

# Qual è la differenza tra `const int *`, `const int * const` e `int const *`?

Mi incasino sempre su come usare `const int *`, `const int * const` e `int const *` correttamente. Esiste un insieme di regole che definiscono ciò che puoi o non puoi fare?

Voglio sapere tutte le cose da fare e quelle da non fare in termini di incarichi, passaggio alle funzioni, ecc.

Leggilo indietro (come guidato da [Regola in senso orario/a spirale](#)):

- `int*` - puntatore a int
- `int const *` - puntatore a const int
- `int * const` - const puntatore a int
- `int const * const` - const puntatore a const int

Ora il primo `const` può essere su entrambi i lati del tipo, quindi:

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

Se vuoi diventare veramente pazzo puoi fare cose del genere:

- `int **` - puntatore al puntatore su int
- `int ** const` - un puntatore const a un puntatore a un int
- `int * const *` - un puntatore a un puntatore const su un int
- `int const **` - un puntatore a un puntatore a un const int
- `int * const * const` - un puntatore const a un puntatore const su un int
- ...

E per essere sicuri di essere chiari sul significato di `const`

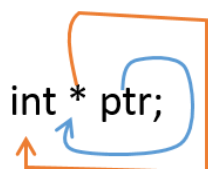
```
const int* foo;
int *const bar; //note, you actually need to set the pointer
                //here because you can't change it later ;)
```

`foo` è un puntatore variabile a un numero intero costante. Ciò ti consente di modificare ciò che hai indicato, ma non il valore a cui punti. Molto spesso questo è visto con stringhe in stile C in cui si ha un puntatore a un `const char`. È possibile modificare la stringa a cui si punta ma non è possibile modificare il contenuto di queste stringhe. Questo è importante quando la stringa stessa si trova nel segmento dati di un programma e non dovrebbe essere modificata.

`bar` è un puntatore costante o fisso a un valore che può essere modificato. Questo è come un riferimento senza lo zucchero sintattico extra. A causa di questo fatto, di solito si utilizza un riferimento in cui si utilizza un puntatore `T* const` a meno che non sia necessario consentire i puntatori `NULL`.

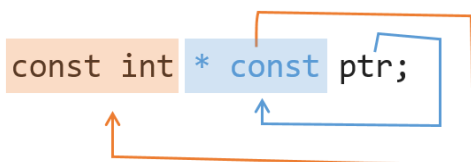
Per coloro che non conoscono la regola orario/a spirale: Inizia dal nome della variabile, spostati in senso orario (in questo caso, vai indietro) al prossimo **puntatore** o **tipo**. Ripeti fino al termine dell'espressione.

ecco una demo:



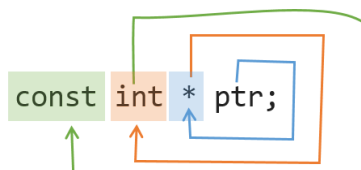
`int * ptr;`

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



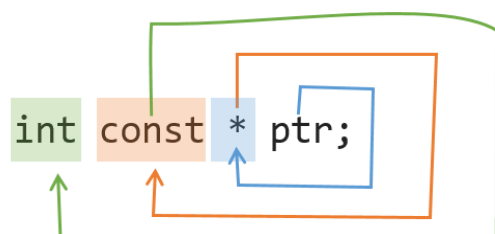
`const int * const ptr;`

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



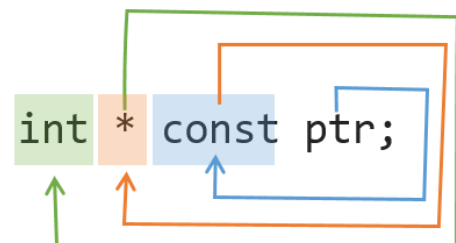
`const int * ptr;`

`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**

Penso che tutto sia già stato risolto qui, ma voglio solo aggiungere che dovresti fare attenzione a **typedefs**! NON sono solo sostituzioni di testo.

Per esempio:

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

Il tipo di `astring` è `char * const`, non `const char *`. Questa è una ragione per cui tendo sempre a mettere `const` alla destra del tipo, e mai all'inizio.

Come quasi tutti hanno sottolineato:

Qual è la differenza tra `const X* p`, `X* const p` e `const X* const p`?

Devi leggere le dichiarazioni dei puntatori da destra a sinistra.

- `const X* p` significa "p punta a una X che è const": l'oggetto X non può essere modificato tramite p.
- `X* const p` significa "p è un puntatore const su una X che non è const": non puoi cambiare il puntatore p stesso, ma puoi cambiare l'oggetto X tramite p.

- `const X* const p` significa "p è un puntatore const su una X che è const": non puoi cambiare il puntatore p stesso, né puoi cambiare l'oggetto X tramite p.

### 1. Riferimento costante:

Un riferimento a una variabile (qui int), che è costante. Passiamo principalmente la variabile come riferimento, perché i riferimenti sono di dimensioni più piccole rispetto al valore effettivo, ma c'è un effetto collaterale e questo è perché è come un alias della variabile attuale. Potremmo cambiare accidentalmente la variabile principale attraverso il nostro pieno accesso all'alias, quindi la rendiamo costante per prevenire questo effetto collaterale.

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

### 2. Puntatori costanti

Una volta che un puntatore costante punta a una variabile, non può puntare a nessuna altra variabile.

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

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

### 3. Puntatore a costante

Un puntatore attraverso il quale non è possibile modificare il valore di una variabile che punta è noto come puntatore a costante.

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

### 4. Puntatore costante a una costante

Un puntatore costante a una costante è un puntatore che non può né modificare l'indirizzo a cui punta e né può modificare il valore mantenuto a quell'indirizzo.

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

Questa domanda mostra **precisamente** perché mi piace fare le cose come ho detto nella mia domanda [è const dopo che il tipo id è accettabile?](#)

In breve, trovo che il modo più semplice per ricordare la regola è che il "const" diventa *after* la cosa a cui si applica. Quindi nella tua domanda "int const \*" significa che l'int è costante, mentre "int \* const" significherebbe che il puntatore è costante.

Se qualcuno decide di metterlo in prima fila (es: "const int \*"), come eccezione speciale in quel caso si applica alla cosa dopo di essa.

A molte persone piace usare quell'eccezione speciale perché pensano che sia più carina. Non mi piace, perché è un'eccezione e quindi confonde le cose.

---

La regola generale è che la parola chiave `const` si applica a ciò che la precede immediatamente. Eccezione, un `const` di partenza si applica a quanto segue.

- `const int*` è uguale a `int const*` e significa **"puntatore a costante int"**.
- `const int* const` è uguale a `int const* const` e significa **"puntatore costante a costante int"**.

**Modifica:** Per le cose da fare e da non fare, se [questa risposta](#) non è sufficiente, potresti essere più preciso su ciò che vuoi?

---

Semplice utilizzo di 'const'

L'uso più semplice è dichiarare una costante nominata. Per fare ciò, si dichiara una costante come se fosse una variabile ma si aggiunga 'const' prima di essa. Si deve inizializzarlo immediatamente nel costruttore, perché, ovviamente, non è possibile impostare il valore in un secondo momento in quanto lo cambierebbe. Per esempio,

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

creerà una costante intera, chiamata "Constant1" in modo inimmaginabile, con il valore 96.

Tali costanti sono utili per i parametri utilizzati nel programma, ma non devono essere modificati dopo la compilazione del programma. Ha un vantaggio per i programmatori rispetto al comando "#define" del preprocessore C in quanto è compreso e utilizzato dal compilatore stesso, non solo sostituito nel testo del programma dal preprocessore prima di raggiungere il compilatore principale, quindi i messaggi di errore sono molto più utili .

Funziona anche con i puntatori, ma bisogna stare attenti dove 'const' per determinare se il puntatore o ciò a cui punta è costante o entrambi. Per esempio,

```
const int * Constant2
```

dichiara che Constant2 è un puntatore variabile su un numero intero costante e

```
int const * Constant2
```

è una sintassi alternativa che fa lo stesso, mentre

```
int * const Constant3
```

dichiara che Constant3 è puntatore costante a un numero intero variabile e

```
int const * const Constant4
```

dichiara che Constant4 è un puntatore costante a un numero intero costante. Fondamentalmente 'const' si applica a tutto ciò che è alla sua sinistra immediata (a parte se non c'è nulla nel qual caso si applica a qualunque sia il suo diritto immediato).

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

Ho avuto lo stesso dubbio di te fino a quando mi sono imbattuto in questo [libro](#) dal Guru del C++ Scott Meyers. Riferisci il terzo elemento in questo libro in cui parla in dettaglio sull'uso di `const`.

Segui questo consiglio

1. Se la parola `const` appare a sinistra dell'asterisco, ciò che viene indicato è costante
2. Se la parola `const` appare a destra dell'asterisco, il puntatore stesso è costante
3. Se `const` appare su entrambi i lati, entrambi sono costanti

È semplice ma difficile. Si noti che è possibile scambiare il qualificatore `const` con qualsiasi tipo di dati (`int`, `char`, `float`, ecc.).

Vediamo gli esempi di seguito.

`const int *p` ==> `*p` è di sola lettura [`p` è un puntatore a un numero intero costante]

`int const *p` ==> `*p` è di sola lettura [`p` è un puntatore a un numero intero costante]

`int *p const` ==> **Wrong** Statement. Il compilatore genera un errore di sintassi.

`int *const p` ==> `p` è di sola lettura [`p` è un puntatore costante a un intero] . Come pointer `p` here is read-only, la dichiarazione e la definizione dovrebbero essere nella stessa posizione.

`const int *p const` ==> **Wrong** Statement. Il compilatore genera un errore di sintassi.

`const int const *p` ==> `*p` è di sola lettura

`const int *const p1` ==> `*p` e `p` sono di sola lettura [`p` è un puntatore costante a un numero intero costante]. Poiché pointer `p` here è di sola lettura, la dichiarazione e la definizione devono essere nello stesso posto.

`int const *p const` ==> **Wrong** Statement. Il compilatore genera un errore di sintassi.

`int const int *p` ==> **Wrong** Statement. Il compilatore genera un errore di sintassi.

`int const const *p` ==> `*p` è di sola lettura ed è equivalente a `int const *p`

`int const *const p` ==> `*p` e `p` sono di sola lettura [`p` è un puntatore costante a un numero intero costante]. Poiché pointer `p` here è di sola lettura, la dichiarazione e la definizione devono essere nello stesso posto.

Il const con l'int su entrambi i lati renderà **puntatore a costante int** .

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

o

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

const dopo '\*' renderà **puntatore costante a int** .

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

In questo caso tutti questi sono **puntatore a numero intero costante** , ma nessuno di questi è puntatore costante.

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

In questo caso tutti sono **puntatore a numero intero costante** e ptr2 è **puntatore costante a numero intero costante**. Ma ptr1 non è un puntatore costante.

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

Questo riguarda principalmente la seconda riga: best practice, assegnazioni, parametri di funzione, ecc.

Pratica generale. Cerca di rendere tutto `const` che puoi. O per dirla in un altro modo, fai in modo che tutto `const` inizi e rimuovi esattamente il set minimo di `consts` necessario per consentire al programma di funzionare. Questo sarà di grande aiuto per raggiungere la correttezza delle costanti, e contribuirà a garantire che i bug sottili non vengano introdotti quando le persone cercano e assegnano cose che non dovrebbero modificare.

Evita `const_cast <>` come la peste. Ci sono uno o due casi d'uso legittimi, ma sono molto pochi e distanti tra loro. Se stai provando a cambiare un oggetto `const`, farai molto meglio a trovare chi ha dichiarato `const` al primo passo e discuti la questione con loro per raggiungere un consenso su cosa dovrebbe accadere.

Il che porta molto ordinatamente in compiti. Puoi assegnare qualcosa solo se non è `const`. Se vuoi assegnare qualcosa che è `const`, vedi sopra. Ricorda che nelle dichiarazioni `int const *foo;` e `int * const bar;` le cose diverse sono `const` - altre risposte qui hanno trattato mirabilmente questo problema, quindi non ci andrò.

Parametri di funzione:

Passa per valore: ad es. `void func(int param)` non ti interessa in un modo o nell'altro nel sito di chiamata. L'argomento può essere fatto che esistono casi d'uso per dichiarare la funzione come `void func(int const param)` ma che non ha alcun effetto sul chiamante, solo sulla funzione stessa, in quanto qualsiasi valore viene passato non può essere modificato dalla funzione durante la chiamata.

Passa per riferimento: ad es. `void func(int &param)` Ora fa la differenza. Come appena dichiarato `func` è permesso di cambiare `param`, e qualsiasi sito di chiamata dovrebbe essere pronto ad affrontare le conseguenze. La modifica della dichiarazione a `void func(int const &param)` modifica il contratto e garantisce che `func` non possa ora cambiare `param`, cioè che cosa viene passato è ciò che verrà restituito. Come altri hanno notato, questo è molto utile per passare a basso costo un oggetto di grandi dimensioni che non si desidera modificare. Il passaggio di un riferimento è molto più economico rispetto al passaggio di un oggetto di grandi dimensioni in base al valore.

Passa col puntatore: ad es. `void func(int *param)` e `void func(int const *param)` Questi due sono praticamente sinonimi delle rispettive controparti di riferimento, con l'avvertenza che la funzione chiamata ora deve controllare `nullptr` a meno che qualche altra garanzia contrattuale assicuri `func` che non riceverà mai un `nullptr` in `param`.

Parere di opinione su questo argomento. Dimostrare la correttezza in un caso come questo è terribilmente difficile, è semplicemente troppo facile commettere un errore. Quindi non correre rischi e controlla sempre i parametri del puntatore per `nullptr`. Ti risparmiassi dolore e sofferenza e non riuscirai a trovare insetti a lungo termine. E per quanto riguarda il costo del controllo, è poco costoso, e nei casi in cui l'analisi statica incorporata nel compilatore può gestirlo, l'ottimizzatore lo eliderà comunque. Attiva Link Time Code Generation per MSVC, o WOPR (credo) per GCC, e otterrai il programma in modo esteso, vale a dire anche nelle chiamate di funzione che attraversano un limite del modulo del codice sorgente.

Alla fine della giornata tutto quanto sopra rende un caso molto solido per preferire sempre i riferimenti ai puntatori. Sono solo più al sicuro.

Per me, la posizione di `const` vale a dire se appare a SINISTRA o DESTRA o sia a SINISTRA che a DESTRA rispetto a `*` mi aiuta a capire il significato attuale.

1. Un `const` alla sinistra di `*` indica che l'oggetto puntato dal puntatore è un oggetto `const`.
2. Un `const` alla DESTRA di `*` indica che il puntatore è un puntatore `const`.

La seguente tabella è tratta dal lettore di corsi di laboratorio standard di programmazione C++ Stanford CS106L.

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

Declaration Syntax	Name	Can reassign?	Can modify pointee?
<code>const Type* myPtr</code>	Pointer-to- <code>const</code>	Yes	No
<code>Type const* myPtr</code>	Pointer-to- <code>const</code>	Yes	No
<code>Type* const myPtr</code>	<code>const</code> pointer	No	Yes
<code>const Type* const myPtr</code>	<code>const</code> pointer-to- <code>const</code>	No	No
<code>Type const* const myPtr</code>	<code>const</code> pointer-to- <code>const</code>	No	No

Solo per completezza per C seguendo le altre spiegazioni, non sono sicuro per C++.

- `pp` - puntatore al puntatore
- `p` - puntatore
- `dati` - la cosa puntata, negli esempi `x`
- **`bold`** - variabile di sola lettura

## Pointer

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

## Puntatore al puntatore

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-livelli di Dereferenza

Continua ad andare, ma l'umanità può scomunicarti.

```

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

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

```