Printing in C

1 How does one print in C, and how does it differ from C0/C1?

Recall that in C0 there were five functions we used in order to print things:

- `print()`
- `println()`
- `printint()`
- `printchar()`
- `printbool()`

Each of these functions allowed you to print a different type of argument, but only that argument. In C, all of these functions are replaced with one, much more powerful function: `printf`. This function makes use of the more powerful aspects of C to allow the user to have full control over how they wish to print.

`printf`'s first argument is the string that you wish to print, optionally containing some *format specifiers*. If the string you wish to print contains format specifiers, then during printing those format specifiers will be replaced by the values contained in the subsequent arguments. As an example, the following code would print out `I have 500 apples - or 1f4 in hexadecimal`.

```
unsigned int num_apples = 500;
printf("I have %u apples - or %x in hexadecimal\n", num_apples, num_apples);
```

There are a couple of key ideas to see in this example. The first is that the arguments after the format string replace the format specifiers (here `%u` and `%x`) in the same order as the format specifiers. Thus, if we replaced the first `num_apples` in the print with 200, the code would print out `I have 200 apples - or 1f4 in hexadecimal`. The second is that the format specifier indicates how an argument will be interpreted — the same variable can be printed as a decimal number, a hexadecimal number, or even a character. This can be incredibly helpful when trying to understand casting between integer types — and incredibly confusing when the format specifier does not match the type of the variable being printed.

Important: Due to security considerations, it is generally considered bad practice to give a variable as the sole argument of `printf` — for example `printf(my_string)`. If you wish to print out `my_string`, it is considered good practice to print it using `printf("%s", my_string)`.

The rest of this guide will deal with the specifics of how to create a format specifier to print out an argument exactly how you want, but first it is important to understand the general format that one follows. Most format specifiers are either a percent sign followed by a specifier character (e.g., `%d`) or a percent sign followed by a length sub-specifier followed by a specifier character (e.g., `%ld`). The general form for format specifiers is discussed in Section Advanced Printing.

2 The Specifier Character

The specifier character indicates what "kind" of thing will be printed and how the argument should be interpreted on a basic level — whether that be as a string, an integer, a pointer or any number of other things. Below is a summary of the most common specifier characters.

Signed Decimal Integers. In order to print a *signed decimal integer*, one must use the format character `d` or `i`. Both of these accomplish the exact same thing, but for historical reasons, `d` is more common.

Some examples are provided below:

```
printf("%d\n", 50);              // Prints 50
printf("%d\n", -213);            // Prints -213
printf("%i\n", -213);            // Prints -213
short x = 32767;                 // 32,767 is 2^15 - 1
printf("%d\n", x);               // Prints 32767
```

Important: An argument corresponding to `%d` (or `%i`) **must** have type **`int`** (or smaller signed types like **`short`** and **`signed char`**). Providing an argument of any other type is undefined behavior — it may print the expected result, or it may not on any given execution.

C provides flags to print all these other types (see below). Alternatively, one can use `%d` by explicitly casting the expression to **`int`**:

```
unsigned int y = 4294966796;
printf("%d\n", (int)y);          // Prints -500
```

As this example shows, what gets printed may not be what one had in mind.

Unsigned Decimal Integers. In order to print an unsigned decimal integer, one must use the format character `u`. Similarly to the signed integer specifier from before, this specifier character expects the argument passed to it to have type **`unsigned int`** (or a smaller unsigned type).

Some examples are provided below:

```
unsigned int w = 500;
printf("%u\n", w);               // Prints 500
unsigned int x = -500;           // -500 is implicitly cast to 4294966796
printf("%u\n", x);               // Prints 4294966796
unsigned short y = 65535;        // 65,535 is 2^16 - 1
printf("%d\n", y);               // Prints 32767
int z = 15122;
printf("%u\n", (unsigned int)z); // Prints 15122
```

As the last example shows, other types need to be cast explicitly to **`unsigned int`**.

Unsigned Hexadecimal Integer. In order to print an unsigned hexadecimal integer, one must use either the format character `x` or `X`. The only difference is that `x` represents alphabetic characters with a lowercase letter, and `X` represents them with an uppercase letter. Just like with `%u`, if the corresponding argument must be an **`unsigned int`** (or smaller unsigned type).

Some examples are provided below:

```
printf("%x\n", 31);               // Prints 1f
printf("%X\n", 31);               // Prints 1F
printf("%x\n", (unsigned int)-2); // Prints fffffffe
printf("%X\n", (unsigned int)-2); // Prints FFFFFFFE
unsigned int a = 4294967295;      // 4,294,967,295 is 2^32 - 1
a++;                              // 0 mod 2^32
printf("%x\n", a);                // Prints 0
```

Note: One useful property of the `%x` specifier is that it exactly represents how an integer is stored in the computer. As such it can be very helpful when debugging bitwise operations and casting between integer types — make sure to include explicit casts to print types other than **`unsigned int`**.

Unsigned Octal Integer. In order to print an unsigned octal integer, one must use the format character `o`. This format character is almost identical to the unsigned hexadecimal integer, except it prints the number in octal (base 8).

Some examples are provided below:

```
printf("%o\n", 31);               // Prints 37
printf("%o\n", -2);               // Prints 37777777776
```

We don't have much use for octal in 15-122.

Character. In order to print an ASCII character, one must use the format character `c`. This format character takes in a **`char`** and prints out the ASCII character that it represents. Given that a **`char`** is an integer type, this specifier can be used to help convert between the ASCII representation and the integral value.

Some examples are provided below:

```
printf("%c\n", 'a');              // Prints a
printf("%c\n", 97);               // Prints a
printf("%d\n", 'a');              // Prints 97
char* s = "I love 15-122";
printf("%c;%c\n", s[0], s[4]);    // Prints I;v
```

String. In order to print an entire string, one must use the format character `s`. This format character assumes that the input is a `NUL`-terminated string (e.g. a **`char*`**). If any other pointer type is used for the argument, the resulting output is likely to be garbled and not understandable

Some examples are provided below:

```
printf("Hi, my name is %s\n", "Alex");  // Prints Hi, my name is Alex
char *s = "15-122";
printf("Hi, my name is %s\n", s);       // Prints Hi, my name is 15-122
```

Pointer Address. In order to print the address of a pointer, one must use the format character `p`. This can be useful when debugging (for instance to track a specific pointer). However, because exact pointer addresses are not static, it is possible for the value printed to change between runs.

Some examples are provided below. The comments show what this program printed when run — your results will likely be different:

```
char *s = "15-122";
printf("%p\n", s);                  // Prints 0x4006a0
int *xp = malloc(sizeof(int));
printf("%p\n", (void*)xp);          // Prints 0x1ba2010
int x = 5;
int *xpp = &x;
printf("%p\n", (void*)xpp);         // Prints 0x7ffc406364b4
printf("%p\n", (void*)0xdeadbeef); // Prints 0xdeadbeef
printf("%p\n", NULL);               // Prints (nil)
free(xp);
```

Note: [Other Specifier Characters]

There are several other specifier characters, for example to deal with floating point numbers. As floating point numbers are not necessary for 15-122, we will not go over these specifier characters.

3 Length Sub-Specifiers

As mentioned in the previous section, all of the integer specifier characters assume that the input is a 32-bit integer. If you wish to print something that is not a 32-bit integer you must explicitly provide the size of the integer through the character immediately preceding the specifier character. Below we provide a list of all of the length specifiers, as well as the integer types that those length specifiers correspond to.

- `hh`: The `hh` length specifier indicates that the integer to be printed has the same length as a **`char`**. In 15-122 programming assignments, this will mean any integer that is 8 bits.
- `h`: The `h` length specifier indicates that the integer to be printed has the same length as a **`short`**. In 15-122 programming assignments, this will mean any integer that is 16 bits.
- *(no length specifier)*: When there is no length specifier, this indicates that the integer to be printed has the same length as an **`int`**. In 15-122 programming assignments, this will mean any integer that is 32 bits.
- `l`: The `l` length specifier indicates that the integer to be printed has the same length as a **`long`**. In 15-122 programming assignments, this will mean any integer that is 64 bits.
- `z`: The `z` length specifier indicates that the integer to be printed has the same length as a `size_t`. In 15-122 programming assignments, this will mean any integer that is 64 bits.

There are several other length specifiers out there, however they are not necessary in almost all cases.

For convenience, here is a table converting between integer type and the format specifier needed to print that type.

4 Advanced Printing

The format specifiers have a number of ways to fine-tune the exact way that an argument is printed out. These are not important unless you wish to write print statements that look elegant on your screen.

Type	Format Specifier
`char`	%`hhd`
`unsigned char`	%`hhu`
`short`	%`hd`
`unsigned short`	%`hu`
`int`	%`d`
`unsigned int`	%`u`
`long`	%`ld`
`unsigned long`	%`lu`
`size_t`	%`zu`

To understand these parameters, one must first explore the most general layout for a format specifier. All format specifiers take the form of `%[flags][width][.precision][length]specifier`. In other words, all format specifiers start with a percent sign, optionally followed by one or more flags, optionally followed by the width, optionally followed by a period and then the precision, optionally followed by a length, and finally followed by specifier character.

4.1 Flags

The first optional parameter in the format specifier is the flags. When present, each flag modifies the output. The flags can be combined arbitrarily, however if two flags contradict each other, then only one of those flags will be considered. The rest of this section explores the possible flags

`-`. When the `-` flag is present in conjunction with the width field (See Section Width below), the resulting printed value will be left-justified within the width rather than being right-justified.

Some illustrative examples:

```
printf("<%5d>\n", 10);        // Prints <   10>
printf("<%-5d>\n", 10);       // Prints <10   >
printf("<%6s>\n", "hello");   // Prints < hello>
printf("<%-6s>\n", "hello");  // Prints <hello >
```

`+`. When the `+` flag is present, positive numbers will be prepended with the `+` sign. If the specifier character is for something other than a number (like a string), this flag is nonsensical and will result in a compiler error.

Some illustrative examples:

```
printf("<%d>\n", 10);     // Prints <10>
printf("<%d>\n", -10);    // Prints <-10>
printf("<%+d>\n", 10);    // Prints <+10>
printf("<%+d>\n", -10);   // Prints <-10>
```

`[SPACE]`. When the `[SPACE]` flag is present, positive numbers will be prepended with a space. If the specifier character is for something other than a number (like a string), this flag is nonsensical and will result in a compiler error.

Some illustrative examples:

```
printf("<%d>\n", 10);       // Prints <10>
printf("<%d>\n", -10);      // Prints <-10>
printf("<% d>\n", 10);      // Prints < 10>
printf("<% d>\n", -10);     // Prints <-10>
//printf("<% s>\n", "hi");  // Compilation error!
```

#. When the **#** flag is present and the specifier character is `x`, `X` or `o`, the printed number will be prepended with `0x`, `0X` or `0`; respectively. When the **#** flag is present and the specifier character is anything else, this flag is nonsensical and is ignored.

Some illustrative examples:

```
unsigned int x = 10;
printf("<%x>\n", x);        // Prints <a>
printf("<%X>\n", x);        // Prints <A>
printf("<%o>\n", x);        // Prints <12>
printf("<%d>\n", x);        // Prints <10>
printf("<%#x>\n", x);       // Prints <0xa>
printf("<%#X>\n", x);       // Prints <0XA>
printf("<%#o>\n", x);       // Prints <012>
//printf("<%#d>\n", 10);    // Compilation error!
```

0. When the `0` flag is present in conjunction with the width field (See Section Width below), the resulting printed value will be padded with 0's if necessary rather than spaces. Note that this flag only works when the specifier character indicates a number is to be printed out.

Some illustrative examples:

```
printf("<%5d>\n", 10);        // Prints <   10>
printf("<%05d>\n", 10);       // Prints <00010>
```

4.2 Width

The width specifier is simply a number which indicates the minimum number of characters to be printed. By default, if the output needs to be padded, it will be left-aligned and padded with spaces. It is also possible to replace the number with the $*$ character, which indicates that the number is an additional argument rather than baked into the string

Some illustrative examples:

```
printf("<%5d>\n", 10);          // Prints <   10>
printf("<%*d>\n", 5, 10);       // Prints <   10>
printf("<%8s>\n", "hello");     // Prints <    hello>
printf("<%*s>\n", 8, "hello");  // Prints <    hello>
```

4.3 Precision

The precision specifier is a period (`.`) followed by the precision needed. Exactly what the precision means depends on the specifier character. For integer specifiers (e.g. `d`, `x`, `o`, `u`, `x` and `X`) the precision is similar to the width specifier, but always pads with leading zeros. For the string specifier (e.g. `s`),

the precision indicates the maximum number of characters to print. For the character and pointer address specifiers (`c` and `p`), the precision specifier is meaningless and results in a compiler error.

Much like the width specifier, it is also possible to replace the number with the $*$ character, which indicates that the number is an additional argument rather than baked into the string

Some illustrative examples:

```
printf("<%.5d>\n", 10);                 // Prints <00010>
printf("<%.*d>\n", 5, 10);              // Prints <00010>
printf("<%.5s>\n", "hello friends");    // Prints <hello>
printf("<%.*s>\n", 5, "hello friends"); // Prints <hello>
```

Generally speaking the precision specifier is used for floating point numbers, which are not necessary in 15-122. See the note on Other Specifier Characters.