6.8b — C-style string symbolic constants

BY ALEX ON AUGUST 15TH, 2015 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

C-style string symbolic constants

In the lesson **6.6 -- C-style strings**, we discussed how you could create and initialize a C-style string, like this:

```
#include <iostream>

int main()
{
    char myName[]{ "Alex" }; // fixed array
    std::cout << myName << '\n';

    return 0;
}
```

C++ also supports a way to create C-style string symbolic constants using pointers:

```
#include <iostream>

int main()
{
    const char *myName{ "Alex" }; // pointer to symbolic constant
    std::cout << myName << '\n';

    return 0;
}
```

While these above two programs operate and produce the same results, C++ deals with the memory allocation for these slightly differently.

In the fixed array case, the program allocates memory for a fixed array of length 5, and initializes that memory with the string “Alex\0”. Because memory has been specifically allocated for the array, you’re free to alter the contents of the array. The array itself is treated as a normal local variable, so when the array goes out of scope, the memory used by the array is freed up for other uses.

In the symbolic constant case, how the compiler handles this is implementation defined. What *usually* happens is that the compiler places the string “Alex\0” into read-only memory somewhere, and then sets the pointer to point to it. Because this memory may be read-only, best practice is to make sure the string is const.

For optimization purposes, multiple string literals may be consolidated into a single value. For example:

```
const char *name1{ "Alex" };
const char *name2{ "Alex" };
```

These are two different string literals with the same value. The compiler may opt to combine these into a single shared string literal, with both name1 and name2 pointed at the same address. Thus, if name1 was not const, making a change to name1 could also impact name2 (which might not be expected).

As a result of string literals being stored in a fixed location in memory, string literals have static duration rather than automatic duration (that is, they die at the end of the program, not the end of the block in which they are defined). That means that when we use string literals, we don’t have to worry about scoping issues. Thus, the following is okay:

```
const char* getName()
{
    return "Alex";
```

```
}
```

In the above code, `getName()` will return a pointer to C-style string "Alex". If this function were returning any other kind of literal by address, the literal would be destroyed at the end of `getName()`, and we'd return a dangling pointer back to the caller. However, because string literals have static duration, "Alex" will not be destroyed when `getName()` terminates, so the caller can still successfully access it.

C-style strings are used in a lot of old or low-level code, because they have a very small memory footprint. Modern code should favor the use `std::string` and `std::string_view`, as those provide safe and easy access to the string.

std::cout and char pointers

At this point, you may have noticed something interesting about the way `std::cout` handles pointers of different types.

Consider the following example:

```
#include <iostream>

int main()
{
    int nArray[5]{ 9, 7, 5, 3, 1 };
    char cArray[]{ "Hello!" };
    const char *name{ "Alex" };

    std::cout << nArray << '\n'; // nArray will decay to type int*
    std::cout << cArray << '\n'; // cArray will decay to type char*
    std::cout << name << '\n'; // name is already type char*

    return 0;
}
```

On the author's machine, this printed:

```
003AF738
Hello!
Alex
```

Why did the int array print an address, but the character arrays printed strings?

The answer is that `std::cout` makes some assumptions about your intent. If you pass it a non-char pointer, it will simply print the contents of that pointer (the address that the pointer is holding). However, if you pass it an object of type `char*` or `const char*`, it will assume you're intending to print a string. Consequently, instead of printing the pointer's value, it will print the string being pointed to instead!

While this is great 99% of the time, it can lead to unexpected results. Consider the following case:

```
#include <iostream>

int main()
{
    char c{ 'Q' };
    std::cout << &c;

    return 0;
}
```

In this case, the programmer is intending to print the address of variable c. However, &c has type char*, so std::cout tries to print this as a string! On the author's machine, this printed:

Q╠╠╠╩╡4;¿■A

Why did it do this? Well, it assumed &c (which has type char*) was a string. So it printed the 'Q', and then kept going. Next in memory was a bunch of garbage. Eventually, it ran into some memory holding a 0 value, which it interpreted as a null terminator, so it stopped. What you see may be different depending on what's in memory after variable c.

This case is somewhat unlikely to occur in real-life (as you're not likely to actually want to print memory addresses), but it is illustrative of how things work under the hood, and how programs can inadvertently go off the rails.

6.9 -- Dynamic memory allocation with new and delete

Index

6.8a -- Pointer arithmetic and array indexing

C++ TUTORIAL | PRINT THIS POST

155 comments to 6.8b — C-style string symbolic constants

« Older Comments 1 2

phjiang
February 2, 2020 at 11:41 pm · Reply

[code]
#include <iostream>

char c{ 'Q' };

```
int main()
{
    std::cout << &c;

    return 0;
}
/[code]
```

Move the declaration out of the main(). Run the program will print Q. Why?

nascardriver
February 4, 2020 at 8:27 am · Reply

It's undefined behavior what happens after 'Q' is printed. Anything can happen.

ErwanDL
November 7, 2019 at 5:38 am · Reply

Hey Alex and Nascar,

I am a bit confused about this line you wrote :
"Rule: Feel free to use C-style string symbolic constants if you need read-only strings in your program, but always make them const!"

I think you said in a previous lesson that C-style strings should be avoided altogether : are C-style string symbolic constants an exception to this piece of advice ? Or can they be replaced by a cleaner, more modern C++ alternative involving std::string (something like

```
1 const std::string
```

?

Cheers

nascardriver
November 7, 2019 at 6:01 am · Reply

Hi again,

the lesson about `std::string_view` was added about a week ago. No lessons after it have been updated yet, so you'll still see some old code and rules.

`char*` and `const char*` are used for buffers and in code that doesn't have access to `std::string_view` or is limited on resources. For strings, use `std::string` and `std::string_view`.

The problem with `const std::string` is that it creates a copy of the string and that it's not compile-time constant. `std::string_view` is fast and can be used at compile-time (There's no lesson about compile-time programming yet, look into `constexpr` functions when you're done with learncpp).

Thanks again for your feedback! I'll think about moving some of the last 6 lessons around so that they make more sense and can show an example of modifying a `std::string_view`'s observed string.

Samira Ferdi
August 12, 2019 at 5:52 pm · Reply

Hi, Alex and Nascardriver!

What do you think about my code?

```
#include <iostream>
#include <cstring>

void printMessage(const char *message)
{
    std::cout << message << '\n';
    std::cout << strlen(message) << '\n';
}

int main()
{
    printMessage("Hello world");
    return 0;
}
```

nascardriver
August 12, 2019 at 11:46 pm · Reply

Line 7 should use `std::strlen`.

Samira Ferdi
August 13, 2019 at 4:48 pm · Reply

Thank you, Nascardriver! But, should I prefer std::string instead of C-Style string symbolic constant? If I should, it means I have to pass std::string by reference instead of pass by value?

nascardriver
August 14, 2019 at 2:36 am · Reply

Yes.
Only use C-style strings if you need to worry about performance or memory usage.
Otherwise, use `std::string` and `std::string_view`.

CHERIS
July 11, 2019 at 10:46 pm · Reply

#include <iostream>
int main()
{
  char c = 'Q';
  std::cout << &c;

  return 0;
}

Output:

cheris@cheris:~$ cd /home/cheris/Desktop/C++
cheris@cheris:~/Desktop/C++$ g++ test.cpp
cheris@cheris:~/Desktop/C++$ ./a.out
Qcheris@cheris:~/Desktop/C++$

my question is why it print only Q??

nascardriver
July 11, 2019 at 11:36 pm · Reply

You're not allowed to access the memory after `c`. Doing so anyway causes undefined behavior. (`std::cout <<` keeps on reading until it finds a 0-byte).

Darshan
May 2, 2019 at 11:05 pm · Reply

Hi alex,

```
#include <iostream>

int main()
{
    char cArray[] = "Hello!";
    const char *name = "Alex";
    char z[] = {'H','e','l','l','o'};

    std::cout<<cArray<<'\n';
    std::cout<<name<<'\n';
    std::cout<<z<<'\n';

    return 0;
}
```

Why is this code giving output as:
Hello!
Alex
HelloHello!

why does outputting the variable 'z' prints HelloHello!
?

nascardriver
May 3, 2019 at 2:29 am · Reply

@z is not 0-terminated. @std::cout.operator<< prints until it finds a 0-terminator. Once get get to the end of @z, behavior is undefined, because you're accessing memory that you're not supposed to access through @z.

```
char z[]{'H','e','l','l','o', '\0'};
//                        ^^^^^^
```

Darshan
May 3, 2019 at 2:31 am · Reply

Oh yeah. Thanks a lot @nascardriver

Dimitri
March 7, 2019 at 6:35 am · Reply

```
#include <iostream>
```

```
2
3  int main()
4  {
5      char c = 'Q';
6      std::cout << &c;
7
8      return 0;
9  }
```

Q╠╠╠╩╡4;¿■A

"What you see may be different depending on what's in memory after variable c."

Maybe a dumb, not the topic, question but this does not give me rest. Why do these simbols ╠╠╠ always come out in console, no matter how many times you run the program. Not the first time I notice them. I understand that this is garbage memory, but why it does not change.

Alex
March 8, 2019 at 4:31 pm · Reply

It's possible your program is or was using that memory location for something else, and that's the garbage that happens to be consistently left.

hassan magaji
January 24, 2019 at 10:18 pm · Reply

Hi everyone,
can constant references be used here instead pointers to constant?

```
1  const char &strRef = "my name";
2  // instead of
3  const char *name = "my name";
```

nascardriver
January 25, 2019 at 6:28 am · Reply

Hi Hassan!

No. String literals are char arrays, they can decay into pointers, but not references.

Piyush
January 16, 2019 at 6:50 am · Reply

```
1  #include <iostream>
2  int main()
3  {
4      char s1[4] = "cat";
5      char s2[4] = "dog";
6      char *p1 = s1;
7      char *p2 = s2;
8
9      // the following is a string copy operation
10     while (*p1++ = *p2++)
11         ;
12
13     // s2 was copied into s1, so now they are both equal to "dog"
```

```
14      printf_s("%s %s", s1, s2);
15  }
```

How string copying works in this program ??

Source: microsoft docs.
Useful for others: Click on my name to see the whole article posted at microsoft docs.

Thanks a lot.

nascardriver
January 16, 2019 at 7:08 am · Reply

*p++ returns a reference to the character @p is pointing to and increases @p by 1.
*p1++ = *p2++ returns the value that *p2++ returned.
C-style strings a 0-terminated. Once the string is done, *p2++ returns 0. The loop stops.

Piyush
January 16, 2019 at 7:18 am · Reply

Thanks a lot. It means that the condition remains true while one or both of the string is not terminated and as soon as it terminates the condition becomes false and the loop exits. :)

Please also reply to my previous comment. I think that's more important for me to understand.

Pointers are giving me headaches. That's tough and there are many possibilities with pointers but i love to play with it. Thanks for the quick reply.

nascardriver
January 16, 2019 at 7:23 am · Reply

> the condition remains true while one or both of the string is not terminated
The value or length of @p1 is ignored. The code produces undefined behavior if @p2 is longer than @p1

> as soon as it terminates the condition becomes false
The other way around. As soon as the condition becomes false, it terminates.

> Pointers are giving me headaches
That's normal. Once you understood them, you won't know what took you so long.

Piyush
January 15, 2019 at 2:23 am · Reply

```
1   #include<iostream>
2   using std::cout;
3   using std::endl;
4   int main(){
5       const char *name{"alex"};
6       const char *assign{name};
7       char *cat{"dolly"};
8       name=cat;
9       cout<<assign;
10      return 0;
```

```
11 }
```

output :
alex

I thought assign will print dolly as assign points to name and name points to cat and the value at the address pointed by cat is "dolly". Please expain it

Thanks a lot.

nascardriver
January 16, 2019 at 7:27 am · Reply

Let's add imaginary addresses and see what happens

```
1  int main()
2  {
3    const char *name{ "alex" }; // "alex" is at 0x1000. @name is 0x1000. @name points t
4    const char *assign{ name }; // @name is 0x1000. @assign is 0x1000
5    char *cat{ "dolly" }; // "dolly" is at 0x2000. @cat is 0x2000. @cat points to "doll
6
7    name = cat; // @cat is 0x2000. @name is 0x2000. @name points to "dolly"
8
9    std::cout << assign; // @assign is 0x1000. @assign points to "alex"
10
11   return 0;
12 }
```

magaji::hussaini
December 12, 2018 at 8:59 am · Reply

Hi Alex , nascardriver
Please I am confused about getName() returning a dangling pointer bcos I tried this:

```
1  const char* getName() { return "Hussaini"; }
2  char* getName1() { return "Magaji"; }
```

and the below statements works (though with a warning):

```
1  const char *name { getName() };
2  char *name1 { getName1() };
3  std::cout << name << '\n' << name1 << '\n';
```

Alex
December 14, 2018 at 7:32 pm · Reply

String literals have static duration, and thus won't create dangling pointers.

magaji::hussaini
December 16, 2018 at 2:08 am · Reply

Thanks

C++guy
November 4, 2018 at 12:55 am · Reply

"The answer is that std::cout makes some assumptions about your intent. If you pass it a non-char pointer, it will simply print the contents of that pointer (the address that the pointer is holding). However, if you pass it an object of type char* or const char*, it will assume you're intending to print a string. Consequently, instead of printing the pointer's value, it will print the string being pointed to instead!"

So basically, when std::cout runs a string it just dereferences all the elements of the array if it's put as an array after decaying all of them into pointers right ? And that's why C++ allows to write something like const char* myName = "Alex" ? Cause theorically a char can't be equal to a string but if it's a pointer then it's ok isn't it ?

nascardriver
November 4, 2018 at 2:26 am · Reply

yes

Liam
September 8, 2018 at 4:10 pm · Reply

Hello,

When I'm trying to create a symbolic constant using constexpr, like so:

```
1 constexpr char *myName = "Alex";
```

I'm getting a compiler warning:
"warning: ISO C++ forbids converting a string constant to 'char*' [-Wwrite-strings]"

While using const instead doesn't trigger a warning. Why so?

nascardriver
September 9, 2018 at 1:48 am · Reply

Hi Liam!

```
1  // (1) A pointer to a const char (array).
2  const char *szText{ "Hello World!" };
3
4  szText[3] = 'R'; // Illegal, each character is const
5  szText = nullptr; // OK, the pointer itself is not const
6
7  // (2) A pointer to a char (array).
8  char *szText{ "Hello World!" }; // Illegal by definition. "Hello World!" is a symboli
9
10 // (3) A constant pointer to a char (array).
11 char *const szText{ "Hello World!" }; // Illegal by definition. Same reason as (2).
12
13 // (4) A constant pointer to a const char (array)
14 const char *const szText{ "Hello World!" };
15
16 szText[3] = 'R'; // Illegal, each character is const
17 szText = nullptr; // Illegal, the pointer is const
```

Here comes the slightly confusing part, constexpr refers to the pointer, not the elements.

```
1 // (5) A compile-time-constant pointer to a _non-const_ char (array).
2 constexpr char *szText{ "Hello World!" }; // Illegal by definition. Same reason as (2)
3
4 // (6) A compile-time-constant pointer to a const char (array).
5 constexpr const char *szText{ "Hello World!" };
```

```
6
7  szText[3] = 'R'; // Illegal, each character is const
8  szText = nullptr; // Illegal, the pointer is const
```

Liam
September 10, 2018 at 12:13 pm · Reply

Thank you nascardriver!

I would recommend to include this part in the tutorial since I would never have guessed this without your explanation! First, I didn't realize how to create a const pointer, instead of a pointer to a const (I just found out that this is covered in a future lesson, that I didn't go through yet). Second, now I see that the constexpr pointer doesn't even follow that syntax and worth separate explanation (took some head-scratching to figure out what's going on, must be worth mentioning constexpr in the 6.10 "Pointers and const" lesson).

Piyush
January 15, 2019 at 1:56 am · Reply

```
1  const char *szText{ "Hello World!" };
```

It means that i can change the address pointed by szText but can't change the value of the address pointed by azText currently.
So doing so-

```
1  char *a{"aa"};
2  szText{a};
```

Will change the address pointed by szText but "Hello World!" will still be stored in the previous address.

we also need to make the pointer itself const to let these things not happen.

nascardriver
January 15, 2019 at 8:11 am · Reply

> It means that i can change the address pointed by szText but can't change the value of the address pointed by azText currently.

Correct.

> char *a{"aa"};
That's illegal.

> szText{a};
That's invalid syntax.

```
1  const char *szText{ "Hello World!" };
2  const char *szOtherText{ "Piyush" };
3
4  szText = szOtherText;
```

You can declare the pointer const if you want to. There's no downside from overwriting the pointer.

Piyush
January 16, 2019 at 6:04 am · Reply

Thanks a lot.

Nguyen
August 30, 2018 at 7:45 pm · Reply

Hi nascardriver,

Unlike 6.6 - C-style strings, C-style string symbolic constants using pointers can not have value based on user input.
Is it right?

Thanks.

nascardriver
September 1, 2018 at 7:11 am · Reply

right

Michael Stef
July 15, 2018 at 11:51 am · Reply

i'am not sure but shouldn't "Use a const pointer to a string literal" be "Use a pointer to a const string literal" because the pointer can be changed to point to other read only strings but string literal it self can't be changed ?

Alex
July 17, 2018 at 4:21 pm · Reply

Yes, thanks for the correction.

Ishak
June 28, 2018 at 10:11 am · Reply

I can do this:

```
1 const char *name1 = "Alex";
```

but I cant do this ? if yes then why?

[code] const int *value = 5; [code]

as always thanks for the help

nascardriver
June 28, 2018 at 10:16 am · Reply

Hi Ishak!

5 is neither a pointer nor a string. You need to either add quotation marks or a cast.

Conor
July 26, 2018 at 5:28 am · Reply

This still doesn't make sense to me, this is a pointer declaration/assignment but its being assigned a string value, should it not be something like &someChar previously defined?

nascardriver
July 26, 2018 at 5:31 am · Reply

Whenever you write

```
1 "some text"
```

in C++, it's a const char* (an array of characters), not an @std::string. @std::string just has a constructor that accepts a const char *, so you can use a const char * in places where an @std::string is required.

Conor McGuigan
July 26, 2018 at 5:33 am · Reply

I think I understand it now ("Alex" is placed in memory and myName points to it), its just confusing syntax. Why does "const int *myNumber = 5", its the same concept. Does the compiler not put the value 5 in some memory location and myNumber points to it like the previous case?

nascardriver
July 26, 2018 at 5:42 am · Reply

The type of

```
1 "hello world"
```

is const char *.
The type of

```
1 5
```

is int.

You can store a const char * in a const char *, but you cannot store an int in a const int *.

Conor McGuigan
July 26, 2018 at 5:54 am · Reply

I did a bit of messing around and I'm starting to get it a bit better. I just don't like the varying consistency here, where you use pointer declaration/initialisation in a different context as well as the strange behaviour when you want to print the address of a char*. but it's something you won't be doing regularly so I can not get too annoyed about it. Thanks for replies

nascardriver
July 26, 2018 at 6:03 am · Reply

Keep in mind that an array is nothing but a pointer to the first element.
This syntax might suit you better

```
const char szName[]{ "Alex" };
```

@std::cout was programmed to print the array value rather than the address when it's passed a const char *. You can print the address by casting the const char * to a const void * or using @std::printf.

```
const char szName[]{ "Alex" };

// Print value
std::cout << szName << std::endl;
std::printf("%s\n", szName);

// Print address
std::cout << reinterpret_cast<const void *>(szName) << std::endl;
std::printf("%p\n", szName);
```

Conor McGuigan
July 26, 2018 at 6:16 am · Reply

Yes this helps a lot. Do you have to do the cast? This also printed the address:

```
std::cout << &szName;
```

nascardriver
July 26, 2018 at 6:23 am · Reply

That will print the address of the pointer (a const char **), not the address the pointer is pointing to.

Conor McGuigan
July 26, 2018 at 6:28 am

Oh right that makes sense. Thanks again and apologies for going into to such details.

Xin Shi
June 26, 2018 at 11:36 pm · Reply

hi, Alex
how can I get the adress of a char type value when I really want do that?

nascardriver
June 27, 2018 at 2:04 am · Reply

Hi Xin Shi!

```
char c{ 'n' }; // Char you want to get the address of
char *p{ &c }; // Address of @c, you're probably talking about printing it.

```

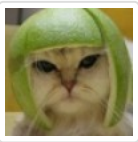
```
// @std::uintptr_t from <cstdint> guarantees lossless conversion from any pointer to @
// @reinterpret_cast performs a cast without conversions.
std::uintptr_t uiAddress{ reinterpret_cast<std::uintptr_t>(p) };

// Print it
std::cout << std::hex << uiAddress << std::endl;
```

John Kennedy
April 14, 2018 at 11:52 pm · Reply

```
const char* getName()
{
    return "Alex";
}
```

"In the above code, getName() will return a pointer to C-style string “Alex”. This is okay since “Alex” will not go out of scope when getName() terminates, so the caller can still successfully access it."

I don't get those lines. What do you mean by telling ""Alex" will not go out of scope"



Alex
April 16, 2018 at 2:41 pm · Reply

When returning a pointer (or reference) from a function, you have to be careful that the object being pointed to (or referenced) does not get destroyed when the function ends. For example:

```
int * getX()
{
     int x = 5;
     return &x;
}
```

This is bad. x is a local variable, and it will get destroyed at the end of the function. The pointer passed back to the caller will be left dangling, and accessing it will result in undefined behavior.

However, because string literals have special handling, they aren't destroyed at the end of the function. So it's safe to return pointers to them.

Peter Baum
March 20, 2018 at 5:08 pm · Reply

You might consider making the first section "C-style string symbolic constants" clearer by separating the two issues:

1. using pointers or using an explicit array declaration
2. the use of const or not

Because of the previous lesson, I was thinking mostly about 1. and had to go back and discovered that the topic was actually mostly about 2.

Overall... doing a great job explaining pointers.

Alex
March 22, 2018 at 1:33 pm · Reply

The two considerations are linked:
* If you use the pointer method, you should always use const
* If you use the array method, you choose whether to const or not (but if you're going to const, then you might as well use the pointer method)

I'll add a summary to the end of the section to make it more clear what the takeaways are. Appreciate the feedback!

ASP
March 2, 2018 at 2:36 am · Reply

A small typo:
Thus, if name1 (were) not const, making a change to name1 could also impact name2.
were -> was

Alex
March 3, 2018 at 8:28 pm · Reply

Fixed! Thanks for pointing this out.

Vinicius
April 13, 2018 at 7:46 am · Reply

Wow, you're still updating the lessons in 2018. Good for who wanna learn C++ this year =)

erad
December 15, 2017 at 1:38 am · Reply

Hi Alex,

I am sorely confused about this segment in this lesson!:

"For optimization purposes, multiple string literals with the same content may point to the same location. For example:

```
1 const char *name1 = "Alex";
2 const char *name2 = "Alex";
```

These are two different string literals with the same value."

I thought the string literals are the quoted expressions(i.e. "Alex") on the right of the assignment operator while the objects doing the 'pointing' (i.e. pointers name1 and name2) are the ones on the left.

1. So why did you write that "multiple string literals ... may point to ... "? String literals are essentially 'values' which theoretically do not point to other objects, right?

2. With respect to the last line of the quote, did you mean VARIABLES in place of the LITERALS you wrote? Because then it makes more logical sense that the VARIABLES are "different" but could contain the same value. In my view, 'literal' and 'value' are somewhat synonymous. Am I wrong?

Alex
December 16, 2017 at 5:25 pm · Reply

Poor wording on my part. I've updated the text to be more technically accurate.

Alex
November 24, 2017 at 4:41 pm · Reply

Assuming c is a char, &c will get the address of the char. It's just that you can't print this, because it'll print as a string rather than an address. If you want to print the address, you could try casting it to a void pointer and see if that works -- though if so, I'm not sure whether that behavior is standard across all compilers.

Joe
October 27, 2017 at 4:20 am · Reply

Hi Alex,

I think there is a typo in the below lines.

const char name1 = "Alex";
const char name2 = "Alex";

Aren't above lines supposed to be

const char name1[] = "Alex";
const char name2[] = "Alex";

Alex
October 30, 2017 at 2:03 pm · Reply

No, they're actually supposed to be:

```
1 const char *name1 = "Alex";
2 const char *name2 = "Alex";
```

Thanks for pointing out the error!

Kushagra
October 24, 2017 at 6:35 am · Reply

Please explain this

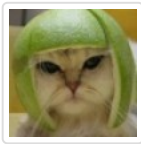
In the symbolic constant case, how the compiler handles this is implementation defined. What usually happens is that the compiler places the string “Alex\0” into read-only memory somewhere, and then sets the pointer to point to it. Multiple string literals with the same content may point to the same location. Because this memory may be read-only, and because making a change to a string literal may impact other uses of that literal, best practice is to make sure the string is const.

Alex
October 25, 2017 at 11:53 am · Reply

What part of the paragraph do you need clarification on? There's a lot going on there. :)

Kushagra
October 25, 2017 at 11:39 pm · Reply

From multiple string literals.



Alex
October 26, 2017 at 1:48 pm · Reply

Gotcha. By multiple string literals, I mean something like this:

```
const char name1 = "Alex";
const char name2 = "Alex";
```

These are two different string literals with the same value. The compiler may opt to combine these into a single shared string literal, with both name1 and name2 pointed at the same address.

Lim Che Ling
September 27, 2017 at 8:36 pm · Reply

Hi, can you enlighten me on this stupid question?
I can do this:

```
char str[] = "myString\n";
char* ptr = str;
cout << ptr;  //print 'myString'
```

But when I do this, I get warning "ISO C++11 does not allow conversion from string literal to 'char*'

```
char *ptr ="myString";
```

Why is it so?

Alex
October 2, 2017 at 9:12 pm · Reply

Because a string literal has type const char*, not char*.

Astronoid
July 4, 2017 at 6:42 am · Reply

Hey Alex,
Thank you for your efforts; I am confused about many points in that lesson.

```
#include <iostream>

int main()
{
    char c = 'Q';
    std::cout << &c;

    return 0;
}
```

The first one is why the compiler treat "&c" as a string not a pointer; the second one is how to get the memory address of that pointer. And thanks;

Alex
July 5, 2017 at 1:01 pm · Reply

std::cout was designed to treat objects of type (const) char* as strings since this is the native type of C-style string literals. Most of the time, this is what we want. Occasionally it causes issues.

I'm not sure what you actually mean by "get the memory address of that pointer". &c returns a pointer to variable c, but it doesn't have an address itself since it's an r-value.

Harshit
November 23, 2017 at 10:53 am · Reply

I also have the same question.
How can we then get the address of the char variable c then.

Rex Lucas
November 30, 2017 at 2:49 am · Reply

Hi Harshit
I had the same question in sect 6.7. If you want to use cout then Alex said "cast to void*" ie

```
#include<iostream>
using namespace std;
int main()
{
char c = 'Q';
void *ptr = &c; //Alex actually said use static_cast but implicit cast appears
cout << ptr;
}
```

« Older Comments 1 2