Date: 18th April 2019

Project: Addition of a filter to recursive_directory_iterator Standard Proposal

Reply-to: Noel Tchidjo Moyo ⟨ noel.tchidjomoyo@murex.com ⟩

Table of Contents

I	Introduction	2
II	Motivation and Scope	4
III	Design Decisions	4
IV	Impact On the Standard	10
V	Technical Specifications	10
VI	Acknowledgements	15

Bibliography **16**

I Introduction

With the constant increase of complex applications, which have several years of life, it becomes necessary for the tools processing the files of these applications to be able to quickly list the types of files that interest them.
The object type recursive_directory_iterator introduced in the C++ 17 standard, allows to list the entries of a directory and its subdirectories. Its use is usually coupled with a test to check if the extension of the file corresponds to that which interests us. This use can in some complex projects suffer of performance issues since we will provide to users many unwanted entries. This is the reason why we have investigated two API changes that allow to pass directly a filter as parameter of recursive_directory_iterator constructor.
That is :

- The addition of a new recursive_directory_iterator constructor with a regular expression parameter
- The addition of a new recursive_directory_iterator constructor with a user-provided lambda parameter.

Analysis of code expressiveness and time performance in both approaches, shows that the user-provided lambda version provides significant benefits to C++ community. Finally these results justify our choice to propose for the standardization the addition of a predicate concept as filter to recursive_directory_iterator constructor.

Need & Prior Art

The recursive_directory_iterator object type is an InputIterator that allows to iterate over the entries of a directory, and recursively over all entries of its subdirectories. It is defined under <filesystem> header and has seven constructors.

```
recursive_directory_iterator() noexcept;

recursive_directory_iterator(
const recursive_directory_iterator& rhs );

recursive_directory_iterator(
recursive_directory_iterator&& rhs ) noexcept;

explicit recursive_directory_iterator(
```

```
const std::filesystem::path& p );

recursive_directory_iterator(
const std::filesystem::path& p,
std::filesystem::directory_options options );

recursive_directory_iterator(
const std::filesystem::path& p,
std::filesystem::directory_options options,
std::error_code& ec );

recursive_directory_iterator(
const std::filesystem::path& p,
std::error_code& ec );
```

Unfortunately, none of these constructors allows to provide a filter, that can be apply systematically on each entry, in order to provide user with entries that directly satisfy the filter. Currently, users rely on a `if` statement in a `for` loop in order to apply their filter (as in below code).

```
for(auto& entry:
std::filesystem::recursive_directory_iterator(folder))
{
  auto extension = entry.path().extension();
  auto ext = extension.c_str();

  if( strcmp(ext, ".c") == 0
  || strcmp(ext, ".h") == 0
  || strcmp(ext, ".cpp") == 0
  || strcmp(ext, ".hpp") == 0 )
  {
   do_some_work(entry.path().filename().c_str());
  }
}
```

However, filter could be complex, and in huge projects, with very large files set this usage could suffer from performance issues.

II Motivation and Scope

Why is this important? What kinds of problems does it address?

The motivation for the addition of a filter to recursive_directory_iterator is to provide better code expressiveness and improve time performance when using recursive_directory_iterator to list entries on a complex project.

What is the intended user community?

The intended community is anyone interested in operating on entries of complex project with very large entries set.

III Design Decisions

In order to evaluate what could be the best approach, we have implemented a new constructor with a regex, and another one with a std :: function in two compilers: GCC7-3 and Clang5. The resulting signature of recursive_directory_iterator with a user-provided lambda is

```
explicit recursive_directory_iterator(
const std::filesystem::path& __p,
const std::function<bool(const char* )> lambda,
const std::filesystem::entry_restriction& po)
```

The parameter std :: filesystem :: entry_restriction is also added in order to specify the type of entry on which the filter should be apply. This corresponds also to the type of entry to be returned by the iterator. The definition of this type is

```
enum class entry_restriction
{
 file ,
 directory ,
 symlink ,
 none
};
```

When transforming the above sample example by using this new signature, we obtain this

```
auto filter = [](const char* filename)
{
const char* ext = get_filename_ext(filename);
return strcmp(ext, ".c") == 0
|| strcmp(ext, ".h") == 0
|| strcmp(ext, ".cpp") == 0
|| strcmp(ext, ".hpp") == 0;
};

for(auto& entry :
std::filesystem::recursive_directory_iterator(folder,
        filter,
        std::filesystem::entry_restriction::file ))
{
  do_some_work(entry.path().filename().c_str());
}
```

We also assess the possibility to add a regular expression as filter with the below signature

```
explicit recursive_directory_iterator(
const std::filesystem::path& __p,
const std::regex& reg,
const std::filesystem::entry_restriction& po)
```

The above sample example will become

```
for(auto& entry:
std::filesystem::recursive_directory_iterator(folder,
        std::regex(".*\\.h|.*\\.c|.*\\.cpp|.*\\.hpp"),
        std::filesystem::entry_restriction::file))
{
        do_some_work(entry.path().filename().c_str());
}
```

Expressiveness

Clearly both transformations improve code readability.

Time performance

For time performance measurements, we have run a directory traverser program over several large open source file sets. We choose to present here time performance on three of them: Linux project file set, Gecko project file set and Hadoop project file set.
We chose to present the results when running on these 3 projects because they allow to have benchmarks in the case where we have a majority of entries that pass the filter (Linux), the case where we have a majority of entries that do not pass the filter (Hadoop), and where there is some balance between those who pass and those who do not pass the filter (Gecko). Also, note that these tests have been performed on Unix (for GCC implementation) and Windows (for CLANG implementation) platforms.

Tests program code and compiler's patches are available on https://github.com/bonpiedlaroute/cppcon2018

Here are some results on Unix platform (Smallest bar is the faster)

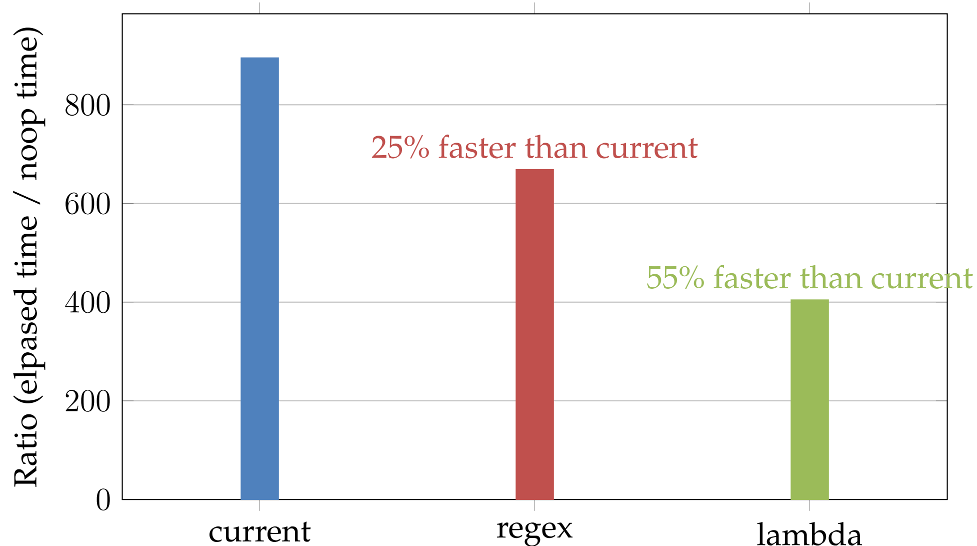

Figure 1: Unix - Executions over Gecko file set

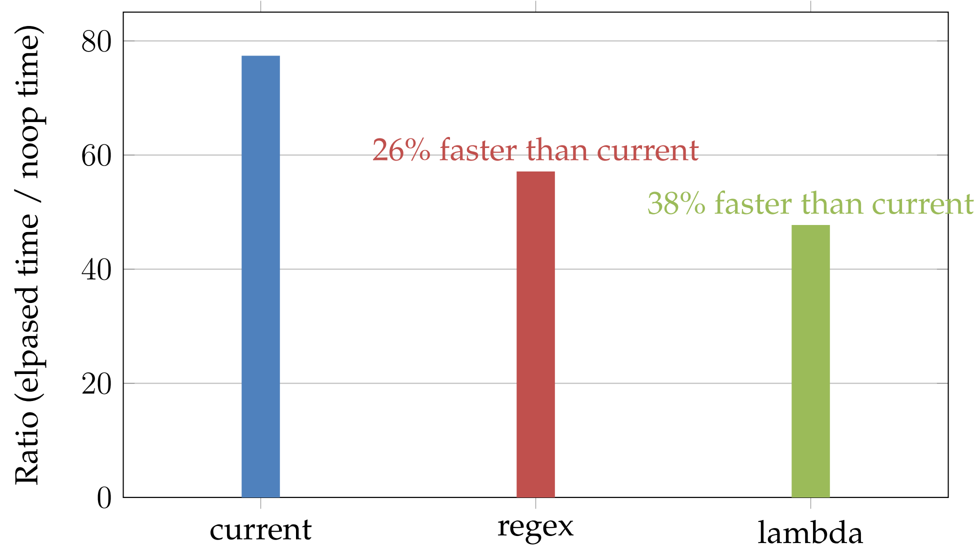

Figure 2: Unix - Executions over Hadoop file set

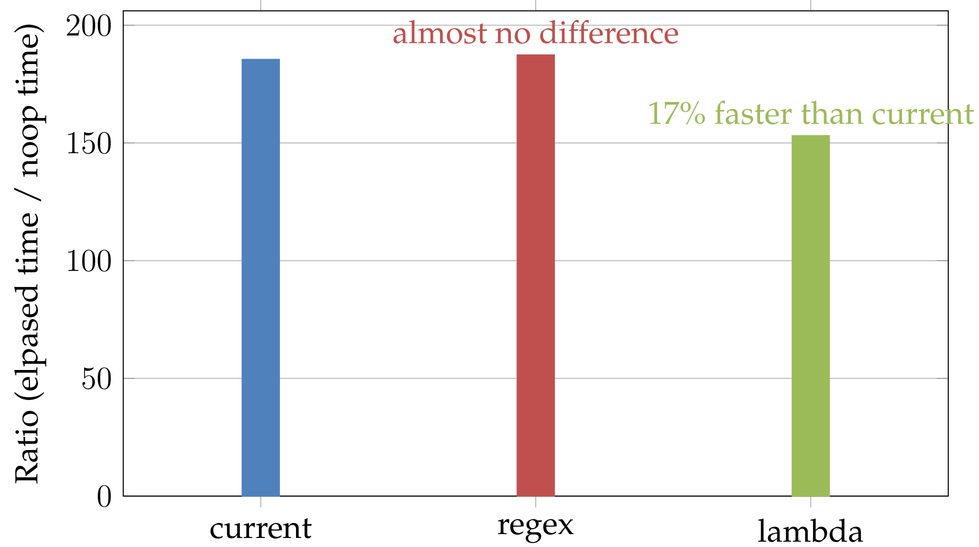

Figure 3: Unix - Executions over Linux file set

Running the same tests on windows platform provides the below results

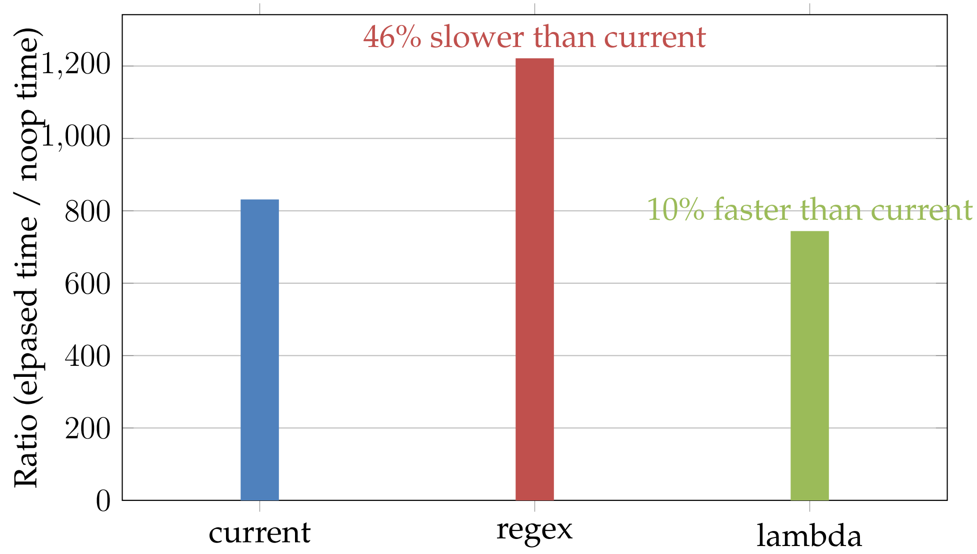

Figure 4: Windows - Executions over Gecko file set

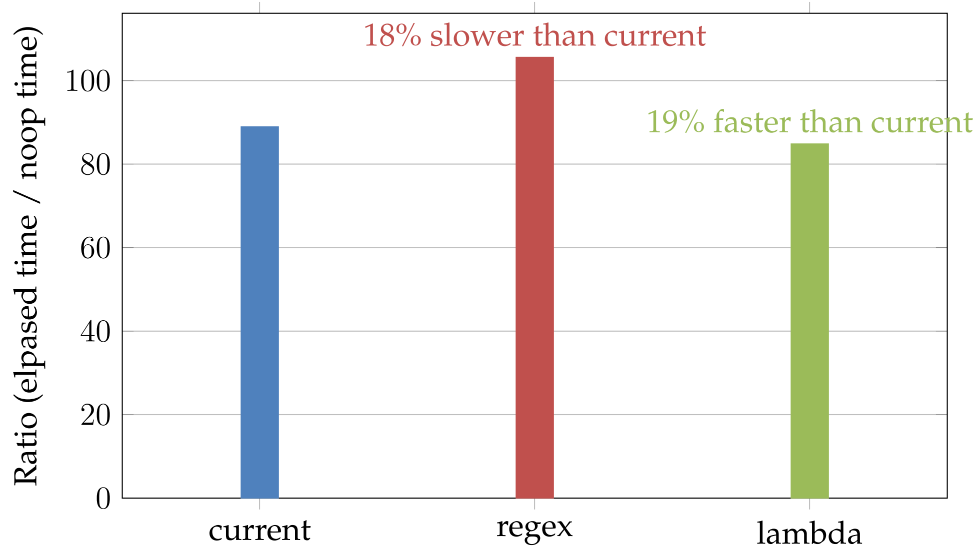

Figure 5: Windows - Executions over Hadoop file set

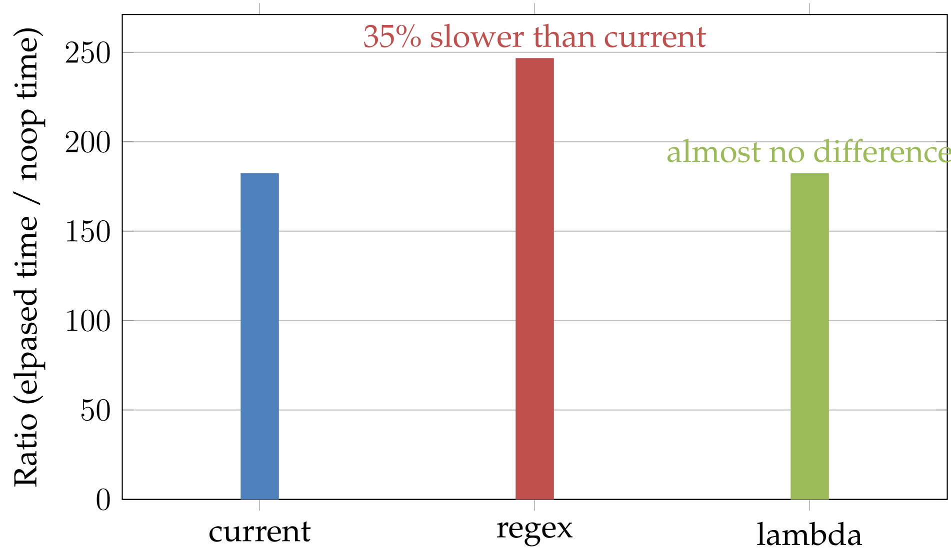

Figure 6: Windows - Executions over Linux file set

Analysis and Discussions

Above results show that the lambda version has always better time performance than the regex version. This in fact makes sense since the user-provided lambda is already tuned to the user specific need. When comparing to the current version the lambda version has generally better time performance, except in rare cases as Figure 6 where the gain is not significant.
We compared the implementations of both versions on Windows in order to understand the reason for the poor performance of the regex version on Windows. After many measurements, the significant difference in execution time appeared during calls of std :: regex_match compared to calls to the lambda. A call to std :: regex_match took 1000 times more execution time than a call to the lambda on Windows. Thus, the poor performance of the regex version on Windows compare to Unix is due to std :: regex_match Windows visual studio implementation. We also note that adding std :: regex as parameter to std :: recursive_directory_iterator also add a dependency between <filesystem> and <regex>. The last point that justifies the choice of a lambda (that we have finally generalized to a predicate concept) over a regular expression is that with the lambda version the filter can easily evolve and be more complex. Users can for example check for file last modification time, or for the presence of a specific pattern in the contents of the file etc. This will satisfy more community users need.

Passing a path or passing const char* to the predicate concept

When sending the first draft proposal, we have proposed a new constructor with filter as a predicate concept on a std :: filesystem :: path. Some community members argue that we have not performed profiling with a lambda taking std :: filesystem :: path as a parameter (profiling was done with a lambda with `const char*`), and that using a std :: filesystem :: path instead of `const char*` will reduce performance gain. We have then realized profiling on recursive_director_iterator constructor with a lambda that takes a std :: filesystem :: path to check if constructing a path has not a negligible cost. Results show that, for actual compilers implementation, the cost of constructing a path is not negligible.
That is the reason why we proposed both solutions: the addition of a new recursive_directory_iterator constructor with a filter as a predicate concept on std :: filesystem :: path (for more generality), and the addition of recursive_directory_iterator constructor with a filter as predicate concept on `const char *` (mostly for performance gain)

IV Impact On the Standard

Although we only investigated on recursive_directory_iterator, the filter addition will work in the same way for directory_iterator object type. Thus, this proposal implies new constructors for each existing standard object types recursive_directory_iterator and directory_iterator.
It is also worth mentioning that adding a filter with a predicate concept does not add any dependency to <filesystem> standard module, since concepts are core language feature. Also, note that the usage of a predicate concept implies that the filter cannot have a mutable state.

V Technical Specifications

```
template <class F>
concept EntryPredicate =
CopyConstructible<F> &&
Predicate<F&, const char*>;

class recursive_directory_iterator {
public:
template <EntryPredicate P>
```

```
explicit recursive_directory_iterator(
const std::filesystem::path& path,
P predicate);

template <EntryPredicate P>
explicit recursive_directory_iterator(
const std::filesystem::path& path,
P predicate,
std::error_code& ec);

template <EntryPredicate P>
explicit recursive_directory_iterator(
const std::filesystem::path& path,
P predicate,
const std::filesystem::directory_options& options);

template <EntryPredicate P>
explicit recursive_directory_iterator(
const std::filesystem::path& path,
P predicate,
const std::filesystem::directory_options& options,
std::error_code& ec);
};



class directory_iterator {
public:
template <EntryPredicate P>
explicit directory_iterator(
const std::filesystem::path& path,
P predicate);

template <EntryPredicate P>
explicit directory_iterator(
const std::filesystem::path& path,
P predicate,
std::error_code& ec);

template <EntryPredicate P>
```

```
explicit directory_iterator(
const std::filesystem::path& path,
P predicate,
const std::filesystem::directory_options& options);

template <EntryPredicate P>
explicit directory_iterator(
const std::filesystem::path& path,
P predicate,
const std::filesystem::directory_options& options,
std::error_code& ec);
};
```

```
template <class F>
concept EntryPathPredicate =
CopyConstructible<F> &&
Predicate<F&, const std::filesystem::path&>;

class recursive_directory_iterator {
public:
template <EntryPathPredicate P>
explicit recursive_directory_iterator(
const std::filesystem::path& path,
P predicate);

template <EntryPathPredicate P>
explicit recursive_directory_iterator(
const std::filesystem::path& path,
P predicate,
std::error_code& ec);

template <EntryPathPredicate P>
explicit recursive_directory_iterator(
const std::filesystem::path& path,
P predicate,
const std::filesystem::directory_options& options);

template <EntryPathPredicate P>
explicit recursive_directory_iterator(
const std::filesystem::path& path,
```

```
P predicate,
const std::filesystem::directory_options& options,
std::error_code& ec);
};



class directory_iterator {
public:
template <EntryPathPredicate P>
explicit directory_iterator(
const std::filesystem::path& path,
P predicate);

template <EntryPathPredicate P>
explicit directory_iterator(
const std::filesystem::path& path,
P predicate,
std::error_code& ec);

template <EntryPathPredicate P>
explicit directory_iterator(
const std::filesystem::path& path,
P predicate,
const std::filesystem::directory_options& options);

template <EntryPathPredicate P>
explicit directory_iterator(
const std::filesystem::path& path,
P predicate,
const std::filesystem::directory_options& options,
std::error_code& ec);
};
```

```
enum class directory_options {
none = 0 /*unspecified*/,
follow_directory_symlink,
skip_permission_denied,
/* Addition */
filter_only_file,
```

```
filter_only_directory,
filter_only_symlink
};
```

Name	Value	Meaning
filter_only_file	3	apply filter on files and provide only entries that are files
filter_only_directory	4	apply filter on directories and provide only entries that are directories
filter_only_symlinks	5	apply filter on symlinks and provide only entries that are symlinks

Table 1: Meaning of new values for enum class directory_options

Usage example

```
auto filter = [](const char* entry)
{
auto ext = get_filename_ext(entry);
return strcmp(ext, ".c") == 0
|| strcmp(ext, ".h") == 0
|| strcmp(ext, ".cpp") == 0
|| strcmp(ext, ".hpp") == 0;
};

for(auto& entry :
std::filesystem::recursive_directory_iterator(folder,
filter,
std::filesystem::directory_options::filter_only_file ))
{
auto filename = entry.path().filename();
do_some_work(filename.c_str());
}
```

VI Acknowledgements

I would like to thank my colleagues: Jeremy Demeule, Jonathan Boccara, Frederic Tingaud, Georges Schumacher, and all C++ community members for their insightful comments and encouragements in the writing of this proposal.

Bibliography

[1] cppreference.com , *Cpp Reference recursive_directory_iterator*, 2018.

[2] Noel Tchidjo Moyo, Feedback on practical use of std :: recursive_directory_iterator , Poster, CppCon, September 2018.