C Programming for Java Programmers

Java based its syntax off of C, so if you know Java, you'll find a lot of things work the same way in C. There are some differences that we'll note below. Here is a sample C program that asks the user to enter two numbers and whether to add or subtract them. Then it does the appropriate operation.

```c
#include <stdio.h>

int main()
{
    int num1, num2;
    char type;

    printf("Enter the first number:\n");
    scanf("%d", &num1);

    printf("Enter the second number:\n");
    scanf("%d", &num2);

    printf("Enter a for add, s for subtract:\n");
    scanf(" %c", &type);

    if (type == 'a')
        printf("%d", num1+num2);
    else if (type == 's')
        printf("%d", num1-num2);
    else
        printf("I don't understand.");

    return 0;
}
```

Notice that the basic syntax of braces, semicolons, operators, and if statements is the same as in Java. Some differences to note:

- Importing things is done with `#include`.
- Instead of Java's `public static void main(String[] args)`, C uses `int main()`. You will also often see `int main(int argc, char *argv[])` if the program takes command line arguments. The main function has a return type of int. This is the status code that the C program will send when it closes. Usually we use `return 0` at the end of `main` to return the status code 0, for success.
- Input and output in C is different than in Java. Above we use `scanf` and `printf`, though there are other options. There is more on these a little later.

Loops also work the same in C as in Java, except that in older versions of C, you may need to declare the loop variable before the loop instead of inside it, like below.

```c
int i;
for (i=0; i<10; i++) {
    // something happens in here...
}
```

In those older versions, you have to declare all your variables at the start of functions, and not anywhere in the middle.

Data types Here are some similarities and differences between C and Java data types:

- Just like Java, C has int, float, double, and char data types.

- C has some data types, such as `unsigned char`, that are useful for low-level programming. Java doesn't have these.
- C originally didn't have a boolean data type, though newer versions did introduce one. Mostly, people use an int instead, with 0 = False and 1 = True.
- C does not have a string data type like Java. We'll look at strings a little later in these notes.
- The char data type can be interpreted as either a character or as an integer corresponding to the character's ASCII code. For instance, `char c='a'` and `char c=97` have the same effect.
- Arrays are similar, but the declaration syntax is a bit different. Where Java would have `int[] a = new int[10];`, C would have `int a[10];`.
- C is not an object-oriented programming language. There are no classes or objects.

See https://introcs.cs.princeton.edu/java/faq/c2java.html for a nice side-by-side comparison of C and Java.

Functions Functions work similarly in C and in Java except that C functions need to have a prototype, which is basically just a repeat of the declaration line. Here is an example:

```c
#include <stdio.h>

int add(int x, int y); // this is the prototype

int main() {
    printf("%d", add(2,2));
    return 0;
}

int add(int x, int y) {
    return x + y;
}
```

Since C is not object-oriented, there are no `public` or `static` keywords. The declaration just contains the return type, function name, and parameters.

The printf and scanf functions

A useful function for printing things is `printf`. It uses formatting codes to specify how things look. These formatting codes have been adopted in many other languages, so it's good to know them. Here is a basic example:

```c
printf("x is %d", x);
```

The `%d` acts as a placeholder, and the value of `x` is inserted there. Here is an example with two variables.

```c
printf("x is %d and y is %d", x, y);
```

There are a variety of different formatting codes. Here are some of the most common:

code	description
`%d`	for displaying integers in decimal
`%f`	for floating point numbers
`%g`	chooses between regular or scientific notation for floats
`%s`	for strings
`%c`	for single characters
`%x`	for displaying integers in hexadecimal
`%p`	for pointers (covered later)

The codes can be customized in a few ways. For example, to allot 9 spots for a floating point number with 2 after the decimal point, use `%9.2f`. Any slots not used will be replaced with spaces, which means you can use the code to right-justify numbers. If you want an integer to have leading 0s, like 0034 instead of 34, you can use `%04d`. Here is a short program with some examples.

```
#include <stdio.h>

int main() {
    int x = 14;
    double y = 12.934;
    printf("%6.2f\n", y);
    printf("%18.10f\n", y);
    printf("%d\n", x);
    printf("%5d\n", x);
    printf("%05d\n", x);
    printf("%x=%d and y=%f\n", x, y);
    printf("%s\n", "abcdefg");
    printf("%c %d\n", 'a', 'a');
    return 0;
}
```

Here is the output of the code above:

```
 12.93
    12.9340000000
14
   14
00014
e=439326816 and y=12.934000
abcdefg
a 97
```

A few notes: To use `printf`, include `stdio.h`. The `printf` function doesn't add a newline, so if you want things to appear on separate lines, you need to manually add a `\n` at the end.

scanf The `scanf` function can be used to get input from the keyboard. Here is an example that gets an integer.

```
int a;
char s[10]; % strings are just arrays of characters
printf("Enter a number");
scanf("%d", &a);
```

The function uses formatting codes to indicate the data type we're looking for. The variable name goes at the end, preceded by a `&` character. The `&` is the "address of" operator. It's needed here for `scanf`. There is more about it in the section on pointers. Like `printf`, the `scanf` function is in `stdio.h`.

Strings

C doesn't have a string data type. Instead, we use arrays of characters. Here are a few examples of declaring strings:

```
char a[] = {'a', 'b', 'c', '\0'};
char b[] = "abc";
char c[7];
```

Notice in the first example, the `'\0'` or null character at the end. Strings in C must be terminated with a null character. That is how C determines where the string ends. A very common error in C is to forget the null character, which can cause string methods to end up reading past the intended end of the string. That can cause serious problems. The declaration in the second line is a shorthand for what was done in the first line, and the

null character is automatically inserted. The third line declares a string long enough to hold 6 characters (leaving one for the null character). In the other examples, the C compiler infers the string length from the right side of the declaration.

Because of how pointers work (see below), if you want to use `scanf` to get a string, no `&` character is needed, like below.

```
char s[100];
printf("Enter a string");
scanf("%s", s);
```

In Java, if you want the length of a string `s`, you would use `s.length()`. But since C is not object-oriented, we can't do something like that. Instead, string operations are done with functions, which are mostly in `string.h`. For instance, here are a few examples:

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

int main()
{
    char a[10] = "abc", b[] = "defg";
    printf("Length of b: %d\n", strlen(b));
    strcat(a, b);  // concatenate b to a, like a = a + b in Java
    printf("%s", a);
    return 0;
}
```

Pointers

One of the trickiest things for people coming to C from Java are *pointers*. They are a way to directly access memory locations in a process's address space. A pointer variable is one that holds memory locations. To create a pointer variable and point it at an int variable called `x`, use the line below:

```
int *ptr = &x;
```

If we want to change `ptr` to point at a variable called `y`, use the following:

```
ptr = &y;
```

The `&` operator is the "address of" operator, which returns the memory location of its operand. Pointers are declared by putting `*` before the variable name. The `*` is also use to "dereference" the pointer, which means to get the value at the memory location that the pointer holds. Here is a program showing the basics of pointers:

```
#include <stdio.h>

int main() {
    int x = 2;
    printf("Memory location of x: %p\n", &x);

    int *ptr = &x;
    printf("Address held by ptr: %p\n", ptr);
    *ptr = 12;
    printf("%d\n", x);
    return 0;
}
```

Note that the formatting code `%p` is used to print out pointer variables. Note also, if we wanted a pointer pointing to something else, like a double, then the declaration would be `double *ptr`.

Pointers are used extensively in C. They give you low-level access to the machine, and that is especially helpful when doing low-level programming. They also help to save memory since you can pass around a pointer to something rather than making copies of it.

Pointers and functions Pointers are often as arguments to functions so that the function can change values passed to it. Here is an example:

```
#include <stdio.h>

void func(int *x, int y);

int main() {
    int a=4, b=7;
    func(&a, b);
    printf("a=%d b=%d", a, b);
    return 0;
}

void func(int *x, int y) {
    *x = 78;
    y = 99;
}
```

The result of this program is that the variable a in main is changed to 78, while b is unchanged. This has to do with the parameters of the function. The *x parameter is a pointer, which means it will hold the memory location of the variable passed to it (note the &a in the function call). Thus we can use *x to change the value of things passed to it since we have direct access to the memory location. On the other hand, the y parameter is not a pointer. It is instead a local variable and it receives a copy of what's passed to it. Any changes to y will only affect that local copy and will have no effect on the calling variable (b in this case).

Below is another example that uses pointers to create a function that swaps two variables. It's not possible to write something like this in Java or Python.

```
#include <stdio.h>

void swap(int *x, int *y);

int main() {
    int a=4, b=7;
    printf("a=%d b=%d", a, b);
    swap(&a, &b);
    printf("a=%d b=%d", a, b);

    return 0;
}

void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

Memory allocation

In C, if you need a large chunk of memory, you have to allocate it yourself. The function that does this is called `malloc`. It is found in `stdlib.h`, so that file needs to be included. Here is an example of how malloc works:

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

int main() {
    char *buffer = malloc(10000);
    buffer[0] = 'a';
    buffer[1] = 'b';
    buffer[2] = '\0';;

    printf("%s\n", buffer);
```

```
    free(buffer);

    int *buffer2 = malloc(50000*sizeof(int));
    return 0;
}
```

The `malloc` function takes one argument, which is the number of bytes to allocate. It returns a pointer to where that memory is located, which will always be on the heap. Arrays, on the other hand, are local variables on the stack. The heap is much larger than the stack, so it's better to use malloc when you need a large block of space.

Notice that in the second call to `malloc` above, we used `sizeof(int)`, but not in the first call. To see why, note that the argument to `malloc` is the number of bytes to allocate. A `char` in C is one byte, while an `int` in C is typically 4 bytes. In order to allocate enough space for 50,000 integers, we would need $50{,}000 \times 4 = 200{,}000$ bytes. In general, it's a good idea to use the `sizeof` function when using `malloc`, even for `char`.

Once the memory has been allocated, you can treat it just like an array, accessing elements using square brackets. When you are done with the memory, you can release it by calling the `free` function. If you're not using much memory, you don't have to free it. It will automatically be freed when your program exits. But if you are using lots of memory, especially in a long-running program, then it's good to call `free` to prevent your program from running out. Freed memory goes back to a pool of free memory that can be allocated for other things. Many higher level languages, like Java and Python, take care of this for you automatically, using a process called *garbage collection* that looks for things that are no longer being used and frees them . In C, you have to do this yourself. It's easy to accidentally lose track of the pointer to where the memory is allocated. This is a *memory leak*. Here is an example:

```
int *buf = malloc(10000*sizeof(int));
*buf = malloc(2000*sizeof(int));
```

After the first line, `buf` stores the memory location of the start of the 10,000 integer allocation. But we lose this location when we run the second line. After that, we have no way of knowing what that first location was and so we can't free it. That memory is essentially lost and wasted until the program exits.

Arrays vs. malloc If we need space for 2000 ints, we could either create an array via something like `int a[2000];` or we could use `malloc`. What's the difference? A process's memory is typically broken up into a couple of areas for different purposes. One of those holds the program's code, another holds global variables. The other two are the *stack* and *heap*. The stack is used to hold information needed for function calls. In particular, local variables for functions are stored on the stack. The stack generally isn't all that big, so if we need a large enough array, it won't be able to hold it. That's when to use `malloc`. An array defined on the stack will also be destroyed when the function ends, whereas memory on the heap persists. So if you want something accessible by multiple functions, heap memory allocated by `malloc` is a better choice.

Structs

C is not object-oriented, but it does have a way to group variables together into something a little like an object. This is called a *structure*. Here is an example:

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

struct Person {
    int age;
    char name[32];
};

int main() {
    struct Person bob;
    bob.age = 20;
    strcpy(bob.name, "what");
    printf("%d %s", bob.age, bob.name);
```

```
    return 0;
}
```

We access members of a structure using the dot operator just like in Java. If we have a pointer to a structure, however, then we need to use the -> operator to access the members. Here is what that would look like for the `Person` structure above:

```
struct Person *ptr = &bob;
ptr->age = 30;
```

Signed and unsigned types

C makes a distinction between *signed* and *unsigned* integer data types. For instance, C has `char` and `unsigned char` data types that are 8 bits. In a `char`, the leftmost bit indicates the sign of the number, whether it is positive or negative. It is not used for signs in an `unsigned char`. The end result is that an `unsigned char` can store values from 0 to 255, while a `char` stores values from −128 to 127.

Similarly, an `int` is a 4-byte (32-bit) data type. An `unsigned int` has values from 0 to from 4294967295 (which is $2^{32} - 1$). A regular (signed) `int` has values from −2147483648 to 2147483647.

If we try to store a value larger than a variable can hold, a wraparound will happen. For instance, if we have an `unsigned char` variable that is holding 255 and we add 1 to it, we get 256, which is larger than the max of 255. This ends up wrapping around back to 0. Similarly, if we have a `char` holding 127 and add 1, we get 128. But this wraps around to the minimum, which is −128. Both of these are examples of *overflows*. Overflows are particularly common with the `int` data type and are a well-known source of security vulnerabilities.

Bitwise operations

C is used for low-level and hardware programming, and at that level, it's sometimes important to work with individual bits of memory. C provides the following operators (which are also the same in many other programming languages):

- `&` — bitwise AND
- `|` — bitwise OR
- `^` — bitwise XOR
- `~` — bitwise NOT
- `<<` — left shift
- `>>` — right shift

Bitwise operations work on bits and groups of bits. The left shift operation shifts all the left by a specified amount. For instance, `x << 2` will shift all the bits of `x` left two positions. The two leftmost bits will essentially fall off the end of the number and be lost. At the right two new zeros will be inserted. Note in particular, it is a shift and not a rotation. Right shifts work similarly, except that bits are shifted to the right. Bits at the right will fall off the end and be lost, and zeros will come in on the left.

As an example, if `x` is 00001100, here are the results of some shifts:

`x << 1` = 00011000
`x << 2` = 00110000
`x << 3` = 01100000
`x << 4` = 11000000
`x << 5` = 10000000
`x << 6` = 00000000

Until bits start falling off the end, a left shift behaves like multiplying a number by 2. For instance, the binary value of x used above is 12, and the results of the shifts above are 24, 48, 96, 192, and 128, assuming that x is an unsigned character. A right shift behaves like dividing by 2 and rounding down.

Here are the rules for the bitwise AND, OR, and XOR operations:

x	y	x & y
0	0	0
1	0	0
0	1	0
1	1	1

x	y	x | y
0	0	0
1	0	1
0	1	1
1	1	1

x	y	x ^ y
0	0	0
1	0	1
0	1	1
1	1	0

When used on larger values, like a `char` or an `int`, the operation is applied to each bit separately. For instance, here is the result of a bitwise AND on two 8-bit values.

```
  10110010
& 11000110
----------
  10000010
```

Binary values can be represented in C using notation starting with `0b`. For instance, the binary number 10110010 would be represented by `0b10110010` in C. This notation is not a part of standard C, though many compilers do support it. Instead of pure binary like this, sometimes people use hex, which are indicated with `0x`. For instance the binary value above would be represented as `0xB3` in C.

Checking and setting specific bits

In low-level and embedded systems programming, it's often important to be able to access individual bits of a number. In embedded systems, memory is often scarce, and in networking, even wasted bit adds overhead that slows down the connection. If we have some values acting as flags, we could make separate variables for each, but a more efficient approach is to use individual bits of a variable to act as those flags. If we have an 8-bit number, each of those 8 bits can act as a separate flag. If we used 8 different variables instead, the total space usage would be 64 bits, which is much less efficient than the 8 bits if each bit is its own flag.

Checking the value of a bit To test a single bit, the `&` operator can be used to isolate it. Namely, if we AND with a binary number that is 0 everywhere except with a 1 in the bit we are interested, then the zeros will have the effect of wiping out everything else, and the 1 will isolate the desired bit. For instance, if we want to check if bit #6 (the sixth from the right) in an 8-bit variable x is set to 1, we would use the following if statement:

```
if (x & 0b00100000 != 0)
```

Here is how it would work with some examples:

	1	1	0	1	0	0	1	0
&	0	0	1	0	0	0	0	0
	0	0	0	0	0	0	0	0

	1	1	1	1	0	0	1	0
&	0	0	1	0	0	0	0	0
	0	0	1	0	0	0	0	0

Notice how everything is zeroed out except for possibly the sixth bit. Note that instead of using `0b00100000`, we could use the decimal number 32 or the hexadecimal number `0x20`. Sometimes you'll also see people use the operation `x & (1<<6)` for this.

Setting bits to 1 To set bit #6 to 1, we would use the `|` operation, specifically `x = x | 0b00100000`. The zeroes in all the other positions will have no effect on the other bits, and the 1 in the sixth position will have the effect of turning bit #6 to 1 if it is 0 and leaving it set at 1 if it is already 1. Here is an example:

```
    1 1 0 1 0 0 1 0          1 1 1 1 0 0 1 0
|   0 0 1 0 0 0 0 0      |   0 0 1 0 0 0 0 0
-----------------        -----------------
    1 1 1 1 0 0 1 0          1 1 1 1 0 0 1 0
```

Setting bits to 0 To set bit #6 to 0, we can AND with `0b11011111`. The ones will leave everything alone, and the 0 in bit #6 will force that bit to become 0. Here is an example:

```
    1 1 0 1 0 0 1 0          1 1 1 1 0 0 1 0
&   1 1 0 1 1 1 1 1      &   1 1 0 1 1 1 1 1
-----------------        -----------------
    1 1 0 1 0 0 1 0          1 1 0 1 0 0 1 0
```

Sometimes you'll see people write this as `x & ~(1 << 6)`, where
`1 << 6 gives a binary value with a 1 in bit \#6 and 0s elsewhere, and then the \verb++` operator flips all the bits so it becomes the same as `0b11011111`.

Flipping a bit Flipping a bit is where if it is 1, then it gets set to 0 and if it is 0, then it gets set to 1. The XOR operation is used for this. Specifically, to flip bit #6, we do `x ^ 0b00100000`. The zeroes have no effect on the number, and XOR-ing with 1 has the effect of flipping things. Here is an example:

```
    1 1 0 1 0 0 1 0          1 1 1 1 0 0 1 0
^   0 0 1 0 0 0 0 0      ^   0 0 1 0 0 0 0 0
-----------------        -----------------
    1 1 1 1 0 0 1 0          1 1 0 1 0 0 1 0
```

Extracting multiple bits Sometimes we want to extract multiple bits at a time. Maybe we have an 8-bit variable `x` and we want to extract the lower 3 bits (bits #0 to #2 from the right). We can use the `&` operation like this: `x & 0b00000111`. Alternately, we could use the decimal number 31 or the hexadecimal value `0x1F`. The value we `&` with to extract the desired bits is sometimes called a *bitmask* or just a *mask*. Here is an example:

```
    1 1 0 1 0 0 1 0
&   0 0 0 0 0 1 1 1
-----------------
    0 0 0 0 0 0 1 0
```

If we want the upper 3 bits, we could try a mask by `0b11100000`, but thinking of those 3 bits as representing a number from 0 to 7, we wouldn't get the right result. Instead, use a shift right like this: `x >> 5`.

Bitwise XOR and encryption The bitwise XOR operation gives a reversible way to combine two values. It is especially used in encryption. For example, say we have the binary value 11001010. We can encrypt it by XOR-ing it with a random binary key. Suppose that key is 00100001. The result is shown below.

```
  11001010
^ 00100001
  --------
  11101011
```

To decrypt, we XOR the encrypted value, 11101011, with the key again, and we get back the original 11001010. This is a basis for a very important encryption type called a stream cipher.

Buffer overflows

A very common error in C that can have disastrous security implications is a *buffer overflow*. C won't try to stop you if you try to read or write past the end of an array or string. Many other languages will crash your program with an index out of bounds error, but not C. Here is an example:

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

int main() {
    char a[10] = "abcdefg";
    char b[10] = "wxyz";

    strcpy(b, "0001112223334445556666");
    printf("a=%s\n", a);
    printf("b=%s\n", b);

    return 0;
}
```

The program creates a string `b` large enough to hold 9 characters (and a null), and then uses the `strcpy` function to copy a string much longer than 9 characters into it. This has the effect of writing the long string into the memory locations that are right after `b`, which will very likely be string `a`. When I ran it, it ended up printing that `a` is equal to `33444555666`.

Buffer overflows are arguably the most serious class of vulnerability out there since they can lead to total takeover a machine if an attacker can overwrite the right parts of memory. In C, a stack frame is generated for each function call. It holds information related to the function call, like local variables and the place in the code segment to return to once the function is done. If an attacker can cause one of those local variables to overflow far enough to overwrite the return address, then they can control what code is run after the function. They often overwrite that return address with the address of code that opens up a shell prompt, allowing them to run commands on the system.

So many of the biggest vulnerabilities for the past few decades have been buffer overflows, often in very widely used software. It's unfortunately very easy for a programmer, even an experienced one, to accidentally introduce a buffer overflow vulnerability into their code. There are coding practices to help avoid them, but they can be a pain, and it's easy to accidentally forget to use them.