Tip 55: Mark input pointers with `const`

Ben Klemens

19 January 2011

level: Not just writing code: writing code with style
purpose: Clarify your intent

Early in your life, you learned that *copies* of input data are passed to functions, but you can still have functions that change input data by sending in a *copy of a pointer* to data.

When you see that an input is plain, not-pointer data, then you know that the caller's original version of the variable won't change. When you see a pointer input, it's unclear. Lists and strings are naturally pointers, so the pointer input could be data to be modified, or it could just be a string.

The `const` keyword is a literary device for you, the author, to make your code more readable. It is a *type modifier* indicating that the data pointed to by the input pointer will not change over the course of the function.

As a literary device, however, it is problematic. The compiler does not lock down the data being pointed to against all modification. In the following example, `a` and `b` point to the same data, but because `b` is not `const` in the header for `set_elmt`, it can change the third element of the `a` array.

```
void set_elmt(int const *a, int *b){
    b[0] = 3;
}

int main(){
    int const a[10];
    int *b = (int*)a;
    set_elmt(a, b);
}
```

So it is a literary device, not a lock on the data.

noun-adjective form You'll notice that I write `int const` instead of `const int`. ¿What is this, Spanish? We English speakers like to put the adjective before the noun. Both are valid in the syntax, and there's an easy grammatical resolution: read `const` as *that is constant*. Thus:
`int const *` = a pointer to (an integer that is constant)

`int * const` = (a pointer to an integer) that is constant
`int * const *` = a pointer to ((a pointer to an integer) that is constant)
`int const *const * const` = a pointer that is constant to a pointer that is constant to an integer that is constant

These are all different things.

Tension In practice, you will find that the keyword sometimes creates tension that needs to be resolved, when you have a pointer that is marked const, but want to send it as an input to a function that does not have a `const` marker in the right place. Maybe the author read somewhere that the keyword was too much trouble, or believes the chatter about how shorter code is always better code, or just forgot.

Before proceeding, you'll have to ask yourself if there is any way in which the pointer could change in the `const`-less function being called. There may be an edge case where something gets changed, or some other odd reason. This is stuff worth knowing anyway.

If you've established that the function does not break the promise of `const`-ness that you made with your pointer, then it is entirely appropriate to cheat and cast your `const` pointer to a non-const for the sake of quieting the compiler. I actually did that above with the line about `int *b = (int*)a`. Without the cast from `const int*` to `int*`, the compiler would have complained.

When sending a const pointer to a function without a const in the header, we can use the same casting to quiet the compiler:

```
//no const in the header this time:
void set_elmt(int *a, int *b){ b[0] = 3; }

int main(){
    int const a[10];
    int *b = (int *) a;
    set_elmt((int*)b, b);
}
```

The rule seems reasonable to me. You can override the compiler's const-checking, as long as you are explicit about it and indicate that you know what you are doing.

If you are worried that the function you are calling won't fulfill your promise of `const`-ness, then you can take one step further and make a full copy of the data, not just an alias. You should get the same result from the function either way.

Constness lacks depth Let us say that we have a structure, `typedef struct {int *list, int list_leng` and we send in a pointer to this structure. Are the elements of the structure const or not?

Let's try it: the following program generates a struct with two pointers, and in `check_all_counters` that struct becomes `const`, yet when we send one of the pointers held by the structure to the const-less subfunction, the compiler doesn't complain.

```
typedef struct {
    int *counter1, *counter2;
} counters;

void check_counter(int *ctr){ assert(*ctr !=0); }

void check_all_counters(counters const *in){
    check_counter(in->counter2);
}

int main(){
    counters cc = {.counter1=malloc(sizeof(int)), .counter2=malloc(sizeof(int))}
    *cc.counter1= *cc.counter2=1;
    check_all_counters(&cc);
}
```

If you really need to protect an intermediate level in your hierarchy of types, like you have a list of pointers that can't move but the data in the pointers can change, then maybe you're better off just documenting the promise. If you do `const` only an intermediate level in the hierarchy, do your readers a favor and document that too.

So there are the problems: if you send a const pointer to a non-const function, you'll need to cast; if you have a const struct, its pointer elements are not const, and so you'll have to keep an eye on them without the compiler's help.

As literature goes, that isn't all that problematic, and the recommendation that you add `const` to your headers as often as appropriate still stands—don't just grumble about how the people who came before you didn't provide the right headers. After all, some day somebody else will use your code, and you don't want them grumbling about how they can't use the `const` keyword because your functions don't have the right headers.