4.11 — Chars

BY ALEX ON JUNE 9TH, 2007 | LAST MODIFIED BY ALEX ON DECEMBER 19TH, 2019

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? The char data type was designed for such a purpose.

The char data type is an integral type, meaning the underlying value is stored as an integer, and it's guaranteed to be 1-byte in size. However, similar to how a boolean value is interpreted as true or false, a char value is 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 example, ASCII code 97 is interpreted as the character 'a'.

Character literals 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, 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


    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); // 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, 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 (**6.16 -- 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 quo

    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	\\	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:

```
1 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:

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

However, strings are not fundamental data types in C++, and are a little more complex, so we'll reserve discussion of those until we cover compound types.

Best practice

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.


4.12 -- Literals


Index


4.10 -- Introduction to if statements

C++ TUTORIAL | PRINT THIS POST

223 comments to 4.11 — Chars

« Older Comments 1 2 3

Air Paul
January 27, 2020 at 9:39 am · Reply

What is the windows API?

nascardriver
January 28, 2020 at 2:07 am · Reply

It's a set of functions and types that you can use to control Windows (the OS), eg. moving windows around or playing sounds.

Kermit
January 19, 2020 at 3:44 am · Reply

does this counts as useful program ? xD

```
1 #include <iostream>
2
3 using namespace std;
```

```
4
5   bool isGay(string input)
6   {
7       if (input == "Rudy")
8       {
9           return true;
10      }
11      else
12      {
13          return false;
14      }
15      return 0;
16  }
17  /*char userInput()
18  {
19      cout << "Enter Rudy if he is gay: ";
20      char x{};
21      cin >> x;
22      return x;
23  }*/
24  int main()
25  {
26      //char input{ userInput() };
27      cout << "Enter Rudy if he is gay: ";
28      string input{};
29      cin >> input;
30      if (isGay(input))
31      {
32          cout << "Yup he is";
33      }
34      else
35      {
36          cout << "Still he is";
37      }
38      return 0;
39  }
```

also had a question the type

```
char
```

is used to just store a single character? as u look in the code i wrote a function

```
userInput()
```

of type char and when i tested, it was just printing the first letter of that word.
for eg.
<pre>
Enter Sentance: Hello
You Entered: H
</pre>

nascardriver
January 19, 2020 at 4:47 am · Reply

`isGay` can be reduced to

```
return (input == "Rudy");
```

A `char` is a single character. If you want text, use a `std::string`

Kermit
January 19, 2020 at 4:56 am · Reply

is there any difference in these 2 codes.

```
1 return (input == "Rudy");
```

```
1 return input == "Rudy";
```

seems no difference works alright without the Parenthesis and with the Parenthesis.

nascardriver
January 19, 2020 at 5:05 am · Reply

They're the same. I like parentheses. The more parentheses, the easier the code is to read for someone who doesn't know the order of evaluation.

giang
January 9, 2020 at 1:40 am · Reply

Thanks very much for the lessons. But I'm still stuck with some questions:
1.If a char is 1-byte-size. Then why ASCII code only has 128 characters? Shouldn't it have 256 characters??
2.What is a signed or unsigned char? Does it mean that the char's ASCII code is signed or unsigned?
3.What is a signed char??

nascardriver
January 9, 2020 at 4:22 am · Reply

A `char` is just a number. It's often represented as a character, but that's just a representation, it's still a mere integer that can be signed or unsigned like all other integers.
There's no type that's smaller than a byte. If you want to store 3 values, you'll have to use at least 1 byte, even if that means wasting 6 bits (You can combine multiple values into a single byte, let's not get into that).
ASCII only uses 7 bits, but the 8th bit can be used for extended ASCII, eg. the letters with dots and slashes on top.

Alex
January 9, 2020 at 1:22 pm · Reply

I've always assumed ASCII used 127 characters because the range 0-127 can be held by char regardless of whether it defaults to signed or unsigned.
Signed and unsigned chars are just 8-bit signed and unsigned integers. See the lesson on those topics.

TheDoctor
December 28, 2019 at 3:25 am · Reply

This works perfectly, without any warning-

```
1 char ch{'A'+'4'};
```

But this does not, or at least gives a warning on some other compilers/ IDE's-

```
1 char ch1{'A'};
```

```
char ch2{'4'};
char ch3{ch1+ch2};
```

Anyone have any idea why?

nascardriver
December 28, 2019 at 5:00 am · Reply

Both

```
'A' + '4'
// and
ch1 + ch2
```

produce an `int` as a result (Because the built-in arithmetic operators don't accept anything smaller than an `int`).
Since `A` and `4` are constant expressions, the result is a constant expression too.
`ch1` and `ch2` are not constant expressions, so the result is also not a constant expression.

Brace initialization disallows narrowing conversions, unless the type we're trying to convert is a constant expression and the value fits into the new type.
`'A' + '4'` is a constant expression and its result ('u') fits into a `char`, so the conversion is legal. If you used larger chars, eg. `'A' + 'A'`, you'd get an error.

If we make `ch1` and `ch2` constant expressions, we can make the code work

```
// constexpr is covered later in this chapter
constexpr char ch1{'A'};
constexpr char ch2{'4'};
char ch3{ch1 + ch2}; // Ok
```

TheDoctor
December 29, 2019 at 2:31 am · Reply

So basically since brace initialization can't know what would be the value of the variables (ch1, ch2) it can't determine if the resultant int will safely be cast or overflow the char, right?

nascardriver
December 29, 2019 at 3:00 am · Reply

I never thought of it like that, but it makes sense!

TheDoctor
December 29, 2019 at 3:49 am · Reply

Thanks a ton!
You and Alex are the heroes we don't even deserve.

giang
January 9, 2020 at 1:45 am · Reply

'A' + '4'` is a constant expression and its result ('u')-> Can you explain why??

Alex
January 9, 2020 at 1:33 pm · Reply

'A' is a constant and '4' is a constant. The compiler is capable of adding together constants to produce a constant result.
'A' = ascii code 65, '4' = ascii code 52. Therefore, 'A'+'4' = 65 + 52 = 117, which is the ascii code for 'u'

salah
February 11, 2020 at 12:38 am · Reply

Hi Alex, Hi nascardriver ,

Why then is it legal to do that without any warring :

int a {1} ;
int b {2} ;
int c {a + b};

here a and b are not const variable, and c is Ok .
why does that give an error with char data type ?

HolzsotckG
December 23, 2019 at 1:53 am · Reply

Is buffor of std::cin its variable?

nascardriver
December 23, 2019 at 7:47 am · Reply

Types can have sub-types. The buffer is a part of std::cin. This is covered later.

koe
December 17, 2019 at 7:44 pm · Reply

"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."

Hi, wording here is a bit hard to unravel since in the previous example a char was cast into an integer. Maybe say instead:

"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."

UPD: question
How do you empty the cin buffer? For example this code, if you input "abc" the result is a b 0.

```
1 char a{};
2 std::cin >> a;
3 std::cout << a << '\n';
4 std::cin >> a;
5 std::cout << a << '\n';
6 int y{3};
7 std::cin >> y;
8 std::cout << y << '\n';
```

UPD2: another question
I can print "A question?" just fine. "A question\?" works too. What scenarios are the escape sequences useful for?

nascardriver
December 18, 2019 at 4:42 am · Reply

Lesson updated to make it clearer which cast is performed.
Extraction failure and handling is covered in lesson C.5.10.

```
int i{};
std::cin >> i;

if (std::cin.fail()) // Check if extraction failed
{
  std::cin.clear(); // Tell std::cin that we solved the problem.
  // Remove all input from the current line
  std::cin.clear(std::numeric_limits<std::streamsize>::max(), '\n');
}

// Try extracting again.
```

\? was used to prevent the compiler from interpreting a string as a trigraph (Special character sequences to be used instead of a character if you have a weird keyboard) in certain situations. Trigraphs have been removed from C++, \? shouldn't be useful for anything anymore.

koe
December 18, 2019 at 3:26 pm · Reply

Could you update the lesson indicating which escape sequences still have utility? Or maybe say instead of 'notable', 'still useful in the modern world' for the three listed.

nascardriver
December 19, 2019 at 1:57 am · Reply

I'm not sure about form feeds and octal, apart from \? everything is still used. I added a note to the \? row in the table.

Ganesh Dhawale
October 23, 2019 at 9:50 am · Reply

Hi

Ganesh's Biggest Fan
October 24, 2019 at 12:22 pm · Reply

Hi!

Anastasia
September 10, 2019 at 4:22 am · Reply

Hi,
what should be used to represent characters larger than ASCII? I tried `wchar_t`, but clang says

that character is too large for the type, even though its (wchar_t) size is 4 bytes on my system...
I've re-read this lesson and it says not to use wchar_t, so I tested it with char32_t as well - same result :(

```
#include <iostream>

int main()
{
    wchar_t f { 'ϕ' }; // error: character too large for enclosing character literal type

    wchar_t u_umlaut { 'ü' }; // same error

    std::cout << sizeof(wchar_t) << '\n'; // 4

    char32_t l { 'л' }; // same error

    char32_t o_umlaut { 'ö' }; // same error

    return 0;
}
```

I guess c-style strings won't work either? Strangely enough std::string works perfectly with unicode, is it implementation specific?

nascardriver
September 10, 2019 at 4:33 am · Reply

The variable's type is irrelevant to your error. The problem are your characters.

```
'*' // No matter how hard you try, this is 8 bits wide.
```

You need to prefix your character literal, in your case with 'U'

```
char32_t ch{ U'л' }; // U means UTF-8
```

See cppreference for a complete list of prefixes
https://en.cppreference.com/w/cpp/language/character_literal

Anastasia
September 10, 2019 at 4:44 am · Reply

It really works, thanks! And the link is very useful too. I wish some of it was mentioned in the lesson...
How about std::string? Does it normally support all the unicode characters?

nascardriver
September 10, 2019 at 4:55 am · Reply

The same prefixes (I think) apply to strings (U"Hello"). `std::string` can store them, but its access functions will yield wrong results, eg. `std::string::length` returns the number of bytes, not characters. There's `std::u32string` and more (https://en.cppreference.com/w/cpp/string/basic_string), which use types other than `char` to store the string. I haven't used them, let me know how it goes :)

Anastasia
September 10, 2019 at 5:34 am · Reply

I did some messy testing... It seems that 'U' prefix is not necessary for normal std::strings and c-style strings, but unicode std::strings (e.g. std::u32string) need it. And for some reason std::cout can't output std::u32string, maybe it needs a some sort of unicode flag as well?

```
#include <iostream>
#include <string>

int main()
{
    std::string hello { "Привет!" }; // perfectly works

    std::cout << hello << " - " << hello.size() <<
            '\n'; // size 13. It counts each cyrillic char as 2 bytes + '

    std::string Uhello { U"Привет!" }; // error: no matching constructor

    const char* c_style_hello { "Привет!" }; // no error here
    std::cout << c_style_hello << '\n'; // correct output

    std::u32string another_hello { U"Привет!" }; // this one seems to rec

    std::cout << another_hello << " - " << another_hello.size() << '\n';
    /** cout doesn't know how to treat it ? :/
      *
      * error: invalid operands to binary expression ('std::ostream'
      * (aka 'basic_ostream<char>') and 'std::u32string' (aka
      * 'basic_string<char32_t>'))
      */

    return 0;
}
```

> `std::string::length` returns the number of bytes
it's the same thing with ascii chars (num bytes == num chars), but I think it should still display correctly number of bytes in other encodings and it should be possible to count how many characters it has knowing how many bytes every char takes.

nascardriver
September 10, 2019 at 5:45 am · Reply

> knowing how many bytes every char takes
UTF-8 doesn't have a fixed character width. Characters can be 1-4 bytes wide. If you use a fixed-width encoding, you can divide the number of bytes to get the length.

I suppose `std::cout<<` can only print 0-terminated strings. Whether or not the strings are displayed correctly depends on your terminal, this is nothing you can do something about. You can overload `operator<<` for the other string classes and convert the string to something your terminal understands. There are conversion functions in the standard library, I haven't used them.

Anastasia
September 10, 2019 at 6:21 am · Reply

This all seems much more complicated than I thought :/ I see now why Alex tried to avoid this subject in the lesson...

Thank you for the clarifications, it was very helpful, as usual.

learning
June 19, 2019 at 10:08 am · Reply

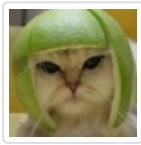
hey, what is the meaning of the '\h'?

```
1    std::cout << ch << " has ASCII code " << static_cast<int>(ch) << '\h';
```

nascardriver
June 19, 2019 at 10:15 am · Reply

Hi!

It's supposed to be '\n'.
@Alex



Alex
June 19, 2019 at 2:41 pm · Reply

'\h' gets no love these days.

Fixed!

Samira Ferdi
June 11, 2019 at 7:12 pm · Reply

Hi Alex and Nascardriver!

I've found that the addition of char type is like the addition of it's ASCII code.
For example '5' + '5' means 53 + 53 and prints in the console 106. Is this a standard or just compiler-specific behavior?

nascardriver
June 12, 2019 at 3:16 am · Reply

It's standard. `char` is an integral type and can be used in arithmetic (With care to integral promotion). The only thing special about `char` is that in addition to it's integral representation, it has a character representation.

BP
June 4, 2019 at 2:25 am · Reply

Hello!

A quick question:
How does the \a work?
Because I can't seem to get an alert of any kind...

Thanks,

Edit: Or \f, tried it in std::cout and I got a symbol. I'm guessing that we will learn more about \f when we learn about logical pages?

nascardriver
June 4, 2019 at 8:19 am · Reply

They're both characters from the past which were used for typewriters and other old machinery.
Whether or not they do anything now is up to your terminal (console).

BP
June 4, 2019 at 8:24 am · Reply

Thanks!

alfonso
June 3, 2019 at 1:34 am · Reply

Unexpected thing:

```
#include <iostream>

int main () {
    // Some random comment ended with a backslash comments out
    // the next line and I do not know for sure why
    // It looks like the last backslash escapes the ENTER \
    return 0; // return is commented here
}
```

error: multi-line comment [-Werror=comment]

Alex
June 11, 2019 at 3:00 pm · Reply

Yes, \ at the end of a line is a line-continuation character, meaning the next line is treated as an extension of the previous line.

alfonso
June 3, 2019 at 1:23 am · Reply

```
#include <iostream>

int main()
{
    std::cout << "6F in hex is char \'\x6F\'\n";
    return 0;
}
```

Why to escape ' (single quote) there? More generally, why to escape a single quote? Ok, I can see one use case for escaping single quote

```
char ch {'\''};
```

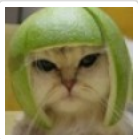
Alex
June 11, 2019 at 2:53 pm · Reply

I've removed the extraneous backslashes in the quoted string. The primary use of \' is as you suggest in your latter example.

Louis Cloete
April 28, 2019 at 11:50 am · Reply

Hi @Alex!

The program example just above "Char size, range, and default sign" still uses std::endl for line breaks.



Alex
May 2, 2019 at 5:02 pm · Reply

Fixed. Thanks!

Somoshree Datta
March 29, 2019 at 7:04 am · Reply

Hi Alex, your tutorials are really awesome! I have this one doubt:
In the previous tutorials, you had said that std::cout flushes the output, i.e. it makes sure that the output shows up on the screen immediately and so the use of std::endl isnt preferred; rather we prefer '\n' instead.
But here in this tutorial, u have written that "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."
So isnt this statement contradicting with the statement that u made earlier?
Eagerly waiting for your response..

nascardriver
March 29, 2019 at 7:12 am · Reply

@std::endl flushes the buffer, not @std::cout.

Somoshree Datta
March 29, 2019 at 7:31 am · Reply

Thanks nascardriver for the reply :)
In the tutorial named Introduction to iostream: cout,cin and endl, there under the heading of std::endl vs '\n', it is mentioned that : "Using std::endl can be a bit inefficient, as it actually does two jobs: it moves the cursor to the next line, and it “flushes” the output (makes sure that it shows up on the screen immediately). When writing text to the console using std::cout, std::cout usually flushes output anyway (and if it doesn’t, it usually doesn’t matter), so having std::endl flush is rarely important."
So isnt this a contradiction?

nascardriver
March 29, 2019 at 7:52 am · Reply

No.

```
std::cout << "Hello World\n";
// The text may or may not have been printed at this point.
// "std::cout usually flushes output".

std::cout << "Hello World" << std::endl;
// Both texts have been printed, because @std::endl flushed the buffer.
```

Even if you don't use @std::endl, the buffer will be flushed at some point. Most likely you won't notice a difference between using @std::endl and '\n'.
@std::endl is bad when you print multiple lines.

```
std::cout << "Hello Somoshree" << std::endl;
std::cout << "How are you today?" << std::endl;
std::cout << "Do you like cats?" << std::endl;
std::cout << "I like cats." << std::endl;
```

The buffer will be flushed after every line, that's slow. It's enough to flush the buffer once after the last line or let @std::cout take care of it when it wants to.

```
std::cout << "Hello Somoshree\n";
std::cout << "How are you today?\n";
std::cout << "Do you like cats?\n";
std::cout << "I like cats.\n";
```

All 4 lines can remain in the buffer and then be printed all at once. Or one after the other, whatever @std::cout likes.

Somoshree Datta
March 29, 2019 at 7:59 am · Reply

ok..got it now :) So that means that std::cout may or may not flush the input buffer, but if used with std::endl, input buffer is definitely flushed because of endl, am I right?

nascardriver
March 29, 2019 at 8:00 am · Reply

Right

Gejsi
February 25, 2019 at 4:17 pm · Reply

Hi! What is the purpose of char variable? Just so we are able to input and output special literals other than integers?

(Besides memory usage, sunce other data types take more space)

Alex
February 26, 2019 at 7:59 pm · Reply

Char variables exist to hold and manipulate individual (ASCII) characters.

Hassan
December 19, 2018 at 4:48 am · Reply

Hi everyone,
here is my simple program to invert the case of an alphabet input by the user.
please correct me if something is wrong including comments-style(good/bad).

```
1  #include <iostream>
2  int main()
3  {
4    std::cout << "Enter an  alphabet: ";
5    signed char ch{0};
6    std::cin >> ch;
7    if ('A' <= ch && ch <= 'Z')
8       ch = ch + 32; // ASCII difference b/w upper-lower case chars.
9    else
10      ch = ch - 32;
11
12    std::cout << "The inverted case is: " << ch << "\n";
13
14   return 0;
15 }
```

nascardriver
December 19, 2018 at 5:17 am · Reply

* Chars are signed by default. There's no need to write it out.
* Line 7: You're only checking in ch <= 'Z'. Give it another look.
* Magic number: 32. Declare a constant or replace it with 'a' - 'A'.
* Line 12: I'm guessing it'd be more efficient to pass \n as a char, not a string.

> including comments-style
I don't like comments that are on the same line as code. This is personal preference.

Hassan
December 20, 2018 at 2:48 am · Reply

correction noted.
thanks.

kaworu
December 17, 2018 at 9:26 pm · Reply

char c = '\78';
cout << "c="<<c;
why c=8?

nascardriver
December 18, 2018 at 7:57 am · Reply

\ means escape what comes next. If the next character is a digit, it's interpreted as octal. \7 is an octal 7, which is the bell character (http://www.asciitable.com/). That char is done. You added an extra 8. This 8 is just the character 8, it's not interpreted as a number. You now have to characters in single quotation marks ('\7' and '8'). Behavior is implementation defined. Your compilers appears to ignore the '\7' and just store '8' in @c. Make sure you followed lesson 0.11. Your compiler should have warned you about this.

Hassan
December 16, 2018 at 9:55 am · Reply

hi everyone,
suppose i am working on a project where memory management is of paramount importance and i am only gonna need integers in the range 0-255, so i chose the type "char".
Is there a way to treat and manipulates(arithmetically) them as numbers without converting it to short or larger type?

nascardriver
December 17, 2018 at 3:50 am · Reply

Hi Hassan!
Arithmetic operators may promote their operands to int. You can get around this by writing a wrapper class that handles the casts for you. You should understand this after you finished chapter 10. I can't think of a solution that you could use with what you learned so far.

```
class CSmallInteger
{
private:
  using tType = std::int_least8_t;

private:
  tType m_i{ 0 };

public:
  operator tType(void)
  {
    return this->m_i;
  }

  CSmallInteger operator+(CSmallInteger that)
  {
    return { static_cast<tType>(this->m_i + that.m_i) };
  }

  CSmallInteger operator+(tType that)
  {
    return { static_cast<tType>(this->m_i + that) };
  }

  // ... more operators ...

  CSmallInteger(tType i)
      : m_i{ i }
  {
  }
};

int main(void)
{
  CSmallInteger a{ 12 };
  CSmallInteger b{ 45 };
  CSmallInteger c{ a + b };

  return 0;
}
```

Hassan
December 19, 2018 at 2:36 am · Reply

thank you.

Rai
November 10, 2018 at 10:08 am · Reply

So I tried making a char program with if and else statement. But I have a few questions on improving it.

```
// Char.cpp : This file contains the 'main' function. Program execution begins and ends the
//

#include <iostream>

char getCharacter()
{
    std::cout << "Enter a keyboard character: " << "\n";
    char ch;
    std::cin >> ch;
    return ch;
}

void isValid(char x, char y)
{
    if (static_cast<int>(x, y) > 31)
    {
        if (static_cast<int>(x, y) < 128)
            std::cout << x << " has an ASCII character of: " << static_cast<int>(x) << "\n"
        std::cout << y << " has an ASCII character of: " << static_cast<int>(y);
    }
    else
        std::cout << "Char overflowed. Exiting.";
}

int main()
{
    char x = getCharacter();
    char y = getCharacter();
    isValid(x, y);
    return 0;
}
```

So I got the if else statement working. If I put a character that isn't in the ASCII table e.g "ø", "Ü" it'll say "Char overflowed.Exiting."

However I'm having 2 issues!

1. How could I take as many characters from the user and output all the character's ascii code. For example the user types in "apple" and it'll output all ascii code for each letter. Or will this be covered in another lesson?

2. If I don't type in 2 characters for std::cout << "Enter a keyboard character: "
, it will ask the question again! How do I make it so it asks once and will print out as many characters and only ask the question once.

It will be highly appreciated if anyone can help me out on this. I'm currently looking at other forums to see if I can find a solution.

nascardriver
November 10, 2018 at 10:32 am · Reply

Hi Rai!

1. You'll need a loop. Loops are covered later.
2. You could add a bool parameter to @getCharacter and only print the message if the argument is true.

Notes:
* Initialize your variables with uniform initialization
* Your static_cast's don't do what you want them to do. They only check @y. "x, y" is using the comma operator, which evaluates to its last argument, in this case @y. @x is unchecked. You need to check @x and @y separately.
* It's bad to let your program exit without printing a trailing line feed, because whatever is printed after your program finished will be in the same line.

R310
July 3, 2018 at 6:26 am · Reply

Hello!
Why do I get negative 49 for \pi?

[code]
Enter a character: π
 has ASCII code -49

Process finished with exit code 0
[\code]

nascardriver
July 4, 2018 at 8:45 am · Reply

Hi R310!

π isn't part of the standard ASCII-set and cannot be stored in a single char.

MorbidPitaya
June 30, 2018 at 10:50 am · Reply

Hey!)

In which cases (except when space is extremely valuable) is using the unsigned char more useful than, or expected instead of, the other integer data types? Is the use of the unsigned char not dangerous?

Concerning the direct output of char literals, does the method work with the newest Visual Studio? When compiling, I get a mistake that states <identifier "cout" is undefined>, even though I copied the code line in the tutorial above.

Thanks in advance!

nascardriver
July 1, 2018 at 6:18 am · Reply

Hi MorbidPitaya!

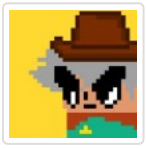
> In which cases (except when space is extremely valuable) is using the unsigned char more useful than, or expected instead of, the other integer data types?
Bytes, because they range from 0x00 to 0xFF. But they can also be implemented as signed chars.

> identifier "cout" is undefined
You either forgot to include <iostream> or you forgot "std::". If that's not it, please share you code.

MorbidPitaya
July 1, 2018 at 12:30 pm · Reply

nascardriver,

Thanks for the info! Also, apparently the issue was in the missing "std::"!)



Larry
June 10, 2018 at 7:58 am · Reply

If anyone here's attempting to read every character that the person puts inside the "cin", you can use a while true loop to go forever, or use a custom variable to see when cin's buffer is empty.

MoAl
May 10, 2018 at 1:10 pm · Reply

Hi.. I was playing around with the code and I am not understanding the output of the this code:

```
1  int main()
2  {
3      char ch('97');
4      std::cout <<ch<< std::endl;
5      std::cout << static_cast <int>(ch)<<std::endl;
6      std::cout << ch << std::endl;
7      return 0;
8  }
```

I am getting:
7
55
7
I understand that 55 is the ASCII code for 7, but where is the 7 come from?
Thanks..

Alex
May 10, 2018 at 10:45 pm · Reply

The result of this code:

```
1  char ch('97');
```

is implementation defined. C++ expects char literals to be one character, but in your case '97' is two. It looks like your compiler is simply discarding the '9' and treating this as if you'd typed '7'.

cppLearner
April 1, 2018 at 11:02 pm · Reply

Consider the following code:

```
int16_t a = 3; //case 1
char b='3'; //case 2
```

Is the following statement true:
In case1 3 is stored as 0000 0000 0000 0011
and in case2 3 is converted to its ascii value and then that ascii value is stored in binary form, I mean '3' --> ascii value 51 --> stored as 0000 0000 0011 0011.
Thanks in advance.

nascardriver
April 2, 2018 at 6:36 am · Reply

Hi cppLearner!

You're right.

seb
March 22, 2018 at 1:14 pm · Reply

"If we want to output a char as a number instead of a character"

"If you’re using a char to hold small integers"

Is there a reason that someone would use char over a true integer type like int or short?

nascardriver
March 23, 2018 at 10:55 am · Reply

Hi seb!

Memory usage, eg. when dealing with bytes.

Gabe
March 6, 2018 at 4:14 pm · Reply

If I make the user enter true or false could I change that value into 1 or 0 to make it work with a boolean and if statement?

nascardriver
March 7, 2018 at 3:15 am · Reply

Hi Gabe!

You can use the std::boolalpha flag to tell std::iostream to use true/false instead of 1/0.

```
bool b{ false };

// This allows the user to enter true of false (case sensitive).
// Note: 0 and 1 are both considered false this way.
std::cin >> std::boolalpha >> b;

if (b)
{
  // code
}
```

« Older Comments 1 2 3