4.11 — Chars

learncpp.com/cpp-tutorial/chars/

To this point, the fundamental data types we've looked at have been used to hold numbers (integers and floating points) or true/false values (Booleans). But what if we want to store letters or punctuation?

```
#include <iostream>

int main()
{
    std::cout << "Would you like a burrito? (y/n)";

    // We want the user to enter a 'y' or 'n' character
    // How do we do this?

    return 0;
}
```

The **char** data type was designed to hold a single `character`. A **character** can be a single letter, number, symbol, or whitespace.

The char data type is an integral type, meaning the underlying value is stored as an integer. Similar to how a Boolean value `0` is interpreted as `false` and non-zero is interpreted as `true`, the integer stored by a `char` variable are intepreted 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 example, ASCII code 97 is interpreted as the character 'a'.

Character literals are always placed between single quotes (e.g. 'g', '1', ' ').

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 and 127 are called the unprintable chars. These codes were designed to control peripheral devices such as printers (e.g. by instructing the printer how to move the print head). Most of these are obsolete now. If you try to print these chars, the results are dependent upon your OS (you may get some emoji-like characters).

Codes 32-126 are called the printable characters, and they represent the letters, number characters, and punctuation that most computers use to display basic English text.

Initializing chars

You can initialize char variables using character literals:

```
char ch2{ 'a' }; // initialize with code point for 'a' (stored as integer 97) (preferred)
```

You can initialize chars with integers as well, but this should be avoided if possible

```
char ch1{ 97 }; // initialize with integer 97 ('a') (not preferred)
```

Warning

Be careful not to mix up character numbers with integer numbers. The following two initializations are not the same:

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

Character numbers are intended to be used when we want to represent numbers as text, rather than as numbers to apply mathematical operations to.

Printing chars

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

```
#include <iostream>

int main()
{
    char ch1{ 'a' }; // (preferred)
    std::cout << ch1; // cout prints character 'a'

    char ch2{ 98 }; // code point for 'b' (not preferred)
    std::cout << ch2; // cout prints a character ('b')


    return 0;
}
```

This produces the result:

```
ab
```

We can also output char literals directly:

```
std::cout << 'c';
```

This produces the result:

```
c
```

Inputting chars

The following program asks the user to input a character, then prints out the character:

```
#include <iostream>

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

    char ch{};
    std::cin >> ch;
    std::cout << "You entered: " << ch << '\n';

    return 0;
}
```

Here's the output from one run:

```
Input a keyboard character: q
You entered: q
```

Note that std::cin will let you enter multiple characters. However, variable *ch* can only hold 1 character. Consequently, only the first input character is extracted into variable *ch*. The rest of the user input is left in the input buffer that std::cin uses, and can be extracted with

subsequent calls to std::cin.

You can see this behavior in the following example:

```
#include <iostream>

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 << "You entered: " << ch << '\n';

    // 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 << "You entered: " << ch << '\n';

    return 0;
}
```

```
Input a keyboard character: abcd
You entered: a
You entered: b
```

If you want to read in more than one char at a time (e.g. to read in a name, word, or sentence), you'll want to use a string instead of a char. A string is a collection of sequential characters (and thus, a string can hold multiple symbols). We discuss this in upcoming lesson (5.9 -- Introduction to std::string).

Char size, range, and default sign

Char is defined by C++ to always be 1 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 (something you should not do unless you're explicitly optimizing for space), 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

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

You've already seen the most common escape sequence: `'\n'`, which can be used to print a newline:

```
#include <iostream>

int main()
{
    int x { 5 };
    std::cout << "The value of x is: " << x << '\n'; // standalone \n goes in single
quotes
    std::cout << "First line\nSecond line\n";        // \n can be embedded in double
quotes
    return 0;
}
```

This outputs:

```
The value of x is: 5
First line
Second line
```

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

```
#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. No longer relevant. You can use question marks unescaped.
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 \\\n";
    std::cout << "6F in hex is char '\x6F'\n";
    return 0;
}
```

Prints:

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

Warning

Escape sequences start with a backslash (\), not a forward slash (/). If you use a forward slash by accident, it may still compile, but will not yield the desired result.

Newline (\n) vs. std::endl

We cover this topic in lesson 1.5 -- Introduction to iostream: cout, cin, and endl.

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

Single 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).

Text between double quotes (e.g. “Hello, world!”) is treated as a string of multiple characters. We discuss strings in lesson 5.2 -- Literals.

Best practice

Put stand-alone chars in single quotes (e.g. `'t'` or `'\n'`, not `"t"` or `"\n"`). This helps the compiler optimize more effectively.

Avoid multicharacter literals

For backwards compatibility reasons, many C++ compilers support **multicharacter literals**, which are char literals that contain multiple characters (e.g. `'56'`). If supported, these have an implementation-defined value (meaning it varies depending on the compiler). Because they are not part of the C++ standard, and their value is not strictly defined, multicharacter literals should be avoided.

Best practice

Avoid multicharacter literals (e.g. `'56'`).

Multicharacter literal support sometimes causes problems for new programmers. Consider the following simple program:

```
#include <iostream>

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

int main()
{
	std::cout << add(1, 2) << '/n';

	return 0;
}
```

The programmer expects this program to print the value `3` and a newline. But instead, on the author’s machine, it outputs the following:

```
312142
```

The issue here is that the programmer accidentally used `'/n'` (a multicharacter literal consisting of a forward slash and an `'n'` character) instead of `'\n'` (the escape sequence for a newline). The program first prints `3` (the result of `add(1, 2)`) correctly. But then it prints the

value of `'/n'`, which on the author's machine had the implementation-defined value `12142`.

Warning

Make sure that your newlines are using escape sequence `'\n'` , not multicharacter literal `'/n'`.

Here's another example:

```
#include <iostream>

int main()
{
    int x { 5 };
    std::cout << "The value of x is " << x << '\n';

    return 0;
}
```

This program outputs exactly as you expect:

```
The value of x is 5
```

But this output isn't nearly exciting enough, so we decide to add an exclamation point before the newline:

```
#include <iostream>

int main()
{
    int x { 5 };
    std::cout << "The value of x is " << x << '!\n'; // added exclamation point

    return 0;
}
```

Because `'!\n'` is a multicharacter literal, this program now prints:

```
The value of x is 58458
```

That's probably not what you were expecting.

What about the other char types, `wchar_t`, `char8_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.

As an aside…

The term “deprecated” means “still supported, but no longer recommended for use, because it has been replaced by something better or is no longer considered safe”.

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 144,000 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. These char types have the same size as `std::uint_least16_t` and `std::uint_least32_t` respectively (but are distinct types). `char8_t` was added in C++20 to provide support for 8-bit Unicode (UTF-8). It is a distinct type that uses the same representation as `unsigned char`.

You won’t need to use `char8_t`, `char16_t`, or `char32_t` unless you’re planning on making your program Unicode compatible. Unicode and localization are 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 characters to display incorrectly.