**Pointers and const.**

Pointers can be used to access a variable by its address, and this access may include modifying the value pointed. But it is also possible to declare pointers that can access the pointed value to read it, but not to modify it. For this, it is enough with qualifying the type pointed to by the pointer as const. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 | int x;  int y = 10;  const int \* p = &y;  x = \*p; // ok: reading p  \*p = x; // error: modifying p, which is const-qualified |  |

Here *p* points to a variable, but points to it in a const-qualified manner, meaning that it can read the value pointed, but it cannot modify it. Note also, that the expression *&y* is of type *int\**, but this is assigned to a pointer of type *const int\**. This is allowed: a pointer to non-const can be implicitly converted to a pointer to const. But not the other way around! As a safety feature, pointers to const are not implicitly convertible to pointers to non-const.

One of the use cases of pointers to const elements is as function parameters: a function that takes a pointer to non-const as parameter can modify the value passed as argument, while a function that takes a pointer to const as parameter cannot.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 | // pointers as arguments:  #include <iostream>  using namespace std;  void increment\_all (int\* start, int\* stop)  {  int \* current = start;  while (current != stop) {  ++(\*current); // increment value pointed  ++current; // increment pointer  }  }  void print\_all (const int\* start, const int\* stop)  {  const int \* current = start;  while (current != stop) {  cout << \*current << '\n';  ++current; // increment pointer  }  }  int main ()  {  int numbers[] = {10,20,30};  increment\_all (numbers,numbers+3);  print\_all (numbers,numbers+3);  return 0;  } | 11  21  31 | [Edit & Run](https://www32.cplusplus.com/doc/tutorial/pointers/) |

Note that *print\_all* uses pointers that point to constant elements. These pointers point to constant content they cannot modify, but they are not constant themselves: i.e., the pointers can still be incremented or assigned different addresses, although they cannot modify the content they point to.

And this is where a second dimension to constness is added to pointers: Pointers can also be themselves const. And this is specified by appending const to the pointed type (after the asterisk):

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 | int x;  int \*p1 = &x; // non-const pointer to non-const int  const int \*p2 = &x; // non-const pointer to const int  int \*const p3 = &x; // const pointer to non-const int  const int \*const p4 = &x; // const pointer to const int |  |

The syntax with const and pointers is definitely tricky, and recognizing the cases that best suit each use tends to require some experience. In any case, it is important to get constness with pointers (and references) right sooner rather than later, but you should not worry too much about grasping everything if this is the first time you are exposed to the mix of const and pointers. More use cases will show up in coming chapters.

To add a little bit more confusion to the syntax of const with pointers, the const qualifier can either precede or follow the pointed type, with the exact same meaning:

|  |  |  |
| --- | --- | --- |
| 1 2 | const int \* p2a = &x; // non-const pointer to const int  int const \* p2b = &x; // also non-const pointer to const int |  |

As with the spaces surrounding the asterisk, the order of const in this case is simply a matter of style. This chapter uses a prefix const, as for historical reasons this seems to be more extended, but both are exactly equivalent. The merits of each style are still intensely debated on the internet.