and %. You can also use things like ++ and +=.

Casting in C is also the same as casting in Java:

```
int num = 7;
double d = (double) num;        //casts num to a double, d is now 7.0
```

Simulating Booleans

As I mentioned, there is no boolean type in C. Instead, ints are used to simulate booleans. In this simulation, 0 means false, and **anything else means true**. For example:

```
int flag = 0;   //flag is false
flag = 6;       //flag is true
```

Printing

As you've seen, the `printf` function is used to display output in C. For example, to display a string of text:

```
printf("Hello\n");
```

Note that you always need to specify the newline character. There is no `println` equivalent in C.

Printing Variables

Printing variables works a bit differently. First, you specify the kind of variable that's going to be printed (called a **control string**). Then, outside the string, you give the corresponding variable name.

Here are the different control strings:

Type	Control String
`int`	`%d`
`double`	`%lf`
`float`	`%f`
`char`	`%c`
`char*` (string)	`%s` (see String section)

It's best to see an example to figure out how printing works. Here's how to print the value of an integer to the screen:

```
int num = 4;
printf("The value of num is %d\n", num);
```

Notice that where we want to print a variable, we put the control string (`%d` for `int`). After we've listed the entire string, we put the corresponding variable names as the next arguments to `printf`. The above example will print "`The value of num is 4`" to the screen.

We can also print several variables at once:

```
char letter = 'A';
int val = (int) letter;
printf("The ASCII value of %c is %d\n", letter, val);
```

This prints "`The ASCII value of A is 65`" to the screen. Notice that the `%c` corresponds to the letter argument, and the `%d` corresponds to the `val` argument.

Formatting

The `printf` function also allows you some control over formatting your output. For example, if you want a value to take up exactly 6 spaces (padded with space characters on the left, if necessary), put a 6 between the % and the control string character. For example:

```
int num = 4;
printf("The value of num is %6d\n", num);
```

This will print "`The value of num is      4`" to the screen (note the padding on the left of the 4).

You can also only display a certain number of digits for decimal numbers. For example, put a .2 in between the % and the control string character to only display two decimal places. For example:

```
double val = 3.14159;
printf("Pi is %.2lf\n", val);
```

This will display "`Pi is 3.14`". You can specify both the width of the output (for example, six spaces) and the number of decimals to display by doing something like this:

```
double val = 3.14159;
printf("Pi is %6.2lf\n", val);
```

User Input

User input in C is, in short, a pain. There are two major input functions – `getchar()`, which reads a single character, and `scanf(...)`, which reads formatted input. `getchar()` can be very tedious to use because if you want to read in a number, you must read in one character at a time and then convert to an int. `scanf(...)` allows more options, but it is notoriously buggy and has serious security problems. Thankfully, input gets better in C++, but we're stuck with these two functions for now.

To use either `getchar()` or `scanf(...)`, you must include the `stdio.h` library.

getchar()

The `getchar()` function takes no arguments and returns the very next character in the standard input stream. **If there are no more characters in the input stream, it returns the constant `EOF`**. Here's an example that reads a student's letter grade and then prints it back to the console.

```
char grade;
printf("Enter your grade: ");
grade = getchar();
printf("Your grade is %c\n", grade);
```

scanf(...)

The `scanf(...)` function allows us to read formatted input, like ints and doubles. The first argument to `scanf` is the **format string**, which specifies the kind of data you expect to read. To specify the data types you expect, use the same control string characters you used for

`printf` -- `%d`, `%f`, `%lf`, `%c`, and `%s`. If you want to read an int, the first argument to `scanf` should be "`%d`". If you want to read two ints, put "`%d %d`".

The next arguments to `scanf` are the corresponding variables that you want to store the input in. We won't go into details now, but `scanf` needs the memory address of these variables so it can modify their value. To get the address of a variable, put a `&` in front of the variable name.

Here's a simple example that prompts the user for an integer, and then reads in the value:

```
int num;
printf("Enter a number: ");
scanf("%d", &num);   //%d: we're reading in an integer
                     //&num: we're storing that integer in the num variable
```

Here's an example that reads in an integer and a double:

```
int num1;
double num2;
printf("Enter an int and a double: ");
scanf("%d %lf", &num1, &num2);
```

The first number typed will get stored in `num1`, and the second number will get stored in `num2`. Our format string specified that these numbers should be separated by a space, but they can be separated by any amount of whitespace (multiple spaces, tabs, or newlines).

Suppose that the user is entering a fraction, like 9/5. Here's how we could read in that information:

```
int numerator, denominator;
printf("Enter a fraction (like 9/5): ");
scanf("%d/%d", &numerator, &denominator);
```

By putting the "`/`" in the format string, we specify that we expect the input to have a `/` there, but we don't wish to store it in a variable.

`scanf` returns the number of variables that were correctly read in. If an error occurred during input, the constant `EOF` is returned.

Here is an (incomplete) list of subtleties when using `scanf`:

- In most cases, whitespace is skipped by `scanf`. However, if you type a space (or tab or newline) where `scanf` expects a character, that whitespace will get read into your `char` variable

- If you use `scanf` to read a single character, then the user will type an input character and then hit return. `scanf` will read the input character, but the newline will remain

in the input buffer. This can cause problems if you call `scanf` a second time – the newline character will then be read, and not any new input. To fix this problem, add a call to `getchar()` after reading a `char` to read the extra newline character.

- If `scanf` reads input that it does not expect (for example, if it sees a character but is supposed to be reading an int), it will not discard the bad input. The bad input will still be in the input buffer if you call `scanf` again. To fix this, call `getchar` until you reach `EOF`. This will clear the input buffer.

Selection Structures

C has if-statements and switch statements that work just like Java's. Here is a sample if-statement:

```
int age;
//initialize age

//print either Child, Teenager, or Adult, depending on age
if (num < 11) {
     printf("Child\n");
}
else if (num < 18) {
     printf("Teenager\n");
}
else printf("Adult\n)");
```

Here is a sample switch statement. The expression in the switch clause must evaluate to either a character or an integer, just like in Java:

```
char grade;
printf("Enter your grade: ");
grade = getchar();
getchar();          //read and discard newline character
switch (grade) {
     case 'A':
          printf("Excellent\n");
          break;
     case 'B':
          printf("Good\n");
          break;
     case 'C':
          printf("Average\n");
          break;
     case 'D':
          printf("Poor\n");
```

```
            break;
        case 'F':
            printf("Failing\n");
            break;
        default:
            printf("Invalid grade\n");
    }
```

Loops

There are three kinds of loops in C – while, do-while, and for. Their syntax is exactly the same as loops in Java.

While Loop

The code in a while loop executes repeatedly until a specified condition becomes false. If the condition is false before the first execution of the loop, then the entire loop will be skipped. This example will read and print every character typed by the user (up until they press enter):

```
char c = ' ';
printf("Type some text: ");
while (c != EOF) {
    c = getchar();
    printf("%c\n", c);
}
```

Do-While Loop

Like a while loop, the code in a do-while loop executes repeatedly until a specified condition becomes false. However, the condition in a do-while loop is not checked until after the first iteration of a loop. So, a do-while loop always executes at least once. Here's the same example using a do-while loop:

```
char c;
printf("Type some text: ");
do {
c = getchar();
    printf("%c\n", c);

} while (c != EOF);
```

Notice that we don't have to give `c` a dummy initial value, as we did in the while loop.

For-Loop

The syntax of a for-loop is just like it was in Java:

```
for (initialization; condition; update) {
     //code
}
```

The only caveat is that the loop variable must not be declared in the initialization section (like "`int i = 0`"). This is because variables in C should only be declared at the beginning of a block (an opening `{`). If you forget, your code may compile for you, but it won't necessarily work anywhere else.

Here's an example that computes the factorial of a number entered by the user:

```
int i, num;
int factorial = 1;
printf("Enter a positive integer: ");
scanf("%d", &num);

for (i = 1; i <= num; i++) {
     factorial *= num;
}

printf("%d! = %d\n", num, factorial);
```

Break

The "break" statement immediately stops execution of a loop. For example, this code allows us to get and print 10 numbers, unless the user types a 0:

```
int i, num;
for (i = 0; i < 10; i++) {
     printf("Enter a number: ");
     scanf("%d", &num);
     if (num == 0) break;
     printf("You entered %d\n", num);
}
```

Continue

The "continue" statement skips the remaining code inside the loop, and continues with the next iteration. For example, this code allows us to add together 10 numbers inputted by the user, except any numbers that are negative:

```
int i, num;
int sum = 0;
for (i = 0; i < 10; i++) {
     printf("Enter a number: ");
     scanf("%d", &num);
     if (num < 0) continue;    //Won't add num to sum
```

```
        sum+=num;
    }
    printf("The sum of the positive numbers is %d\n", sum);
```

Functions

Functions in C are very similar to methods in Java, except functions are not associated with any class. (They are like static methods in Java.) They take a number of arguments, perform on operation on those arguments, and may or may not return a value.

Function Prototypes

Some C compilers will complain if they see a call to a function before they've seen the function itself. To avoid this problem, it's best to include a prototype for a function at the top of the file, and then to implement it someplace else in the file. A **prototype** lists the name, return type, and arguments for a function – but it does not implement the function. Here's an example:

```
int max(int num1, int num2);
```

We list the return type (`int`), the function name (`max`), and the arguments (`int num1` and `int num2`). This is a prototype, so we don't implement the function here. Instead, we end it with a semi-colon. (Note that this already looks very similar to Java – the only difference is that C functions are not associated with classes, so they don't need a visibility modifier like `private` or `public`.)

Return types can be any valid type (like `char`, `int`, `double`, etc.). If the function does not return a value, its return type should be `void`.

In a prototype, you don't have to list the names of the arguments – you just need to list the types. For example, this would also be a valid prototype for the `max` function:

```
int max(int, int);
```

If you are writing a prototype for a function that takes no arguments, you can leave the argument list blank, like this;

```
int getNum();
```

However, a better way would be to put "`void`" in the argument list, like this:

```
int getNum(void);
```

When the compiler sees a blank argument list in the prototype, it will allow ANY argument list in the function implementation. However, if it sees `void` in the prototype's argument list, then the implementation must also have a `void` argument list.

Function Implementations

A function implementation starts off the same as a prototype. Instead of ending with a semi-colon, it includes the code for the function in brackets `{ }`. Here's the implementation of the `max` function:

```
int max(int num1, int num2) {
     if (num1 >= num2) return num1;
     else return num2;
}
```

Like in Java, if a C function has a non-`void` return type, it must include a return statement that returns a value of the designated type.

Calling Functions

For now, all our programs are in one file, so calling functions is pretty easy. You just put the function name and include appropriate arguments, as if you were calling function from within the same class in Java. Here's a full e
xample of a C program that uses the `max` function:

```
#include <stdio.h>

int max(int, int);   //max function prototype

int main() {
     int val1, val2, big;

     printf("Enter two ints, separated by spaces: ");
     scanf("%d %d", &val1, &val2);

     big = max(val1, val2);
     printf("The max is %d\n", big);

     return 0;       //This suggests the program is ending normally
}

//max function implementation
int max(int num1, int num2) {
     if (num1 >= num2) return num1;
     else return num2;
}
```

Global Variables

All the variables we've seen so far have been **local variables** – variables that are defined within a function. These variables are only visible within that function. Consider this function:

```
int count(void) {
     int sum = 0;
     sum++;
     return sum;
}
```

Each time we call `count`, the `sum` variable is set back to 0, and the return value is 1. `sum` does not retain its value across function calls. If we did want this function to keep track of how many times it had been called, we could store `sum` as a global variable. **Global variables** are declared outside any function, and are visible to any function in the same file:

```
int sum = 0;
int count(void) {
     sum++;
     return sum;
}
```

Now, `sum` does not get set back to 0 each time the function is called.

Global variables should be declared at the top of the file, near the function prototypes (but before any function implementation). If a global variable is declared in the middle of a file, some compilers will not allow you to refer to the variable in any function that comes before its declaration.