# C Primer

*Fall 2017*

## 1 Introduction

This document will cover everything that you need to know about C for the first CS33 lab.

### 1.1 C vs. Java

Unlike Java, an object-oriented programming language, C is a procedural language. This means that C has no classes or objects. As a result, the organization of your C programs will be different than that of your Java programs. Instead of designing your programs around the different types of objects involved as in Java, you will design your programs around the tasks you need to complete.

This document will attempt to highlight the major similarities and differences between C and Java.

# 2 Functions

Java has methods, C has functions. The purpose of each C function is to perform a task when called. However, while Java methods belong to a particular object or class, C functions do not, standing on their own. Java's static methods are a reasonable, if imperfect, comparison to functions, as static methods do not belong to a particular object.

## 2.1 The `printf()` Function

Much as every Java program must contain a method with the header.

```
public static void main(String[] args)
```

each C program must have a function with the header

```
int main(int argc, char **argv)
```

This function serves as the entry point into your program; it is the first code called upon execution.

Unlike its Java counterpart, the C **`main()`** function is of type **`int`** rather than **`void`**. This means that it should return an integer value. This value is generally used to let the system know if the program ran successfully, and if not, what error occurred. A return value of 0 indicates the program ran successfully.

Additionally, C's **`main()`** function takes in two arguments instead of one. The notation **`char **`** will be explained below, but you can think of the second argument of the C function as corresponding roughly to the `String` array parameter of the Java **`main()`** method. **`argv[0]`** is the name of the command used to execute the program, while **`argv[1]`** is the first command-line argument, **`argv[2]`** is the second command-line argument, and so on. The int argument in the C function is the length of **`argv`**. So if the user runs the program with

```
./myCProgram test.txt
```

then **`argc`** will be 2, **`argv[0]`** will be **`"./myCProgram"`** and **`argv[1]`** will be **`"test.txt"`**.

## 2.2 Calling Functions

Calling a function in C is similar to calling a method in Java. Since there are no classes or objects, only the name of the function and the values of its parameters are needed. To call a function **`foo()`** with the parameter 0, you would use

```
foo(0);
```

One major difference between C and Java is that in C, you may call only those functions that have been declared above the function call in your code. This restriction may seem arbitrary, but it makes code compilation easier. Therefore, the following code is invalid:

```
void foo() {
    bar();
}

void bar() {
    /* ... */
}
```

It is possible, however, to get around this restriction with something called function prototypes.

## 2.3 Function Prototypes

Function prototypes alert the compiler of the functions that you intend to use in a file. A function prototype consists of the function header followed by a semicolon. For example, the following is a function prototype.

```
int foo(char bar);
```

The above line would allow all functions below it in your code to call **`foo()`**, even though **`foo()`**'s function body has not yet been given. Therefore, the following code fixes the error of the previous section's example.

```
void bar();

void foo() {
    bar();
}

void bar() {
    /* ... */
}
```

If you use a function prototype, you must include a body for that function later in your code. It is probably simplest to include at the top of your file prototypes for each of the functions that you use so that you need not worry about the order in which your functions are declared.

If you need to use a function in multiple different files, use a header file. Header files contain function prototypes. Their extension is **`.h`**, so you could create a file called **`myHeader.h`** with the following contents:

```
void bar();
```

In **`myCProgram.c`**:

```
#include “myHeader.h”

void foo() {
    bar();
}

void bar() {
    /* ... */
}
```

The **`#include`** directive tells the preprocessor to perform a substitution of the contents of the named file into the **`.c`** file before it’s compiled.

## 2.4 Other Notes on Functions

- Function overloading, having two functions with the same name but different parameter types, is not allowed in C, even though it is in Java.
- Since C has no classes or objects, **`private`**, **`protected`**, and **`public`** are not keywords.

# 3 Data Types

The primitive data types in C are **`int`**, **`char`**, **`short (int)`**, **`long (int)`**, **`double`** and **`float`**.

Note in particular that there is no **`boolean`** type. The **`int`** type is used in its place. All operations that normally would work on a boolean, like **`!`**, work on **`ints`** in C. A function that would return true or false should instead return an integer value, with 0 corresponding to false.

It is also worth noting that, unlike Java, C will not automatically initialize the values of variables within functions. In Java, a newly created variable of type **`int`** will have an initial value of 0. In C, the initial value of such a variable could be anything.

These data types are passed by value in functions. You will learn how to pass them by reference later.

## 3.1 Arrays

Arrays in C are somewhat different than arrays in Java. This document will cover how to create and use arrays as local variables on the stack. There is another way to create arrays, dynamic memory allocation, which you will learn about later.

To create a one-dimensional array of 5 **`ints`** called **`arr`**, one would use the following code:

```
int arr[5];
```

In the C99 standard, you can also declare a variable-length array whose size is determined at runtime.

```
int n = 5;
int arr[n];
```

If we want to create an array with specific initial values, we can use the following code:

```
int arr[5] = {1, 2, 3, 4, 5};
```

Variable-length arrays cannot be initialized in this manner.

You can access elements in an array as you would in Java. However, unlike Java, C does not perform bounds checking on array accesses. That is, if **`arr`** is a one-dimensional array of length 5, both **`arr[-1]`** and **`arr[10]`** are valid C expressions, even though they are outside the array. Therefore, be very careful when you access arrays to make sure you are accessing the elements you really mean to access.  Reading or writing to indices outside of the array could cause your program to have unexpected behavior. It is also noteworthy that C arrays are not objects and so have no **`length`** field. You will need to keep track of the length of an array yourself.

While the elements of a C array can be modified, the array variable itself is immutable. For instance, the following code is invalid:

```
int arr[3] = {5, 10, 15};
int brr[3];
brr = arr;
```

However, an array variable can be assigned to a pointer whose “pointee” type is the array element type. The pointer can then be used in the same manner as the array.

```
int arr[3] = {5, 10, 15};
int *aptr = arr; //aptr and arr now refer to the same array
aptr[0] = 20; //Set the first element of arr to 20
```

The syntax for passing arrays in C is different than it would be in Java. The following is a valid method header in Java:

```
    public static int getMax(int[] arr)
```

In C, the function header would be:

```
    int getMax(int *arr)
```

or[1]:

```
    int getMax(int arr[])
```

C arrays are passed as pointers, so, as in Java, modifying the contents of an array passed as a parameter will modify the original array. Unlike Java, arrays created inside a function call are deleted when the function returns, so returning a C array created inside a function will not behave as you would like. To get around this, one approach is to pass in an array as an argument to a function of return type **`void`** that sets or modifies the array's values as desired.

### 3.1.1 Multi-dimensional Arrays

C also allows for multi-dimensional arrays. Such arrays are declared using a similar syntax to one-dimensional arrays. For example, to declare and initialize a 2x3 array of **`ints`**, you would write:

```
    int arr[2][3] = {{0, 1, 2}, {3, 4, 5}};
```

Note that this syntax does not allow you to create "ragged" arrays, where different rows have different lengths.

Passing multi-dimensional arrays to functions requires a little more care than one-dimensional arrays. In order to correctly access elements of a multi-dimensional array, C must know all dimensions of the array, except the first. For instance, the function header for a function **`foo()`** that accepts an *m*x3 `int` array, for any *m*, you would write:

```
    void foo(int[][3])
```

In C99, you can have variable-length array parameters, but you still must specify all dimensions except the first.[2] For instance, the following function accepts a three-dimensional `int` array whose second and third dimensions are specified by parameters `n` and `m`:

```
    void foo(int n, int m, int[][n][m])
```

Syntactically, C treats an *n*-dimensional array as a one-dimensional array of fixed-size *n* − 1-dimensional arrays. The following code declares a 10x12 array of **`chars`** and creates a pointer to it:

[1] Caution: a parameter of the form type **`arr[]`** is semantically identical to a parameter of the form type **`*arr`**. This means, for example, that while **`sizeof()`** applied to a local array will yield the total size in memory of the array, **`sizeof()`** applied to an array parameter will yield the size of a pointer

[2] The reason for this has to do with how C handles arrays in memory. It's a bit complicated for this section, but ask a TA if you're curious.

```
char arr[10][12];
char (* arr_ptr)[12] = arr;
```

## 3.2 Strings

C has no explicit string type. However, a **`char`** array can be thought of as a string, which explains why the **`char **`** argument in **`main()`** is like a **`String`** array. The null character, '**`\0`**', is used to mark the end of a string, and such a string is said to be *null-terminated*. C supports string literals within double quotes, such as **`"Hello World!"`**, and treats them as a null-terminated **`char`** array. C strings are not as flexible as Java **`Strings`**, however. For example, you may not combine two strings using the **`+`** operator.

For the first lab, you will not need to create string variables, just print them. Later in this document is information about how to print strings. Information about how to create and manipulate strings will be presented later in the course.

## 3.3 `structs`

A **`struct`** is a feature of the C language that stores multiple variables, called fields, at once. One can think of them as Java objects without methods.

To define a **`struct`** whose type is **`Foo`** with two fields, an **`int x`** and a **`char y`**, you can use the following code above the functions in your .c file:

```
struct Foo {
    int x;
    char y;
};
```

You can then create a **`Foo`** named **`bar`** with the following code:

```
struct Foo bar;
```

Note that the type is actually **`struct Foo`**. To avoid needing to type **`struct`** every time you create a `Foo`, you can employ the **`typedef`** keyword. As an alternative to the above code for defining and creating a **`Foo`**, you can use the below code instead.

```
typedef struct {
    int x;
    char y;
} foo_t;

foo_t bar;
```

To access the field **`x`** in **`bar`**, our new **`Foo`**, we can use **`bar.x.`**

Note that, like primitives, structs are passed by value. We will cover a way to avoid this later in the course[3].

## 4 Loops and Conditionals

Loops and conditionals are similar in C and in Java. Like Java, C has **`for`** loops, **`while`** loops, **`do-while`** loops, and **`if`** statements. Since C does not have boolean variables, the conditions for these statements are instead **`ints`**. In C, any non-zero value signifies true, while zero signifies false. In other words, the following is valid code.

```
if(a+2) {
    printf("This prints if a is not -2");
} else {
    printf("This prints only if a is -2");
}
```

## 5 `#include` Statements and Libraries

Much like including your own headers, you can use **`#include`** statements for external code. You will sometimes wish to employ functions from the C standard library. These functions can perform tasks such as printing to the terminal, manipulating strings, and allocating arrays whose size is not known at compile time. Unlike Java, where the **`java.lang`** package is automatically imported for you, C does not automatically include any functions. You must include libraries yourself to use these functions. The C equivalent of Java's **`import`** statement is **`#include`**. Suppose that you want to use a function in **`stdio.h`**, which contains functions that handle input and output for your program. To do so, you must ensure that the top of your source file contains the line

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

More libraries will be introduced as they are needed.

## 6 `printf()`

The function **`printf()`** is a library function in **`stdio.h`**. It is used to output text, somewhat like Java's **`System.out.print()`**.[4] To output a string constant like **`"Welcome to CS33!"`** on its own line, you would use the following code:

```
printf("Welcome to CS33!\n");
```

[3] If you pass a struct by value, each of its fields is copied. A function that edits a struct which was passed to it by value does not edit the original **`struct`** but rather the copy, which has no effect beyond the scope of the function. The alternative to passing by value is passing by reference using what are called pointers, which we will learn about later.

[4] It would be more apt to compare it to Java's `System.out.printf()`.

Suppose, now, that we wanted to generalize this message for any CS course. We have an integer variable **`coursenum`**; if **`coursenum`** equals 32, we'd want to print "`Welcome to CS32!`". In Java, we could use a print statement like

```
System.out.print("Welcome to CS" + course_num + "!\n");
```

However, C strings cannot be appended like that. Instead, **`printf()`** allows us a different way to complete this task. We can use the statement:

```
printf("Welcome to CS%d!\n", course_num);
```

The **`%d`** signifies that we want to print the value of a base 10 integer that is given in the function arguments. You may find the following options useful:

- `%d` prints a base 10 integer, as explained above
- `%f` prints a floating point number
- `%u` prints an unsigned base 10 integer
- `%s` prints a null-terminated string
- `%c` prints the character associated with the provided ASCII value

One may include as many of these symbols in the same **`printf()`** statement as one likes. Assuming `a` and `b` are variables of type **`int`**, then the following code will print their values.

```
printf("The value of a is %d and the value of b is %d.\n", a, b);
```

Note that, if `a` is an `int`, Java permits

```
System.out.print(a);
```

while in C,

```
printf(a);
```

is a syntax error, since `a` is not a string

## 6.1 An Example - Hello CS33

The following simple program prints "`Hello World!`" to the terminal on its own line, followed by "`Welcome to CS33!`" on the next line.

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


int main(int argc, char **argv) {
    int course_num;
    course_num = 33;
    printf("Hello World!\nWelcome to CS%d!\n", coursenum);
```

```
        return 0;
    }
```

## 7 `Assert`

Assert statements form a very useful tool for testing and debugging C programs. **`assert()`** in C is like a function[5] provided by the header file ***assert.h*. `assert()`** takes a single argument, which is a boolean expression. If the expression evaluates to 1, then assert does nothing; if the expression evaluates to 0, then assert terminates execution of your program, and prints the statement that failed and the line number at which the code failed to **`stderr`**. In short, it *asserts* that the state of your program exhibits a particular characteristic.

Assert statements are so useful since, unlike other code errors or ways to debug code, they can inform you exactly where and when your code started to function incorrectly.

If you have assertions in your program and wish to turn them off, you can insert **`#define NDEBUG`** in your program above **`#include<assert.h>`** or compile with the option **`-DNDEBUG`**.

## 8 Compiling Code with `gcc`

We will use `gcc` as our C compiler. This will take your .c source files and find errors in them. If there are no errors, it will compile them into an executable binary file.

First, you should go to the folder with your source code.

If your project consists of the single file **`hello.c`** and you wish your executable file to be called **`hello`**, you would run the command

```
    gcc hello.c -o hello
```

You can compile multiple files in a similar way. If in addition to **`hello.c`** you have **`world.c`**, you would run the command

```
    gcc hello.c world.c -o hello
```

Adding the **`-Wall`** flag will show warnings, if any.

## 9 Running Your Code

Once you have created an executable binary, you may run it from the command line. Continuing our example from above, once we've created the binary **`hello`**, we can run the program by going to the folder with the binary and running the command

[5] Not precisely; `assert()` isn't really a function but rather a macro. This is how it can provide line number and code information.

```
    ./hello
```

This will execute the binary. Note that if you change your code, you will need to recompile it to update the binary.