4.11 — Chars

ALEX AUGUST 28, 2021

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

```
#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 `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	</td>	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, 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 a character

    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:

```
cout << 'c';
```

This produces the result:

```
c
```

A reminder

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 std::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 }; // initialize an integer with the value of ch
    std::cout << i << '\n'; // 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, bool, char, double).

Key insight

Whenever you see C++ syntax (excluding the preprocessor) that makes use of angled brackets, the thing between the angled brackets will most likely be a type. This is typically how C++ deals with concepts that need a parameterizable type.

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

```
#include <iostream>

int main()
{
    char ch{ 'a' };
    std::cout << ch << '\n';
    std::cout << static_cast<int>(ch) << '\n';
    std::cout << ch << '\n';
    return 0;
}
```

This results in:

```
a
97
a
```

It's important to note that the parameter to *static_cast* evaluates as an expression. 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 to a new type. In the above case, variable *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 a large integer 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 (4.5 -- Explicit type conversion (casting) and static_cast).

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) <<
'\n';

    return 0;
}
```

Here's the output from one run:

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

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 << ch << " has ASCII code " << static_cast<int>(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 << ch << " has ASCII code " << static_cast<int>(ch) << '\n';

    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 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 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 embed a newline in a string of text:

```
#include <iostream>

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

This outputs:

```
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	\</td>	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'
```

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?

Stand-alone 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:

```
char ch{'56'}; // a char can only hold one
symbol
```

Text put between double quotes (e.g. "Hello, world!") is called a string. 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:

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

We'll discuss strings in the next lesson (4.12 -- An introduction to std::string).

Rule

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

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.

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 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. char8_t has been added in C++20.

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.

Next lesson

4.12 An introduction to std::string

Back to table of contents

Previous lesson

4.10 Introduction to if statements

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest