The fundamental data types

C provides many fundamental data types, some of which we have already discussed. Other data types, such as arrays, pointers, and structures, can be described in terms of fundamental data types. Here, we discuss limitations on what can be stored for each fundamental data type shown in the table below.

Fundamental data types: long form		
`char`	`signed char`	`unsigned char`
`signed short int`	`signed int`	`signed long int`
`unsigned short int`	`unsigned int`	`unsigned long int`
`float`	`double`	`long double`

Usually, the `signed` keyword is not used. For example, `signed int` is equivalent to `int`. Thus, we use the shorter name (easy to type) `int`. *However, the type* `char` *is special in this regard.* The type `char` is equivalent to either `signed char` or `unsigned char` depending on the compiler. Further, the keywords keywords `short int`, `long int`, and `unsigned int` are shortened to just `short`, `long`, and `unsigned`, respectively. With all these notations, we make a new list shown in the table below.

Fundamental data types		
`char`	`signed char`	`unsigned char`
`short`	`int`	`long`
`unsigned short`	`unsigned`	`unsigned long`
`float`	`double`	`long double`

The data type char

In C, variables of any integral type can be used to represent characters. In particular, `char` and `int` variables are used for this purpose. Constants such as 'B' and '*', that we think of as characters, are actually represented by integer values. Thus, a `char` variable can represent a character as well as an integer. Similarly, an `int` variable can represent an integer as well as a character. However, each `char` is stored in memory using 1 byte (compare to the usual 4 byte for an `int`), which is usually composed of 8 bits. Since each bit is either 0 or 1, we can store $2^8 = 256$ distinct values in 1 byte. Since the number of characters is small, only a subset of these 256 distinct values is used to store the set of characters. The character set includes lower- and uppercase letters, digits, punctuation, and special characters such as % and +. The character set also includes white space characters, such as blank, tab, and newline. The table below shows some character constants and their corresponding integer values.

Some character constants and their corresponding integer values							
Character constants	'a'	'b'	'c'	.....	'z'		
Corresponding integer values	97	98	99	.....	122		
Character constants	'A'	'B'	'C'	.....	'Z'		
Corresponding integer values	65	66	67	.....	90		
Character constants	'0'	'1'	'2'	.....	'9'		
Corresponding integer values	48	49	50	.....	57		
Character constants	'#'	'%'	'&'	.....	'*'		
Corresponding integer values	35	37	38	.....	42		
Non-printable characters	newline		tab	...	blank		
Corresponding integer values	10		9	...	32		

The following simple program prints a few characters and their corresponding values. Note that `%c` prints a character and `%d` prints the corresponding integer values.

```
//Program to print integer values of some characters
#include<stdio.h>
int main()
{
 printf("Characters followed by digits:\n");
 printf("%c %d %c %d %c %d\n",'a','a','c','c','z','z');
 printf("%c %d %c %d %c %d\n",'A','A','C','C','Z','Z');
 printf("%c %d %c %d %c %d\n",'0','0','2','2','9','9');
 printf("%c %d %c %d %c %d\n",'#','#','%','%','*','*');
 printf("newline %d tab %d blank %d\n",'\n','\t',' ');
 return 0;
}
```

Listing 11: C program "ch_int.c"

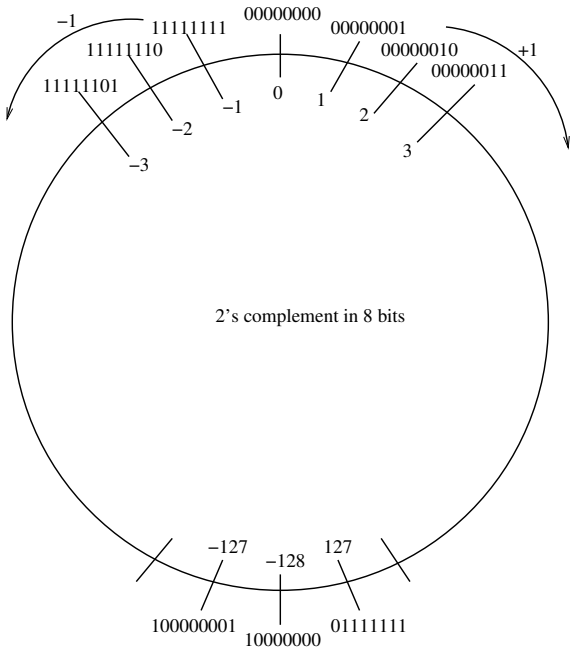
Compiling with `$ gcc ch_int.c ↩` and running the default executable with `$ ./a.out ↩` produce the following output:

```
Characters followed by digits:
a 97 c 99 z 122
A 65 C 67 Z 90
0 48 2 50 9 57
# 35 % 37 * 42
newline 10 tab 9 blank 32
```

Note that there is no particular relationship between a character constant representing a digit and the corresponding integer values. However, `'0', '1', '2',...,'9'` occur in order of increasing integer values. The same is true for the alphabet as well. Thus, observe the output of the following code segment:

```
printf("%d %d\n",'d'-'a','9'-'1'); // 3  8 is printed
printf("%c %c %c\n",'a'+1,98,'b'-('a'-'A')); // b  b B is printed
```

Since the lower case alphabets are stored consecutively as integers, the difference of `'d'-'a'` as an integer is 3, and a similar explanation holds for `'9'-'0'`. Also, due to consecutive storage of lower case alphabets as well as upper case alphabets, we have `'b'-'B'='a'-'A'` and hence `'b'-('a'-'A')='b'-('b'-'B')='B'`
So, a lowercase alphabet can be converted to an upper case alphabet by adding `'A'-'a'` to it.



Now we discuss the storage of `char` in memory. We discuss the storage of `signed char` (uses both positive and negative integers) first, which closely matches with that of `int` (uses both positive and negative integers), the only difference is that 4 bytes is used for `int` instead of 1 byte for `signed char`. Integer values are stored using the two's complement scheme.

In the 8-bit configuration, the most significant bit (MSB) determines the sign of the number. An MSB 0 denotes a non-negative integer, and an MSB 1 denotes a negative integer. The range of integers used in `signed char` is from 2^{-7} to $2^7 - 1$, i.e., from -128 to 127. These integers are stored according to the circular diagram shown in the figure. The integers increase from 0 in the clockwise direction and decrease from 0 along the anti-clockwise direction. However, due to the circular arrangement, adding 1 to 127 gives -128 and subtracting 1 from -128 gives 127. To see this, see the following code segment:

```
char a=127,b,c=-128,d,e,f;
b=a+1;
d=c-1;
e=a+10;
f=c-10;
printf("%d %d %d %d\n",a,b,c,d); //127 -128 -128 127 is the output
printf("%d %d\n",e,f); // -119 118 is printed
```

Similarly, adding 10 to 127 gives -119 and subtracting 10 from -128 gives 118. Thus, whenever we exceed the range of values in `char`, it wraps around the circle. Thus, to avoid unpredictable results, we must be careful not to exceed the range.

Now we consider `unsigned char`, which is a data type in C that stores a single byte (8 bits) of non-negative integer values, ranging from 0 to $2^8 - 1 = 255$. Unlike a `signed char`, it does not have a sign bit, meaning it cannot store negative numbers. Here, too, the `unsigned char` value that exceeds the range wraps around.

Finally, as mentioned before, `char` is compiler dependent. It may represent `signed char` or `unsigned char`. Both of these cases have been discussed above.

The data type int

We first discuss the representation of `int` assuming that 4-byte (i.e., 32 bits) is used to store an `int`. The discussion follows that of `signed char` closely. Using two's complement, the range of integers in `int` is from $-2^{31} = -2147483648$ to $2^{31} - 1 = 2147483647$. Next, we consider `unsigned int`, which is a data type in C that stores 4 bytes (32 bits) of non-negative integer values, ranging from 0 to $2^{32} - 1$. Unlike an `int`, it does not have a sign bit, meaning it cannot store negative numbers. The storage and effects of exceeding the range for both the cases of `int` and `unsigned int` follow those of `signed char` and `unsigned char` respectively.

The table below summarizes the range of values by different `int` types. The sizes of various `int` types are those of my macOS.

Range of various int types in memory		
Types of `int`	Sizes in bytes	Range
`short`	2	-2^{15} to $2^{15} - 1$
`unsigned short`	2	0 to $2^{16} - 1$
`int`	4	-2^{31} to $2^{31} - 1$
`unsigned`	4	0 to $2^{32} - 1$
`long`	8	-2^{63} to $2^{63} - 1$
`unsigned long`	8	0 to $2^{64} - 1$

Return to Lecture Topics