What is the difference between const int*, const int * const, and int const *?

Asked 10 years, 3 months ago Active 25 days ago Viewed 460k times

1244

1039

I always mess up how to use `const int*`, `const int * const`, and `int const *` correctly. Is there a set of rules defining what you can and cannot do?

I want to know all the do's and all don'ts in terms of assignments, passing to the functions, etc.

c++ c pointers int const

edited Mar 9 '17 at 18:48
MD XF
4,438 ● 5 ● 31 ● 57

asked Jul 17 '09 at 13:28
ultraman

156 You can use the "Clockwise/Spiral Rule" to decipher most C and C++ declarations. – James McNellis Jun 13 '10 at 20:49

45 cdecl.org is a great website which auto-translates C declarations for you. – Dave Gallagher Nov 2 '10 at 19:37

6 @Calmarius: **start where the type-name is / should be, move right when you can, left when you must**. `int *(*)(char const * const)`. Start to the right of the parenthesized `*` then we have to move left: `pointer`. Outside the parens, we can move right: `pointer to function of ...`. Then we have to move left: `pointer to function of ... that returns pointer to int`. Repeat to expand the parameter (the `...`): `pointer to function of (constant pointer to constant char) that returns pointer to int`. What would the equivalent one-line declaration be in a easy-reading language like Pascal? – Mark K Cowan Jul 9 '15 at 17:08

1 @MarkKCowan In Pascal it would be something like `function(x:^char):^int`. There function types are imply a pointer to a function so no need to specify it, and Pascal doesn't enforce const correctness. It can be read from left to right. – Calmarius Jul 9 '15 at 20:54

4 The first thing to the left of the "const" is what's constant. If "const" is the thing the farthest to the left, then the first thing to the right of it is what's constant. – Ngineer Jul 31 '16 at 4:41

17 Answers

2072

✓

Read it backwards (as driven by Clockwise/Spiral Rule):

- `int*` - pointer to int
- `int const *` - pointer to const int
- `int * const` - const pointer to int
- `int const * const` - const pointer to const int

Now the first `const` can be on either side of the type so:

- `const int *` == `int const *`
- `const int * const` == `int const * const`

If you want to go really crazy you can do things like this:

- `int **` - pointer to pointer to int
- `int ** const` - a const pointer to a pointer to an int
- `int * const *` - a pointer to a const pointer to an int

- `int const **` - a pointer to a pointer to a const int
- `int * const * const` - a const pointer to a const pointer to an int
- ...

And to make sure we are clear on the meaning of const

```
const int* foo;
int *const bar; //note, you actually need to set the pointer
                //here because you can't change it later ;)
```

`foo` is a variable pointer to a constant integer. This lets you change what you point to but not the value that you point to. Most often this is seen with C-style strings where you have a pointer to a `const char`. You may change which string you point to but you can't change the content of these strings. This is important when the string itself is in the data segment of a program and shouldn't be changed.

`bar` is a constant or fixed pointer to a value that can be changed. This is like a reference without the extra syntactic sugar. Because of this fact, usually you would use a reference where you would use a `T* const` pointer unless you need to allow `NULL` pointers.

edited Apr 17 '18 at 5:10
Azeem
3,329 ●4 ●12 ●25

answered Jul 17 '09 at 13:29
Matt Price
31.4k ●9 ●31 ●41

445 I would like to append a rule of thumb which may help you remember how to discover whether 'const' applies to pointer or to pointed data: split the statement at asterix sign, then, if the const keyword appears in the left part (like in 'const int * foo') - it belongs to pointed data, if it's in the right part ('int * const bar') - it's about the pointer. – Michael Jul 17 '09 at 17:26

11 @Michael: Kudos to Michael for such a simple rule for remembering/understanding const rule. – sivabudh Feb 11 '10 at 19:00

9 @Jeffrey: read it backwards works well as long as there are no parenthesis. Then, well... use typedefs – Mooing Duck May 28 '13 at 19:53

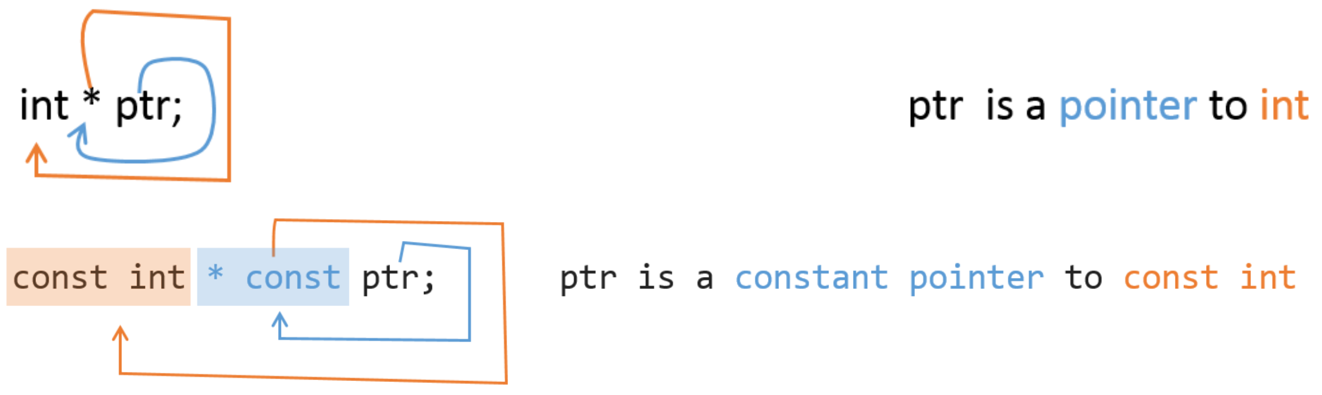
10 +1, though a better summary would be: **read pointer declarations backwards**, that means, close to @Michael 's statement: stop the normal left-to-right reading at the *first* asterisk. – Wolf Jun 18 '14 at 9:21

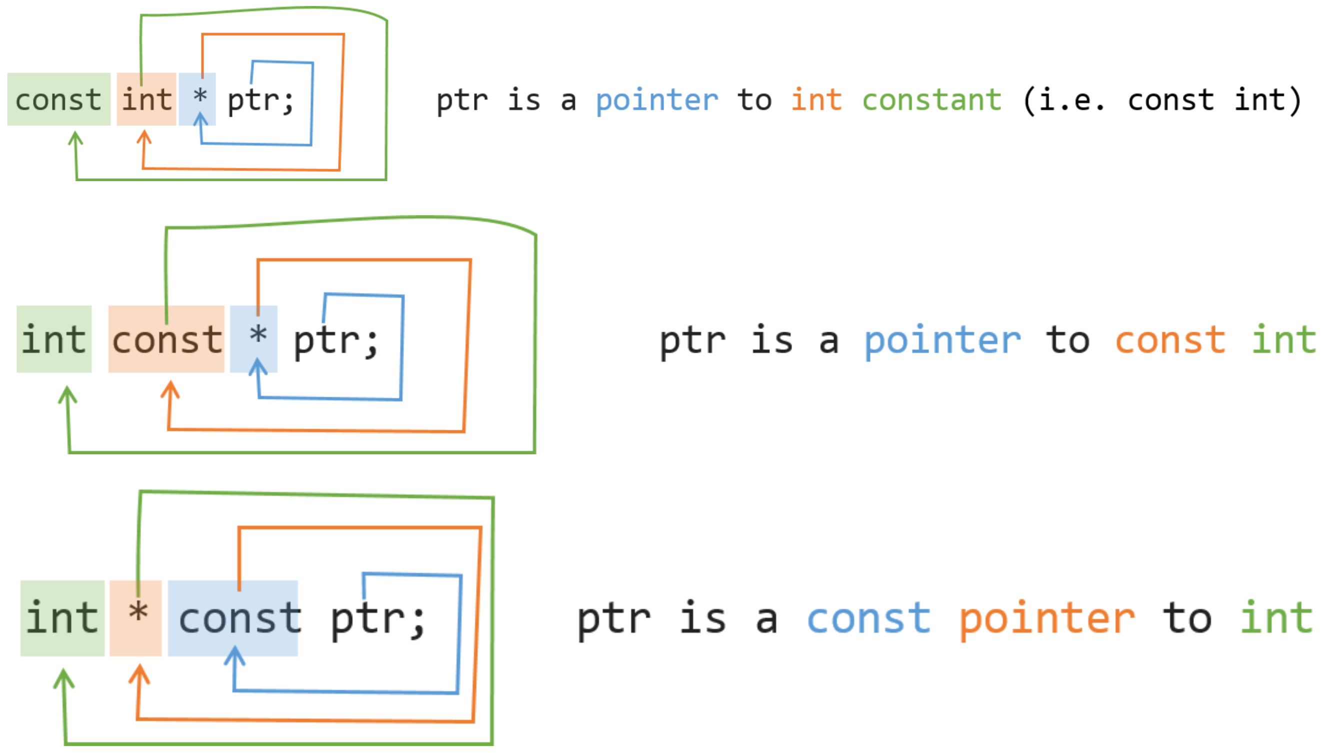
2 Why no C/C++ book presents it this way? It's great – Artur Mar 26 '15 at 10:19

322

For those who don't know about Clockwise/Spiral Rule: Start from the name of the variable, move clockwisely (in this case, move backward) to the next **pointer** or **type**. Repeat until expression ends.

Here is a demo:





`const int * ptr;` ptr is a pointer to int constant (i.e. const int)

`int const * ptr;` ptr is a pointer to const int

`int * const ptr;` ptr is a const pointer to int

edited Jun 8 at 6:53
Felipe Augusto
2,673 ● 4 ● 15 ● 34

answered Jul 10 '15 at 2:15
Shijing Lv
3,584 ● 1 ● 12 ● 9

125 thats just reading it right to left. – Dave Sep 25 '15 at 16:52

8 @Jan the link for the complex example does not have permissions. can you post it directly here, or remove the viewing restrictions? – R71 Apr 8 '16 at 12:03

8 @Rog it used to have all open access permissions... I didn't write the article and don't have access permissions myself, unfortunately. However, here is an archived version of the article that still works: archive.is/SsfMX – Jan Rüegg Apr 8 '16 at 13:34

7 The complex example is still just right to left, but includes resolving parentheses the way one would normally. The whole clockwise spiral thing doesn't make that any easier. – Matthew Read Sep 18 '16 at 23:04

3 I like the colours. *Just a simple man passing through* – rfcoder89 Oct 1 '16 at 7:10

137

I think everything is answered here already, but I just want to add that you should beware of `typedef`s! They're NOT just text replacements.

For example:

```
typedef char *ASTRING;
const ASTRING astring;
```

The type of `astring` is `char * const`, not `const char *`. This is one reason I always tend to put `const` to the right of the type, and never at the start.

edited Apr 17 '18 at 5:15
Azeem
3,329 ● 4 ● 12 ● 25

answered Jul 17 '09 at 13:39
Kaz Dragon
5,556 ● 2 ● 30 ● 44

18 And for me this is the reason to never typedef pointers. I don't see the benefit in things like `typedef int* PINT` (I assume its something that came from practices in C and many developers kept doing it). Great, I replaced that `*` with a `P`, it doesn't speed up typing, plus introducing the issue you mention. – Mephane Jan 28 '11 at 13:01

1 @Mephane - I can see that. However, to me it seems kind of backwards to avoid a nice language feature in order to keep using an exceptional syntactical rule (about "const" placement), rather than avoiding using the exceptional syntactic rule so you can safely make use of this language feature. – T.E.D. Oct 17 '12 at 14:06

6 @Mephane `PINT` is indeed a rather dumb usage of a typedef, especially cuz it makes me think that the system stores uses beer for memory. typedef s are pretty useful for dealing with pointers to functions, though. – ApproachingDarknessFish Dec 26 '13 at 21:07

5 @KazDragon THANKS! Without it, I would've messed up with all those typedefed `PVOID`, `LPTSTR` stuff in Win32 api! – David Lee May 8 '14 at 12:29

2 @Mephane: I've had to use pSomething a couple of times when using certain legacy macros which were written to accept a type, but would break apart if the type wasn't a single alphanumeric identifier. :) – Groo May 8 '17 at 16:37

50

Like pretty much everyone pointed out:

What's the difference between `const X* p`, `X* const p` and `const X* const p`?

> You have to read pointer declarations right-to-left.
>
> - `const X* p` means "p points to an X that is const": the X object can't be changed via p.
> - `X* const p` means "p is a const pointer to an X that is non-const": you can't change the pointer p itself, but you can change the X object via p.
> - `const X* const p` means "p is a const pointer to an X that is const": you can't change the pointer p itself, nor can you change the X object via p.

edited May 26 '17 at 14:20
Donald Duck
4,393 ●14 ●43 ●68

answered Jul 17 '09 at 13:36
luke
28.9k ●7 ●53 ●78

3 Don't forget that `const X* p;` `==` `X const * p;` as in `"p points to an X that is const": the X object can't be changed via p.` – Jesse Chisholm Jan 17 at 20:06

simple and nice explaintion! – Edison Lo Sep 17 at 15:05

47

1. **Constant reference:**

   A reference to a variable (here int), which is constant. We pass the variable as a reference mainly, because references are smaller in size than the actual value, but there is a side effect and that is because it is like an alias to the actual variable. We may accidentally change the main variable through our full access to the alias, so we make it constant to prevent this side effect.

   ```
   int var0 = 0;
   const int &ptr1 = var0;
   ptr1 = 8; // Error
   var0 = 6; // OK
   ```

2. **Constant pointers**

   Once a constant pointer points to a variable then it cannot point to any other variable.

```
int var1 = 1;
int var2 = 0;

int *const ptr2 = &var1;
ptr2 = &var2; // Error
```

3. **Pointer to constant**

   A pointer through which one cannot change the value of a variable it points is known as a pointer to constant.

```
int const * ptr3 = &var2;
*ptr3 = 4; // Error
```

4. **Constant pointer to a constant**

   A constant pointer to a constant is a pointer that can neither change the address it's pointing to and nor can it change the value kept at that address.

```
int var3 = 0;
int var4 = 0;
const int * const ptr4 = &var3;
*ptr4 = 1;     // Error
 ptr4 = &var4; // Error
```

edited Mar 13 '15 at 19:32
Peter Mortensen
14.5k ● 19 ● 89 ● 118

answered May 17 '14 at 20:21
Behrooz Tabesh
1,113 ● 9 ● 6

19

The general rule is that the `const` keyword applies to what precedes it immediately. Exception, a starting `const` applies to what follows.

- `const int*` is the same as `int const*` and means **"pointer to constant int"**.
- `const int* const` is the same as `int const* const` and means **"constant pointer to constant int"**.

Edit: For the Dos and Don'ts, if this answer isn't enough, could you be more precise about what you want?

edited Apr 17 '18 at 5:13
Azeem
3,329 ● 4 ● 12 ● 25

answered Jul 17 '09 at 13:30
AProgrammer
44.1k ● 7 ● 75 ● 128

18

This question shows **precisely** why I like to do things the way I mentioned in my question is const after type id acceptable?

In short, I find the easiest way to remember the rule is that the "const" goes *after* the thing it applies to. So in your question, "int const *" means that the int is constant, while "int * const" would mean that the pointer is constant.

If someone decides to put it at the very front (eg: "const int *"), as a special exception in that case it applies to the thing after it.

Many people like to use that special exception because they think it looks nicer. I dislike it, because it is an exception, and thus confuses things.

edited May 23 '17 at 12:02

answered Jul 17 '09 at 13:52

Community ♦
1 ● 1

T.E.D.
37.2k ● 7 ● 59 ● 129

2 I'm torn on this issue. Logically it makes sense. However most c++ developers would write `const T*` and it has become more natural. How often do you ever use a `T* const` anyways, usually a reference will do just fine. I got bit by all this once when wanting a `boost::shared_ptr<const T>` and instead wrote `const boost::shared_ptr<T>`. Same issue in a slightly different context. – Matt Price Jul 17 '09 at 14:08

Actually, I use constant pointers more often than I use constants. Also, you have to think about how you are going to react in the presence of pointers to pointers (etc.) Admittedly those are rarer, but it would be nice to think about things in a way where you can handle these situations with applomb. – T.E.D. Jul 17 '09 at 14:19

1 The one other nice advantage of placing the const on the right of the type is that now everything to the left of any `const` is the type of that which is const, and everything to its right is that which is actually const. Take `int const * const * p;` as an example. No I don't normally write like that, this is just an example. First `const`: type int, And the int that is const is the contents of the const pointer that is the contents of `p`. Second const: type is pointer to `const` int, const oblect is the contents of `p` – dgnuff Mar 15 '18 at 7:18

16

Simple Use of `const`.

The simplest use is to declare a named constant. To do this, one declares a constant as if it was a variable but add `const` before it. One has to initialize it immediately in the constructor because, of course, one cannot set the value later as that would be altering it. For example:

```
const int Constant1=96;
```

will create an integer constant, unimaginatively called `Constant1`, with the value 96.

Such constants are useful for parameters which are used in the program but are do not need to be changed after the program is compiled. It has an advantage for programmers over the C preprocessor `#define` command in that it is understood & used by the compiler itself, not just substituted into the program text by the preprocessor before reaching the main compiler, so error messages are much more helpful.

It also works with pointers but one has to be careful where `const` to determine whether the pointer or what it points to is constant or both. For example:

```
const int * Constant2
```

declares that `Constant2` is variable pointer to a constant integer and:

```
int const * Constant2
```

is an alternative syntax which does the same, whereas

```
int * const Constant3
```

declares that `Constant3` is constant pointer to a variable integer and

```
int const * const Constant4
```

declares that `Constant4` is constant pointer to a constant integer. Basically ‘const’ applies to whatever is on its immediate left (other than if there is nothing there in which case it applies to whatever is its immediate right).

ref: http://duramecho.com/ComputerInformation/WhyHowCppConst.html

edited Jun 8 at 6:55
Felipe Augusto
2,673 ●4 ●15 ●34

answered Jul 17 '09 at 13:31
ufukgun
4,351 ●6 ●28 ●51

8

I had the same doubt as you until I came across this book by the C++ Guru Scott Meyers. Refer the third Item in this book where he talks in details about using `const`.

Just follow this advice

1. If the word `const` appears to the left of the asterisk, what's pointed to is constant
2. If the word `const` appears to the right of the asterisk, the pointer itself is constant
3. If `const` appears on both sides, both are constant

answered Mar 21 '15 at 13:56
rgk
589 ●11 ●24

7

It's simple but tricky. Please note that we can swap the `const` qualifier with any data type (`int` , `char` , `float` , etc.).

Let's see the below examples.

`const int *p` ==> `*p` is read-only [`p` is a pointer to a constant integer]

`int const *p` ==> `*p` is read-only [`p` is a pointer to a constant integer]

`int *p const` ==> **Wrong** Statement. Compiler throws a syntax error.

`int *const p` ==> `p` is read-only [`p` is a constant pointer to an integer]. As pointer `p` here is read-only, the declaration and definition should be in same place.

`const int *p const` ==> **Wrong** Statement. Compiler throws a syntax error.

`const int const *p` ==> `*p` is read-only

`const int *const p1` ==> `*p` and `p` are read-only [`p` is a constant pointer to a constant integer]. As pointer `p` here is read-only, the declaration and definition should be in same place.

`int const *p const` ==> **Wrong** Statement. Compiler throws a syntax error.

`int const int *p` ==> **Wrong** Statement. Compiler throws a syntax error.

`int const const *p` ==> `*p` is read-only and is equivalent to `int const *p`

`int const *const p` ==> `*p` and `p` are read-only [`p` is a constant pointer to a constant integer]. As pointer `p` here is read-only, the declaration and definition should be in same place.

edited Aug 17 '15 at 21:12
Kimbluey
477 ●1 ●7 ●21

answered Jan 4 '15 at 13:10
Abhijit Sahu
92 ●1 ●8

6

The C and C++ declaration syntax has repeatedly been described as a failed experiment, by the original designers.

Instead, let's *name* the type “pointer to `Type`”; I’ll call it `Ptr_`:

```
template< class Type >
using Ptr_ = Type*;
```

Now `Ptr_<char>` is a pointer to `char`.

`Ptr_<const char>` is a pointer to `const char`.

And `const Ptr_<const char>` is a `const` pointer to `const char`.

There.

edited Jan 6 '16 at 0:18

answered Jan 6 '16 at 0:12

Cheers and hth. - Alf
127k ● 14 ● 173 ● 282

3 do you have a quote for the first sentence? – sp2danny Sep 14 '16 at 7:20

@sp2danny: Googling “C syntax failed experiment” only coughs up a number of interviews with Bjarne Stroustrup where he expresses *his* opinion in that direction, e.g. “I consider the C declarator syntax an experiment that failed” in the Slashdot interview. So I have no reference for the claim about the viewpoints of the original designers of C. I guess it can be found by a sufficiently strong research effort, or maybe disproved simply by asking them, but I think it's better the way it is now. with that part of the claim, still undecided and likely true:) – Cheers and hth. - Alf Sep 14 '16 at 10:42

Oh, Dennis Ritchie has passed away. Brian Kernighan still going strong. But Wikipedia's article about him says, " Kernighan affirmed that he had no part in the design of the C language ("it's entirely Dennis Ritchie's work").". – Cheers and hth. - Alf Sep 14 '16 at 10:47

1 "The C and C++ declaration syntax has repeatedly been described as a failed experiment, by the original designers." wrong for C please change your sentence about C or provide some quotes. – Stargateur Jan 1 '18 at 14:22

3 @Stargateur: Apparently you have read the preceding comments and found something you could leverage for pedantry. Good luck with your life. Anyway, old-timers like me remember a lot that we can't prove without engaging in very time-consuming research. You could just take my word. – Cheers and hth. - Alf Jan 1 '18 at 14:25

6

There are many other subtle points surrounding const correctness in C++. I suppose the question here has simply been about C, but I'll give some related examples since the tag is C++ :

- You often pass large arguments like strings as `TYPE const &` which prevents the object from being either modified or copied. Example :

  `TYPE& TYPE::operator=(const TYPE &rhs) { ... return *this; }`

  But `TYPE &  const` is meaningless because references are always const.

- You should always label class methods that do not modify the class as `const`, otherwise you cannot call the method from a `TYPE const &` reference. Example :

  `bool TYPE::operator==(const TYPE &rhs) const { ... }`

- There are common situations where both the return value and the method should be const. Example :

`const TYPE TYPE::operator+(const TYPE &rhs) const { ... }`

In fact, const methods must not return internal class data as a reference-to-non-const.

- As a result, one must often create both a const and a non-const method using const overloading. For example, if you define `T const& operator[] (unsigned i) const;` , then you'll probably also want the non-const version given by :

```
inline T& operator[] (unsigned i) {
  return const_cast<char&>(
      static_cast<const TYPE&>(*this)[](i)
  );
}
```

Afaik, there are no const functions in C, non-member functions cannot themselves be const in C++, const methods might have side effects, and the compiler cannot use const functions to avoid duplicate function calls. In fact, even a simple `int const &` reference might witness the value to which it refers be changed elsewhere.

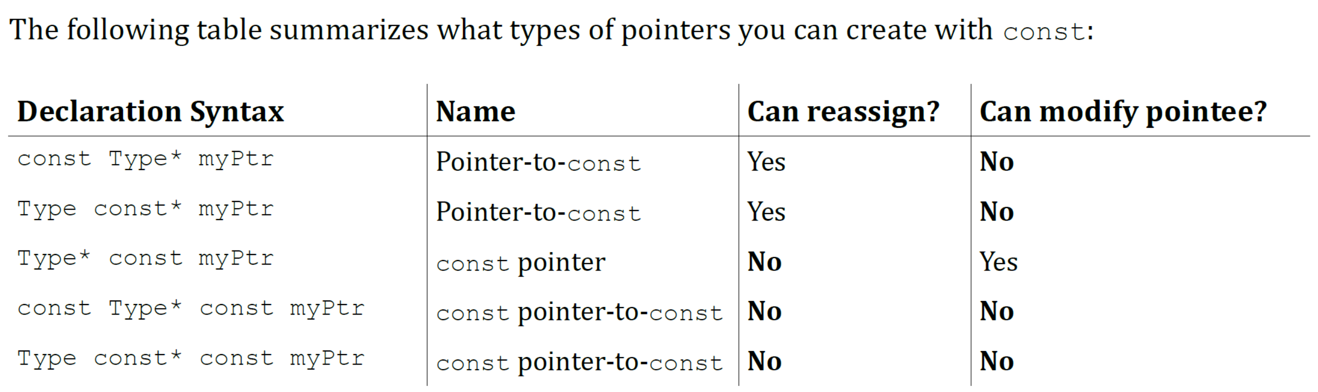
answered Sep 13 '11 at 10:50
Jeff Burdges
2,571 ● 16 ● 36

4

For me, the position of `const` i.e. whether it appears to the LEFT or RIGHT or on both LEFT and RIGHT relative to the `*` helps me figure out the actual meaning.

1. A `const` to the LEFT of `*` indicates that the object pointed by the pointer is a `const` object.
2. A `const` to the RIGHT of `*` indicates that the pointer is a `const` pointer.

The following table is taken from Stanford CS106L Standard C++ Programming Laboratory Course Reader.


The following table summarizes what types of pointers you can create with `const`:

Declaration Syntax	Name	Can reassign?	Can modify pointee?
`const Type* myPtr`	Pointer-to-`const`	Yes	**No**
`Type const* myPtr`	Pointer-to-`const`	Yes	**No**
`Type* const myPtr`	`const` pointer	**No**	Yes
`const Type* const myPtr`	`const` pointer-to-`const`	**No**	**No**
`Type const* const myPtr`	`const` pointer-to-`const`	**No**	**No**

answered Feb 4 at 7:48
srivatsahc
143 ● 3

3

The const with the int on either sides will make **pointer to constant int**:

```
const int *ptr=&i;
```

or:

```
int const *ptr=&i;
```

`const` after `*` will make **constant pointer to int**:

```
int *const ptr=&i;
```

In this case all of these are **pointer to constant integer**, but none of these are constant pointer:

```
 const int *ptr1=&i, *ptr2=&j;
```

In this case all are **pointer to constant integer** and ptr2 is **constant pointer to constant integer**. But ptr1 is not constant pointer:

```
int const *ptr1=&i, *const ptr2=&j;
```

edited Jun 8 at 6:51
Felipe Augusto
2,673 ● 4 ● 15 ● 34

answered Sep 23 '18 at 8:44
Hunter
31 ● 3

2

This mostly addresses the second line: best practices, assignments, function parameters etc.

General practice. Try to make everything `const` that you can. Or to put that another way, make everything `const` to begin with, and then remove exactly the minimum set of `const`s necessary to allow the program to function. This will be a big help in attaining const-correctness, and will help ensure that subtle bugs don't get introduced when people try and assign into things they're not supposed to modify.

Avoid const_cast<> like the plague. There are one or two legitimate use cases for it, but they are very few and far between. If you're trying to change a `const` object, you'll do a lot better to find whoever declared it `const` in the first pace and talk the matter over with them to reach a consensus as to what should happen.

Which leads very neatly into assignments. You can assign into something only if it is non-const. If you want to assign into something that is const, see above. Remember that in the declarations `int const *foo;` and `int * const bar;` different things are `const` - other answers here have covered that issue admirably, so I won't go into it.

Function parameters:

Pass by value: e.g. `void func(int param)` you don't care one way or the other at the calling site. The argument can be made that there are use cases for declaring the function as `void func(int const param)` but that has no effect on the caller, only on the function itself, in that whatever value is passed cannot be changed by the function during the call.

Pass by reference: e.g. `void func(int &param)` Now it does make a difference. As just declared `func` is allowed to change `param`, and any calling site should be ready to deal with the consequences. Changing the declaration to `void func(int const &param)` changes the contract, and guarantees that `func` can now not change `param`, meaning what is passed in is what will come back out. As other have noted this is very useful for cheaply passing a large object that you don't want to change. Passing a reference is a lot cheaper than passing a large object by value.

Pass by pointer: e.g. `void func(int *param)` and `void func(int const *param)` These two are pretty much synonymous with their reference counterparts, with the caveat that the called function now needs to check for `nullptr` unless some other contractual guarantee assures `func` that it will never receive a `nullptr` in `param`.

Opinion piece on that topic. Proving correctness in a case like this is hellishly difficult, it's just too damn easy to make a mistake. So don't take chances, and always check pointer parameters for `nullptr`. You will save yourself pain and suffering and hard to find bugs in the long term. And as for the cost of the check, it's dirt cheap, and in cases where the static analysis built into the compiler can manage it, the optimizer will elide it anyway. Turn on Link Time Code Generation for MSVC, or WOPR (I think) for GCC, and you'll get it program wide, i.e. even in function calls that cross a source code module boundary.

At the end of the day all of the above makes a very solid case to always prefer references to pointers. They're just safer all round.

answered Mar 15 '18 at 7:59
dgnuff
2,247 ● 11 ● 22

2

Just for the sake of completeness for C following the others explanations, not sure for C++.

- pp - pointer to pointer
- p - pointer
- data - the thing pointed, in examples `x`
- **bold** - read-only variable

Pointer

- p data - `int *p;`
- p **data** - `int const *p;`
- **p** data - `int * const p;`
- **p** **data** - `int const * const p;`

Pointer to pointer

1. pp p data - `int **pp;`
2. **pp** p data - `int ** const pp;`
3. pp **p** data - `int * const *pp;`
4. pp p **data** - `int const **pp;`
5. **pp** **p** data - `int * const * const pp;`
6. **pp** p **data** - `int const ** const pp;`
7. pp **p** **data** - `int const * const *pp;`
8. **pp** **p** **data** - `int const * const * const pp;`

```
// Example 1
int x;
x = 10;
int *p = NULL;
p = &x;
int **pp = NULL;
pp = &p;
printf("%d\n", **pp);

// Example 2
int x;
x = 10;
int *p = NULL;
p = &x;
int ** const pp = &p; // Definition must happen during declaration
```

```
printf("%d\n", **pp);

// Example 3
int x;
x = 10;
int * const p = &x; // Definition must happen during declaration
int * const *pp = NULL;
pp = &p;
printf("%d\n", **pp);

// Example 4
int const x = 10; // Definition must happen during declaration
int const * p = NULL;
p = &x;
int const **pp = NULL;
pp = &p;
printf("%d\n", **pp);

// Example 5
int x;
x = 10;
int * const p = &x; // Definition must happen during declaration
int * const * const pp = &p; // Definition must happen during declaration
printf("%d\n", **pp);

// Example 6
int const x = 10; // Definition must happen during declaration
int const *p = NULL;
p = &x;
int const ** const pp = &p; // Definition must happen during declaration
printf("%d\n", **pp);

// Example 7
int const x = 10; // Definition must happen during declaration
int const * const p = &x; // Definition must happen during declaration
int const * const *pp = NULL;
pp = &p;
printf("%d\n", **pp);

// Example 8
int const x = 10; // Definition must happen during declaration
int const * const p = &x; // Definition must happen during declaration
int const * const * const pp = &p; // Definition must happen during declaration
printf("%d\n", **pp);
```

N-levels of Dereference

Just keep going, but may the humanity excommunicate you.

```
int x = 10;
int *p = &x;
int **pp = &p;
int ***ppp = &pp;
int ****pppp = &ppp;

printf("%d \n", ****pppp);
```

edited Jun 8 at 6:50
Felipe Augusto
2,673 ● 4 ● 15 ● 34

answered Feb 17 at 16:30
Undefined Behavior
936 ● 3 ● 14 ● 28

- if `const` is *to the left* of `*`, it refers to the value (it doesn't matter whether it's `const int` or `int const`)
- if `const` is *to the right* of `*`, it refers to the pointer itself

2

- it can be both at the same time

An important point: `const int *p` **does not mean the value you are referring to is constant!!**. It means that you can't change it **through that pointer** (meaning, you can't assign $*p = ...`). The value itself may be changed in other ways. Eg

```
int x = 5;
const int *p = &x;
x = 6; //legal
printf("%d", *p) // prints 6
*p = 7; //error
```

This is meant to be used mostly in function signatures, to guarantee that the function can't accidentally change the arguments passed.

edited Sep 24 at 10:21

answered Jul 31 at 17:55
blue_note
16.1k ● 3 ● 30 ● 47

protected by Sheldore Jul 19 at 13:01

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?