const and constexpr

J.-C. Chappelier, J. Sam

version 1.0 of september 2016

The C++11 standard introduced the reserved word "constexpr" which is significantly different from the reserved word "const", but their similarity is sometimes confusing. Let's try to clarify their jobs.

1 const does not mean “constant”

As a reminder (course), the reserved word “const” qualifies a variable name that [1] to indicate through this **name ,** the value cannot be modified.

For example :

```
int const i(3);
```

prevents changing the value of i: the following code will not compile:

```
i = 5;
```

Why did you specify “through **this** name”?

This nuance uses notions presented at the end of the reference course. [2] : pointers and Let's just give an example here:

```
int const i(3); int* ptr(&i);
```

ptr points to i, but is not itself const.
(Warning! This is precisely bad programming!... but it is possible [3] .)

1. This is the only usage seen so far in this course and therefore the only one discussed here. Other uses of const will be discussed in the "Introduction to Object-Oriented Programming" course.
2. Do not dwell on it if you read this supplement before having seen the course on pointers and references. You can always come back to it when you want.
3. This is so bad programming that some compilers won't compile such code unless they add options, like -fpermissive to the latest versions of g++

SO

i = 5;

still won't compile, but on the other hand

*ptr = 5;

is entirely possible and will change the value of i! The const on i therefore does not mean that the value of i cannot change absolutely, but that it cannot change through the name i.

const therefore does not mean "constant", but "read-only" (implied, "not writable/assignable").

2 A const value is not necessarily known at compile time

The qualifier "const" is a "dynamic (runtime)" notion in the sense that it can apply to values not known at compile-time ("compile-time").

Let's take an example: let's suppose that we ask the user of our program for a value i and that, once entered, the value of i will not be modified in the sequel. We could of course write:

```
int i; cout
<< "Enter a value: "; cin >> i;
```

but this does not underline, does not force, the fact that the value i is not to be modified any more in the sequel.

To make this clear, it would be better to write:

```
read int;
cout << "Enter a value: "; cin >> read; const int
i(read); // i is
initialized with the read value
```

It is clear from this example that: —

once given, the value of i cannot be modified; — yet the value that i actually takes is not known at the time of com pound the program.

3 constexpr means "known at compile time and constant"

It is precisely to compensate for the two subtleties of the reserved word const pointed out by the two previous sections that the C++ committee decided in the 2011 version to introduce

a new reserved word, constexpr, which means "known at compile time and constant ».

A variable (or more broadly an expression, but we won't go into that here) can be qualified as constexpr if precisely these two conditions are met: • we know its value at compile time; — this value will not change during the program.

For example :

```
constexpr double pi(3.141592653589793238463);
```

The fact that the value must be known at compile time of course forbids using constexpr in the example in the previous section:

```
read int;
cout << "Enter a value: "; cin >> read; constexpr
int i(read);
```

Such code does not compile.

Now that you have read this add-in, we advise you to use constexpr wherever both conditions for its application are met.