const

/run/media/lijunjie/资料/Learning Video/pointers-arrays-recursion/01_pointers/04_pointers-to-sophisticated-types/03_const_instructions.html

As we discuss pointers to various types, it is good time to introduce the notion of *const* data—data which we tell the compiler we are not allowed to change. When we declare a variable, we can add **const** to its type to specify that the compiler should not allow us to change the data:

```
const int x = 3; // assigning to x is illegal
```

If we try to change the value of *x* , the compiler will produce an error. Declaring data as **const** can be useful, as it removes a certain class of mistakes that we can make in our program: changing things that we should not.

When we have pointers, there are two different things that can be const: the data that the pointer points at (what is in the box at the end of the arrow), or the pointer itself (where it points). If we write:

```
const int * p = &x;
```

We have declared *p* as a pointer to a **const int** —that is, *p* points at a int, and we are not allowed to change that int. We can change where *p* points (*e.g., p = &y* ; is legal—if *y* is an int). However, changing the value in the box that *p* points at (*e.g., *p = 4* ;) is illegal—we have said that the int which *p* points at is **const** . If we do try to write something like **p = 4* ;, we will get a compiler error like this:

```
assignment of read-only location '*p'
```

We can achieve exactly the same effect by writing:

```
int const * p = &x; // same as const int * p
```

If we want to specify that we can change *p, but not p itself, we would write

```
int * const p = &x;
```

This declaration says that *p* is a **const** pointer to a (modifiable) **int** . Writing **p=4* ; would be legal, but writing *p =&y* ; would be illegal. If we so desire, we can combine both to prevent changing either where the pointer points, or the value it points at:

```
const int * const p = &x;
```

The same principle applies to pointers to pointers (to pointers to pointers...). For example, with an int **, we have the following combinations:

	Can we change **p	Can we change *p	Can we change p

	Can we change **p	Can we change *p	Can we change p
int ** p	Yes	Yes	Yes
const int ** p	No	Yes	Yes
int * const * p	Yes	No	Yes
int ** const p	Yes	Yes	No
const int * const * p	No	No	Yes
const int ** const p	No	Yes	No
int * const * const p	Yes	No	No
const int * const * const p	No	No	No

Note that a declaration of **const** tells the compiler to give us an error only if we try to change the data through the variable declared as **const** , or perform an operation where the **const** -ness gets dropped. For example, the following is legal:

```
int x = 3;
const int * p = &x;
x = 4;
```

Here, we are not allowed to change **p* , however, the value we find at **p* can still be changed by assigning to *x* (since *x* is not **const** , it is not an error to assign to it). However, if we write:

```
const int y = 3;
int * q = &y;
*q = 4;
```

then we will receive a compiler warning (which we should treat as an error):

```
initialization discards 'const' qualifier from pointer target type [enabled by
default]
```

The error is on line 2, in which we assign *&y* (which has type **const int** *) to q (which has type **int** *)—discarding the **const** *qualifier* (const is called a qualifier because it modifies a type). This snippet of code is an error because * *q=4* ; (on line 3) would be perfectly legal

(q is not declared with the **const** qualifier on the type it points to), but would do something we have explicitly said we do not want to do: modify the value of y .

Novice programmers often express some confusion at the fact that the first example is legal and the second is not—in both cases, we have tried to declare a variable and a pointer (to that variable), with one **const** and the other not. We have then tried to modify the value in that box through whichever is not **const** —but one is ok, and the other is not. These rules do actually make sense: in the second case, we have said “y cannot be modified” then we try to say “q is a pointer (which I can use to modify or read a value) to y”—that clearly violates what we said about “y” (that it cannot be modified). In the first case, however, we are saying “x is a variable that can be modified” and then “p is a pointer, which we can only use to read the value it points at, not modify it”—this does not impose new (nor violate existing) restrictions on “x”, only tells us what we can and cannot do with “p”.