2.7 — Chars

BY ALEX ON JUNE 9TH, 2007 | LAST MODIFIED BY ALEX ON JULY 5TH, 2017

Even though the char data type is an integer (and thus follows all of the normal integer rules), we typically work with chars in a different way than normal integers. A char variable holds a 1-byte integer. However, instead of interpreting the value of the char as an integer, the value of a char variable is typically *interpreted* as an ASCII character.

ASCII stands for American Standard Code for Information Interchange, and it defines a particular way to represent English characters (plus a few other symbols) as numbers between 0 and 127 (called an **ASCII code** or code point). For instance, the character 'a' is code 97. 'b' is code 98. Characters are always placed between single quotes.

Here's a full table of ASCII characters:

Code	Symbol	Code	Symbol	Code	Symbol	Code	Symbol
0	NUL (null)	32	(space)	64	@	96	`
1	SOH (start of header)	33	!	65	A	97	a
2	STX (start of text)	34	”	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	’	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	HT (horizontal tab)	41	)	73	I	105	i
10	LF (line feed/new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (form feed / new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (data control 1)	49	1	81	Q	113	q
18	DC2 (data control 2)	50	2	82	R	114	r
19	DC3 (data control 3)	51	3	83	S	115	s
20	DC4 (data control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of transmission block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	|
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL (delete)

Codes 0-31 are called the unprintable chars, and they're mostly used to do formatting and control printers. Most of these are obsolete now.

Codes 32-127 are called the printable characters, and they represent the letters, numbers, and punctuation that most computers use to display basic English text.

The following two initializations both initialize the char with integer value 97:

```
char ch1(97); // initialize with integer 97
char ch2('a'); // initialize with code point for 'a' (97)
```

One word of caution: be careful not to mix up character (keyboard) numbers with actual numbers. The following two initializations are not the same

```
char ch(5); // initialize with integer 5
char ch('5'); // initialize with code point for '5' (53)
```

Printing chars

When using cout to print a char, cout outputs the char variable as an ASCII character instead of a number:

```
#include <iostream>

int main()
{
    char ch(97); // even though we're initializing ch with an integer
    std::cout << ch; // cout prints a character
    return 0;
}
```

This produces the result:

a

We can also output char literals directly:

```
cout << 'b';
```

This produces the result:

b

Note: The fixed width integer int8_t is usually treated the same as a signed char in C++, so it will generally print as a char instead of an integer.

Printing chars as integers via type casting

If we want to output a char as a number instead of a character, we have to tell cout to print the char as if it were an integer. One (poor) way to do this is by assigning the char to an integer, and printing the integer:

```
#include <iostream>

int main()
{
    char ch(97);
    int i(ch); // assign the value of ch to an integer
    std::cout << i; // print the integer value
    return 0;
}
```

However, this is clunky. A better way is to use a type cast. A **type cast** creates a value of one type from a value of another type. To convert between fundamental data types (for example, from a char to an int, or vice versa), we use a type cast called a **static cast**.

The syntax for the static cast looks a little funny:

```
static_cast<new_type>(expression)
```

static_cast takes the value from an expression as input, and converts it into whatever fundamental type *new_type* represents (e.g. int, boolean, char, double).

Here's using a static cast to create an integer value from our char value:

```
#include <iostream>

int main()
{
    char ch(97);
    std::cout << ch << std::endl;
    std::cout << static_cast<int>(ch) << std::endl;
    std::cout << ch << std::endl;
    return 0;
}
```

This results in:

```
a
97
a
```

It's important to note that static_cast takes an expression as input. When we pass in a variable, that variable is evaluated to produce its value, which is then converted to the new type. The variable is *not* affected by casting its value. In the above case, ch is still a char, and still holds the same value.

Also note that static casting doesn't do any range checking, so if you cast an integer that is too big to fit into a char, you'll overflow your char.

We'll talk more about static casts and the different types of casts in a future lesson.

Inputting chars

The following program asks the user to input a character, then prints out both the character and its ASCII code:

```
#include <iostream>

int main()
{
    std::cout << "Input a keyboard character: ";

    char ch;
    std::cin >> ch;
    std::cout << ch << " has ASCII code " << static_cast<int>(ch) << std::endl;

    return 0;
}
```

Here's the output from one run:

```
Input a keyboard character: q
q has ASCII code 113
```

Note that even though cin will let you enter multiple characters, ch will only hold 1 character. Consequently, only the first input character is placed in ch. The rest of the user input is left in the input buffer that cin uses, and can be accessed with subsequent calls to cin.

You can see this behavior in the following example:

```
#include <iostream>

```

```cpp
int main()
{
    std::cout << "Input a keyboard character: "; // assume the user enters "abcd" (without quotes)

    char ch;
    std::cin >> ch; // ch = 'a', "bcd" is left queued.
    std::cout << ch << " has ASCII code " << static_cast<int>(ch) << std::endl;

    // Note: The following cin doesn't ask the user for input, it grabs queued input!
    std::cin >> ch; // ch = 'b', "cd" is left queued.
    std::cout << ch << " has ASCII code " << static_cast<int>(ch) << std::endl;

    return 0;
}
```

```
Input a keyboard character: abcd
a has ASCII code 97
b has ASCII code 98
```

Char size, range, and default sign

Char is defined by C++ to always be one byte in size. By default, a char may be signed or unsigned (though it's usually signed). If you're using chars to hold ASCII characters, you don't need to specify a sign (since both signed and unsigned chars can hold values between 0 and 127).

If you're using a char to hold small integers, you should always specify whether it is signed or unsigned. A signed char can hold a number between -128 and 127. An unsigned char can hold a number between 0 and 255.

Escape sequences

C++ has some characters that have special meaning. These characters are called **escape sequences**. An escape sequence starts with a '\' (backslash) , and then a following letter or number.

The most common escape sequence is '\n', which can be used to embed a newline in a string of text:

```cpp
#include <iostream>

int main()
{
    std::cout << "First line\nSecond line" << std::endl;
    return 0;
}
```

This outputs:

```
First line
Second line
```

Another commonly used escape sequence is '\t', which embeds a tab:

```cpp
#include <iostream>

int main()
{
    std::cout << "First part\tSecond part";
    return 0;
}
```

Which outputs:

```
First part      Second part
```

Three other notable escape sequences are:
\' prints a single quote
\" prints a double quote
\\ prints a backslash

Here's a table of all of the escape sequences:

Name	Symbol	Meaning
Alert	\a	Makes an alert, such as a beep
Backspace	\b	Moves the cursor back one space
Formfeed	\f	Moves the cursor to next logical page
Newline	\n	Moves cursor to next line
Carriage return	\r	Moves cursor to beginning of line
Horizontal tab	\t	Prints a horizontal tab
Vertical tab	\v	Prints a vertical tab
Single quote	\'	Prints a single quote
Double quote	\"	Prints a double quote
Backslash	\\	Prints a backslash
Question mark	\?	Prints a question mark
Octal number	\(number)	Translates into char represented by octal
Hex number	\x(number)	Translates into char represented by hex number

Here are some examples:

```
#include <iostream>

int main()
{
    std::cout << "\"This is quoted text\"\n";
    std::cout << "This string contains a single backslash \\" << std::endl;
    std::cout << "6F in hex is char \'\x6F\'" << std::endl;
    return 0;
}
```

Prints:

```
"This is quoted text"
This string contains a single backslash \
6F in hex is char 'o'
```

Newline (\n) vs. std::endl -- which should you use?

You may have noticed in the last example that we can use \n to move the cursor to the next line, which appears to duplicate the functionality of std::endl. However, they are slightly different.

When std::cout is used for output, the output may be buffered -- that is, std::cout may not send the text to the output device immediately. Instead, it may opt to "collect output" for a while before writing it out. This is done for performance reasons, as in some cases, writing one larger blob of data is much faster than making many small writes.

Both '\n' and std::endl will move the cursor to the next line. In addition, std::endl will also ensure that any queued output is actually output before continuing.

So when should you use '\n' vs std::endl? The short answer is:

- Use *std::endl* when you need to ensure your output is output immediately (e.g. when writing a record to a file, or when updating a progress bar). Note that this may have a performance cost, particularly if writing to the output device is slow (e.g. when writing a file to a disk).

- Use *'\n'* in other cases.

(Administrative note: We're in the process of converting other articles in this tutorial over to '\n' instead of std::endl, because for console output, this is almost always the better choice.)

What's the difference between putting symbols in single and double quotes?

As you learned above, chars are always put in single quotes (e.g. 'a', '+', '5'). A char can only represent one symbol (e.g. the letter a, the plus symbol, the number 5). Something like this is illegal:

```
1 char ch('56'); // a char can only hold one symbol
```

Text put between double quotes is called a string (e.g. "Hello, world!"). A **string** is a collection of sequential characters (and thus, a string can hold multiple symbols).

For now, you're welcome to use string literals in your code:

```
1 std::cout << "Hello, world!"; // "Hello, world!" is a string literal
```

However, string variables are a little more complex than chars or the fundamental data types, so we'll reserve discussion of those until later.

What about the other char types, wchar_t, char16_t, and char32_t?

wchar_t should be avoided in almost all cases (except when interfacing with the Windows API). Its size is implementation defined, and is not reliable. It has largely been deprecated.

Much like ASCII maps the integers 0-127 to American English characters, other character encoding standards exist to map integers (of varying sizes) to characters in other languages. The most well-known mapping outside of ASCII is the Unicode standard, which maps over 110,000 integers to characters in many different languages. Because Unicode contains so many code points, a single Unicode code point needs 32-bits to represent a character (called UTF-32). However, Unicode characters can also be encoded using multiple 16-bit or 8-bit characters (called UTF-16 and UTF-8 respectively).

char16_t and char32_t were added to C++11 to provide explicit support for 16-bit and 32-bit Unicode characters (8-bit Unicode characters are already supported by the normal char type).

You won't need to use char16_t or char32_t unless you're planning on making your program Unicode compatible and you are using 16-bit or 32-bit Unicode characters. Unicode and localization is generally outside the scope of these tutorials, so we won't cover it further.

In the meantime, you should only use ASCII characters when working with characters (and strings). Using characters from other character sets may cause your code to work incorrectly.

 **2.8 -- Literals**

 **Index**

 **2.6 -- Boolean values and an introduction to if statements**