Benchmark execution starts with finding all benchmarks to run.

```
size_t RunSpecifiedBenchmarks(BenchmarkReporter* display_reporter,
  if (!fname.empty()) {
    file_reporter->SetErrorStream(&output_file);
  }

  std::vector<internal::BenchmarkInstance> benchmarks;
  if (!FindBenchmarksInternal(spec, &benchmarks, &Err)) {
    Out.flush();
    Err.flush();
    return 0;
  }

  if (benchmarks.empty()) {
    Err << "Failed to match any benchmarks against regex: " << spec << "\n";
    Out.flush();
    Err.flush();
    return 0;
  }

  if (FLAGS_benchmark_list_tests) {
    for (auto const& benchmark : benchmarks) {
      Out << benchmark.name().str() << "\n";
    }
  } else {
    internal::RunBenchmarks(benchmarks, display_reporter, file_reporter);
  }
```

find benchmarks

↓

returns a vector of BenchmarkInstance

After all BenchmarkInstances are found, creating a runner for each instance is the next stage.

It is important to note that a single Benchmark may create more than one BenchmarkInstance, check here:

```
// Loop through all benchmarks
for (const BenchmarkInstance& benchmark : benchmarks) {
  BenchmarkReporter::PerFamilyRunReports* reports_for_family = nullptr;
  if (benchmark.complexity() != oNone) {
    reports_for_family = &per_family_reports[benchmark.family_index()];
  }
  benchmarks_with_threads += static_cast<int>(benchmark.threads() > 1);
  runners.emplace_back(benchmark, &perfcounters, reports_for_family);
  int num_repeats_of_this_instance = runners.back().GetNumRepeats();
  num_repetitions_total +=
      static_cast<size_t>(num_repeats_of_this_instance);
  if (reports_for_family != nullptr) {
    reports_for_family->num_runs_total += num_repeats_of_this_instance;
  }
}
assert(runners.size() == benchmarks.size() && "Unexpected runner count.");
```

The next stage is creating a BenchmarkRunner for each instance.

Each runner will have to repeat R times. So, the next step is creating a data structure to ensure that a runner is repeating R times.

```
std::vector<size_t> repetition_indices;
repetition_indices.reserve(num_repetitions_total);
for (size_t runner_index = 0, num_runners = runners.size();
     runner_index != num_runners; ++runner_index) {
  const internal::BenchmarkRunner& runner = runners[runner_index];
  std::fill_n(first: std::back_inserter( [&] repetition_indices),
              n: runner.GetNumRepeats(), runner_index);
}
assert(repetition_indices.size() == num_repetitions_total &&
       "Unexpected number of repetition indexes.");
```

Create repetition-indices

See here:

An instance of Runner has repetitions defined by GetNum.
. Repeats, which comes from BenchmarkInstance, which gets from Benchmark.
By default, repetition = 1.
There are 100 ways to define repetitions:

1) while defining a benchmark with BENCHMARK macro.

2) As command line argument

```
BenchmarkRunner::BenchmarkRunner(
    BenchmarkReporter::PerFamilyRunReports* reports_for_family_)
    : b(b_),
      reports_for_family(reports_for_family_),
      parsed_benchtime_flag(ParseBenchMinTime(FLAGS_benchmark_min_time)),
      min_time(FLAGS_benchmark_dry_run
                   ? 0
                   : ComputeMinTime(b_, parsed_benchtime_flag)),
      min_warmup_time(
          FLAGS_benchmark_dry_run
              ? 0
              : ((!IsZero( n: b.min_time()) && b.min_warmup_time() > 0.0)
                     ? b.min_warmup_time()
                     : FLAGS_benchmark_min_warmup_time)),
      warmup_done(FLAGS_benchmark_dry_run ? true : !(min_warmup_time > 0.0)),
      repeats(FLAGS_benchmark_dry_run
                  ? 1
                  : (b.repetitions() != 0 ? b.repetitions()
                                          : FLAGS_benchmark_repetitions)),
```

repeats of BenchmarkRunner comes from BenchmarkInstance

or from FLAGS_benchmark_repetitions

FLAGS_benchmark_repetitions

```
// The number of runs of each benchmark. If greater than 1, the mean and
// standard deviation of the runs will be reported.
BM_DEFINE_int32(benchmark_repetitions, 1);
```

name → default value

Google benchmark creates some macros which are used to create command line flags.

Above image uses the macro: BM_DEFINE_int32 to create a variable of type int32_t with name as FLAGS_benchmark_repetitions. This variable is assigned a function which reads

a command line flag:

benchmark_repetitions.

```
#define BM_DEFINE_int32(name, default_val) \
  BENCHMARK_EXPORT int32_t FLAG(name) =    \
      benchmark::Int32FromEnv(#name, default_val)
```

```
#define FLAG(name) FLAGS_##name
```

Variable name →

FLAGS_benchmark_repetitions.

```
BENCHMARK_EXPORT
int32_t Int32FromEnv(const char* flag, int32_t default_val) {
  const std::string env_var = FlagToEnvVar(flag);
  const char* const value_str = getenv( name: env_var.c_str());
  int32_t value = default_val;
  if (value_str == nullptr ||
      !ParseInt32( src_text: std::string( s: "Environment variable ") + env_var, value_str,
                  &value)) {
    return default_val;
  }
  return value;
}
```

Function to read the command line argument, & parse to int32.

Coming back to benchmark execution, after repetition_indices are created, each runner needs to do its repetitions.

Each repetitions involves:

Warmup.

↓

Infinite loop of iterations

↓ each iteration:

Setup, N iterations, Teardown

```cpp
for (size_t repetition_index : repetition_indices) {
  internal::BenchmarkRunner& runner = runners[repetition_index];
  runner.DoOneRepetition();
  if (runner.HasRepeatsRemaining()) {
    continue;
  }
```

Each Runner undergoes its repetitions.

One Repetition

```cpp
void BenchmarkRunner::DoOneRepetition() {
  // simply use that precomputed iteration count.
  for (;;) {
    b.Setup();
    i = DoNIterations();
    b.Teardown();

    // Do we consider the results to be significant?
    // If we are doing repetitions, and the first repetition was already done,
    // it has calculated the correct iteration time, so we have run that very
    // iteration count just now. No need to calculate anything. Just report.
    // Else, the normal rules apply.
    const bool results_are_significant = !is_the_first_repetition ||
                                         has_explicit_iteration_count ||
                                         ShouldReportIterationResults(i);

    // Good, let's report them!
    if (results_are_significant) {
      break;
    }

    // Nope, bad iteration. Let's re-estimate the hopefully-sufficient
    // iteration count, and run the benchmark again...

    iters = PredictNumItersNeeded(i);
    assert(iters > i.iters &&
           "if we did more iterations than we want to do the next time, "
           "then we should have accepted the current iteration run.");
  }
```

Warm up is done before the for loop.

Every time the loop runs, Setup N iterations, and teardown are called.

If enough iterations are done, the loop breaks, else the system predicts next iterations.

```
bool BenchmarkRunner::ShouldReportIterationResults(
    const IterationResults& i) const {
// Determine if this run should be reported;
 // Either it has run for a sufficient amount of time
 // or because an error was reported.
 return (i.results.skipped_ != 0u) || FLAGS_benchmark_dry_run ||
        i.iters >= kMaxIterations ||  // Too many iterations already.
        i.seconds >=
            GetMinTimeToApply() ||  // The elapsed time is large enough.
        // CPU time is specified but the elapsed real time greatly exceeds
        // the minimum time.
        // Note that user provided timers are except from this test.
        ((i.results.real_time_used >= 5 * GetMinTimeToApply()) &&
         !b.use_manual_time());
}
```

enough iterations done?

(for now)

1) iteration time $\geq$ min bench time.

2) iterations $\geq$ max supported iterations.