Boost.Stopwatches 0.2.0

Vicente J. Botet Escriba

Table of Contents

Warning

Stopwatches is not part of the Boost libraries.

Overview

How to Use This Documentation

This documentation makes use of the following naming and formatting conventions.

- Code is in `fixed width font` and is syntax-highlighted.
- Replaceable text that you will need to supply is in *`italics`*.
- Free functions are rendered in the code font followed by (), as in `free_function()`.
- If a name refers to a class template, it is specified like this: `class_template<>`; that is, it is in code font and its name is followed by <> to indicate that it is a class template.

- If a name refers to a function-like macro, it is specified like this: `MACRO()`; that is, it is uppercase in code font and its name is followed by `()` to indicate that it is a function-like macro. Object-like macros appear without the trailing `()`.
- Names that refer to *concepts* in the generic programming sense are specified in CamelCase.

Note

In addition, notes such as this one specify non-essential information that provides additional background or rationale.

Finally, you can mentally add the following to any code fragments in this document:

```
// Include all of Stopwatches files
#include <boost/stopwatches.hpp>
using namespace boost::chrono;
using namespace boost::stopwatches;
```

Motivation

Measuring elapsed time

Knowing how long a program, a function or a specific code block takes to execute is useful in both test and production environments. **Boost.Stopwatches** introduces the `Stopwatch` concept which is a mechanism to measure the elapsed time. A Stopwatch allows to start, stop, suspend and resume measuring the elapsed time. `stopwatch<>` is the basic model of `Stopwatch`.

Reporting elapsed time

It is often necessary to report elapsed time on a user display or in a log file. `stopwatch_reporter<>` provides a runtime reporting mechanism for this purpose which can be invoked in just one line of code.

```
using namespace boost::chrono;
using namespace boost::stopwatches;
int main()
{
    stopwatch_reporter<stopwatch<process_cpu_clock> > _;
  // ...
}
```

Will produce the following output

```
real 0.034s, cpu 0.031s (93.0%), user 0.031s, system 0.000s
```

As this is one of the expression more commonly use, the library provides a stopclock shortcut so the preceding can be writen as

```
using namespace boost::stopwatches;
int main()
{
    stopclock<> _;
  // ...
}
```

How reliable are these measures?

There are a number of things that can lead to unreliable measurement (see here for more details), but they mostly amount to reporting overhead. Boost.Chrono provides two ways to improve reliability of time measurements. A `stopwatch_accumulator` only reports statistics once all measurements have been acquired, which removes reporting overhead from the measurements. The other approach

is to use a `SuspendibleClock` such that the reporting overhead can be ignored by suspending elapsed time tracking during reporting operations

Description

On top of the standard facilities of **Boost.Chrono**, **Boost.Stopwatches** provides:

- Stopwatches: A facility to measure elapsed time with the ability to start, stop, suspend, or resume measurement.
  - `Stopwatch` concept
  - Scoped helper classes allowing to pairwise start/stop operations, suspend/resume and resume/suspend a `Stopwatch`.
  - `stopwatch`, model of `Stopwatch` capturing elapsed `Clock` times.
  - `stopwatch_accumulator`, model of `Stopwatch` capturing cummulated elapsed Clock times.
- Stopclocks: a complete time reporting package that can be invoked in a single line of code.
  - `stopwatch_reporter`, convenient reporting to an output stream (including wide char streams) of the elapsed time of models of `Stopwatch` results.
  - `stopclock<Clock>` shortcut of `stopwatch_reporter<stopwatch<Clock>>`

Users'Guide

Getting Started

Installing Boost.Stopwatches

Getting Boost.Stopwatches

You can get the last stable release of **Boost.Stopwatches** by downloading `stopwatches.zip` from the Boost Vault.

You can also access the latest (unstable?) state from the Boost Sandbox, directories boost/stopwatches and libs/stopwatches. Just go to here and follow the instructions there for anonymous SVN access.

Where to install Boost.Stopwatches?

The simple way is to decompress (or checkout from SVN) the file in your BOOST_ROOT directory.

Othesewise, if you decompress in a different directory, you will need to comment some lines, and uncomment and change others in the build/Jamfile and test/Jamfile. Sorry for this, but I have not reached yet to write a Jamfile that is able to work in both environements and use the BOOST_ROOT variable. Any help is welcome.

Building Boost.Stopwatches

Boost.Stopwatches is a header only library.

Requirements

Boost.Stopwatches depends on some Boost libraries. For these specific parts you must use either Boost version 1.44.0 or the version in SVN trunk (even if older versions should works also).

In particular, **Boost.Stopwatches** depends on:

Boost.Chrono for duration, time_point and clocks, ...

Boost.Config for configuration purposes, ...

Boost.Exception for throw_exception, ...

Boost.MPL for MPL Assert and bool, ...

Boost.System for error_code, ...

Boost.Input/Output for io_state, ...

Boost.StaticAssert for STATIC_ASSERT, ...

Boost.TypeTraits for is_same, ...

Boost.Utility for base_from_memeber, ...

Boost.Stopwatches depends optionaly on:

Boost.Thread for thread_specific_ptr when suspendible_clock.hpp is included

Boost.Accumulator for accumulator_set, and statistics features when stopwatch_accumulator.hpp is included

Building an executable that uses Boost.Stopwatches

In addition to linking with the Boost.Chrono library you need also to link with the Boost.System library. If you use Suspendible clocks you will need also with Boost.Thread.

Exceptions safety

All functions in the library are exception-neutral and provide strong guarantee of exception safety as long as the underlying parameters provide it.

Thread safety

All functions in the library are thread-unsafe except when noted explicitly.

Tested compilers

The implementation will eventually work with most C++03 conforming compilers. Current version has been tested on:

Windows with

- MSVC 10.0

Cygwin 1.5 with

- GCC 3.4.4

Cygwin 1.7 with

- GCC 4.3.4

MinGW with

- GCC 4.4.0
- GCC 4.5.0
- GCC 4.5.0 -std=c++0x
- GCC 4.6.0
- GCC 4.6.0 -std=c++0x

Ubuntu 10.10

- GCC 4.4.5
- GCC 4.4.5 -std=c++0x

> **Note**
>
> Please let us know how this works on other platforms/compilers.

> **Note**
>
> Please send any questions, comments and bug reports to boost <at> lists <dot> boost <dot> org.

Hello World!

If all you want to do is to time a program's execution:

```
#include <boost/stopwatches/stopclock.hpp>

...

// add this in the scope you want to time,
// at the point you want the timer to start.
boost::stopwatches::stopclock<> rt;
```

Here is a complete program (stopclock_example.cpp):

```
#include <boost/stopwatches/stopclock.hpp>
#include <cmath>

int main()
{
  boost::stopwatches::stopclock<> t;

  for ( long i = 0; i < 10000000; ++i )
    std::sqrt( 123.456L ); // burn some time

  return 0;
}
```

Debug build output was:

```
real 0.832s, cpu 0.813s (97.7%), user 0.813s, system 0.000s
```

In other words, the program took 0.832 real-time (i.e. wall clock) seconds to execute, while the operating system (Windows in this case) charged 0.813 seconds of CPU time to the user and 0 seconds to the system. The total CPU time reported was 0.813 seconds, and that represented utilization of 97.7% of the real-time seconds.

Tutorial

Stopwatches and Stopclocks

Knowing how long a program, a function or a specific block takes to execute is useful in both test and production environments. **Boost.Stopwatches** introduces the `Stopwatch` concept which captures the mechanism to measure the elapsed time. A `Stopwatch`

allows to `start`, `stop`, `suspend` and `resume` measuring the elapsed time. `stopwatch<>` is the basic model of `Stopwatch` allowing to make a single measure.

At the user level, the main use case of measuring the elapsed time is to report these measures on the display. `stopwatch_reporter<>` provides a run time reporting package that can be invoked in a single line of code to report the usage of a `Clock`. For example

```
using namespace boost::chrono;
using namespace boost::stopwatches;

int f1(long j) {
    stopwatch_reporter<stopwatch<> > _;

    for ( long i = 0; i < j; ++i )
        std::sqrt( 123.456L );  // burn some time

    return 0;
}
int main()  {
    f1(100000);
    f1(200000);
    f1(300000);
    return 0;
}
```

Will produce the following output

```
0.006s
0.011s
0.017s
```

Stopwatches accumulation and statistics

The preceding stopwatch manage only with a measure. It is also interesting to have an statisitical view of these times, for example the sum, min, max and mean. `stopwatch_accumulator<>` associates an accumulator with a `stopwatch`, so we are able to retrieve any statistical feature Boost.Accumulator provides.

For example

```
using namespace boost::stopwatches;

int f1(long j) {
    static stopwatch_reporter<stopwatch_accumulator<> > sw;
    stopwatch_reporter<stopwatch_accumulator<> >::scoped_run _(sw);

    for ( long i = 0; i < j; ++i )
        std::sqrt( 123.456L );  // burn some time

    return 0;
}
int main()  {
    f1(100000);
    f1(200000);
    f1(300000);
    return 0;
}
```

Will produce the following output

```
3 times, sum=0.034s, min=0.006s, max=0.017s, mean=0.011s
```

How can I prefix each report with `BOOST_CURRENT_FUNCTION` function signature?

You will need to give a specific format to your `stopclock`. You just need to concatenate your specific pattern to the default_format of the formatter.

For example, for a `stopclock_accumulator` the default formatter is `stopwatch_accumulator_formatter`, you will need to do something like:

```
static stopclock_accumulator<> acc(
    std::string(BOOST_CURRENT_FUNCTION) + ":  "
        + stopwatch_accumulator_formatter::default_format()
);
stopclock_accumulator<>::scoped_run _(acc);
```

Some of you will say that this is too long to type just to get the a report. You can of course define your own macro as

```
#define REPORT_FUNCTION_ACCUMULATED_LIFETIME\
    static boost::stopwatches::stopclock_accumulator<> \
            BOOST_JOIN(_accumulator_, __LINE__)_
        ( std::string(BOOST_CURRENT_FUNCTION) + ":  " + \
            boost::stopwatches::stopwatch_accumulator_formatter::default_format() \
    ); \
    boost::stopwatches::stopclock_accumulator<>::scoped_run \
            BOOST_JOIN(_accumulator_run_, __LINE__) \
        (BOOST_JOIN(_accumulator_, __LINE__))
```

With this macro you will just have to write

```
void foo()
{
    REPORT_FUNCTION_ACCUMULATED_LIFETIME() ;
    boost::this_thread::sleep(boost::posix_time::milliseconds(100));
    // ...
}
```

How can I prefix each report with `__FILE__[__LINE__]` pattern?

When you want to prefix with the `__FILE__[__LINE__]` pattern you can follow the same technique as described below:

```
#define REPORT_LINE_ACCUMULATED_LIFETIME \
    static stopclock_accumulator<>
            BOOST_JOIN(_accumulator_, __LINE__) \
            ( std::string(__FILE__) + "[" + BOOST_STRINGIZE(__LINE__) + "] "
                + stopwatch_accumulator_formatter::default_format() \
    ); \
    stopclock_accumulator<>::scoped_run \
            BOOST_JOIN(_accumulator_run_, __LINE__)
            (BOOST_JOIN(_accumulator_, __LINE__))
```

Now you can mix fcntion and line reports as follows

```
void foo()
{
    REPORT_FUNCTION_ACCUMULATED_LIFETIME;
    boost::this_thread::sleep(boost::posix_time::milliseconds(100));
    {
        REPORT_LINE_ACCUMULATED_LIFETIME;
        boost::this_thread::sleep(boost::posix_time::milliseconds(200));
    }

}
```

Can I use an stopclock accumulator which is not static?

The typical example of stopclock_accumulator is to get statistical measures of the time a function takes for each one of its calls. You can also use `stopclock_accumulator` to get statistical measures of the time a given loop takes for each one of its laps.

```
stopclock_accumulator<> acc(
    std::string(__FILE__) + "[" + BOOST_STRINGIZE(__LINE__) + "] "
        + stopwatch_accumulator_formatter::default_format()
);
for (int i=0; i<N; i++) {
    stopclock_accumulator<>::scoped_run _(acc);
    // ...
}
```

How can I suspend a stopwatch?

```
#include <boost/stopwatches/stopwatch.hpp>
#include <cmath>
#include <boost/thread.hpp>


using namespace boost::stopwatches;
double res;
void f1(long j)
{
    stopwatch_reporter<stopwatch<> >:: _(BOOST_STOPWATCHES_STOPWATCH_FUNCTION_FORMAT);
    for (long i =0; i< j; i+=1)
        res+=std::sqrt( res+123.456L+i );  // burn some time
    stopwatch_reporter<stopwatch<> >::scoped_suspend s(_);
    boost::this_thread::sleep(boost::posix_time::milliseconds(200));
}
```

How to get specific statistics from stopwatches accumulator?

There are two use cases that coul need to change the statistics associated to a stopwatches accumulator:

1. We want to reduce the default reporting and we preffer to adapt the statistics to the reporting

2. We want to report other statistics of the samples

For the first case we just need to change the accumulator_set and the format we want to get. Imagin we want to get only the count, sam and mean statistics, no need to calculate the min neither the max.

```
using namespace boost::accumulators;

typedef stopwatch_reporter<stopwatch_accumulator<process_real_cpu_clock,
            accumulator_set<process_real_cpu_clock::rep,
                features<
                        tag::count,
                        tag::sum,
                        tag::mean
                >
            >
  > my_stopwatch_accumulator_reporter;

int f1(long j)
{
  static my_stopwatch_accumulator_reporter acc("%c times, sum=%ss, mean=%as\n");
  my_stopwatch_accumulator_reporter::scoped_run _(acc);

  for ( long i = 0; i < j; ++i )
    std::sqrt( 123.456L );  // burn some time

  return 0;
}
```

But what would hapend if we haven't forced the format:

```
static my_stopwatch_accumulator_reporter acc;
my_stopwatch_accumulator_reporter::scoped_run _(acc);
```

Unfortunately there is no error at compile time. Fortunately, the run-time execution is not undefined and will return 0 for the missing statistics.

Formatters

How can I make a specific formatter when the default do not satisfy my expectations

Imagine then that we want to report the `tag::variance(lazy)`. We will need to include the specific accumulator file

```
...
#include <boost/accumulators/statistics/variance.hpp>
...
typedef stopwatch_reporter<stopwatch_accumulator<process_real_cpu_clock,
            accumulator_set<process_real_cpu_clock::rep,
                features<
                        tag::count,
                        tag::sum,
                        tag::mean,
                        tag::variance(lazy)
                >
            >
> my_stopwatch_accumulator_reporter;
```

But what happens if we add new statistics to the accumulator_set that are not taken in account by the default formatter? These statistics will simply be ignored. So we will need to define our own accumulator formatter.

```
typedef stopwatch_reporter<stopwatch_accumulator<process_real_cpu_clock,
            accumulator_set<process_real_cpu_clock::rep,
                features<
                        tag::count,
                        tag::sum,
                        tag::mean,
                        tag::variance(lazy)
                >
            >,
            my_stopwatch_accumulator_formatter
> my_stopwatch_accumulator_reporter;
```

Next follow the definition of a formatter taking care of count, sum, mean and variance

```
class my_stopwatch_accumulator_formatter {
public:
    typedef std::string string_type;
    typedef char char_type;
    typedef std::ostream ostream_type;

    static ostream_type &  default_os() {return std::cout;}
    static const char_type* default_format() {
        return "%c times, sum=%ss, mean=%as, variance=%vs\n";
    }
    static int default_places() { return 3; }

    template <class Stopwatch >
    static void show_time( Stopwatch & stopwatch_, const char_type* format,
                           int places, ostream_type & os, system::error_code & ec)
    {
        typedef typename Stopwatch::duration duration_t;
        typename Stopwatch::accumulator accumulator& acc = stopwatch_.accumulated();

        boost::io::ios_flags_saver ifs( os );
        os.setf( std::ios_base::fixed, std::ios_base::floatfield );
        boost::io::ios_precision_saver ips( os );
        os.precision( places );

        for ( ; *format; ++format ) {
            if ( *format != '%' || !*(format+1) || !std::strchr("acsv", *(format+1)) ) {
                os << *format;
            } else {
                ++format;
                switch ( *format ) {
                case 's':
                    os << boost::chrono::duration<double>(
                            duration_t(accumulators::sum(acc))).count();
                    break;
                case 'a':
                    os << (accumulators::count(acc)>0)
                            ? boost::chrono:: duration<double>(duration_t(
                                duration_t::rep(accumulators::mean(acc)))).count()
                            : 0;
                    break;
                case 'c':
                    os << accumulators::count(acc);
                    break;
                case 'v':
                    os << (accumulators::count(acc)>0)
                            ? boost::chrono:: duration<double>(duration_t(
                                duration_t::rep(accumulators::variance(acc)))).count()
                            : 0;
```

```
                    break;
                default:
                    assert(0 && "my_stopwatch_accumulator_formatter internal logic error");
                }
            }
        }
    }
};
```

Examples

Reporting

stopclock_example.cpp

Here is the stopclock_example.cpp program supplied with the Boost Chrono library:

When the `stopclock<> t` object is created, it starts timing. When it is destroyed at the end of the program, its destructor stops the time counting and displays timing information on cout.

```
#include <boost/stopwatches/stopclock.hpp>
#include <cmath>

int main()
{
  boost::stopwatches::stopclock<> t;

  for ( long i = 0; i < 10000000; ++i )
    std::sqrt( 123.456L );  // burn some time

  return 0;
}
```

The output of this program run looks like this:

```
wall 0.42 s, user 0.41 s + system 0.00 s = total cpu 0.41 s, (96.3%)
```

In other words, this program ran in 0.42 seconds as would be measured by a clock on the wall, the operating system charged it for 0.41 seconds of user CPU time and 0 seconds of system CPU time, the total of these two was 0.41, and that represented 96.3 percent of the wall clock time.

See the source file example/stopclock_example.cpp

stopclock_example2.cpp

The stopclock_example2.cpp program is the same, except that it supplies additional constructor arguments from the command line:

```
#include <boost/stopwatches/stopclock.hpp>
#include <cmath>

int main( int argc, char * argv[] )
{
  const char * format = argc > 1 ? argv[1] : "%t cpu seconds\n";
  int          places = argc > 2 ? std::atoi( argv[2] ) : 2;

  boost::stopwatches::stopclock<> t( format, places );

  for ( long i = 0; i < 10000000; ++i )
    std::sqrt( 123.456L ); // burn some time

  return 0;
}
```

Here is the output for this program for several sets of command line arguments:

```
stopclock_example2
0.42 cpu seconds

stopclock_example2 "%w wall clock seconds\n"
0.41 wall clock seconds

stopclock_example2 "%w wall clock seconds\n" 6
0.421875 wall clock seconds

stopclock_example2 "%t total CPU seconds\n" 3
0.422 total CPU seconds
```

See the source file example/stopclock_example2.cpp

time command

```
#include <boost/stopwatches/stopclock.hpp>
#include <cstdlib>
#include <string>
#include <iostream>

int main( int argc, char * argv[] )
{
  if ( argc == 1 )
  {
    std::cout << "invoke: timex [-v] command [args...]\n"
      "  command will be executed and timings displayed\n"
      "  -v option causes command and args to be displayed\n";
    return 1;
  }

  std::string s;

  bool verbose = false;
  if ( argc > 1 && *argv[1] == '-' && *(argv[1]+1) == 'v' )
  {
    verbose = true;
    ++argv;
    --argc;
  }

  for ( int i = 1; i < argc; ++i )
  {
    if ( i > 1 ) s += ' ';
    s += argv[i];
  }

  if ( verbose )
    { std::cout << "command: \"" << s.c_str() << "\"\n"; }

  boost::stopwatches::stopclock<> t;

  return std::system( s.c_str() );
}
```

See the source file example/timex.cpp

Reference

Header <boost/stopwatches.hpp>

```
#include <boost/stopwatches/stopwatches.hpp>
```

Header <boost/stopwatches/stopwatches.hpp>

This file include all the stopwatches related files except the suspendible related files.

```
#include <boost/stopwatches/scoped_stopclock.hpp>
#include <boost/stopwatches/stopclock.hpp>
#include <boost/stopwatches/stopclock_accumulator.hpp>
#include <boost/stopwatches/stopwatch.hpp>
#include <boost/stopwatches/stopwatch_accumulator.hpp>
#include <boost/stopwatches/stopwatch_accumulator_formatter.hpp>
#include <boost/stopwatches/stopwatch_accumulator_time_formatter.hpp>
#include <boost/stopwatches/stopwatch_formatter.hpp>
#include <boost/stopwatches/stopwatch_reporter.hpp>
#include <boost/stopwatches/stopwatch_scoped.hpp>
#include <boost/stopwatches/time_formatter.hpp>
#include <boost/stopwatches/t24_hours.hpp>
#include <boost/stopwatches/t24_hours_formatter.hpp>
```

Other clocks

`SuspendibleClock` Requirements

A `SuspendibleClock` is a Clock that in addition supports `suspend`/`resume` operations.

A `SuspendibleClock` must meet the requirements in the following Table.

In this table `C` denote `clock` types.

Table 1. SuspendibleClock Requirements

expression	return type	operational semantics
`C::suspend()`	`void`	Suspends the time counting of the clock C.
`C::resume()`	`void`	Resumes the time counting of the clock C.
`C::suspended()`	`duration`	Returns the delay(duration during which the clock has been suspended.

Static Member Function `suspend()`

```
void suspend( system::error_code & ec = throws() );
```

Effect: Suspends the SuspendibleClock.

Throw: Any exception the `Clock::now(ec)` function can throw. Otherwise `ec` is set with the correspoding error code set by `Clock::now(ec)`.

Static Member Function `resume()`

```
void resume( system::error_code & ec = throws() );
```

Effect: Resumes the `SuspendibleClock`.

Throw: Any exception the `Clock::now(ec)` can throw. Otherwise `ec` is set with the correspoding error code set by `Clock::now(ec)`.

Static Member Function `suspended()`

```
duration suspended( system::error_code & ec = throws() );
```

Returns: the cumalative elapsed duration during which the `SuspendibleClock` has been suspendeed.

Throw: Any exception the Clock::now function can throw if `ec == throws()`. Otherwise `ec` is set with the correspoding error code set by `Clock::now(ec)`.

Models of `SuspendibleClock`:

- `suspendible_clock`

Header `<boost/stopwatches/scoped_suspend.hpp>`

```
namespace boost { namespace stopwatches {
    template <class Clock> struct is_suspendible;
    template <class Clock> class scoped_suspend;
}}
```

Meta Function Class `is_suspendible`

```
template <class Clock>
struct is_suspendible : mpl:: false_ {};
```

Template Class `scoped_suspend`

```
template <class Clock>
class scoped_suspend {
public:
    scoped_suspend(system::error_code & ec = throws()) {}
    ~scoped_suspend() {}
private:
    scoped_suspend(); // = delete;
    scoped_suspend(const scoped_suspend&); // = delete;
    scoped_suspend& operator=(const scoped_suspend&); // = delete;
};
```

Header `<boost/stopwatches/suspendible_clock.hpp>`

```
namespace boost { namespace stopwatches {

    template <class Clock>
    class suspendible_clock;

    template <class Clock>
    struct is_suspendible<suspendible_clock<Clock> >;

    template <class Clock>
    class scoped_suspend<suspendible_clock<Clock> >;

}}
```

Template Class `suspendible_clock<>`

Given a `Clock`, `suspendible_clock` < `Clock`> is a model of `SuspendibleClock`.

```
template < class Clock >
class suspendible_clock {
public:
    typedef typename Clock::duration                          duration;
    typedef typename Clock::rep                               rep;
    typedef typename Clock::period                            period;
    typedef chrono:: time_point<suspendible_clock<Clock> >  time_point;
    static const bool is_steady = Clock::is_steady;

    static time_point now( system::error_code & ec = throws() );
    static void suspend( system::error_code & ec = throws() );
    static void resume( system::error_code & ec = throws() );
    static duration suspended(system::error_code & ec = throws());
};
```

`scoped_suspend` specialization for `suspendible_clock<>`

```
template <class Clock>
class scoped_suspend<suspendible_clock<Clock> > {
public:
    scoped_suspend(system::error_code & ec = throws());
    ~scoped_suspend();
};
```

Stopwatches

`Stopwatch` Requirements

A Stopwatch measure the amount of time elapsed from a start point in time to the stop point time or the accumulation of them. Stopwatches can in addition be restarted, suspended and resumed.

A Stopwatch must meet the requirements in the following table. In this table `S`, `S1` and `S2` denote stopwatches types. `s` is an instance of `S`.

Table 2. Stopwatch Requirements

expression	return type	operational semantics
`S::clock`	A model of `Clock`.	The clock associated to this Stopwatch.
`S::duration`	`S::clock::duration`	The `duration` type of the `clock`.
`S::time_point`	`S::clock::time_point`	The `time_point` type of the `clock`.
`S::scoped_run`	`stopwatch_runner<stopwatch<Clock> >`	RAI which `start`/`stop` the `stopwatch`.
`S::scoped_suspend`	`stopwatch_suspender<stopwatch<Clock> >`	RAI which `suspend`/`resume` the `stopwatch`.
`S::scoped_resume`	`stopwatch_resumer<stopwatch<Clock> >`	RAI which `resume`/`suspend` the `stopwatch`.
`s.start()`	`S::time_point`	starts a Stopwatch.
`s.restart()`	`std::pair<S::duration,S::time_point>`	restarts a Stopwatch.
`s.stop()`	`S::duration`	stops a Stopwatch.
`s.resume()`	`S::time_point`	resume a Stopwatch.
`s.suspend()`	`S::duration`	suspends a Stopwatch.
`s.elapsed()`	`S::duration`	the elapsed time while the Stopwatch was running.

Member Function `start()`

```
time_point start( system::error_code & ec = throws() );
```

Effect: Starts running the stopwatch.

Returns: the starting time point.

Throw: Any exception the `Clock::now` function can throw when `ec` is `throws()`

Member Function `stop()`

```
duration stop( system::error_code & ec = throws() );
```

Effect: Stops running the stopwatch.

Returns: The cummulated elapsed time.

Throw: Any exception the `Clock::now` function can throw when `ec` is `throws()`

Member Function `suspend()`

```
duration suspend( system::error_code & ec = throws() );
```

Effect: Suspends the stopwatch.

Throw: Any exception the `Clock::now` function can throw when `ec` is `throws()`

Member Function `resume()`

```
time_point resume( system::error_code & ec = throws() );
```

Effect: Resumes the stopwatch.

Returns: the starting time point.

Throw: Any exception the Clock::now function can throw.

Member Function `restart()`

```
time_point restart( system::error_code & ec = throws() );
```

Effect: `stop`/`start` the stopwatch.

Returns: the starting time point.

Throw: Any exception the `Clock::now` function can throw when `ec` is `throws()`

Member Function `elapsed()`

```
duration elapsed(system::error_code & ec = throws()) const;
```

Returns: the cumulated elapsed time.

Throw: Any exception the `Clock::now` function can throw when `ec` is `throws()`

Models of `Stopwatch`:

- `stopwatch`
- `stopwatch_accumulator`

Header `<boost/stopwatches/lightweight_stopwatch.hpp>`

```
namespace boost { namespace stopwatches  {
    struct dont_start_t;
    static const dont_start_t dont_start;

    template <class Clock=high_resolution_clock,
            typename Features=void,
            typename Weight=void
    > class lightweight_stopwatch;

    typedef <see above> system_lightweight_stopwatch;
    #ifdef BOOST_CHRONO_HAS_CLOCK_STEADY
    typedef <see above> steady_lightweight_stopwatch;
    #endif
    typedef <see above> high_resolution_lightweight_stopwatch;
}}
```

Class `dont_start_t`

Structure used to don't start a `lightweight_stopwatch` at construction time.

```
struct dont_start_t;
static const dont_start_t dont_start;
```

Template Class `lightweight_stopwatch<>`

`lightweight_stopwatch<>` is a model of a `Stopwatch` concept.

Knowing how long a part of a program takes to execute is useful in both test and production environments. A `lightweight_stopwatch` object measures elapsed time. It is recommended to use it with clocks that measure wall clock rather than CPU time since the intended use is performance measurement on systems where total elapsed time is more important than just process or CPU time.

The maximum measurable elapsed time depends on the Clock parameter. The accuracy of timings depends on the accuracy of timing information provided the Clock, and this could varies a great deal from one clock to another.

```
template <class Clock, typename Features, typename Weight>
class lightweight_stopwatch {
public:
    typedef Clock                       clock;
    typedef typename Clock::duration    duration;
    typedef typename Clock::time_point  time_point;
    typedef <see below>                 storage;

    explicit lightweight_stopwatch( storage& st, system::error_code & ec = throws() );
    lightweight_stopwatch( storage& st, const dont_start_t& t );

    ~lightweight_stopwatch();

    time_point start( system::error_code & ec = throws() );
    duration stop( system::error_code & ec = throws() );
    std::pair<duration,time_point> restart( system::error_code & ec = throws() );
    duration suspend( system::error_code & ec = throws() );
    time_point resume( system::error_code & ec = throws() );
    duration elapsed( system::error_code & ec = throws() );
    time_point now( system::error_code & ec = throws() );
    void reset( system::error_code & ec = throws() );

    storage& get_storage( );
    duration lifetime( system::error_code & ec = throws() );

    typedef stopwatch_runner<lightweight_stopwatch<Clock> > scoped_run;
    typedef stopwatch_suspender<lightweight_stopwatch<Clock> > scoped_suspend;
    typedef stopwatch_resumer<lightweight_stopwatch<Clock> > scoped_resume;

};
```

`storage` is either `Clock::duration` if Features and Weight are void and `accumulators::accumulator_set<typename Clock::duration::rep, Features, Weight>` otherwise.

Only the specificities of this class are documented here. See `Stopwatch` concept for the common parts.

Constructor `lightweight_stopwatch( storage&, system::error_code &)`

```
explicit lightweight_stopwatch( storage& st, system::error_code & ec = throws() );
```

Effect: constructs and starts the lightweight_stopwatch.

Throw: Any exception the `Clock::now` function can throw when `ec` is `throws()`

Constructor `lightweight_stopwatch( storage&, dont_start_t &)`

```
explicit lightweight_stopwatch( storage& st, const dont_start_t& t );
```

Effect: constructs the lightweight_stopwatch without starting it.

Throw: Nothing.

Member Function `get_storage()`

```
storage& get_storage();
```

Returns: the assocaited storage reference.

Throw: Nothing.

Member Function `reset()`

```
void reset( );
```

Effect: Stop the lightweight_stopwatch and reinit the storage.

Throw: Nothing.

Header `<boost/stopwatches/stopwatch.hpp>`

```
namespace boost { namespace stopwatches  {
    template <class Clock=high_resolution_clock> class stopwatch;

    template <class Clock>
    struct stopwatch_reporter_default_formatter<stopwatch<Clock> > {
        typedef stopwatch_formatter type;
    };

    template <class Clock>
    struct wstopwatch_reporter_default_formatter<stopwatch<Clock> > {
        typedef wstopwatch_formatter type;
    };

    typedef <see above> system_stopwatch;
    #ifdef BOOST_CHRONO_HAS_CLOCK_STEADY
    typedef <see above> steady_stopwatch;
    #endif
    typedef <see above> high_resolution_stopwatch;
}}
```

Template Class `stopwatch<>`

`stopwatch<>` is a model of a `Stopwatch` concept.

Knowing how long a part of a program takes to execute is useful in both test and production environments. A `stopwatch` object measures elapsed time. It is recommended to use it with clocks that measure wall clock rather than CPU time since the intended use is performance measurement on systems where total elapsed time is more important than just process or CPU time.

The maximum measurable elapsed time depends on the Clock parameter. The accuracy of timings depends on the accuracy of timing information provided the Clock, and this could varies a great deal from one clock to another.

```
template <class Clock>
class stopwatch :
    private base_from_member<typename Clock::duration>,
    public lightweight_stopwatch<Clock>
{
public:
    explicit stopwatch( system::error_code & ec = throws()  );
    explicit stopwatch( const dont_start_t& t );

};
```

Constructor `stopwatch( system::error_code &)`

```
explicit stopwatch( system::error_code & ec = throws() );
```

Effect: constructs and starts the stopwatch.

Throw: Any exception the `Clock::now` function can throw when `ec` is `throws()`

Constructor `stopwatch( dont_start_t &)`

```
explicit stopwatch( const dont_start_t& t );
```

Effect: constructs the stopwatch without starting it.

Throw: Nothing.

`stopwatch_reporter_default_formatter` Specialization

The `stopwatch_reporter_default_formatter` of a `stopwatch<Clock>` is a `stopwatch_formatter`.

```
template <class Clock>
struct stopwatch_reporter_default_formatter<stopwatch<Clock> > {
    typedef stopwatch_formatter type;
};
```

The `wstopwatch_reporter_default_formatter` of a `stopwatch<Clock>` is a `wstopwatch_formatter`.

```
template <class Clock>
struct wstopwatch_reporter_default_formatter<stopwatch<Clock> > {
    typedef wstopwatch_formatter type;
};
```

`stopwatch` useful typedefs

The library provides stopwatch short cuts for all the models of `Clock`, replacing clock by stopwatch.

```
typedef boost::stopwatches::stopwatch< boost::chrono::system_clock >
    system_stopwatch;
#ifdef BOOST_CHRONO_HAS_CLOCK_STEADY
typedef boost::stopwatches::stopwatch< boost::chrono::steady_clock >
    steady_stopwatch;
#endif
typedef boost::stopwatches::stopwatch< boost::chrono::high_resolution_clock >
    high_resolution_stopwatch;
```

Header <boost/stopwatches/stopwatch_accumulator.hpp>

```
namespace boost { namespace stopwatches  {
    template <class Clock,
                typename Features=accumulators::features<
                    accumulators::tag::count,
                    accumulators::tag::sum,
                    accumulators::tag::min,
                    accumulators::tag::max,
                    accumulators::tag::mean >,
                typename Weight=void
    > class stopwatch_accumulator;

    template <class Clock, class Accumulator>
    struct stopwatch_reporter_default_formatter<stopwatch_accumulator<Clock, Accumulator> > {
        typedef stopwatch_accumulator_formatter type;
    };

    template <class Clock, class Accumulator>
    struct wstopwatch_reporter_default_formatter<stopwatch_accumulator<Clock, Accumulator> > {
        typedef wstopwatch_accumulator_formatter type;
    };

    typedef <see below> system_stopwatch_accumulator;
#ifdef BOOST_CHRONO_HAS_CLOCK_STEADY
    typedef <see below> steady_stopwatch_accumulator;
#endif
    typedef <see below> high_resolution_stopwatch_accumulator;

}}
```

Template Class stopwatch_accumulator<>

A stopwatch_accumulator<> is a model of a Stopwatch concept that allows to accumulate the time in several times instead of at once as it is the case of the class stopwatch<>.

```
template <class Clock, typename Features, typename Weight>
class stopwatch_accumulator
    : private base_from_member<
                typename accumulators::accumulator_set<
                    typename Clock::duration::rep, Features, Weight
                >
            >,
      public lightweight_stopwatch<Clock,Features,Weight>
{
public:
    stopwatch_accumulator( );
};
```

Constructor stopwatch_accumulator()

```
stopwatch_accumulator();
```

Effect: Initialize the elapsed duration and the times counter to 0. The stopwatch is not started.

stopwatch_reporter_default_formatter Specialization

The stopwatch_reporter_default_formatter of a stopwatch_accumulator<Clock> is a stopwatch_accumulator_formatter.

```
template <class Clock, class Accumulator>
struct stopwatch_reporter_default_formatter<stopwatch_accumulator<Clock, Accumulator> > {
    typedef stopwatch_accumulator_formatter type;
};
```

The `wstopwatch_reporter_default_formatter` of a `stopwatch_accumulator<Clock>` is a `wstopwatch_accumulator_formatter`.

```
template <class Clock, class Accumulator>
struct wstopwatch_reporter_default_formatter<stopwatch_accumulator<Clock, Accumulator> > {
    typedef wstopwatch_accumulator_formatter type;
};
```

`stopwatch_accumulator` useful typedefs

The library provides stopwatch_accumulator shortcuts for all the models of `Clock`, replacing clock by stopwatch_accumulator.

```
typedef boost::stopwatches::stopwatch_accumulator< boost::chrono::system_clock >
    system_stopwatch_accumulator;
#ifdef BOOST_CHRONO_HAS_CLOCK_STEADY
typedef boost::stopwatches::stopwatch_accumulator< boost::chrono::steady_clock >
    steady_stopwatch_accumulator;
#endif
typedef boost::stopwatches::stopwatch_accumulator< boost::chrono::high_resolution_clock >
    high_resolution_stopwatch_accumulator;
```

Header `<boost/stopwatches/stopwatch_scoped.hpp>`

```
namespace boost { namespace stopwatches  {
    template <class Stopwatch> class stopwatch_runner;
    template <class Stopwatch> class stopwatch_suspender;
    template <class Stopwatch> class stopwatch_resumer;
}}
```

Boost.Chrono provides some helper classes ensuring pairwised operations (start/stop, suspend/resume, resule/suspend).

Template Class `stopwatch_runner<>`

This helper class ensures that the start/stop are pairwised. Starts the associated `Stopwatch` at construction time, and stops it at destruction time.

```
template <class Stopwatch> class stopwatch_runner {
public:
    typedef Stopwatch stopwatch;
    stopwatch_runner(stopwatch & a, system::error_code & ec = throws());
    ~stopwatch_runner();
    stopwatch_runner() = delete;
    stopwatch_runner(const stopwatch_runner&) = delete;
    stopwatch_runner& operator=(const stopwatch_runner&) = delete;
};
```

Usage

```
void f1()
{
     static stopwatch_accumulator<> t;
     stopwatch_runner<stopwatch_accumulator<> > _(t);
     // ...
}
```

Template Class `stopwatch_suspender<>`

This helper class ensures that the suspend/resume are pairwised. Suspends the associated `Stopwatch` at construction time, and resumes it at destruction time.

```
template <class Stopwatch> class stopwatch_suspender {
public:
    typedef Stopwatch stopwatch;
    stopwatch_suspender(stopwatch & a, system::error_code & ec = throws());
    ~stopwatch_suspender();
    stopwatch_suspender() = delete;
    stopwatch_suspender(const stopwatch_suspender&) = delete;
    stopwatch_suspender& operator=(const stopwatch_suspender&) = delete;
}
```

Usage

```
void f1()
{
     static stopwatch_accumulator<> t;
     stopwatch_runner<stopwatch_accumulator<> > _(t);
     // ...

     // call to some function we don't want to measure
     {
        stopwatch_suspender<stopwatch_accumulator<> > _(t);
        external_function();
     }
}
```

Template Class `stopwatch_resumer<>`

This helper class ensures that the resume/suspend are pairwised. Resumes the associated `Stopwatch` at construction time, and suspends it at destruction time.

```
template <class Stopwatch> class stopwatch_resumer {
public:
    typedef Stopwatch stopwatch;
    stopwatch_resumer(stopwatch & a, system::error_code & ec = throws());
    ~stopwatch_resumer();
    stopwatch_resumer() = delete;
    stopwatch_resumer(const stopwatch_resumer&) = delete;
    stopwatch_resumer& operator=(const stopwatch_resumer&) = delete;
}
```

Usage

```
void f1()
{
     static stopwatch_accumulator<> t;
     stopwatch_runner<stopwatch_accumulator<> > _(t);
     // ...

     // call to some function we don't want to measure
     {
        stopwatch_suspender<stopwatch_accumulator<> > _(t);

        {
            stopwatch_resumer<stopwatch_accumulator<> > _(t);

        }
     }
}
```

Stopwatch Reporters

`Formatter` Requirements

A Formatter outputs on a given ostream a formatted string combining informations from a Stopwatch and the format and the double precision.

A Formatter must meet the requirements in the following Table.

In this table `F` denote a Formatter type, `S` is a Stopwatch and `s` is an instance of `S`, `f` is `const char *` , `p` is and int, and `os` is a `std::ostream`, ec is a system::error_code

Table 3. Formatter Requirements

expression	return type	operational semantics
`F::default_os()`	std::otream&	The output stream.
`F::default_places()`	`std::size_t`	The precision when displaying a double.
`F::default_format()`	`const char*`	The default format.
`F::show_time(s,f,p,os,ec)`	`S::time_point`	outputs on `os` a formatted string combining informations from the Stopwatch `s`, the format `f` and the double precision `p`.

Models of `Formatter`:

- `stopwatch_accumulator_formatter`
- `stopwatch_accumulator_formatter`
- `basic_24_hours_formatter`

`Formatter` related traits

```
template <class Stopwatch>
struct stopwatch_reporter_default_formatter {
    typedef <see below> type;
};
```

The nested typedef `type` defines the default formatter used by the `stopwatch_reporter` class when the `Formatter` parameter is not explicit.

`Reporter` Requirements

A `Reporter` provides everything a `Stopwatch` provides and adds reporting capabilities. The reporting is controleed by two parameters:

A `Reporter` must meet the requirements in the following table in addition of the `Stopwatch`. The reporting is controleed by two parameters:

- format : The output format
- places(precision): the number of decimal placess used.

And is sent to an output stream.

In this table `R` denote Reporters types. `r` is an instance of `R`, ec is a system::error_code, os is an std::ostream, f is a std::string, p is an int. From the library perspective a Reporter needs to satisfy the following requirements.

Table 4. Reporter Requirements

expression	return type	operational semantics
`R::stopwatch`	A model of `Stopwatch`.	The Stopwatch associated to this Reporter.
`r.report()`	`void`	Creates a report.
`r.report(ec)`	`void`	Creates a report.
`r.reported()`	`bool`	The reporting has been done.

From the user point of view, the reporter provides often construction with any of the parameters controleed the reporting.

Table 5. Open Reporter Requirements

expression	return type	operational semantics
`R r`	`R`	Creates a Reporter.
`R r(ec)`	`R`	Creates a Reporter.
`R r(os)`	`R`	Creates a Reporter.
`R r(os,ec)`	`R`	Creates a Reporter.
`R r(os,f)`	`R`	Creates a Reporter.
`R r(os,f,ec)`	`R`	Creates a Reporter.
`R r(os,f,p)`	`R`	Creates a Reporter.
`R r(os,f,p,ec)`	`R`	Creates a Reporter.
`R r(os,p)`	`R`	Creates a Reporter.
`R r(os,p,ec)`	`R`	Creates a Reporter.
`R r(f)`	`R`	Creates a Reporter.
`R r(f,ec)`	`R`	Creates a Reporter.
`R r(f,p)`	`R`	Creates a Reporter.
`R r(f,p,ec)`	`R`	Creates a Reporter.
`R r(p)`	`R`	Creates a Reporter.
`R r(p,ec)`	`R`	Creates a Reporter.

Member Function `R()`

```
void R(
    [ std::ostream & os ]
    [, const std::string & format]
    [, int places]
    [, system::error_code & ec = throws()]
    );
```

Effect: Creates a reporter. All the parameters are optional.

Throw: Nothing.

Member Function `report()`

```
void report(system::error_code & ec = throws());
```

Effect: Produce a report taking in account the ostream, format and place.

Throw: Any exception the `Formatter::show_time(*this,f,p,os,ec)` function can throw when `ec` is `throws()`.

Member Function `reported()`

```
bool reported();
```

Returns: True if the reportes has ben reported already.

Throw: Nothing.

Header `<boost/stopwatches/stopwatch_reporter.hpp>`

```
namespace boost { namespace stopwatches  {
    template <class Stopwatch, class Formatter>
    class basic_stopwatch_reporter;

    template <class Stopwatch>
    struct stopwatch_reporter_default_formatter;

    template <class Stopwatch,
              class Formatter=typename stopwatch_reporter_default_formatter<Stopwatch>::type>
    class stopwatch_reporter;

    template <class Stopwatch>
    struct wstopwatch_reporter_default_formatter;

    template <class Stopwatch,
              class Formatter=typename wstopwatch_reporter_default_formatter<Stopwatch>::type>
    class wstopwatch_reporter;
}}
```

Template Class `basic_stopwatch_reporter<>`

class `basic_stopwatch_reporter` provides everything a `Stopwatch` provides and it adds reporting capabilities that can be invoked in a single line of code. The reporting is controleed by two parameters:

- format : The output format
- places(precision): the number of decimal placess used.

The default places is given by Formatter::default_places().

The default format is given by Formatter::default_format().

```
template <class Stopwatch, class Formatter>
class basic_stopwatch_reporter : public Stopwatch {
public:
    typedef typename Stopwatch::clock clock;
    typedef Stopwatch stopwatch;
    typedef Formatter formatter;

    explicit basic_stopwatch_reporter( system::error_code & ec = throws() );
    explicit basic_stopwatch_reporter( std::ostream & os,
                system::error_code & ec = throws() );

    explicit basic_stopwatch_reporter( const std::string & format,
                system::error_code & ec = throws() );
    explicit basic_stopwatch_reporter( std::ostream & os, const std::string & format,
                system::error_code & ec = throws() );

    explicit basic_stopwatch_reporter( const std::string & format, int places,
                system::error_code & ec = throws() );
   explicit basic_stopwatch_reporter( std::ostream & os, const std::string & format, int places,
                system::error_code & ec = throws() );

    explicit basic_stopwatch_reporter( int places,
                system::error_code & ec = throws() );
    explicit basic_stopwatch_reporter( std::ostream & os, int places,
                system::error_code & ec = throws() );

    explicit basic_stopwatch_reporter( int places, const std::string & format,
                system::error_code & ec = throws() );
   explicit basic_stopwatch_reporter( std::ostream & os, int places, const std::string & format,
                system::error_code & ec = throws() );

    ~basic_stopwatch_reporter();

    void report( system::error_code & ec = throws() );
    bool reported() const;


    typedef stopwatch_runner<basic_stopwatch_reporter<Stopwatch> > scoped_run;
    typedef stopwatch_suspender<basic_stopwatch_reporter<Stopwatch> > scoped_suspend;
    typedef stopwatch_resumer<basic_stopwatch_reporter<Stopwatch> > scoped_resume;
};
```

Template Class `stopwatch_reporter<>`

class `stopwatch_reporter` provides a everything a `Stopwatch` provides and it adds reporting capabilities that can be invoked in a single line of code. The reporting is controleed by two parameters:

- format : The output format
- places(precision): the number of decimal placess used.

The default places is given by Formatter::default_places().

The default format is given by Formatter::default_format().

```
template <class Stopwatch, class Formatter>
class stopwatch_reporter : public basic_stopwatch_reporter<Stopwatch,Formatter> {
public:
    typedef typename Stopwatch::clock clock;
    typedef Stopwatch stopwatch;
    typedef Formatter formatter;

    explicit stopwatch_reporter( system::error_code & ec = throws() );
    explicit stopwatch_reporter( std::ostream & os,
                system::error_code & ec = throws() );

    explicit stopwatch_reporter( const std::string & format,
                system::error_code & ec = throws() );
    explicit stopwatch_reporter( std::ostream & os, const std::string & format,
                system::error_code & ec = throws() );

    explicit stopwatch_reporter( const std::string & format, int places,
                system::error_code & ec = throws() );
    explicit stopwatch_reporter( std::ostream & os, const std::string & format, int places,
                system::error_code & ec = throws() );

    explicit stopwatch_reporter( int places,
                system::error_code & ec = throws() );
    explicit stopwatch_reporter( std::ostream & os, int places,
                system::error_code & ec = throws() );

    explicit stopwatch_reporter( int places, const std::string & format,
                system::error_code & ec = throws() );
    explicit stopwatch_reporter( std::ostream & os, int places, const std::string & format,
                system::error_code & ec = throws() );

    ~stopwatch_reporter();

    void report( system::error_code & ec = throws() );
    bool reported() const;


    typedef stopwatch_runner<stopwatch_reporter<Stopwatch> > scoped_run;
    typedef stopwatch_suspender<stopwatch_reporter<Stopwatch> > scoped_suspend;
    typedef stopwatch_resumer<stopwatch_reporter<Stopwatch> > scoped_resume;
};
```

Usage

```
void f1()
{
    typedef stopwatch_reporter<stopwatch_accumulator<> > accumulator;
    static accumulator t;
    accumulator::scoped_run _(t);
    // ...

    // call to some function we don't want to measure
    {
        accumulator::scoped_suspend _(t);
        external_function();
    }
}
```

Template Class `wstopwatch_reporter<>`

class `wstopwatch_reporter` provides a everything a `Stopwatch` provides and it adds reporting capabilities that can be invoked in a single line of code. The reporting is controleed by two parameters:

- format : The output format
- places(precision): the number of decimal placess used.

The default places is given by Formatter::default_places().

The default format is given by Formatter::default_format().

```
template <class Stopwatch, class Formatter>
class wstopwatch_reporter : public basic_wstopwatch_reporter<Stopwatch,Formatter> {
public:
    typedef typename Stopwatch::clock clock;
    typedef Stopwatch stopwatch;
    typedef Formatter formatter;

    explicit wstopwatch_reporter( system::error_code & ec = throws() );
    explicit wstopwatch_reporter( std::ostream & os,
                system::error_code & ec = throws() );

    explicit wstopwatch_reporter( const std::string & format,
                system::error_code & ec = throws() );
    explicit wstopwatch_reporter( std::ostream & os, const std::string & format,
                system::error_code & ec = throws() );

    explicit wstopwatch_reporter( const std::string & format, int places,
                system::error_code & ec = throws() );
    explicit wstopwatch_reporter( std::ostream & os, const std::string & format, int places,
                system::error_code & ec = throws() );

    explicit wstopwatch_reporter( int places,
                system::error_code & ec = throws() );
    explicit wstopwatch_reporter( std::ostream & os, int places,
                system::error_code & ec = throws() );

    explicit wstopwatch_reporter( int places, const std::string & format,
                system::error_code & ec = throws() );
    explicit wstopwatch_reporter( std::ostream & os, int places, const std::string & format,
                system::error_code & ec = throws() );

    ~wstopwatch_reporter();

    void report( system::error_code & ec = throws() );
    bool reported() const;


    typedef stopwatch_runner<wstopwatch_reporter<Stopwatch> > scoped_run;
    typedef stopwatch_suspender<wstopwatch_reporter<Stopwatch> > scoped_suspend;
    typedef stopwatch_resumer<wstopwatch_reporter<Stopwatch> > scoped_resume;
};
```

Usage

```
void f1()
{
    typedef wstopwatch_reporter<stopwatch_accumulator<> > accumulator;
    static accumulator t;
    accumulator::scoped_run _(t);
    // ...

    // call to some function we don't want to measure
    {
        accumulator::scoped_suspend _(t);
        external_function();
    }
}
```

Header <boost/stopwatches/stopclock.hpp>

```
namespace boost { namespace stopwatches {
    template < class Clock, class Formatter > class basic_stopclock;
    template < class Clock, class Formatter > class stopclock;
    template < class Clock, class Formatter > class wstopclock;

    typedef <see above> system_stopclock;
    #ifdef BOOST_CHRONO_HAS_CLOCK_STEADY
    typedef <see above> steady_stopclock;
    #endif
    typedef <see above> high_resolution_stopclock;
    typedef <see above> process_real_cpu_stopclock;
    typedef <see above> process_user_cpu_stopclock;
    typedef <see above> process_system_cpu_stopclock;

    typedef <see above> system_wstopclock;
    #ifdef BOOST_CHRONO_HAS_CLOCK_STEADY
    typedef <see above> steady_wstopclock;
    #endif
    typedef <see above> high_resolution_wstopclock;
    typedef <see above> process_real_cpu_wstopclock;
    typedef <see above> process_user_cpu_wstopclock;
    typedef <see above> process_system_cpu_wstopclock;
}}
```

Template Class basic_stopclock<>

`basic_stopclock<Clock,Formatter>` template class is a shortcut of `basic_stopwatch_reporter<stopwatch<Clock,Formatter>>`

```
template< class Clock, class Formatter>
class basic_stopclock : public basic_stopwatch_reporter<stopwatch<Clock>, Formatter> {
public:
    typedef Clock clock;
    typedef stopwatch<Clock> stopwatch;
    typedef Formatter formatter;
    typedef typename Formatter::string_type string_type;
    typedef typename Formatter::char_type char_type;
    typedef typename Formatter::ostream_type ostream_type;

    explicit basic_stopclock( system::error_code & ec = throws() );
    explicit basic_stopclock( ostream_type & os,
                system::error_code & ec = throws() );
    explicit basic_stopclock( const string_type & format,
                system::error_code & ec = throws() );
    explicit basic_stopclock( int places,
                system::error_code & ec = throws() );

    basic_stopclock( ostream_type & os, const string_type & format,
                system::error_code & ec = throws() );
    basic_stopclock( const string_type & format, int places,
                system::error_code & ec = throws() );
    basic_stopclock( ostream_type & os, int places,
                system::error_code & ec = throws() );
    basic_stopclock( int places, const string_type & format,
                system::error_code & ec = throws() );

    basic_stopclock( ostream_type & os, const string_type & format, int places,
                system::error_code & ec = throws() );
    basic_stopclock( ostream_type & os, int places, const string_type & format,
                system::error_code & ec = throws() );


    typedef typename base_type::scoped_run scoped_run;
    typedef typename base_type::scoped_suspend scoped_suspend;
    typedef typename base_type::scoped_resume scoped_resume;
};
```

Template Class `stopclock<>`

A stopclock is a stopwatch with the ability to report elapsed time on an output stream. `stopclock<Clock>` template class is a shortcut of `basic_stopclock<Clock, typename stopwatch_reporter_default_formatter<stopwatch<Clock>>::type>` with a specific default formatter.

```
template
    < class Clock=process_cpu_clock
    , class Formatter=typename stopwatch_reporter_default_formatter<stopwatch<Clock>>::type
> class stopclock : public basic_stopclock<Clock, Formatter> {
public:
    typedef Clock clock;
    typedef stopwatch<Clock> stopwatch;
    typedef Formatter formatter;
    typedef typename Formatter::string_type string_type;
    typedef typename Formatter::char_type char_type;
    typedef typename Formatter::ostream_type ostream_type;

    explicit stopclock( system::error_code & ec = throws() );
    explicit stopclock( ostream_type & os,
                system::error_code & ec = throws() );
    explicit stopclock( const string_type & format,
                system::error_code & ec = throws() );
    explicit stopclock( int places,
                system::error_code & ec = throws() );

    stopclock( ostream_type & os, const string_type & format,
                system::error_code & ec = throws() );
    stopclock( const string_type & format, int places,
                system::error_code & ec = throws() );
    stopclock( ostream_type & os, int places,
                system::error_code & ec = throws() );
    stopclock( int places, const string_type & format,
                system::error_code & ec = throws() );

    stopclock( ostream_type & os, const string_type & format, int places,
                system::error_code & ec = throws() );
    stopclock( ostream_type & os, int places, const string_type & format,
                system::error_code & ec = throws() );


    typedef typename base_type::scoped_run scoped_run;
    typedef typename base_type::scoped_suspend scoped_suspend;
    typedef typename base_type::scoped_resume scoped_resume;
};
```

`stopclock` useful typedefs

The library provides stopclock shortcuts for all the models of `Clock`, replacing clock by stopclock.

```
typedef boost::stopwatches::stopclock< boost::chrono::system_clock >
    system_stopwatch_stopclock;
#ifdef BOOST_CHRONO_HAS_CLOCK_STEADY
typedef boost::stopwatches::stopclock< boost::chrono::steady_clock >
    steady_stopwatch_stopclock;
#endif
typedef boost::stopwatches::stopclock< boost::chrono::high_resolution_clock >
    high_resolution_stopclock;
typedef boost::stopwatches::stopclock< boost::chrono::process_real_cpu_clock >
    process_real_cpu_stopclock;
typedef boost::stopwatches::stopclock< boost::chrono::process_user_cpu_clock >
    process_user_cpu_stopclock;
typedef boost::stopwatches::stopclock< boost::chrono::process_system_cpu_clock >
    process_system_cpu_stopclock;
```

Template Class `wstopclock<>`

`wstopclock<Clock>` template class is a shortcut of `basic_wstopclock<Clock, typename stopwatch_reporter_default_formatter<stopwatch<Clock>>::type>` with a specific default formatter.

```
template
    < class Clock=process_cpu_clock
    , class Formatter=typename stopwatch_reporter_default_formatter<stopwatch<Clock>>::type
> class wstopclock : public basic_wstopclock<Clock, Formatter> {
public:
    typedef Clock clock;
    typedef stopwatch<Clock> stopwatch;
    typedef Formatter formatter;
    typedef typename Formatter::string_type string_type;
    typedef typename Formatter::char_type char_type;
    typedef typename Formatter::ostream_type ostream_type;

    explicit wstopclock( system::error_code & ec = throws() );
    explicit wstopclock( ostream_type & os,
                system::error_code & ec = throws() );
    explicit wstopclock( const string_type & format,
                system::error_code & ec = throws() );
    explicit wstopclock( int places,
                system::error_code & ec = throws() );

    wstopclock( ostream_type & os, const string_type & format,
                system::error_code & ec = throws() );
    wstopclock( const string_type & format, int places,
                system::error_code & ec = throws() );
    wstopclock( ostream_type & os, int places,
                system::error_code & ec = throws() );
    wstopclock( int places, const string_type & format,
                system::error_code & ec = throws() );

    wstopclock( ostream_type & os, const string_type & format, int places,
                system::error_code & ec = throws() );
    wstopclock( ostream_type & os, int places, const string_type & format,
                system::error_code & ec = throws() );


    typedef typename base_type::scoped_run scoped_run;
    typedef typename base_type::scoped_suspend scoped_suspend;
    typedef typename base_type::scoped_resume scoped_resume;
};
```

`wstopclock` useful typedefs

The library provides wstopclock shortcuts for all the models of `Clock`, replacing clock by wstopclock.

```
typedef boost::stopwatches::wstopclock< boost::chrono::system_clock >
    system_wstopclock;
#ifdef BOOST_CHRONO_HAS_CLOCK_STEADY
typedef boost::stopwatches::wstopclock< boost::chrono::steady_clock >
    steady_wstopclock;
#endif
typedef boost::stopwatches::wstopclock< boost::chrono::high_resolution_clock >
    high_resolution_wstopclock;
typedef boost::stopwatches::wstopclock< boost::chrono::process_real_cpu_clock >
    process_real_cpu_wstopclock;
typedef boost::stopwatches::wstopclock< boost::chrono::process_user_cpu_clock >
    process_user_cpu_wstopclock;
typedef boost::stopwatches::wstopclock< boost::chrono::process_system_cpu_clock >
    process_system_cpu_wstopclock;
```

Header <boost/stopwatches/stopclock_accumulator.hpp>

```
namespace boost { namespace stopwatches {
    template < class Clock, class Formatter >
    class basic_stopclock_accumulator;
    template < class Clock, class Formatter >
    class stopclock_accumulator;
    template < class Clock, class Formatter >
    class wstopclock_accumulator;

    typedef <see above> system_stopclock_accumulator;
    #ifdef BOOST_CHRONO_HAS_CLOCK_STEADY
    typedef <see above> steady_stopclock_accumulator;
    #endif
    typedef <see above> high_resolution_stopclock_accumulator;
    typedef <see above> process_real_cpu_stopclock_accumulator;
    typedef <see above> process_user_cpu_stopclock_accumulator;
    typedef <see above> process_system_cpu_stopclock_accumulator;

    typedef <see above> system_wstopclock_accumulator;
    #ifdef BOOST_CHRONO_HAS_CLOCK_STEADY
    typedef <see above> steady_wstopclock_accumulator;
    #endif
    typedef <see above> high_resolution_wstopclock_accumulator;
    typedef <see above> process_real_cpu_wstopclock_accumulator;
    typedef <see above> process_user_cpu_wstopclock_accumulator;
    typedef <see above> process_system_cpu_wstopclock_accumulator;
}}
```

Template Class basic_stopclock_accumulator<>

`basic_stopclock_accumulator<Clock>` template class is a shortcut of `basic_stopwatch_reporter<stopwatch_accumulator<Clock>, typename stopwatch_reporter_default_formatter<stopwatch_accumulator<Clock>>::type>`

```
template
    < class Clock=high_resolution_clock
    , class Formatter=
        typename stopwatch_reporter_default_formatter<stopwatch_accumulator<Clock>>::type
> class basic_stopclock_accumulator
    : public basic_stopwatch_reporter<stopwatch_accumulator<Clock>, Formatter> {
public:
    typedef Clock clock;
    typedef stopwatch_accumulator<Clock> stopwatch;
    typedef Formatter formatter;
    typedef typename Formatter::string_type string_type;
    typedef typename Formatter::char_type char_type;
    typedef typename Formatter::ostream_type ostream_type;

    explicit basic_stopclock_accumulator( system::error_code & ec = throws() );
    explicit basic_stopclock_accumulator( ostream_type & os,
                system::error_code & ec = throws() );
    explicit basic_stopclock_accumulator( const string_type & format,
                system::error_code & ec = throws() );
    explicit basic_stopclock_accumulator( int places,
                system::error_code & ec = throws() );

    basic_stopclock_accumulator( ostream_type & os, const string_type & format,
                system::error_code & ec = throws() );
    basic_stopclock_accumulator( const string_type & format, int places,
                system::error_code & ec = throws() );
    basic_stopclock_accumulator( ostream_type & os, int places,
                system::error_code & ec = throws() );
    basic_stopclock_accumulator( int places, const string_type & format,
                system::error_code & ec = throws() );

    basic_stopclock_accumulator( ostream_type & os, const string_type & format, int places,
                system::error_code & ec = throws() );
    basic_stopclock_accumulator( ostream_type & os, int places, const string_type & format,
                system::error_code & ec = throws() );


    typedef typename base_type::scoped_run scoped_run;
    typedef typename base_type::scoped_suspend scoped_suspend;
    typedef typename base_type::scoped_resume scoped_resume;
};
```

Template Class `stopclock_accumulator<>`

`stopclock_accumulator<Clock>` template class is a shortcut of `stopwatch_reporter<stopwatch<Clock>>` with a specific formatter.

```
template
    < class Clock=high_resolution_clock
    , class Formatter=
        typename stopwatch_reporter_default_formatter<stopwatch_accumulator<Clock>>::type
> class stopclock_accumulator
    : public basic_stopclock_accumulator<Clock, Formatter> {
public:
    typedef Clock clock;
    typedef stopwatch_accumulator<Clock> stopwatch;
    typedef Formatter formatter;
    typedef typename Formatter::string_type string_type;
    typedef typename Formatter::char_type char_type;
    typedef typename Formatter::ostream_type ostream_type;

    explicit stopclock_accumulator( system::error_code & ec = throws() );
    explicit stopclock_accumulator( ostream_type & os,
                system::error_code & ec = throws() );
    explicit stopclock_accumulator( const string_type & format,
                system::error_code & ec = throws() );
    explicit stopclock_accumulator( int places,
                system::error_code & ec = throws() );

    stopclock_accumulator( ostream_type & os, const string_type & format,
                system::error_code & ec = throws() );
    stopclock_accumulator( const string_type & format, int places,
                system::error_code & ec = throws() );
    stopclock_accumulator( ostream_type & os, int places,
                system::error_code & ec = throws() );
    stopclock_accumulator( int places, const string_type & format,
                system::error_code & ec = throws() );

    stopclock_accumulator( ostream_type & os, const string_type & format, int places,
                system::error_code & ec = throws() );
    stopclock_accumulator( ostream_type & os, int places, const string_type & format,
                system::error_code & ec = throws() );


    typedef typename base_type::scoped_run scoped_run;
    typedef typename base_type::scoped_suspend scoped_suspend;
    typedef typename base_type::scoped_resume scoped_resume;
};
```

`stopclock_accumulator` useful typedefs

The library provides stopclock_accumulator shortcuts for all the models of `Clock`, replacing clock by stopclock_accumulator.

```
typedef boost::stopwatches::stopclock_accumulator< boost::chrono::system_clock >
    system_stopclock_accumulator;
#ifdef BOOST_CHRONO_HAS_CLOCK_STEADY
typedef boost::stopwatches::stopclock_accumulator< boost::chrono::steady_clock >
    steady_stopclock_accumulator;
#endif
typedef boost::stopwatches::stopclock_accumulator< boost::chrono::high_resolution_clock >
    high_resolution_stopclock_accumulator;
typedef boost::stopwatches::stopclock_accumulator< boost::chrono::process_real_cpu_clock >
    process_real_cpu_stopclock_accumulator;
typedef boost::stopwatches::stopclock_accumulator< boost::chrono::process_user_cpu_clock >
    process_user_cpu_stopclock_accumulator;
typedef boost::stopwatches::stopclock_accumulator< boost::chrono::process_system_cpu_clock >
    process_system_cpu_stopclock_accumulator;
```

Template Class `wstopclock_accumulator<>`

`wstopclock_accumulator<Clock>` template class is a shortcut of `stopwatch_reporter<stopwatch<Clock>>` with a specific formatter.

```
template
    < class Clock=high_resolution_clock
    , class Formatter=
        typename stopwatch_reporter_default_formatter<stopwatch_accumulator<Clock>>::type
> class wstopclock_accumulator
    : public basic_wstopclock_accumulator<Clock, Formatter> {
public:
    typedef Clock clock;
    typedef stopwatch_accumulator<Clock> stopwatch;
    typedef Formatter formatter;
    typedef typename Formatter::string_type string_type;
    typedef typename Formatter::char_type char_type;
    typedef typename Formatter::ostream_type ostream_type;

    explicit wstopclock_accumulator( system::error_code & ec = throws() );
    explicit wstopclock_accumulator( ostream_type & os,
                system::error_code & ec = throws() );
    explicit wstopclock_accumulator( const string_type & format,
                system::error_code & ec = throws() );
    explicit wstopclock_accumulator( int places,
                system::error_code & ec = throws() );

    wstopclock_accumulator( ostream_type & os, const string_type & format,
                system::error_code & ec = throws() );
    wstopclock_accumulator( const string_type & format, int places,
                system::error_code & ec = throws() );
    wstopclock_accumulator( ostream_type & os, int places,
                system::error_code & ec = throws() );
    wstopclock_accumulator( int places, const string_type & format,
                system::error_code & ec = throws() );

    wstopclock_accumulator( ostream_type & os, const string_type & format, int places,
                system::error_code & ec = throws() );
    wstopclock_accumulator( ostream_type & os, int places, const string_type & format,
                system::error_code & ec = throws() );


    typedef typename base_type::scoped_run scoped_run;
    typedef typename base_type::scoped_suspend scoped_suspend;
    typedef typename base_type::scoped_resume scoped_resume;
};
```

`wstopclock_accumulator` useful typedefs

The library provides wstopclock_accumulator shortcuts for all the models of `Clock`, replacing clock by wstopclock_accumulator.

```
typedef boost::stopwatches::wstopclock_accumulator< boost::chrono::system_clock >
    system_wstopclock_accumulator;
#ifdef BOOST_CHRONO_HAS_CLOCK_STEADY
typedef boost::stopwatches::wstopclock_accumulator< boost::chrono::steady_clock >
    steady_wstopclock_accumulator;
#endif
typedef boost::stopwatches::wstopclock_accumulator< boost::chrono::high_resolution_clock >
    high_resolution_wstopclock_accumulator;
typedef boost::stopwatches::wstopclock_accumulator< boost::chrono::process_real_cpu_clock >
    process_real_cpu_wstopclock_accumulator;
typedef boost::stopwatches::wstopclock_accumulator< boost::chrono::process_user_cpu_clock >
    process_user_cpu_wstopclock_accumulator;
typedef boost::stopwatches::wstopclock_accumulator< boost::chrono::process_system_cpu_clock >
    process_system_cpu_wstopclock_accumulator;
```

Header <boost/stopwatches/scoped_stopclock.hpp>

```
namespace boost { namespace stopwatches {
    template < class Clock, class Formatter > class scoped_stopclock;
}}
```

Template Class scoped_stopclock<>

`scoped_stopclock<>`is like a `stopclock<>` but that in addition will output a scoped trace. At construction time it will output

```
{{{ <string>
```

and at destruction time

```
}}} <string> <output of stopwatch_reporter>
```

A typical ussage of this class is

```
int f1(long j)
{
  scoped_stopclock<> _(BOOST_CURRENT_FUNCTION);

  for ( long i = 0; i < j; ++i )
    std::sqrt( 123.456L );  // burn some time

  return 0;
}
```

Synopsis

```
template < class Clock=process_cpu_clock
    , class Formatter=typename stopwatch_reporter_default_formatter<stopwatch<Clock>>::type
> class scoped_stopclock
    : public stopwatch_reporter<stopwatch<Clock>, Formatter> {
public:
    typedef Clock clock;
    typedef Stopwatch stopwatch;
    typedef Formatter formatter;
    typedef typename Formatter::string_type string_type;
    typedef typename Formatter::char_type char_type;
    typedef typename Formatter::ostream_type ostream_type;

    explicit scoped_stopclock( const std::string& func,
                system::error_code & ec = throws() );
    scoped_stopclock( const std::string& func, ostream_type & os,
                system::error_code & ec = throws() );

    scoped_stopclock( const std::string& func, const string_type & format,
                system::error_code & ec = throws() );

    scoped_stopclock( const std::string& func, int places,
                system::error_code & ec = throws() );

    scoped_stopclock( const std::string& func, ostream_type & os,
                const string_type & format,
                system::error_code & ec = throws() );

    scoped_stopclock( const std::string& func, const string_type & format,
                int places, system::error_code & ec = throws() );

    scoped_stopclock( const std::string& func, ostream_type & os, int places,
                system::error_code & ec = throws() );

    scoped_stopclock( const std::string& func, int places,
                const string_type & format, system::error_code & ec = throws() );

    scoped_stopclock( const std::string& func, ostream_type & os,
                const string_type & format, int places,
                system::error_code & ec = throws() );

    scoped_stopclock( const std::string& func, ostream_type & os, int places,
                const string_type & format, system::error_code & ec = throws() );

    ~scoped_stopclock();

    typedef typename base_type::scoped_run scoped_run;
    typedef typename base_type::scoped_suspend scoped_suspend;
    typedef typename base_type::scoped_resume scoped_resume;
};
```

Stopwatch Formatters

Header <boost/stopwatches/stopwatch_formatter.hpp>

```
namespace boost { namespace stopwatches  {
    template <
        typename CharT=char,
        typename Traits=std::char_traits<CharT>,
        class Alloc=std::allocator<CharT>
    >
    class basic_stopwatch_formatter;

    typedef basic_stopwatch_formatter<char> stopwatch_formatter;
    typedef basic_stopwatch_formatter<wchar_t> wstopwatch_formatter;
}}
```

Template Class basic_stopwatch_formatter<>

`stopwatch_formatter` is a model of `Formatter`.

```
template <
    typename CharT=char,
    typename Traits=std::char_traits<CharT>,
    class Alloc=std::allocator<CharT>
>
class basic_stopwatch_formatter {
public:
    typedef std::basic_string<CharT,Traits,Alloc> string_type;
    typedef CharT char_type;
    typedef std::basic_ostream<CharT,Traits> ostream_type;
    static ostream_type &  default_os();
    static const char_type * default_format();
    static int default_places();

    template < class Stopwatch >
    static void show_time( Stopwatch & stopwatch_, const char * format, int places,
                           std::ostream & os, system::error_code & ec);
};
```

The default places is given by default_places and is 3.

The default format is "%ts\n", where

- `%d` : the result of elapsed() when the reporting is done.

The time is given using the suffix "s" following the System International d'Unites Std.

Header <boost/stopwatches/stopwatch_accumulator_formatter.hpp>

```
namespace boost { namespace stopwatches  {
    template <
        typename CharT=char,
        typename Traits=std::char_traits<CharT>,
        class Alloc=std::allocator<CharT>
    > basic_stopwatch_accumulator_formatter;
    typedef basic_stopwatch_accumulator_formatter<char> stopwatch_accumulator_formatter;
    typedef basic_stopwatch_accumulator_formatter<wchar_t> wstopwatch_accumulator_formatter;
}}
```

Template Class `basic_stopwatch_accumulator_formatter<>`

`basic_stopwatch_accumulator_formatter` is a model of `Formatter`

```
template <
    typename CharT=char,
    typename Traits=std::char_traits<CharT>,
    class Alloc=std::allocator<CharT>
class basic_stopwatch_accumulator_formatter {
public:
    typedef std::basic_string<CharT,Traits,Alloc> string_type;
    typedef CharT char_type;
    typedef std::basic_ostream<CharT,Traits> ostream_type;
    static ostream_type &  default_os();
    static const char_type * default_format();
    static int default_places();

    template <class Stopwatch >
    static void show_time( Stopwatch & stopwatch_, const char * format, int places,
                           std::ostream & os, system::error_code & ec);
};
```

The default places is given by default_places and is 3.

The default format is "%c times, sum%ss, min%ms, max%Ms, mean%as, frequency%fHz, lifetime%ls, percentage=%p%\n", where

- `%c` : the counter of the number of times the pair srat/stop has been called.
- `%s` : the sum of the samples of elapsed time between the call to start/stop.
- `%m` : the min of the samples of elapsed time between the call to start/stop.
- `%M` : the max of the samples of elapsed time between the call to start/stop.
- `%a` : the mean of the samples of elapsed time between the call to start/stop.
- `%f` : the frequency of calls to start.
- `%l` : the lifetime of the stopwatch_accumulator.
- `%p` : the percentage of time spent by this stopwatch respect to its lifetime.

The time is given using the suffix "s", the frequency is given using the suffix "Hz", both following the System International d'Unites Std.

`basic_stopwatch_accumulator_formatter` useful typedefs

The library provides basic_stopwatch_accumulator_formatter shortcuts for char and wchar_t.

```
typedef basic_stopwatch_accumulator_formatter<char>
    stopwatch_accumulator_formatter;
typedef basic_stopwatch_accumulator_formatter<wchar_t>
    wstopwatch_accumulator_formatter;
```

Header <boost/stopwatches/time_formatter.hpp>

```
namespace boost { namespace stopwatches {

    template <
        typename CharT=char,
        typename Traits=std::char_traits<CharT>,
        class Alloc=std::allocator<CharT>
    >
    class basic_time_formatter;

    typedef basic_time_formatter<char> time_formatter;
    typedef basic_time_formatter<wchar_t> wtime_formatter;

    template <>
    struct stopwatch_reporter_default_formatter<stopwatch<process_cpu_clock> > {
        typedef time_formatter type;
    };

} }
```

Template Class basic_time_formatter<>

basic_time_formatter is a model of Formatter.

```
template <typename CharT, typename Traits, class Alloc>
class basic_time_formatter {
public:
    typedef std::basic_string<CharT,Traits,Alloc> string_type;
    typedef CharT char_type;
    typedef std::basic_ostream<CharT,Traits> ostream_type;
    static ostream_type &  default_os();
    static const char_type * default_format();
    static int default_places();

    template <class Stopwatch >
    static void show_time( Stopwatch & stopwatch_
        , const char * format, int places, std::ostream & os
        , system::error_code & ec);
};
```

The default places is given by default_places and is 3.

The default format is "nreal %rs, cpu %cs (%p%), user %us, system %ss\n", where

- %r : real process clock
- %u : user process clock
- %s : system process clock
- %c : user+system process clock
- %p : percentage (user+system)/real process clock

All the units are given using the suffix "s" following the System International d'Unites Std.

basic_time_formatter useful typedefs

The library provides basic_time_formatter shortcuts for char and wchar_t.

```
typedef basic_time_formatter<char> time_formatter;
typedef basic_time_formatter<wchar_t> wtime_formatter;
```

`stopwatch_reporter_default_formatter` Specialization

```
template <>
struct stopwatch_reporter_default_formatter<stopwatch<process_cpu_clock> > {
    typedef time_formatter type;
};
```

Header `<boost/stopwatches/stopwatch_accumulator_time_formatter.hpp>`

```
namespace boost { namespace stopwatches  {
    template <
        typename CharT=char,
        typename Traits=std::char_traits<CharT>,
        class Alloc=std::allocator<CharT>
    > basic_stopwatch_accumulator_time_formatter;
   typedef basic_stopwatch_accumulator_time_formatter<char> stopwatch_accumulator_time_formatter;
   typedef basic_stopwatch_accumulator_time_formatter<wchar_t> wstopwatch_accumulator_time_format↵
ter;
}}
```

Template Class `basic_stopwatch_accumulator_time_formatter<>`

`basic_stopwatch_accumulator_time_formatter` is a model of `Formatter`

```
template <
    typename CharT=char,
    typename Traits=std::char_traits<CharT>,
    class Alloc=std::allocator<CharT>
class basic_stopwatch_accumulator_time_formatter {
public:
    typedef std::basic_string<CharT,Traits,Alloc> string_type;
    typedef CharT char_type;
    typedef std::basic_ostream<CharT,Traits> ostream_type;
    static ostream_type &  default_os();
    static const char_type * default_format();
    static int default_places();

    template <class Stopwatch >
    static void show_time( Stopwatch & stopwatch_, const char * format, int places,
                           std::ostream & os, system::error_code & ec);
};
```

The default places is given by default_places and is 3.

The default format is "%c times, sum`%s, min`%m, max`%M, mean`%a, frequency`%fHz, lifetime`%ls, percentage=%p%\n|real %rs, cpu %cs (%p%), user %us, system %ss", where

The part before the '|' corresponds to the accumulator format and the part after corresponds to the times format, which will be used for the sum, max, min and mean statistics.

- `%c` : the counter of the number of times the pair srat/stop has been called.
- `%s` : the sum of the samples of elapsed time between the call to start/stop.
- `%m` : the min of the samples of elapsed time between the call to start/stop.

- `%M` : the max of the samples of elapsed time between the call to start/stop.
- `%a` : the mean of the samples of elapsed time between the call to start/stop.
- `%f` : the frequency of calls to start.
- `%l` : the lifetime of the stopwatch_accumulator.
- `%p` : the percentage of time spent by this stopwatch respect to its lifetime.
- `%r` : real process clock
- `%u` : user process clock
- `%s` : system process clock
- `%c` : user+system process clock
- `%p` : percentage (user+system)/real process clock

The time is given using the suffix "s", the frequency is given using the suffix "Hz", both following the System International d'Unites Std.

`basic_stopwatch_accumulator_time_formatter` useful typedefs

The library provides basic_stopwatch_accumulator_time_formatter shortcuts for char and wchar_t.

```
typedef basic_stopwatch_accumulator_time_formatter<char>
    stopwatch_accumulator_time_formatter;
typedef basic_stopwatch_accumulator_time_formatter<wchar_t>
    wstopwatch_accumulator_time_formatter;
```

Header `<boost/stopwatches/t24_hours.hpp>`

```
namespace boost { namespace stopwatches {
    class t24_hours;
}}
```

Class `t24_hours`

`t24_hours` helper class decompose a duration in days, hours, minutes, seconds and nanoseconds. It can be used through its static functions or creating an instance and using its fields.

```
class t24_hours {
public:
    typedef boost::chrono::duration<boost::int_least32_t, ratio<24*3600> > days;
    typedef boost::chrono::hours hours;
    typedef boost::chrono::minutes minutes;
    typedef boost::chrono::seconds seconds;
    typedef boost::chrono::nanoseconds nanoseconds;

    template <class Rep, class Period>
    static days get_days(const boost::chrono::duration<Rep, Period>& d);

    template <class Rep, class Period>
    static hours get_hours(const boost::chrono::duration<Rep, Period>& d);

    template <class Rep, class Period>
    static minutes get_minutes(const boost::chrono::duration<Rep, Period>& d);

    template <class Rep, class Period>
    static seconds get_seconds(const boost::chrono::duration<Rep, Period>& d);

    template <class Rep, class Period>
    static nanoseconds get_nanoseconds(const boost::chrono::duration<Rep, Period>& d);

    days days_;
    hours hours_;
    minutes minutes_;
    seconds seconds_;
    nanoseconds nanoseconds_;

    template <class Rep, class Period>
        explicit t24_hours(const boost::chrono::duration<Rep, Period>& d);
};
```

Header <boost/stopwatches/t24_hours_formatter.hpp>

```
namespace boost { namespace stopwatches {

    template <
        typename CharT=char,
        typename Traits=std::char_traits<CharT>,
        class Alloc=std::allocator<CharT>
    >
    class basic_24_hours_formatter;

    typedef basic_24_hours_formatter<char> t24_hours_formatter;
    typedef basic_24_hours_formatter<wchar_t> wt24_hours_formatter;

} }
```

Template Class basic_24_hours_formatter<>

`basic_24_hours_formatter` is a model of `Formatter`.

```
template <typename CharT, typename Traits, class Alloc>
class basic_24_hours_formatter {
public:
    static std::ostream &  default_os();
    static const char * default_format();
    static int default_places();

    template <class Stopwatch >
    static void show_time( Stopwatch & stopwatch_
        , const char * format, int places, std::ostream & os
        , system::error_code & ec);
};
```

The default places is given by default_places and is 3.

The default format is "%d days(s) %h:%m:%s.%n\n", where

- %d : days
- %h : hours
- %m : minutes
- %s : seconds
- %n : nanoseconds

basic_24_hours_formatter useful typedefs

The library provides basic_24_hours_formatter shortcuts for char and wchar_t.

```
typedef basic_24_hours_formatter<char> t24_hours_formatter;
typedef basic_24_hours_formatter<wchar_t> wt24_hours_formatter;
```

Appendices

Appendix A: History

Version 0.2.0, Feb 6, 2011

Features:

- Adaptation to new Boost.Chrono inteface.

Version 0.1.0, September 10, 2010

Features:

- Extraction from Boost.Chrono of Boost.Stopwatches

Appendix B: Rationale

How reliable are these measures?

There are three cases which can lead to get unreliable measures:

- It is not possible to measure events that transpire at rates of the same order of magnitude as the clock's precision with any reliability. For example, a 10ms clock cannot be used reliably to measure elapsed times of tens of milliseconds. The library provides a [high_resolution_clock] that gives you the highest resolution time available on your platform. That will give the best precision, but can only be used for reliable measurement of events that elapse about an order of magnitude slower than that clock's precision.

```
#include <boost/chrono/chrono.hpp>
    ...
    stopclock< high_resolution_clock> _;
```

- Using a process clock in a multithreaded application will give elapsed time for the process as a whole, including threads other than the calling thread. To get time elapsed for a specific thread, use the supplied `thread_clock` which returns time elapsed for the calling thread only, if supported by the platform.

- When stopclocks are nested, usually from stopclocks appearing in each of several nested function calls, the overhead of the stopclock processing begins to be significant relative to run time of the code being measured. The innermost measurements remain accurate, but those in the outermost layers can measure too much overhead to be trustworthy.

- Nested stopclocks (usually nested function calls where each function contains a stopclock). When the nesting is deep enough, the cumulative overhead of all the stopclock functionality make the data unreliable except for the inner-most trace points. The question is, how much time is related to the application code we want to measure and how much to the fact we are meassuring and logging in inner blocks?

Most of the stopclock overhead is likely due to logging. There are two things we can do to make the difference :

- Don't flush log information while measuring elapsed time. A `stopwatch_accumulator` can make that possible, because it don't report until all the measures have been compiled and then report some statistics. Alternatively, an asynchronous stream would permit normal logging but by a thread other than the one being measured.

- Add a mechanism to track the difference between the application time and stopclock time. If a `Clock` models `SuspendibleClock` and its precision is sufficiently fine, this mechanism could suspend the `Clock`'s counting while reporting measurements and resume it thereafter.

Appendix C: Implementation Notes

Appendix D: FAQ

Why does stopwatch_reporter only display millisecond place precision when the underlying Clock has nanosecond precision?

To avoid giving the impression of precision where none exists. See Caveat emptor. You can always specify additional decimal places if you want to live dangerously.

Why does stopwatch_reporter sometimes report more cpu seconds than real seconds?

Ask your operating system supplier. The results have been inspected with a debugger, and both for Windows and Linux, that's what the OS appears to be reporting at times.

Can I obtain statistics of the time elapsed between calls to a function?

The library does not provides this feature.

What happens if I press Ctrl+C and program terminates? What log would Boost.Stopwatches output?

Appendix E: Acknowledgements

The library's started from the Beman Dawes `timer<>`, `process_clock`, `process_timer`, `run_timer` classes which are now deprecated and replaced by the `stopwatch`, `process_cpu_clock` and `stopclock` classes.

Thanks to Adrew Chinoff for its multiple suggestion on `stopwatch_accumulator`, and helping me to polish the documentation.

Thanks to Tom Tan for reporting some compiler issues with MSVC V10 beta and MinGW-gcc-4.4.0 and for the many suggestion he did concerning the `stopwatch`, `lightweight_stopwatch`, and `stopclock` classes and a deep help with wide characters implementation.

Thanks to Ronald Bock for reporting Valgind issues and for the many suggestion he did concerning the documentation.

Appendix F: Tests

In order to test you need to do.

```
bjam libs/stopwatches/test
```

You can also run a specific suite of test by doing

```
cd libs/stopwatches/test
bjam stopwatch
```

stopwatch

Name	kind	Description	Result	Ticket
test_min_max	compile	test compilation succeeds in the presence of macros min and max.	Pass	#
stopwatch_example	run	...	Pass	#
scoped_stopwatch_example	run	...	Pass	#
stopwatch_accumulator_example	run	...	Pass	#
specific_stopwatch_accumulator_example	run	...	Pass	#
stopclock_example	run	...	Pass	#
stopclock_accumulator_example	run	...	Pass	#
nested_stopclock_accumulator_example	run	...	Pass	#
loop_stopclock_accumulator_example	run	...	Pass	#
t24_hours_example	run	...	Pass	#
scoped_stopclock_example	run	...	Pass	#
timex	link	...	Pass	#
stopclock_constructor_overload_test	run	...	Pass	#
wstopclock_constructor_overload_test	run	...	Pass	#

Appendix G: Tickets

Ticket	Description	Resolution	State
1	suspend doesn't works: partial_ not initialized	initialize with duration::zero()	Closed
2	suspend doesn't works: elapsed doesn't take care of partial_	take care of partial	Closed
3	suspend doesn't works: bad use of system::error_code & ec	replace by system::error_code ec	Closed
4	Use of Specific formatters doesn't works		Closed
5	boost/chrono/scoped_suspend.hpp(31) : warning C4520: 'boost::chrono::scoped_suspend<Clock>' : multiple default constructors specified	Remove the default constructor deletion	Closed
6	suspendible_clock_test doesn't works in my mingw environement	(issue with tss)	**Open**
7	error_code not initialized	Use ec.clear() before throwing a exception.	Closed
8	Valgrind issue: Conditional jump or move depends on uninitialised value(s)	Replace the test	Closed

Appendix H: Performances

We have run some program changing how the reporting is done.

NONE: no report is done on the inner function HIGH: every call to the recursive function is reported using an stopclock SUSPEND: every call to the recursive function is reported using a using an stopclock on a suspendible clock ACCU: every call to the recursive function is tracked using a stopclock_accumulator

We have run the programm with two different clocks, high_resolution_clock and thread_clock.

The programs are either single-threaded or multi-threaded.

Two kind of inner functions are used: recursive or non recursive. In order to test the influence of nesting reports, the non recursive functions use up to 10 nesting levels, depending on its parameter.

the function at level n is defined as follows

```
void fn(long v) {
    // reporting or not
    stopclock<> _;
    // burn some time
    for ( long i = 0; i < v; ++i )
        res+=std::sqrt( res+123.456L+i );  // burn some time
    if (v==0) return;

    if (v%(n-1)==0) fn-1(v-1);
    if (v%(n-2)==0) fn-2(v-1);
    ...
    f1(v-1);
}
```

This gives a variable number in nesting reporting depending on the parameter, with a variable lifetime.

Single-Threaded Recursive function

We have run the same program and changed how the reporting is done.

The programm creates two thread of execution. the thread entry point calls a function which makes some calculation depending on its parameter and call recursivelly itself decreasing the parameter.

NONE: no report is done on the inner function HIGH: every call to the recursive function is reported using an stopclock SUSPEND: every call to the recursive function is reported using a using an stopclock on a suspendible clock ACCU: every call to the recursive function is tracked using a stopclock_accumulator

We have run the programm with two different clocks, high_resolution_clock and thread_clock.

Multi-Threaded Recursive function

We have run the same program and changed how the reporting is done.

The programm creates two thread of execution. the thread entry point calls a function which makes some calculation depending on its parameter and call recursivelly itself decreasing the parameter.

NONE: no report is done on the inner function HIGH: every call to the recursive function is reported using an stopclock SUSPEND: every call to the recursive function is reported using a using an stopclock on a suspendible clock ACCU: every call to the recursive function is tracked using a stopclock_accumulator

We have run the programm with two different clocks, high_resolution_clock and thread_clock.

Appendix I: Future plans

Tasks to do

- Complete documentation
- Fully implement error handling, with test cases.
- Fix open isues.