```
// Helpers for generating unique variable names
#define BENCHMARK_PRIVATE_NAME(...)                                    \
  BENCHMARK_PRIVATE_CONCAT(benchmark_uniq_, BENCHMARK_PRIVATE_UNIQUE_ID, \
                           __VA_ARGS__)

#define BENCHMARK_PRIVATE_CONCAT(a, b, c) BENCHMARK_PRIVATE_CONCAT2(a, b, c)
#define BENCHMARK_PRIVATE_CONCAT2(a, b, c) a##b##c
// Helper for concatenation with macro name expansion
#define BENCHMARK_PRIVATE_CONCAT_NAME(BaseClass, Method) \
  BaseClass##_##Method##_Benchmark

#define BENCHMARK_PRIVATE_DECLARE(n)                                 \
  /* NOLINTNEXTLINE(misc-use-anonymous-namespace) */                 \
  static ::benchmark::internal::Benchmark const* const BENCHMARK_PRIVATE_NAME( \
      n) BENCHMARK_UNUSED

#define BENCHMARK(...)                                               \
  BENCHMARK_PRIVATE_DECLARE(_benchmark_) =                           \
      (::benchmark::internal::RegisterBenchmarkInternal(             \
          ::benchmark::internal::make_unique<                        \
              ::benchmark::internal::FunctionBenchmark>(#__VA_ARGS__, \
                                                        __VA_ARGS__)))
```

RegisterBenchmarkInteral adds a Benchmark pointer to an instance of BenchmarkFamilies

```
Benchmark* RegisterBenchmarkInternal(std::unique_ptr<Benchmark> bench) {
  Benchmark* bench_ptr = bench.get();
  BenchmarkFamilies* families = BenchmarkFamilies::GetInstance();
  families->AddBenchmark(std::move(bench));
  return bench_ptr;
}
```

```
class BenchmarkFamilies {
 public:
  static BenchmarkFamilies* GetInstance();

  // Registers a benchmark family and returns the index assigned to it.
  size_t AddBenchmark(std::unique_ptr<Benchmark> family);

  // Clear all registered benchmark families.
  void ClearBenchmarks();

  // Extract the list of benchmark instances that match the specified
  // regular expression.
  bool FindBenchmarks(std::string spec,
                      std::vector<BenchmarkInstance>* benchmarks,
                      std::ostream* Err);

 private:
  BenchmarkFamilies() {}

  std::vector<std::unique_ptr<Benchmark>> families_;
  Mutex mutex_;
};

BenchmarkFamilies* BenchmarkFamilies::GetInstance() {
  static BenchmarkFamilies instance;
  return &instance;
}

size_t BenchmarkFamilies::AddBenchmark(std::unique_ptr<Benchmark> family) {
  MutexLock l(mutex_);
  size_t index = families_.size();
  families_.push_back(std::move(family));
  return index;
}
```

→ collection of pointers to Benchmark

→ returns a single instance

AddBenchmark has an interesting mutex.

Imagine BENCHMARK(BM_x) & BENCHMARK(BM_y) in 2 different

benchmark files (also called translation units).

Both these statements well result in preprocessor generating the following code (or equivalent code):

→ UNIQUE

Benchmark * p =

RegisterBenchmark

static variable

Internal (·····)

This means RegisterBenchmark Internal may be called in

parallel, thus resulting in AddBenchmark in parallel.

After C11 standard, all local static variables are initialized in thread-safe manner, but the Specification is NOT TRUE for global static variables across translation units.

Hence, AddBenchmark has a mutex.