Summary of “Const as a promise”

Marc Paterno

16 June 2020

Please interrupt

- It is harder to have a “normal” group conversation over video than it is in person, but I would like us to try.

Section 1

My summary

constexpr vs *const*

- Prefer *constexpr*
  - when defining *symbolic constants*, and
  - when you want immutable data.
- Use *const* to prevent modification of data that is otherwise modifiable.
- *const* is mostly for interfaces.
- *const* turns runtime bugs into compile-time errors.

Retrofitting is hard

- Use *const* proactively.
- Declare *const* everything that can be so declared.
  - The resulting code is easier to reason about.
  - It reduces bugs due to reading uninitialized values.

All the declarations. . .

- Mr. Saks spent a great deal of time on understanding declarations.
- See https://en.cppreference.com/w/cpp/language/declarations for a summary.
- I like to add parentheses to make reading easier — so that I don't have to work so hard to read my own code.

Read variable declaration right to left

```
double       *       p1 = ... // read/write mutable pointer
double const*       p2 = ... // readonly mutable pointer
double       * const p3 = ... // read/write immutable pointer
double const* const p4 = ... // readonly immutable pointer
```

Read variable declaration right to left

```
double* p1 = ...              // read/write mutable pointer
double const* p2 = ...        // readonly mutable pointer
double* const p3 = ...        // read/write immutable pointer
double const* const p4 = ... // readonly immutable pointer
```

Useful “tricks” for initializing values

- Use an *immediately invoked lambda expression*.
- Use a *structured binding* and rely on the return slot.

Immediately invoked function expression (IIFE)

- They look like:

```
[/* capture list */](/* parameters */){ /* body */ }(/* arguments */);
```

- Especially useful for use in the *member initializer list* of a constructor.
- Use to wrap up a function with an output parameter.
- A concrete example:

```
char const* some_file = ...;
int const nblocks = [](char const* fname) {
  struct stat data;
  auto rc = stat(fname, &data);
  return (rc == 0) ? data.st_blksize : -1;
}(some_file);
// now go on to use nblocks
```

Avoid output arguments

- Avoid output arguments; they prevent initialization and thus conflict with *const*.
- We can rely on “return slot” (see Arthur O’Dwyer’s talk **Complete guide to *return x***).

```
// assume my_func is a function to be integrated.
// Ugly output parameters
double val = 0., error = 0.;       // can not be const
int ncalls = 0, status = 0;        // can not be const
int const maxcalls = 1000 * 1000; // prefer const
UglyIntegrate(my_func, maxcalls, val, error, ncalls, status);
```

```
// Tidier interface
int const maxcalls = 1000 * 10000;
auto const [val, error, ncalls, status] = TidyIntegrate(my_func, maxcalls);
```

Section 2

Discussion