Understanding Tensorflow 2 source code

D. Gueorguiev 4/27/20

Contents

Preliminaries

The discussion in this document is based on Tensorflow master from **April 26, 2020** which I forked in my github repo https://github.com/dimitarpg13/tensorflow/. For reading this document it is assumed that the reader has a good understanding of C++ v14 , standard template library, utilities and some compiler internals.

Tensorflow globals and macros

The header file for the core Tensorflow macros and globals is tensorflow/core/platform/macros.h.

TF_PREDICT_TRUE and TF_PREDICT_FALSE

If we are not compiling with Nvidia GPU support which defines __NVCC__ global then TF_PREDICT_FALSE|TRUE uses the compiler attribute __builtin_expect(x,y) to issue a hint to the gcc compiler which branch to optimize against. This is shown on the code snippet Code Snippet: TF_PREDICT macro

Code Snippet: TF_PREDICT macro

```
// Compilers can be told that a certain branch is not likely to be taken
// (for instance, a CHECK failure), and use that information in static
// analysis. Giving it this information can help it optimize for the
// common case in the absence of better information (ie.
// -fprofile-arcs).
//
// We need to disable this for GPU builds, though, since nvcc8 and older
// don't recognize `__builtin_expect` as a builtin, and fail compilation.
#if (!defined(__NVCC__)) && \
   (TF_HAS_BUILTIN(__builtin_expect) || (defined(__GNUC__) && __GNUC__ >= 3))
#define TF_PREDICT_FALSE(x) (__builtin_expect(x, 0))
#define TF_PREDICT_TRUE(x) (__builtin_expect(!!(x), 1))
```

```
#else
#define TF_PREDICT_FALSE(x) (x)
#define TF_PREDICT_TRUE(x) (x)
#endif
```

TF_DISALLOW_COPY_AND_ASSIGN

Another useful macro is TF_DISALLOW_COPY_AND_ASSIGN which deletes the copy constructor of the specified type and the assignment operator as well. It is shown on the code snippet Code Snippet: Macro TF_DISALLOW_COPY_AND_ASSIGN below:

Code Snippet: Macro TF_DISALLOW_COPY_AND_ASSIGN

```
// A macro to disallow the copy constructor and operator= functions
// This is usually placed in the private: declarations for a class.
#define TF_DISALLOW_COPY_AND_ASSIGN(TypeName) \
  TypeName(const TypeName&) = delete;                       \
  void operator=(const TypeName&) = delete
```

TF_ARRAYSIZE

The macro TF_ARRAYSIZE returns the number of elements of a generic pointer-based array of any type. Notice that any generic container which implements the dereference operator * even if it is thin enough it will still cause TF_ARRAYSIZE to throw floating point exception which is not standard C++ exception which can be caught by try / catch block but rather it can be intercepted by reading the state of the floating point exception flag using std::fesetexceptflag. Refer to code snippet Code Snippet: Macro TF_ARRAYSIZE

Code Snippet: Macro TF_ARRAYSIZE

```
// The TF_ARRAYSIZE(arr) macro returns the # of elements in an array arr.
//
// The expression TF_ARRAYSIZE(a) is a compile-time constant of type
// size_t.
#define TF_ARRAYSIZE(a)                                  \
  ((sizeof(a) / sizeof(*(a))) /                          \
   static_cast<size_t>(!(sizeof(a) % sizeof(*(a)))))
```

TF_FALLTHROUGH_INTENDED

The next code snippet deals with fallthrough in the case when they are intended. In case we have a new enough compiler we define the macro TF_FALLTHROUGH_INTENDED to set a specific compiler attribute clang::fallthrough which tells the compiler that the specific fallthrough was intended so it should not emit a warning/error on it. If the compiler is not clang or we are not compiling with C++11 then we resort to a do-while trick to convince the compiler not to issue a warning on fallthrough. Details in the code snipper Macro TF_FALLTHROUGH_INTENDED.

Macro TF_FALLTHROUGH_INTENDED

```
#if defined(__GXX_EXPERIMENTAL_CXX0X__) || __cplusplus >= 201103L || \
   (defined(_MSC_VER) && _MSC_VER >= 1900)
// Define this to 1 if the code is compiled in C++11 mode; leave it
// undefined otherwise.  Do NOT define it to 0 -- that causes
// '#ifdef LANG_CXX11' to behave differently from '#if LANG_CXX11'.
#define LANG_CXX11 1
#endif

#if defined(__clang__) && defined(LANG_CXX11) && defined(__has_warning)
#if __has_feature(cxx_attributes) && __has_warning("-Wimplicit-fallthrough")
#define TF_FALLTHROUGH_INTENDED [[clang::fallthrough]]  // NOLINT
#endif
#endif

#ifndef TF_FALLTHROUGH_INTENDED
#define TF_FALLTHROUGH_INTENDED \
  do {                          \
  } while (0)
#endif
```

Macros utilizing compiler attributes

And here are some macros utilizing compiler attributes in their GCC implementation:

```
// Compiler supports GCC-style attributes
```

TF_ATTRIBUTE_NORETURN: hint to the compiler that the function does not return; implemented with the ((noreturn)) attribute.

```
#define TF_ATTRIBUTE_NORETURN __attribute__((noreturn))
```

TF_ATTRIBUTE_ALWAYS_INLINE: hint to the compiler to inline the current function even if the compiler is not in optimizing mode. Implemented with the ((always_inline)) attribute.

```
#define TF_ATTRIBUTE_ALWAYS_INLINE __attribute__((always_inline))
```

TF_ATTRIBUTE_NOINLINE: hint to the compiler not to inline the current function even if the compiler is in optimizing mode. Implemented with the ((noinline)) attribute.

```
#define TF_ATTRIBUTE_NOINLINE __attribute__((noinline))
```

TF_ATTRIBUTE_UNUSED: hint to the compiler not to issue a warning on unused variable as the variable is expected to be unused.

```
#define TF_ATTRIBUTE_UNUSED __attribute__((unused))
```

TF_ATTRIBUTE_COLD: hint to the compiler that the function is cold. Implemented through the ((cold)) attribute. The cold attribute is used to inform the compiler that a function is unlikely executed. The function is optimized for size rather than speed and on many targets it is placed into special subsection of the text section so all cold functions appears close together improving code locality of non-cold parts of program. The paths leading to call of cold functions within code are marked as unlikely by the branch prediction mechanism. It is thus useful to mark functions used to handle unlikely conditions, such as perror, as cold to improve optimization of hot functions that do call marked functions in rare occasions.

```
#define TF_ATTRIBUTE_COLD __attribute__((cold))
```

TF_ATTRIBUTE_WEAK: marks the symbol to be weak rather than global by using the attribute ((weak)). This is primarily useful in defining library functions which can be overridden in user code, though it can also be used with non-function declarations. Weak symbols are supported for ELF targets, and also for a.out targets when using the GNU assembler and linker.

```
#define TF_ATTRIBUTE_WEAK __attribute__((weak))
```

TF_PACKED: marks the struct or union as packed using the compiler attribute ((packed)). The packed attribute, attached to the struct or union type definition, specifies that each member of the structure or union is placed to minimize the memory required. When attached to an enum definition, it indicates that the smallest integral type should be used. Specifying this attribute for struct and union types is equivalent to specifying the packed attribute on each of the structure or union members.

```
#define TF_PACKED __attribute__((packed))
```

TF_MUST_USE_RESULT: uses attribute ((warn_unused_result)) which causes a warning to be emitted if a caller of the function with this attribute does not use its return value. This is useful for functions where not checking the result is either a security problem or always a bug, such as realloc.

```
#define TF_MUST_USE_RESULT __attribute__((warn_unused_result))
```

TF_PRINTF_ATTRIBUTE: issues a printf string where string_index is the index of the string to print within the param list.

```
#define TF_PRINTF_ATTRIBUTE(string_index, first_to_check) \
  __attribute__((__format__(__printf__, string_index, first_to_check)))

#define TF_SCANF_ATTRIBUTE(string_index, first_to_check) \
  __attribute__((__format__(__scanf__, string_index, first_to_check)))
```

Tensorflow Logging Internals

The main Tensorflow logging header file is tensorflow/core/platform/default/logging.h. This header file contains various macros and helper classes for implementing TensorFlow-centric logging internals.
Few logging global constants defining the logging level are shown in code snippet Code Snippet: Logging Level constants

Code Snippet: Logging Level constants

```
namespace tensorflow {
const int INFO = 0;          // base_logging::INFO;
const int WARNING = 1;       // base_logging::WARNING;
const int ERROR = 2;         // base_logging::ERROR;
const int FATAL = 3;         // base_logging::FATAL;
const int NUM_SEVERITIES = 4;  // base_logging::NUM_SEVERITIES;
```

}

Class LogMessage

The base class which controls the logging behavior is **LogMessage** and its declaration is shown in code snippet Code Snippet: Classs LogMessage

Code Snippet: Classs LogMessage

```
namespace tensorflow {
namespace internal {

class LogMessage : public std::basic_ostringstream<char> {
 public:
  LogMessage(const char* fname, int line, int severity);
  ~LogMessage() override;

  // Change the location of the log message.
  LogMessage& AtLocation(const char* fname, int line);

  // Returns the minimum log level for VLOG statements.
  // E.g., if MinVLogLevel() is 2, then VLOG(2) statements will produce output,
  // but VLOG(3) will not. Defaults to 0.
  static int64 MinVLogLevel();

  // Returns whether VLOG level lvl is activated for the file fname.
  //
  // E.g. if the environment variable TF_CPP_VMODULE contains foo=3 and fname is
  // foo.cc and lvl is <= 3, this will return true. It will also return true if
  // the level is lower or equal to TF_CPP_MIN_VLOG_LEVEL (default zero).
  //
  // It is expected that the result of this query will be cached in the VLOG-ing
  // call site to avoid repeated lookups. This routine performs a hash-map
  // access against the VLOG-ing specification provided by the env var.
  static bool VmoduleActivated(const char* fname, int level);

 protected:
  void GenerateLogMessage();

 private:
  const char* fname_;
  int line_;
  int severity_;
};
} // namespace internal
} // namespace tensorflow
```
This class is the entry point for the core Tensorflow logging framework and all log entries are recorded via an instance of this class.
Important method in LogMessage is the static method LogMessage::VmoduleActivate(..) which accepts a zero-terminated char array containing a file name, and an int with the logging level. This method returns if the VLOG logging level lvl is activated for the specified file name fname. If the environment variable TF_CPP_VMODULE contains foo=3 and fname is foo.cc and lvl is <= 3, this will return true. It will also return true if the level is lower or equal to TF_CPP_MIN_VLOG_LEVEL (default zero). The value of the env var TF_CPP_MODULE is supposed to be of the form "foo=1,bar=2,baz=3".

Implementation of this method is given in code snippet Code Snippet: Method VmoduleActivated

Code Snippet: Method VmoduleActivated
```
bool LogMessage::VmoduleActivated(const char* fname, int level) {
  if (level <= MinVLogLevel()) {
    return true;
  }
  static VmoduleMap* vmodules = VmodulesMapFromEnv();
  if (TF_PREDICT_TRUE(vmodules == nullptr)) {
    return false;
  }
  const char* last_slash = strrchr(fname, '/');
  const char* module_start = last_slash == nullptr ? fname : last_slash + 1;
  const char* dot_after = strchr(module_start, '.');
  const char* module_limit =
      dot_after == nullptr ? strchr(fname, '\0') : dot_after;
  StringData module(module_start, module_limit - module_start);
  auto it = vmodules->find(module);
  return it != vmodules->end() && it->second >= level;
}
```
Both the implementation of LogMessage::VmoduleActivated(..) and the global function VmodulesMapFromEnv() use the container StringData. This container provides its own Hasher functor which is based on the DJB hash function given as :

h(0) = 5381
h(i) = 33 * h(i-1) ^ str[i]

The definition of StringData is shown on code snippet Code Snippet: Struct StringData below:

Code Snippet: Struct StringData
```
struct StringData {
  struct Hasher {
    size_t operator()(const StringData& sdata) const {
      // For dependency reasons, we cannot use hash.h here. Use DJBHash instead.
      size_t hash = 5381;
      const char* data = sdata.data;
      for (const char* top = data + sdata.size; data < top; ++data) {
        hash = ((hash << 5) + hash) + (*data);
      }
      return hash;
    }
  };

  StringData() = default;
  StringData(const char* data, size_t size) : data(data), size(size) {}

  bool operator==(const StringData& rhs) const {
    return size == rhs.size && memcmp(data, rhs.data, size) == 0;
  }

  const char* data = nullptr;
  size_t size = 0;
};
```

The implementation of the global function VmodulesMapFromEnv() is shown on code snippet Code Snippet: Function VmodulesMapFromEnv. Notice how the Hasher of StringData is used as a key in the hash table which is returned as a result of the invocation of VmodulesMapFromEnv().

Code Snippet: Function VmodulesMapFromEnv

```
using VmoduleMap = std::unordered_map<StringData, int, StringData::Hasher>;

// Returns a mapping from module name to VLOG level, derived from the
// TF_CPP_VMODULE environment variable; ownership is transferred to the caller.
VmoduleMap* VmodulesMapFromEnv() {
  // The value of the env var is supposed to be of the form:
  //    "foo=1,bar=2,baz=3"
  const char* env = getenv("TF_CPP_VMODULE");
  if (env == nullptr) {
    // If there is no TF_CPP_VMODULE configuration (most common case), return
    // nullptr so that the ShouldVlogModule() API can fast bail out of it.
    return nullptr;
  }
  // The memory returned by getenv() can be invalidated by following getenv() or
  // setenv() calls. And since we keep references to it in the VmoduleMap in
  // form of StringData objects, make a copy of it.
  const char* env_data = strdup(env);
  VmoduleMap* result = new VmoduleMap();
```

```
 while (true) {
  const char* eq = strchr(env_data, '=');
  if (eq == nullptr) {
   break;
  }
  const char* after_eq = eq + 1;

  // Comma either points at the next comma delimiter, or at a null terminator.
  // We check that the integer we parse ends at this delimiter.
  const char* comma = strchr(after_eq, ',');
  const char* new_env_data;
  if (comma == nullptr) {
   comma = strchr(after_eq, '\0');
   new_env_data = comma;
  } else {
   new_env_data = comma + 1;
  }
  (*result)[StringData(env_data, eq - env_data)] =
    ParseInteger(after_eq, comma - after_eq);
  env_data = new_env_data;
 }
 return result;
}
```

The code of the ParseInteger is shown on code snippet Code Snippet: Function ParseInteger – it uses std::istringstream to convert the string to int64. Notice the comment why the safe version of str to int64 was not used:

Code Snippet: Function ParseInteger

```
int ParseInteger(const char* str, size_t size) {
 // Ideally we would use env_var / safe_strto64, but it is
 // hard to use here without pulling in a lot of dependencies,
 // so we use std:istringstream instead
 string integer_str(str, size);
 std::     ss(integer_str);
 int level = 0;
 ss >> level;
 return level;
}
```

The new log entry is added to the associated file by the protected method GenerateLogMessage(). This method is called from the destructor of the base class LogMessage. For platforms different than Andorid GenerateLogMessage() is shown on code snippet Code Snippet: Method GenerateLogMessage. The thread id is included in the log entry if the environment vairable TF_CPP_LOG_THREAD_ID has been defined. Notice the last argument of the fprintf statement str().c_str() which prints the message payload which was streamed into this LogMessage instance via the `<<` streaming operator.

Code Snippet: Method GenerateLogMessage
```
void LogMessage::GenerateLogMessage() {
  static bool log_thread_id = EmitThreadIdFromEnv();
  uint64 now_micros = EnvTime::NowMicros();
  time_t now_seconds = static_cast<time_t>(now_micros / 1000000);
  int32 micros_remainder = static_cast<int32>(now_micros % 1000000);
  const size_t time_buffer_size = 30;
  char time_buffer[time_buffer_size];
  strftime(time_buffer, time_buffer_size, "%Y-%m-%d %H:%M:%S",
           localtime(&now_seconds));
  const size_t tid_buffer_size = 10;
  char tid_buffer[tid_buffer_size] = "";
  if (log_thread_id) {
    snprintf(tid_buffer, sizeof(tid_buffer), " %7u",
             absl::base_internal::GetTID());
  }
  // TODO(jeff,sanjay): Replace this with something that logs through the env.
  fprintf(stderr, "%s.%06d: %c%s %s:%d] %s\n", time_buffer, micros_remainder,
          "IWEF"[severity_], tid_buffer, fname_, line_, str().c_str());
}

bool EmitThreadIdFromEnv() {
  const char* tf_env_var_val = getenv("TF_CPP_LOG_THREAD_ID");
  return tf_env_var_val == nullptr
             ? false
             : ParseInteger(tf_env_var_val, strlen(tf_env_var_val)) != 0;
}
```

And here is one descendant class from LogMessage – LogMessageFatal and another one which shares base class with LogMessage – LogMessageNull shown in the code snippet Code Snippet: LogMessageFatal and LogMessageNull. Notice the use of the compiler attribute noreturn in the LogMessageFatal destructor giving hint to the compiler that this method does not return graciously. Also notice the compiler attrobute cold adorning the LogMessageFatal constructor giving a hint to the compiler to optimizer for size instead of for speed.

Code Snippet: LogMessageFatal and LogMessageNull
```
// LogMessageFatal ensures the process will exit in failure after
// logging this message.
class LogMessageFatal : public LogMessage {
 public:
  LogMessageFatal(const char* file, int line) TF_ATTRIBUTE_COLD;
  TF_ATTRIBUTE_NORETURN ~LogMessageFatal() override;
};

LogMessageFatal::LogMessageFatal(const char* file, int line)
    : LogMessage(file, line, FATAL) {}
LogMessageFatal::~LogMessageFatal() {
  // abort() ensures we don't return (we promised we would not via
  // ATTRIBUTE_NORETURN).
  GenerateLogMessage();
  abort();
}

// LogMessageNull supports the DVLOG macro by simply dropping any log messages.
class LogMessageNull : public std::basic_ostringstream<char> {
 public:
  LogMessageNull() {}
  ~LogMessageNull() override {}
};
```

Minimum Log and Vlog level from the environment

Two global functions used in the macros LOG and VLOG to set the minimum log level which will allow emitting log messages are MinLogLevelFromEnv() and MinVLogLevelFromEnv(). Disabling the logging is relevant when the code is being tested in fuzzer mode using tools such as LLVM's LibFuzzer. For automated Fuzz testing see the following wiki page – Fuzzing.

```
int64 MinLogLevelFromEnv() {
  // We don't want to print logs during fuzzing as that would slow fuzzing down
  // by almost 2x. So, if we are in fuzzing mode (not just running a test), we
  // return a value so that nothing is actually printed. Since LOG uses >=
  // (see ~LogMessage in this file) to see if log messages need to be printed,
  // the value we're interested on to disable printing is the maximum severity.
  // See also http://llvm.org/docs/LibFuzzer.html#fuzzer-friendly-build-mode
#ifdef FUZZING_BUILD_MODE_UNSAFE_FOR_PRODUCTION
  return tensorflow::NUM_SEVERITIES;
#else
  const char* tf_env_var_val = getenv("TF_CPP_MIN_LOG_LEVEL");
  return LogLevelStrToInt(tf_env_var_val);
#endif
}

int64 MinVLogLevelFromEnv() {
  // We don't want to print logs during fuzzing as that would slow fuzzing down
  // by almost 2x. So, if we are in fuzzing mode (not just running a test), we
  // return a value so that nothing is actually printed. Since VLOG uses <=
  // (see VLOG_IS_ON in logging.h) to see if log messages need to be printed,
  // the value we're interested on to disable printing is 0.
  // See also http://llvm.org/docs/LibFuzzer.html#fuzzer-friendly-build-mode
#ifdef FUZZING_BUILD_MODE_UNSAFE_FOR_PRODUCTION
  return 0;
#else
  const char* tf_env_var_val = getenv("TF_CPP_MIN_VLOG_LEVEL");
  return LogLevelStrToInt(tf_env_var_val);
#endif
}
```

The LOG macro

The LOG macro is defined in the following snippet. The code is self-explanatory given the earlier discussion in this section.

```
#define _TF_LOG_INFO \
  ::tensorflow::internal::LogMessage(__FILE__, __LINE__, ::tensorflow::INFO)
#define _TF_LOG_WARNING \
  ::tensorflow::internal::LogMessage(__FILE__, __LINE__, ::tensorflow::WARNING)
#define _TF_LOG_ERROR \
  ::tensorflow::internal::LogMessage(__FILE__, __LINE__, ::tensorflow::ERROR)
#define _TF_LOG_FATAL \
  ::tensorflow::internal::LogMessageFatal(__FILE__, __LINE__)

#define _TF_LOG_QFATAL _TF_LOG_FATAL

#define LOG(severity) _TF_LOG_##severity
```

The VLOG macro

The difference between the LOG macro and the VLOG macro is that the latter enables logging only if the specified logging level is enabled for the specified module (file) using the vmodule settings. This is achieved through a use of lambda function which invokes the static method LogMessage::VmoduleActivated(fname,level) and depending on the result it either creates on the stack a LogMessage instance or it executes a noop of type void. The purpose of the helper struct Voidifier (see below) is , as the name suggests, to alter the return type to void when invoking the constructor of LogMessage and thereby avoiding compilation error "*second operand to the conditional operator is of type 'void', but the third operand is neither a throw-expression nor of type 'void'*" when instantiating the VLOG macro. For the details see code snippet Code Snippet: Voidifier, VLOG_IS_ON and VLOG macros.

Code Snippet: Voidifier, VLOG_IS_ON and VLOG macros

```
#ifdef IS_MOBILE_PLATFORM
// Uses the lower operator & precedence to voidify a LogMessage reference, so
// that the ternary VLOG() implementation is balanced, type wise.
struct Voidifier {
  template <typename T>
  void operator&(const T&)const {}
};
// Turn VLOG off when under mobile devices for considerations of binary size.
#define VLOG_IS_ON(lvl) ((lvl) <= 0)

#else

// Otherwise, set TF_CPP_MIN_VLOG_LEVEL environment to update minimum log level
// of VLOG, or TF_CPP_VMODULE to set the minimum log level for individual
// translation units.
#define VLOG_IS_ON(lvl)                                                  \
  (([](int level, const char* fname) {                                   \
    static const bool vmodule_activated =                                \
        ::tensorflow::internal::LogMessage::VmoduleActivated(fname, level); \
    return vmodule_activated;                                            \
  })(lvl, __FILE__))

#endif

#define VLOG(level)                                                      \
  TF_PREDICT_TRUE(!VLOG_IS_ON(level))                                    \
  ? (void)0                                                              \
  : ::tensorflow::internal::Voidifier() &                                \
        ::tensorflow::internal::LogMessage(__FILE__, __LINE__,           \
                                           tensorflow::INFO)

// `DVLOG` behaves like `VLOG` in debug mode (i.e. `#ifndef NDEBUG`).
// Otherwise, it compiles away and does nothing.
#ifndef NDEBUG
#define DVLOG VLOG
#else
#define DVLOG(verbose_level) \
  while (false && (verbose_level) > 0) ::tensorflow::internal::LogMessageNull()
#endif
```

Helper classes requiring synchronization – LogEveryNState, LogFirstNState, LogEveryPow2State, LogEveryNSecState

The code for the discussion below is shown in code snippet Code Snippet: LogEveryXState helper classes. The first class LogEveryNState, as the name suggests, logs every n-th entry where the atomicity of the transaction is maintained by an std::atomic int restricted to std::memory_order_relaxed which only guarantees that when the counter is being read and updated both the read and the update happen atomically i.e. no synchronization guarantees are offered with this memory order. Note that the concept

of atomicity of the counter updates and reads does not imply atomicity of LossyIncrement execution. This means that if a thread invokes LogEveryNState::ShouldLog(n) there will be no guarantee that LossyIncrement(&counter_) will increment LogEveryNState::counter_.
Looking into the next helper class LogFirstNState, LogFirstNState::ShouldLog(n) returns true if the current LogFirstNState::counter_ value is less than n.
Class LogEveryPow2State logs every time LogEveryPow2State::counter_ becomes 2^n for some integer n. Class LogEveryNSecState is more interesting as it logs the associated state every n seconds. This is achieved by to std::atomic integers – one int32 representing counter_ and another int64 representing the number of cycles until the next log time (next_log_time_cycles_); both initialized with 0 value.
The cycles counting and the conversion of cycles into seconds occurs in the method LogEveryNSecState::ShouldLog(seconds). Let us take a look into it. First we increment the internal counter by invoking LossyIncrement(&counter_). Then we get the number of cycles from the clock by using Abseil internal clock - absl::base_internal::CycleClock::Now(). Next we read the value of the atomic next_log_time_cycles_ into the local variable next_cycles and compare it with the current value for the number of cycles obtained from the Abseil internal clock and if the latter is smaller than the former we return false immediatelly. Otherwise we enter the while clause where using atomic<T>::compare_exchange_weak(..) we compare the current value of next_log_time_cycles_ with the one just loaded into next_cycles and if those have the same value we exchange the current value of next_log_time_cycles_ with now_cycles + seconds * absl::base_internal::CycleClock::Frequency() and return true immediatelly. If those do not have the same value which is the case when another thread executes the same do-while loop and has just incremented the last value of next_log_time_cycles_ with seconds * absl::base_internal::CycleClock::Frequency() then our thread repeats the comparison of the current value of next_log_time_cycles_ with the current value for the number of cycles obtained from the Abseil internal clock inside the body of the do-while loop. Usually this do-while loop converges after just one iteration. Note that the do-while loop is necessary because we are using the weak form of compare-and-exchange which is allowed to fail spuriously but has better multi-threaded performance compared to the strong version.

Code Snippet: LogEveryXState helper classes
```
class LogEveryNState {
 public:
  bool ShouldLog(int n);
  uint32_t counter() { return counter_.load(std::memory_order_relaxed); }

 private:
  std::atomic<uint32> counter_{0};
};

class LogFirstNState {
 public:
  bool ShouldLog(int n);
  uint32 counter() { return counter_.load(std::memory_order_relaxed); }

 private:
  std::atomic<uint32> counter_{0};
};

class LogEveryPow2State {
 public:
  bool ShouldLog(int ignored);
  uint32 counter() { return counter_.load(std::memory_order_relaxed); }

 private:
  std::atomic<uint32> counter_{0};
};

class LogEveryNSecState {
 public:
  bool ShouldLog(double seconds);
  uint32 counter() { return counter_.load(std::memory_order_relaxed); }

 private:
  std::atomic<uint32> counter_{0};
  // Cycle count according to CycleClock that we should next log at.
  std::atomic<int64> next_log_time_cycles_{0};
};

// The following code behaves like AtomicStatsCounter::LossyAdd() for
// speed since it is fine to lose occasional updates.
// Returns old value of *counter.
uint32 LossyIncrement(std::atomic<uint32>* counter) {
  const uint32 value = counter->load(std::memory_order_relaxed);
  counter->store(value + 1, std::memory_order_relaxed);
  return value;
}

bool LogEveryNState::ShouldLog(int n) {
  return n != 0 && (LossyIncrement(&counter_) % n) == 0;
}

bool LogFirstNState::ShouldLog(int n) {
  const uint32 counter_value = counter_.load(std::memory_order_relaxed);
  if (counter_value < n) {
    counter_.store(counter_value + 1, std::memory_order_relaxed);
    return true;
  }
  return false;
}
```

```
bool LogEveryPow2State::ShouldLog(int ignored) {
 const uint32 new_value = LossyIncrement(&counter_) + 1;
 return (new_value & (new_value - 1)) == 0;
}

bool LogEveryNSecState::ShouldLog(double seconds) {
 LossyIncrement(&counter_);
 const int64 now_cycles = absl::base_internal::CycleClock::Now();
 int64 next_cycles = next_log_time_cycles_.load(std::memory_order_relaxed);
 do {
   if (now_cycles <= next_cycles) return false;
 } while (!next_log_time_cycles_.compare_exchange_weak(
     next_cycles,
     now_cycles + seconds * absl::base_internal::CycleClock::Frequency(),
     std::memory_order_relaxed, std::memory_order_relaxed));
 return true;
}
```

The helper macro LOGGING_INTERNAL_STATEFUL_CONDITION and the LOG_EVERY_### macros

Let us digest the helper macro LOGGING_INTERNAL_STATEFUL_CONDITION.
The outer for-loop

```
 for (bool logging_internal_stateful_condition_do_log(condition);  \
      logging_internal_stateful_condition_do_log;                  \
      logging_internal_stateful_condition_do_log = false)          \
```

executes exactly once and its only purpose is to make sure that the two inner for-loops run only once. Inside the macro LOG_EVERY_N LogMessage is instantiated locally on the stack via the LOG macro after instantiating the macro LOGGING_INTERNAL_STATEFUL_CONDITION. This gives us a clue how the LOGGING_INTERNAL_STATEFUL_CONDITION should be used. Here is an example code illustrating the useage of the macro LOG_EVERY_N:

```
 for (const auto& user : all_users) {
     LOG_EVERY_N(INFO, 1000) << "Processing user #" << COUNTER;
     ProcessUser(user);
 }
```

Since LogMessage inherits from std::basic_ostringstream<char> both the string "Processing user #" and the value of COUNTER are sreamed into a LogMessage instance since the condition argument is set to true inside LOG_EVERY_N. The streamed string is output into a file via LogMessage::GenerateLogMessage when the LogMessage instance allocated on the stack goes out of scope. Notice the use of special keyword COUNTER in the code example above. This is connected to the declaration of the local int32 variable COUNTER initialized with the value of LogEvery***X***State.counter()

which is streamed into the LogMessage instance by the costruct `<< COUNTER`. Notice the clever use of ABSL_ATTRIBUTE_UNUSED which suppresses compiler warning on unused variable and is defined in **absl/base/attributes.h:549** as

```
#define ABSL_ATTRIBUTE_UNUSED __attribute__((__unused__))
```

The code of the discussed macros is shown on code snippet Code Snippet: LOGGING_INTERNAL_STATEFUL_CONDITION and LOG_EVERY_X macros below.

Code Snippet: LOGGING_INTERNAL_STATEFUL_CONDITION and LOG_EVERY_X macros

```
// This macro has a lot going on!
//
// * A local static (`logging_internal_stateful_condition_state`) is
//   declared in a scope such that each `LOG_EVERY_N` (etc.) line has its own
//   state.
// * `COUNTER`, the third variable, is used to support `<< COUNTER`. It is not
//   mangled, so shadowing can be a problem, albeit more of a
//   shoot-yourself-in-the-foot one.  Don't name your variables `COUNTER`.
// * A single for loop can declare state and also test
//   `condition && state.ShouldLog()`, but there's no way to constrain it to run
//   only once (or not at all) without declaring another variable.  The outer
//   for-loop declares this variable (`do_log`).
// * Using for loops instead of if statements means there's no risk of an
//   ambiguous dangling else statement.
#define LOGGING_INTERNAL_STATEFUL_CONDITION(kind, condition, arg)   \
  for (bool logging_internal_stateful_condition_do_log(condition);            \
       logging_internal_stateful_condition_do_log;                            \
       logging_internal_stateful_condition_do_log = false)                    \
    for (static ::tensorflow::internal::Log##kind##State                      \
             logging_internal_stateful_condition_state;                       \
         logging_internal_stateful_condition_do_log &&                        \
         logging_internal_stateful_condition_state.ShouldLog(arg);            \
         logging_internal_stateful_condition_do_log = false)                  \
      for (const uint32_t COUNTER ABSL_ATTRIBUTE_UNUSED =                     \
               logging_internal_stateful_condition_state.counter();           \
           logging_internal_stateful_condition_do_log;                        \
           logging_internal_stateful_condition_do_log = false)


// An instance of `LOG_EVERY_N` increments a hidden zero-initialized counter
// every time execution passes through it and logs the specified message when
// the counter's value is a multiple of `n`, doing nothing otherwise.  Each
// instance has its own counter.  The counter's value can be logged by streaming
// the symbol `COUNTER`.  `LOG_EVERY_N` is thread-safe.
// Example:
```

```
//
//   for (const auto& user : all_users) {
//     LOG_EVERY_N(INFO, 1000) << "Processing user #" << COUNTER;
//     ProcessUser(user);
//   }
#define LOG_EVERY_N(severity, n)                                   \
  LOGGING_INTERNAL_STATEFUL_CONDITION(EveryN, true, n) \
  LOG(severity)
// `LOG_FIRST_N` behaves like `LOG_EVERY_N` except that the specified message is
// logged when the counter's value is less than `n`.  `LOG_FIRST_N` is
// thread-safe.
#define LOG_FIRST_N(severity, n)                                  \
  LOGGING_INTERNAL_STATEFUL_CONDITION(FirstN, true, n) \
  LOG(severity)
// `LOG_EVERY_POW_2` behaves like `LOG_EVERY_N` except that the specified
// message is logged when the counter's value is a power of 2.
// `LOG_EVERY_POW_2` is thread-safe.
#define LOG_EVERY_POW_2(severity)                                  \
  LOGGING_INTERNAL_STATEFUL_CONDITION(EveryPow2, true, 0) \
  LOG(severity)
// An instance of `LOG_EVERY_N_SEC` uses a hidden state variable to log the
// specified message at most once every `n_seconds`.  A hidden counter of
// executions (whether a message is logged or not) is also maintained and can be
// logged by streaming the symbol `COUNTER`.  `LOG_EVERY_N_SEC` is thread-safe.
// Example:
//
//   LOG_EVERY_N_SEC(INFO, 2.5) << "Got " << COUNTER << " cookies so far";
#define LOG_EVERY_N_SEC(severity, n_seconds)                              \
  LOGGING_INTERNAL_STATEFUL_CONDITION(EveryNSec, true, n_seconds) \
  LOG(severity)
```

The CHECK_X macros and related internals

The Code Snippet: CHECK macro instantiates on the stack LogMessageFatal which logs the string supplied as argument and executes abort() causing the executing process to exit abnormally.

Code Snippet: CHECK macro

```
// CHECK dies with a fatal error if condition is not true.  It is *not*
// controlled by NDEBUG, so the check will be executed regardless of
// compilation mode.  Therefore, it is safe to do things like:
//    CHECK(fp->Write(x) == 4)
#define CHECK(condition)                \
  if (TF_PREDICT_FALSE(!(condition))) \
  LOG(FATAL) << "Check failed: " #condition " "
```

Next we have an inline helper function Code Snippet: MakeCheckOpValueString which redirects an instance of the type T to an output stream os. It has few specializations which produce taylored output sent to the output stream.

Code Snippet: MakeCheckOpValueString

```
// This formats a value for a failing CHECK_XX statement.  Ordinarily,
// it uses the definition for operator<<, with a few special cases below.
template <typename T>
inline void MakeCheckOpValueString(std::ostream* os, const T& v) {
  (*os) << v;
}

// Overrides for char types provide readable values for unprintable
// characters.
template <>
void MakeCheckOpValueString(std::ostream* os, const char& v) {
  if (v >= 32 && v <= 126) {
    (*os) << "'" << v << "'";
  } else {
    (*os) << "char value " << static_cast<short>(v);
  }
}

template <>
void MakeCheckOpValueString(std::ostream* os, const signed char& v) {
  if (v >= 32 && v <= 126) {
    (*os) << "'" << v << "'";
  } else {
    (*os) << "signed char value " << static_cast<short>(v);
  }
}

template <>
void MakeCheckOpValueString(std::ostream* os, const unsigned char& v) {
  if (v >= 32 && v <= 126) {
    (*os) << "'" << v << "'";
  } else {
    (*os) << "unsigned char value " << static_cast<unsigned short>(v);
  }
}


#if LANG_CXX11
```

```
// We need an explicit specialization for std::nullptr_t.
template <>
void MakeCheckOpValueString(std::ostream* os, const std::nullptr_t& p) {
  (*os) << "nullptr";
}
#endif
```

The next struct Code Snippet: CheckOpString is a helper struct which sole purpose is to hint the compiler that the comparison encoded in the overloaded cast operator CheckOpString::bool() is unlikely to be false **in a heavily optimized code** (using compiler option -O3).

Code Snippet: CheckOpString

```
// A container for a string pointer which can be evaluated to a bool -
// true iff the pointer is non-NULL.
struct CheckOpString {
  CheckOpString(string* str) : str_(str) {}
  // No destructor: if str_ is non-NULL, we're about to LOG(FATAL),
  // so there's no point in cleaning up str_.
  operator bool() const { return TF_PREDICT_FALSE(str_ != NULL); }
  string* str_;
};
```

Another helper class for formating error strings for binary expressions (operator and two operands) is Code Snippet: CheckOpMessageBuilder. The constructor of this class inserts the text specified as constructor argument concatenated with left parenthesis to the ostrings stream instance which it creates. The two methods CheckOpMessageBuilder::ForVar1() and CheckOpMessageBuilder::ForVar2() preprare the stream for each of the two operands prepending the string “versus” before the second operand and appending the right parenthesis after it.

Code Snippet: CheckOpMessageBuilder

```
// A helper class for formatting "expr (V1 vs. V2)" in a CHECK_XX
// statement.  See MakeCheckOpString for sample usage.  Other
// approaches were considered: use of a template method (e.g.,
// base::BuildCheckOpString(exprtext, base::Print<T1>, &v1,
// base::Print<T2>, &v2), however this approach has complications
// related to volatile arguments and function-pointer arguments).
class CheckOpMessageBuilder {
 public:
  // Inserts "exprtext" and " (" to the stream.
  explicit CheckOpMessageBuilder(const char* exprtext);
  // Deletes "stream_".
  ~CheckOpMessageBuilder();
  // For inserting the first variable.
  std::ostream* ForVar1() { return stream_; }
  // For inserting the second variable (adds an intermediate " vs. ").
  std::ostream* ForVar2();
  // Get the result (inserts the closing ")").
  string* NewString();

 private:
  std::ostringstream* stream_;
};

CheckOpMessageBuilder::CheckOpMessageBuilder(const char* exprtext)
    : stream_(new std::ostringstream) {
  *stream_ << "Check failed: " << exprtext << " (";
}

CheckOpMessageBuilder::~CheckOpMessageBuilder() { delete stream_; }

std::ostream* CheckOpMessageBuilder::ForVar2() {
  *stream_ << " vs. ";
  return stream_;
}

string* CheckOpMessageBuilder::NewString() {
  *stream_ << ")";
  return new string(stream_->str());
}
```

Code Snippet: MakeCheckOpString is a templetized helper function for building the error message string (as the comment below indicates). The function is adorned with the **noinline** compiler attribute to make sure the code is optimized for size rather than for performance as it will be invoked rarely but will be present on many places. This function makes use of CheckOpMessageBuilder and MakeCheckOpValueString with which it builds and formats the error message for the binary operator and the two operands v1 and v2.

Code Snippet: MakeCheckOpString

```
// Build the error message string. Specify no inlining for code size.
template <typename T1, typename T2>
string* MakeCheckOpString(const T1& v1, const T2& v2,
                          const char* exprtext) TF_ATTRIBUTE_NOINLINE {
  CheckOpMessageBuilder comb(exprtext);
  MakeCheckOpValueString(comb.ForVar1(), v1);
  MakeCheckOpValueString(comb.ForVar2(), v2);
  return comb.NewString();
}
```

Follow few Helper functions for the CHECK_OP macro.
As the comment indicates the specialization of the helper inline string* name##Impl(T1&,T2&) is due limitation of generic template instantiation when using declared but undefined static const and anonymous enum values as the arguments v1 and v2. *Note: the limitation of using anonymous enum values as typed template arguments has been lifted since c++0x as the text in* `4.3.1[temp.arg.type]/2` *"A local type, a type with no linkage, an unnamed type or a type compounded from any of these types shall not be used as a template-argument for a template type-parameter." has been removed from the standard.*
Next we instantiate all of those helper functions for the following operators : ==, !=, <=, <, >=, >. Finally, we undefine the macro TF_DEFINE_CHECK_OP_IMPL as the we are done with it (the Impl functions have been created) and we do not want namespace polution.

Helper functions for the CHECK_OP macro

```
// Helper functions for CHECK_OP macro.
// The (int, int) specialization works around the issue that the compiler
// will not instantiate the template version of the function on values of
// unnamed enum type - see comment below.
// The (size_t, int) and (int, size_t) specialization are to handle unsigned
// comparison errors while still being thorough with the comparison.
#define TF_DEFINE_CHECK_OP_IMPL(name, op)                         \
  template <typename T1, typename T2>                             \
  inline string* name##Impl(const T1& v1, const T2& v2,           \
                            const char* exprtext) {               \
    if (TF_PREDICT_TRUE(v1 op v2))                                \
      return NULL;                                                \
    else                                                          \
      return ::tensorflow::internal::MakeCheckOpString(v1, v2, exprtext); \
  }                                                               \
  inline string* name##Impl(int v1, int v2, const char* exprtext) { \
    return name##Impl<int, int>(v1, v2, exprtext);                \
  }                                                               \
  inline string* name##Impl(const size_t v1, const int v2,        \
                            const char* exprtext) {               \
    if (TF_PREDICT_FALSE(v2 < 0)) {                               \
      return ::tensorflow::internal::MakeCheckOpString(v1, v2, exprtext); \
    }                                                             \
    return name##Impl<size_t, size_t>(v1, v2, exprtext);          \
  }                                                               \
  inline string* name##Impl(const int v1, const size_t v2,        \
                            const char* exprtext) {               \
    if (TF_PREDICT_FALSE(v2 >= std::numeric_limits<int>::max())) { \
      return ::tensorflow::internal::MakeCheckOpString(v1, v2, exprtext); \
    }                                                             \
    const size_t uval = (size_t)((unsigned)v2);                        \
    return name##Impl<size_t, size_t>(v1, uval, exprtext);             \
  }

// We use the full name Check_EQ, Check_NE, etc. in case the file including
// base/logging.h provides its own #defines for the simpler names EQ, NE, etc.
// This happens if, for example, those are used as token names in a
// yacc grammar.
TF_DEFINE_CHECK_OP_IMPL(Check_EQ,
                        ==)  // Compilation error with CHECK_EQ(NULL, x)?
TF_DEFINE_CHECK_OP_IMPL(Check_NE, !=)  // Use CHECK(x == NULL) instead.
TF_DEFINE_CHECK_OP_IMPL(Check_LE, <=)
TF_DEFINE_CHECK_OP_IMPL(Check_LT, <)
TF_DEFINE_CHECK_OP_IMPL(Check_GE, >=)
TF_DEFINE_CHECK_OP_IMPL(Check_GT, >)
#undef TF_DEFINE_CHECK_OP_IMPL
```

Now we are going to look into another helper macro Code Snippet: CHECK_OP_LOG which is needed in order to define and implement CHECK_OP. As in the code snippet related to the Helper functions for the CHECK_OP macro we end up with a bunch of specializations to accommodate static const integral types declared in classes such as the values of in-class anonymous enums.

Code Snippet: CHECK_OP_LOG

```
// Function is overloaded for integral types to allow static const
// integrals declared in classes and not defined to be used as arguments to
// CHECK* macros. It's not encouraged though.
template <typename T>
inline const T& GetReferenceableValue(const T& t) {
  return t;
}
inline char GetReferenceableValue(char t) { return t; }
inline unsigned char GetReferenceableValue(unsigned char t) { return t; }
inline signed char GetReferenceableValue(signed char t) { return t; }
inline short GetReferenceableValue(short t) { return t; }
inline unsigned short GetReferenceableValue(unsigned short t) { return t; }
inline int GetReferenceableValue(int t) { return t; }
inline unsigned int GetReferenceableValue(unsigned int t) { return t; }
inline long GetReferenceableValue(long t) { return t; }
inline unsigned long GetReferenceableValue(unsigned long t) { return t; }
inline long long GetReferenceableValue(long long t) { return t; }
inline unsigned long long GetReferenceableValue(unsigned long long t) {
  return t;
}
```

```
// In optimized mode, use CheckOpString to hint to compiler that
// the while condition is unlikely.
#define CHECK_OP_LOG(name, op, val1, val2)                    \
  while (::tensorflow::internal::CheckOpString _result =      \
           ::tensorflow::internal::name##Impl(                \
             ::tensorflow::internal::GetReferenceableValue(val1), \
             ::tensorflow::internal::GetReferenceableValue(val2), \
             #val1 " " #op " " #val2))                        \
  ::tensorflow::internal::LogMessageFatal(__FILE__, __LINE__) << *(_result.str_)
```

Finally we are ready to look into the Code Snippet: CHECK_OP macro and derivatives – CHECK_{EQ|NE|LE|LT|GE|GT}, DCHECK_{EQ|NE|LE|LT|GE|GT} and QCHECK_{EQ|NE|LE|LT|GE|GT}. All of the CHECK_{EQ|NE|LE|LT|GE|GT} macros are defined in terms of CHECK_OP where the first and second argument are the helper function (see Helper functions for the CHECK_OP macro) and the corresponding binary operation. The only exception is the CHECK_NOTNULL macro which relies on the templetized function CheckNotNull which is shown below as well. Note that if not in DEBUG mode the macros DCHECK, DCHECK_{EQ|NE|LE|LT|GE|GT} do nothing.

Code Snippet: CHECK_OP macro and derivatives

```
#define CHECK_OP(name, op, val1, val2) CHECK_OP_LOG(name, op, val1, val2)

// CHECK_EQ/NE/...
#define CHECK_EQ(val1, val2) CHECK_OP(Check_EQ, ==, val1, val2)
#define CHECK_NE(val1, val2) CHECK_OP(Check_NE, !=, val1, val2)
#define CHECK_LE(val1, val2) CHECK_OP(Check_LE, <=, val1, val2)
#define CHECK_LT(val1, val2) CHECK_OP(Check_LT, <, val1, val2)
#define CHECK_GE(val1, val2) CHECK_OP(Check_GE, >=, val1, val2)
#define CHECK_GT(val1, val2) CHECK_OP(Check_GT, >, val1, val2)
#define CHECK_NOTNULL(val)                          \
  ::tensorflow::internal::CheckNotNull(__FILE__, __LINE__, \
                                  "'" #val "' Must be non NULL", (val))
template <typename T>
T&& CheckNotNull(const char* file, int line, const char* exprtext, T&& t) {
  if (t == nullptr) {
    LogMessageFatal(file, line) << string(exprtext);
  }
  return std::forward<T>(t);
}

#ifndef NDEBUG
// DCHECK_EQ/NE/...
#define DCHECK(condition) CHECK(condition)
#define DCHECK_EQ(val1, val2) CHECK_EQ(val1, val2)
```

```
#define DCHECK_NE(val1, val2) CHECK_NE(val1, val2)
#define DCHECK_LE(val1, val2) CHECK_LE(val1, val2)
#define DCHECK_LT(val1, val2) CHECK_LT(val1, val2)
#define DCHECK_GE(val1, val2) CHECK_GE(val1, val2)
#define DCHECK_GT(val1, val2) CHECK_GT(val1, val2)

#else

#define DCHECK(condition) \
  while (false && (condition)) LOG(FATAL)

// NDEBUG is defined, so DCHECK_EQ(x, y) and so on do nothing.
// However, we still want the compiler to parse x and y, because
// we don't want to lose potentially useful errors and warnings.
// _DCHECK_NOP is a helper, and should not be used outside of this file.
#define _TF_DCHECK_NOP(x, y) \
  while (false && ((void)(x), (void)(y), 0)) LOG(FATAL)

#define DCHECK_EQ(x, y) _TF_DCHECK_NOP(x, y)
#define DCHECK_NE(x, y) _TF_DCHECK_NOP(x, y)
#define DCHECK_LE(x, y) _TF_DCHECK_NOP(x, y)
#define DCHECK_LT(x, y) _TF_DCHECK_NOP(x, y)
#define DCHECK_GE(x, y) _TF_DCHECK_NOP(x, y)
#define DCHECK_GT(x, y) _TF_DCHECK_NOP(x, y)

#endif

// These are for when you don't want a CHECK failure to print a verbose
// stack trace.  The implementation of CHECK* in this file already doesn't.
#define QCHECK(condition) CHECK(condition)
#define QCHECK_EQ(x, y) CHECK_EQ(x, y)
#define QCHECK_NE(x, y) CHECK_NE(x, y)
#define QCHECK_LE(x, y) CHECK_LE(x, y)
#define QCHECK_LT(x, y) CHECK_LT(x, y)
#define QCHECK_GE(x, y) CHECK_GE(x, y)
#define QCHECK_GT(x, y) CHECK_GT(x, y)
```

Introduction into The Abseil C++ common libraries

The Abseil C++ common libraries were originally part of the internal Google C++ codebase. They were made open source in September 2017 by introducing the Abseil github repository for current work and all releases.
Abseil contains the following C++ library components:

- **base**

The base library contains abseil fundamentals used throughout the whole suite such as initialization code and other code which all other Abseil code depends on. Code within base may not depend on any other code (other than the C++ standard library).

- **algorithm**

The algorithm library contains additions to the C++ <algorithm> library and container-based versions of such algorithms.

- **container**

The container library contains additional STL-style containers, including Abseil's unordered "Swiss table" containers.

- **debugging**

The debugging library contains code useful for enabling leak checks, and stacktrace and symbolization utilities.

- **hash**

The hash library contains the hashing framework and default hash functor implementations for hashable types in Abseil.

- **memory**

The memory library contains C++11-compatible versions of std::make_unique() and related memory management facilities.

- **meta**

The meta library contains C++11-compatible versions of type checks available within C++14 and C++17 versions of the C++ <type_traits> library.

- **numeric**

The numeric library contains C++11-compatible 128-bit integers.

- **strings**

The strings library contains a variety of strings routines and utilities, including a C++11-compatible version of the C++17 std::string_view type.
synchronization
The synchronization library contains concurrency primitives (Abseil's absl::Mutex class, an alternative to std::mutex) and a variety of synchronization abstractions.

- **time**

The time library contains abstractions for computing with absolute points in time, durations of time, and formatting and parsing time within time zones.

- **types**

The types library contains non-container utility types, like a C++11-compatible version of the C++17 std::optional type.

- **utility**

The utility library contains utility and helper code.

The Abseil Hashing Framework

Components of the Abseil Hashing Framework

Components of the Abseil Hashing Framework are-
absl::Hash<T> functor invoking the hasher

AbslHashValue – an extension point that allows for extending types to support hashing without requirement to define hashing algorithm. This is a friend function for the new container class for which we would like to add hashing support.
HashState – a type-erased class which manipulates and updates the hash state represented by the template parameter H. HashState contains members such as HashState::combine() and Hash::combine_contiguous() which are called inside AbslHashValue implementing the hashing support for the new container.

Discussion on Hash<T>

So we have the Hash functor defined in absl/hash/hash.h as:

```
template <typename T>
using Hash = absl::hash_internal::Hash<T>;
```

and inside absl/hash/internal/hash.h as:

```
// HashImpl

// Add a private base class to make sure this type is not an aggregate.
// Aggregates can be aggregate initialized even if the default constructor is
// deleted.
struct PoisonedHash : private AggregateBarrier {
  PoisonedHash() = delete;
  PoisonedHash(const PoisonedHash&) = delete;
  PoisonedHash& operator=(const PoisonedHash&) = delete;
};

template <typename T>
struct HashImpl {
  size_t operator()(const T& value) const { return CityHashState::hash(value); }
};

template <typename T>
struct Hash
    : absl::conditional_t<is_hashable<T>::value, HashImpl<T>, PoisonedHash> {};
```

The enabled specialization of Hash is HashImpl which in turn invokes CityHashState::hash to obtain the hash of the value. The disabled specialization is PoisonedHash which does not allow instantiation and assignment. This is achieved through the usage of constructor deleters and assignment operator deleter. The privately inherited class AggregateBarrier serves as protection against aggergate initialization which would have been allowed had the struct PoisonedHash have no private base classes.
The enabled Hash implementation defines a functor through function call operator() overload. The class Hash supports the following types out-of-the-box:

- All integral types (including bool)
- All enum types
- All floating-point types (although hashing them is discouraged)

- All pointer types, including nullptr_t
- std::pair<T1, T2>, if T1 and T2 are hashable
- std::tuple<Ts...>, if all the Ts... are hashable
- std::unique_ptr and std::shared_ptr
- All string-like types including:
- std::string
- std::string_view (as well as any instance of std::basic_string that
- uses char and std::char_traits)
- All the standard sequence containers (provided the elements are hashable)
- All the standard ordered associative containers (provided the elements are hashable)
- absl types such as the following:
- absl::string_view
- absl::InlinedVector
- absl::FixedArray
- absl::uint128
- absl::Time, absl::Duration, and absl::TimeZone

When invoked, absl::Hash<T> searches for supplied hash functions in the following order:

- Natively supported types out of the box (see above)
- Types for which an AbslHashValue() overload is provided (such as user-defined types). See "Adding Type Support to absl::Hash" below.
- Types which define a HASH_NAMESPACE::hash<T> specialization (aka __gnu_cxx::hash<T> for gcc/Clang or stdext::hash<T> for MSVC)
- Types which define a std::hash<T> specialization

The fallback to legacy hash functions exists mainly for backwards compatibility. If you have a choice, prefer defining an AbslHashValue overload instead of specializing any legacy hash functors.

Adding Type Support to absl::Hash

Let us start with an example
Suppose we have a class Circle for which we want to add hashing:

```
class Circle {
 public:
  ...
 private:
  std::pair<int, int> center_;
  int radius_;
};
```

To add hashing support to Circle, we simply need to add a free (non-member) function AbslHashValue(), and return the combined hash state of the existing hash state and the class state. You can add such a free function using a friend declaration within the body of the class:

```
class Circle {
 public:
  ...
  template <typename H>
  friend H AbslHashValue(H h, const Circle& c) {
    return H::combine(std::move(h), c.center_, c.radius_);
  }
  ...
};
```

To summarize:
In order to add support for your user-defined type, add a proper AbslHashValue() overload as a free (non-member) function. The overload will take an existing hash state and should combine that state with state from the type.

Example:

```
template <typename H>
H AbslHashValue(H state, const MyType& v) {
  return H::combine(std::move(state), v.field1, ..., v.fieldN);
}
```

where (field1, ..., fieldN) are the members you would use on your operator== to define equality.

Notice that AbslHashValue is not a class member, but an ordinary function. An AbslHashValue overload for a type should only be declared in the same file and namespace as said type. The proper AbslHashValue implementation for a given type will be discovered via ADL.

Note: unlike std::hash, absl::Hash should never be specialized. It must only be extended by adding AbslHashValue() overloads.

Note on Argument-Dependent Lookup (ADL):
Argument-Dependent Lookup is the set of rules for looking up the unqualified function names in function-call expressions, including implicit function calls to overloaded operators. These function names are looked up in the namespaces of their arguments in addition to the scopes and namespaces considered by the usual unqualified name lookup.

Argument-dependent lookup makes it possible to use operators defined in a different namespace.

The pattern of Type Erasure

Here are some preliminaries on the type-erased HashState. There is a very good article on type erasure in C++ by Arthur O'Dwyer: https://quuxplusone.github.io/blog/2019/03/18/what-is-type-erasure/
In his post Arthur O'Dwyer has constructed a Code Snippet: classic example of type-erased Callback struct shown below. In it a helper WrappingCallback is used to erase the type of Callback

Code Snippet: classic example of type-erased Callback struct
```
struct AbstractCallback {
  virtual int call(int) const = 0;
  virtual ~AbstractCallback() = default;
};

template<class T>
struct WrappingCallback : AbstractCallback {
  T cb_;
  explicit WrappingCallback(T &&cb) : cb_(std::move(cb)) {}
  int call(int x) const override { return cb_(x); }
};

struct Callback {
  std::unique_ptr<AbstractCallback> ptr_;

  template<class T>
  Callback(T t) {
    ptr_ = std::make_unique<WrappingCallback<T>>(std::move(t));
  }
  int operator()(int x) const {
    return ptr_->call(x);
  }
};

int run_twice(const Callback& callback) {
  return callback(1) + callback(1);
}

int main() {
  int y = run_twice([](int x) { return x+1; });
  assert(y == 4);
}
```

Discussion on HashState

HashState is defined in absl/hash/hash.h.
The comments in the beginning of hash.h are quite complete and need to be read carefully.

Let us see how the type-erasure is implemented in HashState which is declared and defined in absl/hash/hash.h. HashStateBase is declared and defined in absl/hash/internal/hash.h which is shown later.
```
class HashState : public hash_internal::HashStateBase<HashState> {
```

```
public:
  // HashState::Create()
  //
  // Create a new `HashState` instance that wraps `state`. All calls to
  // `combine()` and `combine_contiguous()` on the new instance will be
  // redirected to the original `state` object. The `state` object must outlive
  // the `HashState` instance.
  template <typename T>
  static HashState Create(T* state) {
    HashState s;
    s.Init(state);
    return s;
  }
. . .

private:
  HashState() = default;
  template <typename T>
  static void CombineContiguousImpl(void* p, const unsigned char* first,
                                    size_t size) {
    T& state = *static_cast<T*>(p);
    state = T::combine_contiguous(std::move(state), first, size);
  }

  template <typename T>
  void Init(T* state) {
    state_ = state;
    combine_contiguous_ = &CombineContiguousImpl<T>;
  }

. . .
  void* state_;
  void (*combine_contiguous_)(void*, const unsigned char*, size_t);
}
```

The Hash State Concept, and using HashState for Type Erasure

A hash state object represents an intermediate state in the computation of an unspecified hash algorithm. HashStateBase provides a CRTP style base class for hash state implementations. For details on CRTP see Notes on CRTP (Curiously Recurring Template Pattern) below. Developers adding type support for absl::Hash should not rely on any parts of the state object other than the following member functions:

- HashStateBase::combine()

- HashStateBase::combine_contiguous()

A derived hash state class of type `H` must provide a static member function with a signature similar to the following:

```
static H combine_contiguous(H state, const unsigned char*, size_t)
```

HashStateBase will provide a complete implementation for a hash state object in terms of this method.

Example:

```
// Use CRTP to define your derived class.
struct MyHashState : HashStateBase<MyHashState> {
    static H combine_contiguous(H state, const unsigned char*, size_t);
    using MyHashState::HashStateBase::combine;
    using MyHashState::HashStateBase::combine_contiguous;
};
```

The absl::Hash framework relies on the Concept of a "hash state". Such a hash state is used in several places:

- Within existing implementations of absl::Hash<T> to store the hashed state of an object. Note that it is up to the implementation how it stores such state. A hash table, for example, may mix the state to produce an integer value; a testing framework may simply hold a vector of that state.
- Within implementations of AbslHashValue() used to extend user-defined types. (See "Adding Type Support to absl::Hash" below.)
- Inside a HashState, providing type erasure for the concept of a hash state, which you can use to extend the absl::Hash framework for types that are otherwise difficult to extend using AbslHashValue(). (See the HashState class below.)

The "hash state" concept contains two member functions for mixing hash state:

H::combine(state, values...) combines an arbitrary number of values into a hash state, returning the updated state. Note that the existing hash state is move-only and must be
passed by value.

Each of the value types T must be hashable by H.

Note 1:

```
state = H::combine(std::move(state), value1, value2, value3);
```

must be guaranteed to produce the same hash expansion as

```
state = H::combine(std::move(state), value1);
state = H::combine(std::move(state), value2);
```

state = H::combine(std::move(state), value3);

H::combine_contiguous(state, data, size) combines a contiguous array of **size** elements into a hash state, returning the updated state. Note that the existing hash state is move-only and must be passed by value.

Note 2:
state = H::combine_contiguous(std::move(state), data, size); need ***not*** be guaranteed to produce the same hash expansion as a loop (it may perform internal optimizations). If you need this guarantee, use a loop instead.

To summarize:
HashState is a type erased version of the hash state concept, for use in user-defined AbslHashValue implementations that can't use templates (such as PImpl classes, virtual functions, etc.). The type erasure adds overhead so it should be avoided unless necessary.

Note 3: This wrapper will only erase calls to:
   combine_contiguous(H, const unsigned char*, size_t)

All other calls will be handled internally and will not invoke overloads provided by the wrapped class.

Users of this class should still define a template AbslHashValue function, but can use absl::HashState::Create(&state) to erase the type of the hash state and dispatch to their private hashing logic.

This state can be used like any other hash state. In particular, you can call HashState::combine() and HashState::combine_contiguous() on it.

Example:

```
class Interface {
   public:
    template <typename H>
    friend H AbslHashValue(H state, const Interface& value) {
      state = H::combine(std::move(state), std::type_index(typeid(*this)));
      value.HashValue(absl::HashState::Create(&state));
      return state;
    }
   private:
    virtual void HashValue(absl::HashState state) const = 0;
};

class Impl : Interface {
   private:
    void HashValue(absl::HashState state) const override {
      absl::HashState::combine(std::move(state), v1_, v2_);
    }
    int v1_;
    std::string v2_;
};
```

The contents of absl/hash/hash.h is shown below:

```
// This header defines the Abseil `hash` library and the Abseil hashing
// framework. This framework consists of the following:
//
//   * The `absl::Hash` functor, which is used to invoke the hasher within the
//     Abseil hashing framework. `absl::Hash<T>` supports most basic types and
//     a number of Abseil types out of the box.
//   * `AbslHashValue`, an extension point that allows you to extend types to
//     support Abseil hashing without requiring you to define a hashing
//     algorithm.
//   * `HashState`, a type-erased class which implements the manipulation of the
//     hash state (H) itself, contains member functions `combine()` and
//     `combine_contiguous()`, which you can use to contribute to an existing
//     hash state when hashing your types.
//
// Unlike `std::hash` or other hashing frameworks, the Abseil hashing framework
// provides most of its utility by abstracting away the hash algorithm (and its
// implementation) entirely. Instead, a type invokes the Abseil hashing
// framework by simply combining its state with the state of known, hashable
// types. Hashing of that combined state is separately done by `absl::Hash`.
//
// One should assume that a hash algorithm is chosen randomly at the start of
// each process.  E.g., absl::Hash<int>()(9) in one process and
// absl::Hash<int>()(9) in another process are likely to differ.
//
// Example:
//
//   // Suppose we have a class `Circle` for which we want to add hashing:
//   class Circle {
//    public:
//     ...
//    private:
//     std::pair<int, int> center_;
//     int radius_;
//   };
//
```

```
//   // To add hashing support to `Circle`, we simply need to add a free
//   // (non-member) function `AbslHashValue()`, and return the combined hash
//   // state of the existing hash state and the class state. You can add such a
//   // free function using a friend declaration within the body of the class:
//   class Circle {
//    public:
//     ...
//     template <typename H>
//     friend H AbslHashValue(H h, const Circle& c) {
//       return H::combine(std::move(h), c.center_, c.radius_);
//     }
//     ...
//   };
//
// For more information, see Adding Type Support to `absl::Hash` below.
//
#ifndef ABSL_HASH_HASH_H_
#define ABSL_HASH_HASH_H_

#include "absl/hash/internal/hash.h"

namespace absl {

// -----------------------------------------------------------------------------
// `absl::Hash`
// -----------------------------------------------------------------------------
//
// `absl::Hash<T>` is a convenient general-purpose hash functor for any type `T`
// satisfying any of the following conditions (in order):
//
//  * T is an arithmetic or pointer type
//  * T defines an overload for `AbslHashValue(H, const T&)` for an arbitrary
//    hash state `H`.
//  - T defines a specialization of `HASH_NAMESPACE::hash<T>`
//  - T defines a specialization of `std::hash<T>`
//
// `absl::Hash` intrinsically supports the following types:
//
//   * All integral types (including bool)
//   * All enum types
//   * All floating-point types (although hashing them is discouraged)
//   * All pointer types, including nullptr_t
//   * std::pair<T1, T2>, if T1 and T2 are hashable
//   * std::tuple<Ts...>, if all the Ts... are hashable
//   * std::unique_ptr and std::shared_ptr
//   * All string-like types including:
//     * std::string
//     * std::string_view (as well as any instance of std::basic_string that
//       uses char and std::char_traits)
//  * All the standard sequence containers (provided the elements are hashable)
//  * All the standard ordered associative containers (provided the elements are
//    hashable)
//  * absl types such as the following:
//    * absl::string_view
//    * absl::InlinedVector
//    * absl::FixedArray
//    * absl::uint128
//    * absl::Time, absl::Duration, and absl::TimeZone
//
// Note: the list above is not meant to be exhaustive. Additional type support
// may be added, in which case the above list will be updated.
//
// -----------------------------------------------------------------------------
// absl::Hash Invocation Evaluation
// -----------------------------------------------------------------------------
//
// When invoked, `absl::Hash<T>` searches for supplied hash functions in the
// following order:
//
//   * Natively supported types out of the box (see above)
//   * Types for which an `AbslHashValue()` overload is provided (such as
//     user-defined types). See "Adding Type Support to `absl::Hash`" below.
//   * Types which define a `HASH_NAMESPACE::hash<T>` specialization (aka
//     `__gnu_cxx::hash<T>` for gcc/Clang or `stdext::hash<T>` for MSVC)
//   * Types which define a `std::hash<T>` specialization
//
// The fallback to legacy hash functions exists mainly for backwards
// compatibility. If you have a choice, prefer defining an `AbslHashValue`
// overload instead of specializing any legacy hash functors.
//
// -----------------------------------------------------------------------------
// The Hash State Concept, and using `HashState` for Type Erasure
// -----------------------------------------------------------------------------
//
// The `absl::Hash` framework relies on the Concept of a "hash state." Such a
// hash state is used in several places:
//
// * Within existing implementations of `absl::Hash<T>` to store the hashed
//   state of an object. Note that it is up to the implementation how it stores
//   such state. A hash table, for example, may mix the state to produce an
//   integer value; a testing framework may simply hold a vector of that state.
// * Within implementations of `AbslHashValue()` used to extend user-defined
//   types. (See "Adding Type Support to absl::Hash" below.)
// * Inside a `HashState`, providing type erasure for the concept of a hash
//   state, which you can use to extend the `absl::Hash` framework for types
//   that are otherwise difficult to extend using `AbslHashValue()`. (See the
//   `HashState` class below.)
//
// The "hash state" concept contains two member functions for mixing hash state:
//
// * `H::combine(state, values...)`
//
//   Combines an arbitrary number of values into a hash state, returning the
//   updated state. Note that the existing hash state is move-only and must be
//   passed by value.
//
//   Each of the value types T must be hashable by H.
//
//   NOTE:
//
//     state = H::combine(std::move(state), value1, value2, value3);
//
//   must be guaranteed to produce the same hash expansion as
//
//     state = H::combine(std::move(state), value1);
//     state = H::combine(std::move(state), value2);
//     state = H::combine(std::move(state), value3);
//
// * `H::combine_contiguous(state, data, size)`
//
//    Combines a contiguous array of `size` elements into a hash state,
//    returning the updated state. Note that the existing hash state is
//    move-only and must be passed by value.
//
//    NOTE:
//
//      state = H::combine_contiguous(std::move(state), data, size);
//
//    need NOT be guaranteed to produce the same hash expansion as a loop
//    (it may perform internal optimizations). If you need this guarantee, use a
//    loop instead.
//
```

```
// -----------------------------------------------------------------------------
// Adding Type Support to `absl::Hash`
// -----------------------------------------------------------------------------
//
// To add support for your user-defined type, add a proper `AbslHashValue()`
// overload as a free (non-member) function. The overload will take an
// existing hash state and should combine that state with state from the type.
//
// Example:
//
//   template <typename H>
//   H AbslHashValue(H state, const MyType& v) {
//     return H::combine(std::move(state), v.field1, ..., v.fieldN);
//   }
//
// where `(field1, ..., fieldN)` are the members you would use on your
// `operator==` to define equality.
//
// Notice that `AbslHashValue` is not a class member, but an ordinary function.
// An `AbslHashValue` overload for a type should only be declared in the same
// file and namespace as said type. The proper `AbslHashValue` implementation
// for a given type will be discovered via ADL.
//
// Note: unlike `std::hash', `absl::Hash` should never be specialized. It must
// only be extended by adding `AbslHashValue()` overloads.
//
template <typename T>
using Hash = absl::hash_internal::Hash<T>;

// HashState
//
// A type erased version of the hash state concept, for use in user-defined
// `AbslHashValue` implementations that can't use templates (such as PImpl
// classes, virtual functions, etc.). The type erasure adds overhead so it
// should be avoided unless necessary.
//
// Note: This wrapper will only erase calls to:
//     combine_contiguous(H, const unsigned char*, size_t)
//
// All other calls will be handled internally and will not invoke overloads
// provided by the wrapped class.
//
// Users of this class should still define a template `AbslHashValue` function,
// but can use `absl::HashState::Create(&state)` to erase the type of the hash
// state and dispatch to their private hashing logic.
//
// This state can be used like any other hash state. In particular, you can call
// `HashState::combine()` and `HashState::combine_contiguous()` on it.
//
// Example:
//
//   class Interface {
//    public:
//     template <typename H>
//     friend H AbslHashValue(H state, const Interface& value) {
//       state = H::combine(std::move(state), std::type_index(typeid(*this)));
//       value.HashValue(absl::HashState::Create(&state));
//       return state;
//     }
//    private:
//     virtual void HashValue(absl::HashState state) const = 0;
//   };
//
//   class Impl : Interface {
//    private:
//     void HashValue(absl::HashState state) const override {
//       absl::HashState::combine(std::move(state), v1_, v2_);
//     }
//     int v1_;
//     std::string v2_;
//   };
class HashState : public hash_internal::HashStateBase<HashState> {
 public:
  // HashState::Create()
  //
  // Create a new `HashState` instance that wraps `state`. All calls to
  // `combine()` and `combine_contiguous()` on the new instance will be
  // redirected to the original `state` object. The `state` object must outlive
  // the `HashState` instance.
  template <typename T>
  static HashState Create(T* state) {
    HashState s;
    s.Init(state);
    return s;
  }

  HashState(const HashState&) = delete;
  HashState& operator=(const HashState&) = delete;
  HashState(HashState&&) = default;
  HashState& operator=(HashState&&) = default;

  // HashState::combine()
  //
  // Combines an arbitrary number of values into a hash state, returning the
  // updated state.
  using HashState::HashStateBase::combine;

  // HashState::combine_contiguous()
  //
  // Combines a contiguous array of `size` elements into a hash state, returning
  // the updated state.
  static HashState combine_contiguous(HashState hash_state,
                                      const unsigned char* first, size_t size) {
    hash_state.combine_contiguous_(hash_state.state_, first, size);
    return hash_state;
  }
  using HashState::HashStateBase::combine_contiguous;

 private:
  HashState() = default;

  template <typename T>
  static void CombineContiguousImpl(void* p, const unsigned char* first,
                                    size_t size) {
    T& state = *static_cast<T*>(p);
    state = T::combine_contiguous(std::move(state), first, size);
  }

  template <typename T>
  void Init(T* state) {
    state_ = state;
    combine_contiguous_ = &CombineContiguousImpl<T>;
  }

  // Do not erase an already erased state.
  void Init(HashState* state) {
    state_ = state->state_;
    combine_contiguous_ = state->combine_contiguous_;
  }

  void* state_;
  void (*combine_contiguous_)(void*, const unsigned char*, size_t);
```

```
};

}  // namespace absl
```

Notes on CRTP (Curiously Recurring Template Pattern)

CRTP is used in the absl::hash_internal::HashBase implementation therefore we will discuss it now.

CRTP General form

```
template <class T>
class Base
{
  // methods within Base can use template to access members of Derived
};
class Derived : public Base<Derived>
{
 // ...
};
```

Use cases for CRTP: Static polymorphism

Typically, the base class will take advantage of the fact that the member function bodies (definitions) are not instantiated until long after their declarations and will use members of the derived class within its own member functions via the use of cast e.g.:

```
template <class T>
struct Base
{
  void interface()
   {
     // ...
     static_cast<T*>(this)->implementation();
     // ...
   }

   static void static_func()
   {
      // ...
      T::static_sub_func();
     // ...
   }
};

struct Derived : Base<Derived>
{
   void implementation();
   static void static_sub_func();
};
```

In the above example, note in particular that the function Base<Derived>::interface(), though declared before the existence of the struct Derived is known by the compiler (i.e. before Derived is declared) is not actually instantiated by the compiler until it is actually called by some later code which occurs after the declaration of Derived (not shown in the above example) so that at the time the function "implementation" is instantiated, the declaration of Derived::implementation() is known. This technique achieves a similar effect to the use of virtual functions without the costs and flexibility of dynamic polymorphism. This particular use of the CRTP has been called "simulated dynamic binding".
To elaborate on the above example, consider a base class with *no virtual functions*. Whenever the base class calls another function it will always call its own base class functions. When we derive a class from this base class we inherit the member variables and member functions that were not overridden (no constructors and destructors). If the derived class calls an inherited function which then calls another member function, that function will never call any derived or overridden member functions in the derived class.
However, if member functions use CRTP for all member function calls, the overridden functions in the derived class will be selected at compile time without the costs in size or function call overhead (VTBL structures, and method lookups, multiple-inheritance VTBL machinery) at the disadvantage of not being able to make this choice at runtime.

CRTP Example: Object Counter

The main purpose of object counter is retrieving statistics of object creation and destruction for a given class. This can easily be solved via CRTP.

```
template <typename T>
struct counter
{
  static int objects_created;
  static int objects_alive;

  counter()
  {
   ++objects_created;
   ++objects_alive;
  }

  counter(const counter&)
  {
   ++objects_created;
   ++objects_alive;
  }
 protected:
  ~counter() // objects should never be removed through pointers of this type
  {
    --objects_alive;
  }
};
```

```
template <typename T> int counter<T>::objects_created( 0 );
template <typename T> int counter<T>::objects_alive( 0 );

class X : counter<X>
{
  // ...
}

class Y : counter<Y>
{
  // ...
}
```

Each time an object of class X is created , the constructor of counter<X> is called incrementing both the created and alive count. Each time an object of class X is destroyed the alive count is decremented. It is important to note that counter<X> and counter<Y> are two separate classes and this is why they will keep separate counts of X's and Y's. In this example of CRTP, this distinction of classes is the only use of the template parameter (T in counter<T>) and the reason why we cannot use a simple un-templated base class.

CRTP Example: Polymorphic chaining

Method chaining also known as the named parameter idiom is a common syntax for invoking multiple method calls in OOP languages. Each method returns an object , allowing the calls to be chained together in a single statement without requiring the variables to store intermediate results.
When the named parameter object pattern is applied to an object hierarchy things can go wrong.
Suppose we have such a base class:

```
class Printer
{
public:
  Printer(ostream& pstream) : m_stream(pstream) {}

  template <typename T>
  Printer& print(T&& t) { m_stream << t; return *this; }

  template <typename T>
  Printer& println(T&& t) { m_stream << t << endl; return *this; }
private:
  ostream& m_stream;
}
```

Prints can be easily chained:

```
Printer{myStream}.println("hello").println(500);
```

However, if we define the following derived class:

```
class CoutPrinter : public Printer
{
```

```
public:
  CoutPrinter() : Printer(cout) {}
   CoutPrinter& SetConsoleColor(Color c)
   {
     // ...
     return *this;
   }
}
```

we “loose” the concrete class as soon as we invoke a function of the base:

```
CoutPrinter().print(“Hello ”).SetConsoleColor(Color.red).println(“Printer!”); // compile error
```

CRTP can be used to avoid this problem and implement polymorphic chaining:

```
// Base class
template<typename ConcretePrinter>
class Printer
{
public:
  Printer(ostream& pstream) : m_stream(pstream) {}

  template <typename T>
  ConcretePrinter& print(T&& t)
  {
    m_stream << t;
    return static_cast<ConcretePrinter&>(*this);
  }

  template <typename T>
  ConcretePrinter& println(T&& t)
  {
    m_stream << t;
    return static_cast<ConcretePrinter&>(*this);
  }
private:
  ostream& m_stream;
};

// Derived class
class CoutPrinter : public Printer<CoutPrinter>
{
public:
  CoutPrinter() : Printer(cout) {}
  CoutPrinter& SetConsoleColor(Color c)
  {
    // ...
    return *this;
  }
};
```

Now the expected usage
CoutPrinter().print(“Hello “).SetConsoleColor(Color.red).println(“Printer!”);
does not cause compile error.

CRTP for Polymorphic Copy construction

When using polymorphism one sometimes needs to create copies of objects by the base class pointer. A commonly used idiom for this is adding a virtual clone function that is defined in every derived class. The CRTP can be used to avoid having to duplicate that function or other similar functions in every derived class.

```
// Base class has a pure virtual function for cloning
class AbstractShape {
public:
   virtual ~AbstractShape () = default;
   virtual std::unique_ptr<AbstractShape> clone() const = 0;
} ;

// This CRTP class implements clone() for Derived
template <typename Derived>
class Shape : public AbstractShape {
public:
  std::unique_ptr<AbstractShape> clone() const override {
     return std::make_unique<Derived>(static_cast<Derived const&>(*this));
  }

protected:
  // we are clear Shape class needs to be inherited
  Shape() = default;
  Shape(const Shape&) = default;
} ;

class Square : public Shape<Square>{};
class Circle : public Shape<Circle>{};
```

Drawbacks of using CRTP

One issue with static polymorphism is that without using a general base class like AbstractShape from the above example, derived classes cannot be stored homogeneously – that is putting different types derived from the same base class in the same container.

Discussion on HashStateBase, PiecewiseCombiner and AbslHashValue

HashStateBase is declared and defined in absl/hash/internal/hash.h. It utilizes CRTP to implement static polymorphism with HashStateBase as a base class allowing the base class to access members of the derived classes. The derived class is passed as a template parameter to HashBaseState and denoted by H. The derived class H has to provide declaration and definition of the following static member function:
 H H::combine_contiguous(H state, const unsigned char*, size_t).
HashStateBase will provide implementation for hash state object in terms of this method.

A specialized class template **is_uniquely_represented** is defined and used to determine if the template parameter container has special properties which will allow further optimizations leading to performance benefits. To be uniquely represented, a type must not have multiple ways of representing the same value; for example, float and double are not uniquely represented, because they have distinct representations for +0 and -0. Furthermore, the type's byte representation must consist solely of user-controlled data, with no padding bits and no compiler-controlled data such as vptrs or sanitizer metadata. This is usually very difficult to guarantee, because in most cases the compiler can insert data and padding bits at its own discretion. The template **is_uniquely_represented** is used in the definition and specialization of the function template **hash_range_or_bytes** which uses either **H::combine_contiguous** or **H::combine** depending on the properties of the parameter type T:

```
// hash_range_or_bytes()
//
// Mixes all values in the range [data, data+size) into the hash state.
// This overload accepts only uniquely-represented types, and hashes them by
// hashing the entire range of bytes.
template <typename H, typename T>
typename std::enable_if<is_uniquely_represented<T>::value, H>::type
hash_range_or_bytes(H hash_state, const T* data, size_t size) {
  const auto* bytes = reinterpret_cast<const unsigned char*>(data);
  return H::combine_contiguous(std::move(hash_state), bytes, sizeof(T) * size);
}

// hash_range_or_bytes()
template <typename H, typename T>
typename std::enable_if<!is_uniquely_represented<T>::value, H>::type
hash_range_or_bytes(H hash_state, const T* data, size_t size) {
  for (const auto end = data + size; data < end; ++data) {
    hash_state = H::combine(std::move(hash_state), *data);
  }
  return hash_state;
}
```

PiecewiseCombiner is an internal-only helper class for hashing a piecewise buffer of `char` or `unsigned char` as though it were contiguous. This class provides two methods:

- H add_buffer(state, data, size)
- H finalize(state)

add_buffer can be called zero or more times, followed by a single call to **finalize**. This will produce the same hash expansion as concatenating each buffer piece into a single contiguous buffer, and passing this to **H::combine_contiguous**.

Example usage:

```
PiecewiseCombiner combiner;
```

```
for (const auto& piece : pieces) {
  state = combiner.add_buffer(std::move(state), piece.data, piece.size);
}
return combiner.finalize(std::move(state));
```

The contents of absl/hash/internal/hash.h is shown below

```
namespace hash_internal {

class PiecewiseCombiner;

// Internal detail: Large buffers are hashed in smaller chunks.  This function
// returns the size of these chunks.
constexpr size_t PiecewiseChunkSize() { return 1024; }

// HashStateBase
//
// A hash state object represents an intermediate state in the computation
// of an unspecified hash algorithm. `HashStateBase` provides a CRTP style
// base class for hash state implementations. Developers adding type support
// for `absl::Hash` should not rely on any parts of the state object other than
// the following member functions:
//
//   * HashStateBase::combine()
//   * HashStateBase::combine_contiguous()
//
// A derived hash state class of type `H` must provide a static member function
// with a signature similar to the following:
//
//    `static H combine_contiguous(H state, const unsigned char*, size_t)`.
//
// `HashStateBase` will provide a complete implementation for a hash state
// object in terms of this method.
//
// Example:
//
//   // Use CRTP to define your derived class.
//   struct MyHashState : HashStateBase<MyHashState> {
//       static H combine_contiguous(H state, const unsigned char*, size_t);
//       using MyHashState::HashStateBase::combine;
//       using MyHashState::HashStateBase::combine_contiguous;
//   };
template <typename H>
class HashStateBase {
 public:
  // HashStateBase::combine()
  //
  // Combines an arbitrary number of values into a hash state, returning the
  // updated state.
  //
  // Each of the value types `T` must be separately hashable by the Abseil
  // hashing framework.
  //
  // NOTE:
  //
  //   state = H::combine(std::move(state), value1, value2, value3);
  //
  // is guaranteed to produce the same hash expansion as:
  //
  //   state = H::combine(std::move(state), value1);
  //   state = H::combine(std::move(state), value2);
  //   state = H::combine(std::move(state), value3);
  template <typename T, typename... Ts>
  static H combine(H state, const T& value, const Ts&... values);
  static H combine(H state) { return state; }

  // HashStateBase::combine_contiguous()
  //
  // Combines a contiguous array of `size` elements into a hash state, returning
  // the updated state.
  //
  // NOTE:
  //
  //   state = H::combine_contiguous(std::move(state), data, size);
  //
  // is NOT guaranteed to produce the same hash expansion as a for-loop (it may
  // perform internal optimizations).  If you need this guarantee, use the
  // for-loop instead.
  template <typename T>
  static H combine_contiguous(H state, const T* data, size_t size);

 private:
  friend class PiecewiseCombiner;
};

// is_uniquely_represented
//
// `is_uniquely_represented<T>` is a trait class that indicates whether `T`
// is uniquely represented.
```

```
//
// A type is "uniquely represented" if two equal values of that type are
// guaranteed to have the same bytes in their underlying storage. In other
// words, if `a == b`, then `memcmp(&a, &b, sizeof(T))` is guaranteed to be
// zero. This property cannot be detected automatically, so this trait is false
// by default, but can be specialized by types that wish to assert that they are
// uniquely represented. This makes them eligible for certain optimizations.
//
// If you have any doubt whatsoever, do not specialize this template.
// The default is completely safe, and merely disables some optimizations
// that will not matter for most types. Specializing this template,
// on the other hand, can be very hazardous.
//
// To be uniquely represented, a type must not have multiple ways of
// representing the same value; for example, float and double are not
// uniquely represented, because they have distinct representations for
// +0 and -0. Furthermore, the type's byte representation must consist
// solely of user-controlled data, with no padding bits and no compiler-
// controlled data such as vptrs or sanitizer metadata. This is usually
// very difficult to guarantee, because in most cases the compiler can
// insert data and padding bits at its own discretion.
//
// If you specialize this template for a type `T`, you must do so in the file
// that defines that type (or in this file). If you define that specialization
// anywhere else, `is_uniquely_represented<T>` could have different meanings
// in different places.
//
// The Enable parameter is meaningless; it is provided as a convenience,
// to support certain SFINAE techniques when defining specializations.
template <typename T, typename Enable = void>
struct is_uniquely_represented : std::false_type {};

// is_uniquely_represented<unsigned char>
//
// unsigned char is a synonym for "byte", so it is guaranteed to be
// uniquely represented.
template <>
struct is_uniquely_represented<unsigned char> : std::true_type {};

// is_uniquely_represented for non-standard integral types
//
// Integral types other than bool should be uniquely represented on any
// platform that this will plausibly be ported to.
template <typename Integral>
```

```
struct is_uniquely_represented<
    Integral, typename std::enable_if<std::is_integral<Integral>::value>::type>
    : std::true_type {};

// is_uniquely_represented<bool>
//
//
template <>
struct is_uniquely_represented<bool> : std::false_type {};

// hash_bytes()
//
// Convenience function that combines `hash_state` with the byte representation
// of `value`.
template <typename H, typename T>
H hash_bytes(H hash_state, const T& value) {
  const unsigned char* start = reinterpret_cast<const unsigned char*>(&value);
  return H::combine_contiguous(std::move(hash_state), start, sizeof(value));
}

// PiecewiseCombiner
//
// PiecewiseCombiner is an internal-only helper class for hashing a piecewise
// buffer of `char` or `unsigned char` as though it were contiguous.  This class
// provides two methods:
//
//   H add_buffer(state, data, size)
//   H finalize(state)
//
// `add_buffer` can be called zero or more times, followed by a single call to
// `finalize`.  This will produce the same hash expansion as concatenating each
// buffer piece into a single contiguous buffer, and passing this to
// `H::combine_contiguous`.
//
//  Example usage:
//    PiecewiseCombiner combiner;
//    for (const auto& piece : pieces) {
//      state = combiner.add_buffer(std::move(state), piece.data, piece.size);
//    }
//    return combiner.finalize(std::move(state));
class PiecewiseCombiner {
 public:
  PiecewiseCombiner() : position_(0) {}
  PiecewiseCombiner(const PiecewiseCombiner&) = delete;
  PiecewiseCombiner& operator=(const PiecewiseCombiner&) = delete;

  // PiecewiseCombiner::add_buffer()
  //
  // Appends the given range of bytes to the sequence to be hashed, which may
  // modify the provided hash state.
  template <typename H>
  H add_buffer(H state, const unsigned char* data, size_t size);
  template <typename H>
  H add_buffer(H state, const char* data, size_t size) {
    return add_buffer(std::move(state),
                      reinterpret_cast<const unsigned char*>(data), size);
  }

  // PiecewiseCombiner::finalize()
  //
  // Finishes combining the hash sequence, which may may modify the provided
  // hash state.
  //
  // Once finalize() is called, add_buffer() may no longer be called. The
  // resulting hash state will be the same as if the pieces passed to
  // add_buffer() were concatenated into a single flat buffer, and then provided
  // to H::combine_contiguous().
  template <typename H>
  H finalize(H state);

 private:
  unsigned char buf_[PiecewiseChunkSize()];
  size_t position_;
};

// -----------------------------------------------------------------------------
// AbslHashValue for Basic Types
// -----------------------------------------------------------------------------

// Note: Default `AbslHashValue` implementations live in `hash_internal`. This
// allows us to block lexical scope lookup when doing an unqualified call to
// `AbslHashValue` below. User-defined implementations of `AbslHashValue` can
// only be found via ADL.

// AbslHashValue() for hashing bool values
//
// We use SFINAE to ensure that this overload only accepts bool, not types that
// are convertible to bool.
```

```
template <typename H, typename B>
typename std::enable_if<std::is_same<B, bool>::value, H>::type AbslHashValue(
    H hash_state, B value) {
  return H::combine(std::move(hash_state),
                    static_cast<unsigned char>(value ? 1 : 0));
}

// AbslHashValue() for hashing enum values
template <typename H, typename Enum>
typename std::enable_if<std::is_enum<Enum>::value, H>::type AbslHashValue(
    H hash_state, Enum e) {
  // In practice, we could almost certainly just invoke hash_bytes directly,
  // but it's possible that a sanitizer might one day want to
  // store data in the unused bits of an enum. To avoid that risk, we
  // convert to the underlying type before hashing. Hopefully this will get
  // optimized away; if not, we can reopen discussion with c-toolchain-team.
  return H::combine(std::move(hash_state),
                    static_cast<typename std::underlying_type<Enum>::type>(e));
}
// AbslHashValue() for hashing floating-point values
template <typename H, typename Float>
typename std::enable_if<std::is_same<Float, float>::value ||
                            std::is_same<Float, double>::value,
                        H>::type
AbslHashValue(H hash_state, Float value) {
  return hash_internal::hash_bytes(std::move(hash_state),
                                   value == 0 ? 0 : value);
}

// Long double has the property that it might have extra unused bytes in it.
// For example, in x86 sizeof(long double)==16 but it only really uses 80-bits
// of it. This means we can't use hash_bytes on a long double and have to
// convert it to something else first.
template <typename H, typename LongDouble>
typename std::enable_if<std::is_same<LongDouble, long double>::value, H>::type
AbslHashValue(H hash_state, LongDouble value) {
  const int category = std::fpclassify(value);
  switch (category) {
    case FP_INFINITE:
      // Add the sign bit to differentiate between +Inf and -Inf
      hash_state = H::combine(std::move(hash_state), std::signbit(value));
      break;

    case FP_NAN:
```

```
    case FP_ZERO:
    default:
      // Category is enough for these.
      break;

    case FP_NORMAL:
    case FP_SUBNORMAL:
      // We can't convert `value` directly to double because this would have
      // undefined behavior if the value is out of range.
      // std::frexp gives us a value in the range (-1, -.5] or [.5, 1) that is
      // guaranteed to be in range for `double`. The truncation is
      // implementation defined, but that works as long as it is deterministic.
      int exp;
      auto mantissa = static_cast<double>(std::frexp(value, &exp));
      hash_state = H::combine(std::move(hash_state), mantissa, exp);
  }

  return H::combine(std::move(hash_state), category);
}

// AbslHashValue() for hashing pointers
template <typename H, typename T>
H AbslHashValue(H hash_state, T* ptr) {
  auto v = reinterpret_cast<uintptr_t>(ptr);
  // Due to alignment, pointers tend to have low bits as zero, and the next few
  // bits follow a pattern since they are also multiples of some base value.
  // Mixing the pointer twice helps prevent stuck low bits for certain alignment
  // values.
  return H::combine(std::move(hash_state), v, v);
}

// AbslHashValue() for hashing nullptr_t
template <typename H>
H AbslHashValue(H hash_state, std::nullptr_t) {
  return H::combine(std::move(hash_state), static_cast<void*>(nullptr));
}

// -----------------------------------------------------------------------------
// AbslHashValue for Composite Types
// -----------------------------------------------------------------------------

// is_hashable()
//
// Trait class which returns true if T is hashable by the absl::Hash framework.
// Used for the AbslHashValue implementations for composite types below.
template <typename T>
struct is_hashable;

// AbslHashValue() for hashing pairs
template <typename H, typename T1, typename T2>
typename std::enable_if<is_hashable<T1>::value && is_hashable<T2>::value,
                        H>::type
AbslHashValue(H hash_state, const std::pair<T1, T2>& p) {
  return H::combine(std::move(hash_state), p.first, p.second);
}

// hash_tuple()
//
// Helper function for hashing a tuple. The third argument should
// be an index_sequence running from 0 to tuple_size<Tuple> - 1.
template <typename H, typename Tuple, size_t... Is>
H hash_tuple(H hash_state, const Tuple& t, absl::index_sequence<Is...>) {
  return H::combine(std::move(hash_state), std::get<Is>(t)...);
}

// AbslHashValue for hashing tuples
template <typename H, typename... Ts>
#if defined(_MSC_VER)
// This SFINAE gets MSVC confused under some conditions. Let's just disable it
// for now.
H
#else  // _MSC_VER
typename std::enable_if<absl::conjunction<is_hashable<Ts>...>::value, H>::type
#endif  // _MSC_VER
AbslHashValue(H hash_state, const std::tuple<Ts...>& t) {
  return hash_internal::hash_tuple(std::move(hash_state), t,
                                   absl::make_index_sequence<sizeof...(Ts)>());
}

// -----------------------------------------------------------------------------
// AbslHashValue for Pointers
// -----------------------------------------------------------------------------

// AbslHashValue for hashing unique_ptr
template <typename H, typename T, typename D>
H AbslHashValue(H hash_state, const std::unique_ptr<T, D>& ptr) {
  return H::combine(std::move(hash_state), ptr.get());
}
```

```
// AbslHashValue for hashing shared_ptr
template <typename H, typename T>
H AbslHashValue(H hash_state, const std::shared_ptr<T>& ptr) {
  return H::combine(std::move(hash_state), ptr.get());
}

// -----------------------------------------------------------------------------
// AbslHashValue for String-Like Types
// -----------------------------------------------------------------------------

// AbslHashValue for hashing strings
//
// All the string-like types supported here provide the same hash expansion for
// the same character sequence. These types are:
//
//  - `std::string` (and std::basic_string<char, std::char_traits<char>, A> for
//      any allocator A)
//  - `absl::string_view` and `std::string_view`
//
// For simplicity, we currently support only `char` strings. This support may
// be broadened, if necessary, but with some caution - this overload would
// misbehave in cases where the traits' `eq()` member isn't equivalent to `==`
// on the underlying character type.
template <typename H>
H AbslHashValue(H hash_state, absl::string_view str) {
  return H::combine(
      H::combine_contiguous(std::move(hash_state), str.data(), str.size()),
      str.size());
}

// Support std::wstring, std::u16string and std::u32string.
template <typename Char, typename Alloc, typename H,
          typename = absl::enable_if_t<std::is_same<Char, wchar_t>::value ||
                                       std::is_same<Char, char16_t>::value ||
                                       std::is_same<Char, char32_t>::value>>
H AbslHashValue(
    H hash_state,
    const std::basic_string<Char, std::char_traits<Char>, Alloc>& str) {
  return H::combine(
      H::combine_contiguous(std::move(hash_state), str.data(), str.size()),
      str.size());
}
```

```
// -----------------------------------------------------------------------------
// AbslHashValue for Sequence Containers
// -----------------------------------------------------------------------------

// AbslHashValue for hashing std::array
template <typename H, typename T, size_t N>
typename std::enable_if<is_hashable<T>::value, H>::type AbslHashValue(
    H hash_state, const std::array<T, N>& array) {
  return H::combine_contiguous(std::move(hash_state), array.data(),
                               array.size());
}

// AbslHashValue for hashing std::deque
template <typename H, typename T, typename Allocator>
typename std::enable_if<is_hashable<T>::value, H>::type AbslHashValue(
    H hash_state, const std::deque<T, Allocator>& deque) {
  // TODO(gromer): investigate a more efficient implementation taking
  // advantage of the chunk structure.
  for (const auto& t : deque) {
    hash_state = H::combine(std::move(hash_state), t);
  }
  return H::combine(std::move(hash_state), deque.size());
}

// AbslHashValue for hashing std::forward_list
template <typename H, typename T, typename Allocator>
typename std::enable_if<is_hashable<T>::value, H>::type AbslHashValue(
    H hash_state, const std::forward_list<T, Allocator>& list) {
  size_t size = 0;
  for (const T& t : list) {
    hash_state = H::combine(std::move(hash_state), t);
    ++size;
  }
  return H::combine(std::move(hash_state), size);
}

// AbslHashValue for hashing std::list
template <typename H, typename T, typename Allocator>
typename std::enable_if<is_hashable<T>::value, H>::type AbslHashValue(
    H hash_state, const std::list<T, Allocator>& list) {
  for (const auto& t : list) {
    hash_state = H::combine(std::move(hash_state), t);
  }
  return H::combine(std::move(hash_state), list.size());
}

// AbslHashValue for hashing std::vector
//
// Do not use this for vector<bool>. It does not have a .data(), and a fallback
// for std::hash<> is most likely faster.
template <typename H, typename T, typename Allocator>
typename std::enable_if<is_hashable<T>::value && !std::is_same<T, bool>::value,
                        H>::type
AbslHashValue(H hash_state, const std::vector<T, Allocator>& vector) {
  return H::combine(H::combine_contiguous(std::move(hash_state), vector.data(),
                                          vector.size()),
                    vector.size());
}

// -----------------------------------------------------------------------------
// AbslHashValue for Ordered Associative Containers
// -----------------------------------------------------------------------------

// AbslHashValue for hashing std::map
template <typename H, typename Key, typename T, typename Compare,
          typename Allocator>
typename std::enable_if<is_hashable<Key>::value && is_hashable<T>::value,
                        H>::type
AbslHashValue(H hash_state, const std::map<Key, T, Compare, Allocator>& map) {
  for (const auto& t : map) {
    hash_state = H::combine(std::move(hash_state), t);
  }
  return H::combine(std::move(hash_state), map.size());
}

// AbslHashValue for hashing std::multimap
template <typename H, typename Key, typename T, typename Compare,
          typename Allocator>
typename std::enable_if<is_hashable<Key>::value && is_hashable<T>::value,
                        H>::type
AbslHashValue(H hash_state,
              const std::multimap<Key, T, Compare, Allocator>& map) {
  for (const auto& t : map) {
    hash_state = H::combine(std::move(hash_state), t);
  }
  return H::combine(std::move(hash_state), map.size());
}
```

```
// AbslHashValue for hashing std::set
template <typename H, typename Key, typename Compare, typename Allocator>
typename std::enable_if<is_hashable<Key>::value, H>::type AbslHashValue(
    H hash_state, const std::set<Key, Compare, Allocator>& set) {
  for (const auto& t : set) {
    hash_state = H::combine(std::move(hash_state), t);
  }
  return H::combine(std::move(hash_state), set.size());
}

// AbslHashValue for hashing std::multiset
template <typename H, typename Key, typename Compare, typename Allocator>
typename std::enable_if<is_hashable<Key>::value, H>::type AbslHashValue(
    H hash_state, const std::multiset<Key, Compare, Allocator>& set) {
  for (const auto& t : set) {
    hash_state = H::combine(std::move(hash_state), t);
  }
  return H::combine(std::move(hash_state), set.size());
}

// -----------------------------------------------------------------------------
// AbslHashValue for Wrapper Types
// -----------------------------------------------------------------------------

// AbslHashValue for hashing absl::optional
template <typename H, typename T>
typename std::enable_if<is_hashable<T>::value, H>::type AbslHashValue(
    H hash_state, const absl::optional<T>& opt) {
  if (opt) hash_state = H::combine(std::move(hash_state), *opt);
  return H::combine(std::move(hash_state), opt.has_value());
}

// VariantVisitor
template <typename H>
struct VariantVisitor {
  H&& hash_state;
  template <typename T>
  H operator()(const T& t) const {
    return H::combine(std::move(hash_state), t);
  }
};

// AbslHashValue for hashing absl::variant
template <typename H, typename... T>
```

```
typename std::enable_if<conjunction<is_hashable<T>...>::value, H>::type
AbslHashValue(H hash_state, const absl::variant<T...>& v) {
  if (!v.valueless_by_exception()) {
    hash_state = absl::visit(VariantVisitor<H>{std::move(hash_state)}, v);
  }
  return H::combine(std::move(hash_state), v.index());
}

// -----------------------------------------------------------------------------
// AbslHashValue for Other Types
// -----------------------------------------------------------------------------

// AbslHashValue for hashing std::bitset is not defined, for the same reason as
// for vector<bool> (see std::vector above): It does not expose the raw bytes,
// and a fallback to std::hash<> is most likely faster.

// -----------------------------------------------------------------------------

// hash_range_or_bytes()
//
// Mixes all values in the range [data, data+size) into the hash state.
// This overload accepts only uniquely-represented types, and hashes them by
// hashing the entire range of bytes.
template <typename H, typename T>
typename std::enable_if<is_uniquely_represented<T>::value, H>::type
hash_range_or_bytes(H hash_state, const T* data, size_t size) {
  const auto* bytes = reinterpret_cast<const unsigned char*>(data);
  return H::combine_contiguous(std::move(hash_state), bytes, sizeof(T) * size);
}

// hash_range_or_bytes()
template <typename H, typename T>
typename std::enable_if<!is_uniquely_represented<T>::value, H>::type
hash_range_or_bytes(H hash_state, const T* data, size_t size) {
  for (const auto end = data + size; data < end; ++data) {
    hash_state = H::combine(std::move(hash_state), *data);
  }
  return hash_state;
}

#if defined(ABSL_INTERNAL_LEGACY_HASH_NAMESPACE) && \
    ABSL_META_INTERNAL_STD_HASH_SFINAE_FRIENDLY_
#define ABSL_HASH_INTERNAL_SUPPORT_LEGACY_HASH_ 1
#else
```

```
#define ABSL_HASH_INTERNAL_SUPPORT_LEGACY_HASH_ 0
#endif

// HashSelect
//
// Type trait to select the appropriate hash implementation to use.
// HashSelect::type<T> will give the proper hash implementation, to be invoked
// as:
//   HashSelect::type<T>::Invoke(state, value)
// Also, HashSelect::type<T>::value is a boolean equal to `true` if there is a
// valid `Invoke` function. Types that are not hashable will have a ::value of
// `false`.
struct HashSelect {
 private:
  struct State : HashStateBase<State> {
    static State combine_contiguous(State hash_state, const unsigned char*,
                                    size_t);
    using State::HashStateBase::combine_contiguous;
  };

  struct UniquelyRepresentedProbe {
    template <typename H, typename T>
    static auto Invoke(H state, const T& value)
        -> absl::enable_if_t<is_uniquely_represented<T>::value, H> {
      return hash_internal::hash_bytes(std::move(state), value);
    }
  };

  struct HashValueProbe {
    template <typename H, typename T>
    static auto Invoke(H state, const T& value) -> absl::enable_if_t<
        std::is_same<H,
                     decltype(AbslHashValue(std::move(state), value))>::value,
        H> {
      return AbslHashValue(std::move(state), value);
    }
  };

  struct LegacyHashProbe {
#if ABSL_HASH_INTERNAL_SUPPORT_LEGACY_HASH_
    template <typename H, typename T>
    static auto Invoke(H state, const T& value) -> absl::enable_if_t<
        std::is_convertible<
            decltype(ABSL_INTERNAL_LEGACY_HASH_NAMESPACE::hash<T>()(value)),
            size_t>::value,
        H> {
      return hash_internal::hash_bytes(
          std::move(state),
          ABSL_INTERNAL_LEGACY_HASH_NAMESPACE::hash<T>{}(value));
    }
#endif  // ABSL_HASH_INTERNAL_SUPPORT_LEGACY_HASH_
  };

  struct StdHashProbe {
    template <typename H, typename T>
    static auto Invoke(H state, const T& value)
        -> absl::enable_if_t<type_traits_internal::IsHashable<T>::value, H> {
      return hash_internal::hash_bytes(std::move(state), std::hash<T>{}(value));
    }
  };

  template <typename Hash, typename T>
  struct Probe : Hash {
   private:
    template <typename H, typename = decltype(H::Invoke(
                              std::declval<State>(), std::declval<const T&>()))>
    static std::true_type Test(int);
    template <typename U>
    static std::false_type Test(char);

   public:
    static constexpr bool value = decltype(Test<Hash>(0))::value;
  };

 public:
  // Probe each implementation in order.
  // disjunction provides short circuiting wrt instantiation.
  template <typename T>
  using Apply = absl::disjunction<         //
      Probe<UniquelyRepresentedProbe, T>,  //
      Probe<HashValueProbe, T>,            //
      Probe<LegacyHashProbe, T>,           //
      Probe<StdHashProbe, T>,              //
      std::false_type>;
};

template <typename T>
struct is_hashable
```

```
    : std::integral_constant<bool, HashSelect::template Apply<T>::value> {};

// CityHashState
class ABSL_DLL CityHashState
    : public HashStateBase<CityHashState> {
  // absl::uint128 is not an alias or a thin wrapper around the intrinsic.
  // We use the intrinsic when available to improve performance.
#ifdef ABSL_HAVE_INTRINSIC_INT128
  using uint128 = __uint128_t;
#else   // ABSL_HAVE_INTRINSIC_INT128
  using uint128 = absl::uint128;
#endif  // ABSL_HAVE_INTRINSIC_INT128

  static constexpr uint64_t kMul =
      sizeof(size_t) == 4 ? uint64_t{0xcc9e2d51}
                          : uint64_t{0x9ddfea08eb382d69};

  template <typename T>
  using IntegralFastPath =
      conjunction<std::is_integral<T>, is_uniquely_represented<T>>;

 public:
  // Move only
  CityHashState(CityHashState&&) = default;
  CityHashState& operator=(CityHashState&&) = default;

  // CityHashState::combine_contiguous()
  //
  // Fundamental base case for hash recursion: mixes the given range of bytes
  // into the hash state.
  static CityHashState combine_contiguous(CityHashState hash_state,
                                          const unsigned char* first,
                                          size_t size) {
    return CityHashState(
        CombineContiguousImpl(hash_state.state_, first, size,
                              std::integral_constant<int, sizeof(size_t)>{}));
  }
  using CityHashState::HashStateBase::combine_contiguous;

  // CityHashState::hash()
  //
  // For performance reasons in non-opt mode, we specialize this for
  // integral types.
  // Otherwise we would be instantiating and calling dozens of functions for
  // something that is just one multiplication and a couple xor's.
  // The result should be the same as running the whole algorithm, but faster.
  template <typename T, absl::enable_if_t<IntegralFastPath<T>::value, int> = 0>
  static size_t hash(T value) {
    return static_cast<size_t>(Mix(Seed(), static_cast<uint64_t>(value)));
  }

  // Overload of CityHashState::hash()
  template <typename T, absl::enable_if_t<!IntegralFastPath<T>::value, int> = 0>
  static size_t hash(const T& value) {
    return static_cast<size_t>(combine(CityHashState{}, value).state_);
  }

 private:
  // Invoked only once for a given argument; that plus the fact that this is
  // move-only ensures that there is only one non-moved-from object.
  CityHashState() : state_(Seed()) {}

  // Workaround for MSVC bug.
  // We make the type copyable to fix the calling convention, even though we
  // never actually copy it. Keep it private to not affect the public API of the
  // type.
  CityHashState(const CityHashState&) = default;

  explicit CityHashState(uint64_t state) : state_(state) {}

  // Implementation of the base case for combine_contiguous where we actually
  // mix the bytes into the state.
  // Dispatch to different implementations of the combine_contiguous depending
  // on the value of `sizeof(size_t)`.
  static uint64_t CombineContiguousImpl(uint64_t state,
                                        const unsigned char* first, size_t len,
                                        std::integral_constant<int, 4>
                                        /* sizeof_size_t */);
  static uint64_t CombineContiguousImpl(uint64_t state,
                                        const unsigned char* first, size_t len,
                                        std::integral_constant<int, 8>
                                        /* sizeof_size_t*/);

  // Slow dispatch path for calls to CombineContiguousImpl with a size argument
  // larger than PiecewiseChunkSize().  Has the same effect as calling
  // CombineContiguousImpl() repeatedly with the chunk stride size.
  static uint64_t CombineLargeContiguousImpl32(uint64_t state,
                                               const unsigned char* first,
                                               size_t len);
  static uint64_t CombineLargeContiguousImpl64(uint64_t state,
                                               const unsigned char* first,
                                               size_t len);

  // Reads 9 to 16 bytes from p.
  // The first 8 bytes are in .first, the rest (zero padded) bytes are in
  // .second.
  static std::pair<uint64_t, uint64_t> Read9To16(const unsigned char* p,
                                            size_t len) {
    uint64_t high = little_endian::Load64(p + len - 8);
    return {little_endian::Load64(p), high >> (128 - len * 8)};
  }

  // Reads 4 to 8 bytes from p. Zero pads to fill uint64_t.
  static uint64_t Read4To8(const unsigned char* p, size_t len) {
    return (static_cast<uint64_t>(little_endian::Load32(p + len - 4))
            << (len - 4) * 8) |
           little_endian::Load32(p);
  }

  // Reads 1 to 3 bytes from p. Zero pads to fill uint32_t.
  static uint32_t Read1To3(const unsigned char* p, size_t len) {
    return static_cast<uint32_t>((p[0]) |                         //
                                 (p[len / 2] << (len / 2 * 8)) |  //
                                 (p[len - 1] << ((len - 1) * 8)));
  }

  ABSL_ATTRIBUTE_ALWAYS_INLINE static uint64_t Mix(uint64_t state, uint64_t v) {
    using MultType =
        absl::conditional_t<sizeof(size_t) == 4, uint64_t, uint128>;
    // We do the addition in 64-bit space to make sure the 128-bit
    // multiplication is fast. If we were to do it as MultType the compiler has
    // to assume that the high word is non-zero and needs to perform 2
    // multiplications instead of one.
    MultType m = state + v;
    m *= kMul;
    return static_cast<uint64_t>(m ^ (m >> (sizeof(m) * 8 / 2)));
  }

  // Seed()
  //
  // A non-deterministic seed.
  //
```

```
  // The current purpose of this seed is to generate non-deterministic results
  // and prevent having users depend on the particular hash values.
  // It is not meant as a security feature right now, but it leaves the door
  // open to upgrade it to a true per-process random seed. A true random seed
  // costs more and we don't need to pay for that right now.
  //
  // On platforms with ASLR, we take advantage of it to make a per-process
  // random value.
  // See https://en.wikipedia.org/wiki/Address_space_layout_randomization
  //
  // On other platforms this is still going to be non-deterministic but most
  // probably per-build and not per-process.
  ABSL_ATTRIBUTE_ALWAYS_INLINE static uint64_t Seed() {
    return static_cast<uint64_t>(reinterpret_cast<uintptr_t>(kSeed));
  }
  static const void* const kSeed;

  uint64_t state_;
};

// CityHashState::CombineContiguousImpl()
inline uint64_t CityHashState::CombineContiguousImpl(
    uint64_t state, const unsigned char* first, size_t len,
    std::integral_constant<int, 4> /* sizeof_size_t */) {
  // For large values we use CityHash, for small ones we just use a
  // multiplicative hash.
  uint64_t v;
  if (len > 8) {
    if (ABSL_PREDICT_FALSE(len > PiecewiseChunkSize())) {
      return CombineLargeContiguousImpl32(state, first, len);
    }
    v = absl::hash_internal::CityHash32(reinterpret_cast<const char*>(first), len);
  } else if (len >= 4) {
    v = Read4To8(first, len);
  } else if (len > 0) {
    v = Read1To3(first, len);
  } else {
    // Empty ranges have no effect.
    return state;
  }
  return Mix(state, v);
}

// Overload of CityHashState::CombineContiguousImpl()
```

```
inline uint64_t CityHashState::CombineContiguousImpl(
    uint64_t state, const unsigned char* first, size_t len,
    std::integral_constant<int, 8> /* sizeof_size_t */) {
  // For large values we use CityHash, for small ones we just use a
  // multiplicative hash.
  uint64_t v;
  if (len > 16) {
    if (ABSL_PREDICT_FALSE(len > PiecewiseChunkSize())) {
      return CombineLargeContiguousImpl64(state, first, len);
    }
    v = absl::hash_internal::CityHash64(reinterpret_cast<const char*>(first), len);
  } else if (len > 8) {
    auto p = Read9To16(first, len);
    state = Mix(state, p.first);
    v = p.second;
  } else if (len >= 4) {
    v = Read4To8(first, len);
  } else if (len > 0) {
    v = Read1To3(first, len);
  } else {
    // Empty ranges have no effect.
    return state;
  }
  return Mix(state, v);
}

struct AggregateBarrier {};

// HashImpl

// Add a private base class to make sure this type is not an aggregate.
// Aggregates can be aggregate initialized even if the default constructor is
// deleted.
struct PoisonedHash : private AggregateBarrier {
  PoisonedHash() = delete;
  PoisonedHash(const PoisonedHash&) = delete;
  PoisonedHash& operator=(const PoisonedHash&) = delete;
};

template <typename T>
struct HashImpl {
  size_t operator()(const T& value) const { return CityHashState::hash(value); }
};
```

```
template <typename T>
struct Hash
    : absl::conditional_t<is_hashable<T>::value, HashImpl<T>, PoisonedHash> {};

template <typename H>
template <typename T, typename... Ts>
H HashStateBase<H>::combine(H state, const T& value, const Ts&... values) {
  return H::combine(hash_internal::HashSelect::template Apply<T>::Invoke(
                        std::move(state), value),
                    values...);
}

// HashStateBase::combine_contiguous()
template <typename H>
template <typename T>
H HashStateBase<H>::combine_contiguous(H state, const T* data, size_t size) {
  return hash_internal::hash_range_or_bytes(std::move(state), data, size);
}

// HashStateBase::PiecewiseCombiner::add_buffer()
template <typename H>
H PiecewiseCombiner::add_buffer(H state, const unsigned char* data,
                                size_t size) {
  if (position_ + size < PiecewiseChunkSize()) {
    // This partial chunk does not fill our existing buffer
    memcpy(buf_ + position_, data, size);
    position_ += size;
    return state;
  }

  // Complete the buffer and hash it
  const size_t bytes_needed = PiecewiseChunkSize() - position_;
  memcpy(buf_ + position_, data, bytes_needed);
  state = H::combine_contiguous(std::move(state), buf_, PiecewiseChunkSize());
  data += bytes_needed;
  size -= bytes_needed;

  // Hash whatever chunks we can without copying
  while (size >= PiecewiseChunkSize()) {
    state = H::combine_contiguous(std::move(state), data, PiecewiseChunkSize());
    data += PiecewiseChunkSize();
    size -= PiecewiseChunkSize();
  }
  // Fill the buffer with the remainder
  memcpy(buf_, data, size);
  position_ = size;
  return state;
}

// HashStateBase::PiecewiseCombiner::finalize()
template <typename H>
H PiecewiseCombiner::finalize(H state) {
  // Hash the remainder left in the buffer, which may be empty
  return H::combine_contiguous(std::move(state), buf_, position_);
}

}  // namespace hash_internal
```

Discussion on CombineContiguous implementation

CombineContiguous implementation is defined in absl/hash/internal/hash.cc.
The contents of absl/hash/internal/hash.cc is shown below

```
#include "absl/hash/internal/hash.h"

namespace absl {
ABSL_NAMESPACE_BEGIN
namespace hash_internal {

uint64_t CityHashState::CombineLargeContiguousImpl32(uint64_t state,
                                                     const unsigned char* first,
                                                     size_t len) {
  while (len >= PiecewiseChunkSize()) {
    state =
        Mix(state, absl::hash_internal::CityHash32(reinterpret_cast<const char*>(first),
                                 PiecewiseChunkSize()));
    len -= PiecewiseChunkSize();
    first += PiecewiseChunkSize();
  }
  // Handle the remainder.
  return CombineContiguousImpl(state, first, len,
                               std::integral_constant<int, 4>{});
}

uint64_t CityHashState::CombineLargeContiguousImpl64(uint64_t state,
                                                     const unsigned char* first,
                                                     size_t len) {
  while (len >= PiecewiseChunkSize()) {
    state =
        Mix(state, absl::hash_internal::CityHash64(reinterpret_cast<const char*>(first),
                                 PiecewiseChunkSize()));
    len -= PiecewiseChunkSize();
    first += PiecewiseChunkSize();
  }
  // Handle the remainder.
  return CombineContiguousImpl(state, first, len,
                               std::integral_constant<int, 8>{});
}

ABSL_CONST_INIT const void* const CityHashState::kSeed = &kSeed;

}  // namespace hash_internal
ABSL_NAMESPACE_END
}  // namespace absl
```

The Abseil traits classes and allocator traits

Pointer wrapper templates

Follows a discussion on the implementation of various pointer templates and related wrapper class templates starting with absl::WrapUnique.

Absl::WrapUnique implementation

The whole point of the function template WrapUnique is to deduce the type of the raw pointer and return std::unique_ptr<T>(ptr). Avoid using WrapUnique when the class constructor to be instantiated needs arguments. In such case using std::make_unique<T>(args…) is preferable to absl::WrapUnique(new T(args…)) which will need a raw **new** operator invocation.

Code Snippet: Function template WrapUnique()

```
//
// Adopts ownership from a raw pointer and transfers it to the returned
// `std::unique_ptr`, whose type is deduced. Because of this deduction, *do not*
// specify the template type `T` when calling `WrapUnique`.
//
// Example:
//   X* NewX(int, int);
//   auto x = WrapUnique(NewX(1, 2));  // 'x' is std::unique_ptr<X>.
//
// Do not call WrapUnique with an explicit type, as in
// `WrapUnique<X>(NewX(1, 2))`.  The purpose of WrapUnique is to automatically
// deduce the pointer type. If you wish to make the type explicit, just use
// `std::unique_ptr` directly.
//
//   auto x = std::unique_ptr<X>(NewX(1, 2));
//                  - or -
//   std::unique_ptr<X> x(NewX(1, 2));
//
```

```
// While `absl::WrapUnique` is useful for capturing the output of a raw
// pointer factory, prefer 'absl::make_unique<T>(args...)' over
// 'absl::WrapUnique(new T(args...))'.
//
//   auto x = WrapUnique(new X(1, 2));  // works, but nonideal.
//   auto x = make_unique<X>(1, 2);     // safer, standard, avoids raw 'new'.
//
// Note that `absl::WrapUnique(p)` is valid only if `delete p` is a valid
// expression. In particular, `absl::WrapUnique()` cannot wrap pointers to
// arrays, functions or void, and it must not be used to capture pointers
// obtained from array-new expressions (even though that would compile!).
template <typename T>
std::unique_ptr<T> WrapUnique(T* ptr) {
  static_assert(!std::is_array<T>::value, "array types are unsupported");
  static_assert(std::is_object<T>::value, "non-object types are unsupported");
  return std::unique_ptr<T>(ptr);
}
```

Absl::MakeUnique and absl::make_unique implementation

Next we have the function template MakeUniqueResult which provides two partial specializations – one for array argument type and the other is for an array with known size where the latter actually is invalid way to create unique_ptr instance.

Code Snippet: MakeUniqueResult

```
namespace memory_internal {

// Traits to select proper overload and return type for `absl::make_unique<>`.
template <typename T>
struct MakeUniqueResult {
  using scalar = std::unique_ptr<T>;
};
template <typename T>
struct MakeUniqueResult<T[]> {
  using array = std::unique_ptr<T[]>;
};
template <typename T, size_t N>
struct MakeUniqueResult<T[N]> {
  using invalid = void;
};

}  // namespace memory_internal
```

And now we are going to take a look at the internal abseil implementation of make_unique which is used when we compile with C++ v11 ***and*** the version of the compiler is older than 201103L ***and*** it is not microsoft C++ compiler. Note that this is not the case with TensorFlow V2 which is compiled with C++ v14 but nevertheless we will take a look into this C++ 11-specific abseil implementation of **make_unique**. On first place the reason for existence of function template make_unique is discussed in

an article Exception-safe function calls which explains why invocation using operator ***new*** such as **std::unique_ptr<T>(new T(a,b))** is problematic. Basically, if an exception is thrown during the memory allocation or construction of object ***a*** then we may have a memory leak due to ***b*** for which the C++ standard does not require to be destroyed and memory deallocated.
Example usage is shown below:

```
auto p = make_unique<X>(args...);  // 'p'  is a std::unique_ptr<X>
auto pa = make_unique<X[]>(5);     // 'pa' is a std::unique_ptr<X[]>
```

Three overloads of `absl::make_unique` are supplied:

- For non-array T:

Allocates a T with **new T(std::forward<Args> args...)**, forwarding all **args** to T's constructor. Returns a **std::unique_ptr<T>** owning that object.

```
template <typename T, typename... Args>
typename memory_internal::MakeUniqueResult<T>::scalar make_unique(
    Args&&... args) {
  return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}
```

- For an array of unknown bounds T[]:

absl::make_unique<> will allocate an array T of type U[] with **new U[n]()** and return a **std::unique_ptr<U[]>** owning that array.

Note that **U[n]()** is different from **U[n]**, and elements will be value-initialized. ***Note to myself***: The last statement does not make sense as **new U[n]()** and **new U[n]** will produce the same result invoking the default constructor of U for each element of U[n] after performing the allocation. Note as well that std::unique_ptr will perform its own destruction of the array elements upon leaving scope, even though the array [] does not have a default destructor.

Note: an array of unknown bounds T[] may still be (and often will be)
initialized to have a size, and will still use this overload. E.g:

```
auto my_array = absl::make_unique<int[]>(10);
```

Below It is shown the **absl::make_unique** overload for an array T[] of unknown bounds.
The array allocation needs to use the **new T[size]** form and cannot take
element constructor arguments. The **std::unique_ptr** will manage destructing
these array elements.

```
template <typename T>
typename memory_internal::MakeUniqueResult<T>::array make_unique(size_t n) {
  return std::unique_ptr<T>(new typename absl::remove_extent_t<T>[n]());
}
```

 - For an array of known bounds T[N]:
absl::make_unique<> is deleted (like with **std::make_unique<>**) as this overload is not useful.

Note: an array of known bounds T[N] is not considered a useful construction, and may cause undefined behavior in templates. E.g:

```
auto my_array = absl::make_unique<int[10]>();
```

In those cases, of course, you can still use the overload above and simply initialize it to its desired size:

```
auto my_array = absl::make_unique<int[]>(10);
```

Notes on std::remove_extent

Defined in header <type_traits>

```
template< class T >
struct remove_extent; (since C++11)
```

If T is an array of some type X, provides the member typedef type equal to X, otherwise type is T. Note that if T is a multidimensional array, only the first dimension is removed.

The behavior of a program that adds specializations for remove_extent is undefined.

Member types

Name	**Definition**
type	the type of the element of T

Helper types

```
template< class T >
using remove_extent_t = typename remove_extent<T>::type; (since C++14)
```

Possible implementation

```
template<class T>
struct remove_extent { typedef T type; };

template<class T>
struct remove_extent<T[]> { typedef T type; };

template<class T, std::size_t N>
struct remove_extent<T[N]> { typedef T type; };
```

Below is the full code of the absl::make_unique implementation

Code Snippet: absl::make_unique implementation

```
// gcc 4.8 has __cplusplus at 201301 but the libstdc++ shipped with it doesn't
// define make_unique.  Other supported compilers either just define __cplusplus
// as 201103 but have make_unique (msvc), or have make_unique whenever
// __cplusplus > 201103 (clang).
#if (__cplusplus > 201103L || defined(_MSC_VER)) && \
    !(defined(__GLIBCXX__) && !defined(__cpp_lib_make_unique))
using std::make_unique;
#else
// -----------------------------------------------------------------------------
// Function Template: make_unique<T>()
// -----------------------------------------------------------------------------
//
// Creates a `std::unique_ptr<>`, while avoiding issues creating temporaries
// during the construction process. `absl::make_unique<>` also avoids redundant
// type declarations, by avoiding the need to explicitly use the `new` operator.
//
// This implementation of `absl::make_unique<>` is designed for C++11 code and
// will be replaced in C++14 by the equivalent `std::make_unique<>` abstraction.
// `absl::make_unique<>` is designed to be 100% compatible with
// `std::make_unique<>` so that the eventual migration will involve a simple
// rename operation.
//
// For more background on why `std::unique_ptr<T>(new T(a,b))` is problematic,
// see Herb Sutter's explanation on
// (Exception-Safe Function Calls)[https://herbsutter.com/gotw/_102/].
// (In general, reviewers should treat `new T(a,b)` with scrutiny.)
//
// Example usage:
//
//    auto p = make_unique<X>(args...);  // 'p'  is a std::unique_ptr<X>
//    auto pa = make_unique<X[]>(5);     // 'pa' is a std::unique_ptr<X[]>
//
// Three overloads of `absl::make_unique` are required:
//
//   - For non-array T:
//
//       Allocates a T with `new T(std::forward<Args> args...)`,
//       forwarding all `args` to T's constructor.
//       Returns a `std::unique_ptr<T>` owning that object.
//
//   - For an array of unknown bounds T[]:
//
//       `absl::make_unique<>` will allocate an array T of type U[] with
//       `new U[n]()` and return a `std::unique_ptr<U[]>` owning that array.
//
//       Note that 'U[n]()' is different from 'U[n]', and elements will be
//       value-initialized. Note as well that `std::unique_ptr` will perform its
//       own destruction of the array elements upon leaving scope, even though
//       the array [] does not have a default destructor.
//
//       NOTE: an array of unknown bounds T[] may still be (and often will be)
//       initialized to have a size, and will still use this overload. E.g:
//
```

```
//        auto my_array = absl::make_unique<int[]>(10);
//
//   - For an array of known bounds T[N]:
//
//      `absl::make_unique<>` is deleted (like with `std::make_unique<>`) as
//      this overload is not useful.
//
//      NOTE: an array of known bounds T[N] is not considered a useful
//      construction, and may cause undefined behavior in templates. E.g:
//
//        auto my_array = absl::make_unique<int[10]>();
//
//      In those cases, of course, you can still use the overload above and
//      simply initialize it to its desired size:
//
//        auto my_array = absl::make_unique<int[]>(10);

// `absl::make_unique` overload for non-array types.
template <typename T, typename... Args>
typename memory_internal::MakeUniqueResult<T>::scalar make_unique(
    Args&&... args) {
  return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}

// `absl::make_unique` overload for an array T[] of unknown bounds.
// The array allocation needs to use the `new T[size]` form and cannot take
// element constructor arguments. The `std::unique_ptr` will manage destructing
// these array elements.
template <typename T>
typename memory_internal::MakeUniqueResult<T>::array make_unique(size_t n) {
  return std::unique_ptr<T>(new typename absl::remove_extent_t<T>[n]());
}

// `absl::make_unique` overload for an array T[N] of known bounds.
// This construction will be rejected.
template <typename T, typename... Args>
typename memory_internal::MakeUniqueResult<T>::invalid make_unique(
    Args&&... /* args */) = delete;
#endif
```

Next let us take look at the RawPtr implementation starting with notes on the implementation of std::addressof, followed by std::is_object discussion and std::weak_ptr<T> which are used in RawPtr implementation.

Notes on std::addressof

std::addressof is defined in header <memory>

1)

```
template< class T >
T* addressof(T& arg) noexcept;  (since C++11) (until C++17)
template< class T >
constexpr T* addressof(T& arg) noexcept; (since C++17)
```

2)

```
template <class T>
const T* addressof(const T&&) = delete; (since C++17)
```

1) Obtains the actual address of the object or function arg, even in presence of overloaded operator&
2) Rvalue overload is deleted to prevent taking the address of const rvalues.

The expression std::addressof(E) is a constant subexpression, if E is an lvalue constant subexpression. *(since C++17)*

Parameters
arg: lvalue object or function
Return value: Pointer to **arg**.

Possible implementation

```
template<class T>
typename std::enable_if<std::is_object<T>::value, T*>::type  addressof(T& arg) noexcept
{
   return reinterpret_cast<T*>(
            &const_cast<char&>(
               reinterpret_cast<const volatile char&>(arg)));
}

template<class T>
typename std::enable_if<!std::is_object<T>::value, T*>::type addressof(T& arg) noexcept
{
   return &arg;
}
```

Notes on std::is_object<T> and std::integral_constant<T, T v>

std::integral_constant<T, T v> is defined in header <type_traits> as shown below:

```
template< class T, T v >
struct integral_constant; (since C++11)
```

std::integral_constant wraps a static constant of specified type. It is the base class for the C++ type traits.

Note: The behavior of a program that adds specializations for integral_constant is undefined.

Helper templates

A helper alias template std::bool_constant is defined for the common case where T is bool.

```
template <bool B>
using bool_constant = integral_constant<bool, B>; (since C++17)
```

Helper types

Two typedefs for the common case where T is bool are provided and defined in header <type_traits>

Type	**Definition**
true_type	std::integral_constant<bool, true>
false_type	std::integral_constant<bool, false>

Member types

Type	**Definition**
value_type	T
type	std::integral_constant<T,v>

Member constants

Name	**Value**
constexpr T value *[static]*	static constant of type T with value v *(public static member constant)*

Member functions

operator value_type	returns the wrapped value *(public member function)*
operator() (C++14)	returns the wrapped value *(public member function)*

std::integral_constant::operator value_type

```
constexpr operator value_type() const noexcept;
```

Conversion function. Returns the wrapped value.

std::integral_constant::operator()

```
constexpr value_type operator()() const noexcept; (since C++14)
```

Returns the wrapped value. This function enables std::integral_constant to serve as a source of compile-time function objects.

Possible implementation

```
template<class T, T v>
struct integral_constant {
   static constexpr T value = v;
   using value_type = T;
   using type = integral_constant; // using injected-class-name
   constexpr operator value_type() const noexcept { return value; }
   constexpr value_type operator()() const noexcept { return value; } //since c++14
};
```

std::is_object<T> is defined in header <type_traits> as shown below:

```
template< class T >
struct is_object; (since C++11)
```

If T is an object type (that is any possibly cv-qualified type other than function, reference, or void types), provides the member constant value equal true. For any other type, value is false.

The behavior of a program that adds specializations for is_object or is_object_v (since C++17) is undefined.

Template parameters

T	-	a type to check

Helper variable template
template< class T >
inline constexpr bool is_object_v = is_object<T>::value;
(since C++17)
Inherited from std::integral_constant
Member constants
value

[static]

true if T is an object type , false otherwise
(public static member constant)
Member functions
operator bool

converts the object to bool, returns value
(public member function)
operator()

(C++14)

returns value
(public member function)
Member types

Type	Definition
value_type	bool
type	std::integral_constant<bool, value>

Possible implementation
```
template< class T>
struct is_object : std::integral_constant<bool,
                 std::is_scalar<T>::value ||
                 std::is_array<T>::value  ||
                 std::is_union<T>::value  ||
                 std::is_class<T>::value> {};
```

Notes on std::weak_ptr<T>

RawPtr implementation

```
// -----------------------------------------------------------------------------
// Function Template: RawPtr()
// -----------------------------------------------------------------------------
//
// Extracts the raw pointer from a pointer-like value `ptr`. `absl::RawPtr` is
// useful within templates that need to handle a complement of raw pointers,
// `std::nullptr_t`, and smart pointers.
template <typename T>
auto RawPtr(T&& ptr) -> decltype(std::addressof(*ptr)) {
  // ptr is a forwarding reference to support Ts with non-const operators.
  return (ptr != nullptr) ? std::addressof(*ptr) : nullptr;
}
inline std::nullptr_t RawPtr(std::nullptr_t) { return nullptr; }

// -----------------------------------------------------------------------------
// Function Template: ShareUniquePtr()
// -----------------------------------------------------------------------------
//
// Adopts a `std::unique_ptr` rvalue and returns a `std::shared_ptr` of deduced
// type. Ownership (if any) of the held value is transferred to the returned
// shared pointer.
//
// Example:
//
//     auto up = absl::make_unique<int>(10);
//     auto sp = absl::ShareUniquePtr(std::move(up));  // shared_ptr<int>
//     CHECK_EQ(*sp, 10);
//     CHECK(up == nullptr);
//
// Note that this conversion is correct even when T is an array type, and more
// generally it works for *any* deleter of the `unique_ptr` (single-object
// deleter, array deleter, or any custom deleter), since the deleter is adopted
// by the shared pointer as well. The deleter is copied (unless it is a
// reference).
//
// Implements the resolution of [LWG 2415](http://wg21.link/lwg2415), by which a
// null shared pointer does not attempt to call the deleter.
template <typename T, typename D>
std::shared_ptr<T> ShareUniquePtr(std::unique_ptr<T, D>&& ptr) {
  return ptr ? std::shared_ptr<T>(std::move(ptr)) : std::shared_ptr<T>();
}

// -----------------------------------------------------------------------------
// Function Template: WeakenPtr()
// -----------------------------------------------------------------------------
//
// Creates a weak pointer associated with a given shared pointer. The returned
// value is a `std::weak_ptr` of deduced type.
```

```
//
// Example:
//
//    auto sp = std::make_shared<int>(10);
//    auto wp = absl::WeakenPtr(sp);
//    CHECK_EQ(sp.get(), wp.lock().get());
//    sp.reset();
//    CHECK(wp.lock() == nullptr);
//
template <typename T>
std::weak_ptr<T> WeakenPtr(const std::shared_ptr<T>& ptr) {
  return std::weak_ptr<T>(ptr);
}
```

Helper traits

Let us look into some of the helper traits within the namespace absl::memory_internal (see Code Snippet Code Snippet: Helper traits in absl::memory_internal). Those traits are used heavily elsewhere in the abseil container templates and synchronization primitives. The first trait class ExtractOr<E, O, D> is helper trait which by using SFINAE evaluates to E<O> if this substitution works otherwise evaluates to D. The wrapper class ExtractOrT exposes this concepts for use outside of absl::memory_internal. Notice the usage of the introduced in C++ 17 std::void_t which maps a sequence of any types to the type void.
It is defined in header <type_traits> as:

```
template< class... >
using void_t = void;
```

<u>Note on</u> std::void_t
This metafunction is used in template metaprogramming to detect ill-formed types in SFINAE context:

```
// primary template handles types that have no nested ::type member:
template< class, class = std::void_t<> >
struct has_type_member : std::false_type { };

// specialization recognizes types that do have a nested ::type member:
template< class T >
struct has_type_member<T, std::void_t<typename T::type>> : std::true_type { };
```

It can also be used to detect validity of an expression:

```
// primary template handles types that do not support pre-increment:
template< class, class = std::void_t<> >
struct has_pre_increment_member : std::false_type { };
// specialization recognizes types that do support pre-increment:
template< class T >
struct has_pre_increment_member<T,
          std::void_t<decltype( ++std::declval<T&>() )>
       > : std::true_type { };
```

Until CWG 1558 (a C++14 defect), (see the Note CWG 1558: Unused arguments in alias template specializations below) unused parameters in alias templates were not guaranteed to ensure SFINAE and could be ignored, so earlier compilers require a more complex definition of void_t, such as

```
template<typename... Ts> struct make_void { typedef void type;};
template<typename... Ts> using void_t = typename make_void<Ts...>::type;
```

Note on *CWG 1558: Unused arguments in alias template specializations*
Section: 14.5.7 [temp.alias]   Status: CD4   Submitter: Nikolay Ivchenkov   Date: 2012-09-19
[Moved to DR at the November, 2014 meeting.]

The treatment of unused arguments in an alias template specialization is not specified by the current wording of 14.5.7 [temp.alias]. For example:

```
 #include <iostream>

 template <class T, class...>
   using first_of = T;

 template <class T>
   first_of<void, typename T::type> f(int)
     { std::cout << "1\n"; }

 template <class T>
   void f(...)
     { std::cout << "2\n"; }

 struct X { typedef void type; };

 int main() {
   f<X>(0);
   f<int>(0);
 }
```

Is the reference to first_of<void, T::type> with T being int equivalent to simply void, or is it a substitution failure?

(See also issues 1430, 1520, and 1554.)

Notes from the October, 2012 meeting:

The consensus of CWG was to treat this case as substitution failure.

Proposed resolution (February, 2014):

Add the following as a new paragraph before 14.5.7 [temp.alias] paragraph 3:

When a template-id refers to the specialization of an alias template, it is equivalent...

However, if the template-id is dependent, subsequent template argument substitution still applies to the template-id. [Example:

```
 template<typename...> using void_t = void;
 template<typename T> void_t<typename T::foo> f();
 f<int>(); // error, int does not have a nested type foo
```
—end example]

The type-id in an alias template declaration shall not refer...

For details refer to section 14.5.7 of C++ v11 Specification e.g. version n3690 .

Notes on std::declvalue
Converts any type T to a reference type, making it possible to use member functions in decltype expressions without the need to go through constructors.

declval is commonly used in templates where acceptable template parameters may have no constructor in common, but have the same member function whose return type is needed.

Note that declval can only be used in unevaluated contexts and is not required to be defined; it is an error to evaluate an expression that contains this function. Formally, the program is ill-formed if this function is odr-used.

Parameters
(none)

Return value
Cannot be called and thus never returns a value. The return type is T&& unless T is (possibly cv-qualified) void, in which case the return type is T.
Defined in header <utility>

```
template<class T>
typename std::add_rvalue_reference<T>::type declval() noexcept;
```

Another interesting trait class which uses SFINAE in the code snippet below is GetFirstArg. GetFirstArg accepts a single template parameter. In its partial specialization this single template parameter takes the form of a class template with two template arguments which are a type, followed by a parameter pack.

```
template <typename T>
struct GetFirstArg;

template <template <typename...> class Class, typename T, typename... Args>
struct GetFirstArg<Class<T, Args...>> {
  using type = T;
};
```

The trait GetFirstArg is invoked like this for example:

```
template< typename T, typename... Args >
struct X {
};
using typename GetFirstArg<X<int>>::type;
```

Another class trait which uses SFINAE and partial specialization is ElementType. This trait does two different things depending on the availability of member T::element_type. If such member exists in T then it sets its own ElementType<T>::type to it. Otherwise, it returns the first template argument type of the template class which expands to T.

```
template <typename Ptr, typename = void>
struct ElementType {
  using type = typename GetFirstArg<Ptr>::type;
};

template <typename T>
struct ElementType<T, void_t<typename T::element_type>> {
  using type = typename T::element_type;
};
```

Another interesting class trait which uses SFINAE and partial specialization is RebindFirstArg shown below. RebindFirstArg accepts two template parameters one of which is a class template with two template arguments – the type U and the parameter pack Args… . The exposed type in the maching template specialization of RebindFirstArg defines a type constructed by using the same template class Class but instantiated with parameter U as first argument instead of T. Hence the name of the trait – RebindFirstArg.

```
template <typename T, typename U>
struct RebindFirstArg;

template <template <typename...> class Class, typename T, typename... Args,
        typename U>
struct RebindFirstArg<Class<T, Args...>, U> {
  using type = Class<U, Args...>;
};
```

Another class trait which uses SFINAE and partial specialization is RebindPtr. Notice the access to template type argument T's method template **rebind** via the expression typename T::template rebind<U>. So if template argument type T evaluates to a class template instantiated with a template type as an argument RebindFirstArg reinstantiates it and rebinds it with the template argument type U. Otherwise it instantiate the T's method template rebind with template argument type U.

```
template <typename T, typename U, typename = void>
struct RebindPtr {
  using type = typename RebindFirstArg<T, U>::type;
};

template <typename T, typename U>
struct RebindPtr<T, U, void_t<typename T::template rebind<U>>> {
  using type = typename T::template rebind<U>;
};
```

which is used as:

```
// rebind:
// Ptr::rebind<U> if exists, otherwise Template<U, Args...> if Ptr is a
// template instantiation Template<T, Args...>
template <typename U>
using rebind = typename memory_internal::RebindPtr<Ptr, U>::type;
```

A similar template class which

```
template <typename T, typename U>
constexpr bool HasRebindAlloc(...) {
  return false;
}

template <typename T, typename U>
constexpr bool HasRebindAlloc(typename T::template rebind<U>::other*) {
  return true;
}
```

Code Snippet: Helper traits in absl::memory_internal

```
namespace memory_internal {

// ExtractOr<E, O, D>::type evaluates to E<O> if possible. Otherwise, D.
template <template <typename> class Extract, typename Obj, typename Default,
          typename>
struct ExtractOr {
  using type = Default;
};

template <template <typename> class Extract, typename Obj, typename Default>
struct ExtractOr<Extract, Obj, Default, void_t<Extract<Obj>>> {
  using type = Extract<Obj>;
};

template <template <typename> class Extract, typename Obj, typename Default>
using ExtractOrT = typename ExtractOr<Extract, Obj, Default, void>::type;

// Extractors for the features of allocators.
template <typename T>
using GetPointer = typename T::pointer;
```

```
template <typename T>
using GetConstPointer = typename T::const_pointer;

template <typename T>
using GetVoidPointer = typename T::void_pointer;

template <typename T>
using GetConstVoidPointer = typename T::const_void_pointer;

template <typename T>
using GetDifferenceType = typename T::difference_type;

template <typename T>
using GetSizeType = typename T::size_type;

template <typename T>
using GetPropagateOnContainerCopyAssignment =
    typename T::propagate_on_container_copy_assignment;

template <typename T>
using GetPropagateOnContainerMoveAssignment =
    typename T::propagate_on_container_move_assignment;

template <typename T>
using GetPropagateOnContainerSwap = typename T::propagate_on_container_swap;

template <typename T>
using GetIsAlwaysEqual = typename T::is_always_equal;

template <typename T>
struct GetFirstArg;

template <template <typename...> class Class, typename T, typename... Args>
struct GetFirstArg<Class<T, Args...>> {
  using type = T;
};

template <typename Ptr, typename = void>
struct ElementType {
  using type = typename GetFirstArg<Ptr>::type;
};

template <typename T>
struct ElementType<T, void_t<typename T::element_type>> {
  using type = typename T::element_type;
};
```

```
template <typename T, typename U>
struct RebindFirstArg;

template <template <typename...> class Class, typename T, typename... Args,
          typename U>
struct RebindFirstArg<Class<T, Args...>, U> {
  using type = Class<U, Args...>;
};

template <typename T, typename U, typename = void>
struct RebindPtr {
  using type = typename RebindFirstArg<T, U>::type;
};

template <typename T, typename U>
struct RebindPtr<T, U, void_t<typename T::template rebind<U>>> {
  using type = typename T::template rebind<U>;
};

template <typename T, typename U>
constexpr bool HasRebindAlloc(...) {
  return false;
}

template <typename T, typename U>
constexpr bool HasRebindAlloc(typename T::template rebind<U>::other*) {
  return true;
}

template <typename T, typename U, bool = HasRebindAlloc<T, U>(nullptr)>
struct RebindAlloc {
  using type = typename RebindFirstArg<T, U>::type;
};

template <typename T, typename U>
struct RebindAlloc<T, U, true> {
  using type = typename T::template rebind<U>::other;
};

}  // namespace memory_internal
```

Pointer traits

Code Snippet: pointer_traits

```
// ---------------------------------------------------------------------------
// Class Template: pointer_traits
// ---------------------------------------------------------------------------
```

```
//
// An implementation of C++11's std::pointer_traits.
//
// Provided for portability on toolchains that have a working C++11 compiler,
// but the standard library is lacking in C++11 support. For example, some
// version of the Android NDK.
//

template <typename Ptr>
struct pointer_traits {
  using pointer = Ptr;

  // element_type:
  // Ptr::element_type if present. Otherwise T if Ptr is a template
  // instantiation Template<T, Args...>
  using element_type = typename memory_internal::ElementType<Ptr>::type;

  // difference_type:
  // Ptr::difference_type if present, otherwise std::ptrdiff_t
  using difference_type =
      memory_internal::ExtractOrT<memory_internal::GetDifferenceType, Ptr,
                                  std::ptrdiff_t>;

  // rebind:
  // Ptr::rebind<U> if exists, otherwise Template<U, Args...> if Ptr is a
  // template instantiation Template<T, Args...>
  template <typename U>
  using rebind = typename memory_internal::RebindPtr<Ptr, U>::type;

  // pointer_to:
  // Calls Ptr::pointer_to(r)
  static pointer pointer_to(element_type& r) {  // NOLINT(runtime/references)
    return Ptr::pointer_to(r);
  }
};

// Specialization for T*.
template <typename T>
struct pointer_traits<T*> {
  using pointer = T*;
  using element_type = T;
  using difference_type = std::ptrdiff_t;

  template <typename U>
  using rebind = U*;

  // pointer_to:
  // Calls std::addressof(r)
  static pointer pointer_to(
      element_type& r) noexcept {  // NOLINT(runtime/references)
    return std::addressof(r);
  }
};
```

Allocator traits

Let us take a look at the trait class absl::allocator_traits which is C++ 11 compatible implementation of C++17's std::allocator_traits shown in Code SnippetCode Snippet: Backported to C++11 C++17 std::allocator_traits.

Code Snippet: Backported to C++11 C++17 std::allocator_traits

```
template <typename Alloc>
struct allocator_traits {
  using allocator_type = Alloc;

  // value_type:
  // Alloc::value_type
  using value_type = typename Alloc::value_type;

  // pointer:
  // Alloc::pointer if present, otherwise value_type*
  using pointer = memory_internal::ExtractOrT<memory_internal::GetPointer,
                                              Alloc, value_type*>;

  // const_pointer:
  // Alloc::const_pointer if present, otherwise
  // absl::pointer_traits<pointer>::rebind<const value_type>
  using const_pointer =
      memory_internal::ExtractOrT<memory_internal::GetConstPointer, Alloc,
                                  typename absl::pointer_traits<pointer>::
                                      template rebind<const value_type>>;

  // void_pointer:
  // Alloc::void_pointer if present, otherwise
  // absl::pointer_traits<pointer>::rebind<void>
  using void_pointer = memory_internal::ExtractOrT<
      memory_internal::GetVoidPointer, Alloc,
      typename absl::pointer_traits<pointer>::template rebind<void>>;

  // const_void_pointer:
  // Alloc::const_void_pointer if present, otherwise
  // absl::pointer_traits<pointer>::rebind<const void>
  using const_void_pointer = memory_internal::ExtractOrT<
      memory_internal::GetConstVoidPointer, Alloc,
      typename absl::pointer_traits<pointer>::template rebind<const void>>;

  // difference_type:
  // Alloc::difference_type if present, otherwise
  // absl::pointer_traits<pointer>::difference_type
  using difference_type = memory_internal::ExtractOrT<
      memory_internal::GetDifferenceType, Alloc,
      typename absl::pointer_traits<pointer>::difference_type>;

  // size_type:
  // Alloc::size_type if present, otherwise
  // std::make_unsigned<difference_type>::type
  using size_type = memory_internal::ExtractOrT<
      memory_internal::GetSizeType, Alloc,
      typename std::make_unsigned<difference_type>::type>;

  // propagate_on_container_copy_assignment:
  // Alloc::propagate_on_container_copy_assignment if present, otherwise
  // std::false_type
  using propagate_on_container_copy_assignment = memory_internal::ExtractOrT<
      memory_internal::GetPropagateOnContainerCopyAssignment, Alloc,
      std::false_type>;

  // propagate_on_container_move_assignment:
  // Alloc::propagate_on_container_move_assignment if present, otherwise
  // std::false_type
  using propagate_on_container_move_assignment = memory_internal::ExtractOrT<
      memory_internal::GetPropagateOnContainerMoveAssignment, Alloc,
      std::false_type>;

  // propagate_on_container_swap:
  // Alloc::propagate_on_container_swap if present, otherwise std::false_type
  using propagate_on_container_swap =
      memory_internal::ExtractOrT<memory_internal::GetPropagateOnContainerSwap,
                                  Alloc, std::false_type>;

  // is_always_equal:
  // Alloc::is_always_equal if present, otherwise std::is_empty<Alloc>::type
  using is_always_equal =
      memory_internal::ExtractOrT<memory_internal::GetIsAlwaysEqual, Alloc,
                                  typename std::is_empty<Alloc>::type>;

  // rebind_alloc:
  // Alloc::rebind<T>::other if present, otherwise Alloc<T, Args> if this Alloc
  // is Alloc<U, Args>
  template <typename T>
  using rebind_alloc = typename memory_internal::RebindAlloc<Alloc, T>::type;
```

```
// rebind_traits:
// absl::allocator_traits<rebind_alloc<T>>
template <typename T>
using rebind_traits = absl::allocator_traits<rebind_alloc<T>>;

// allocate(Alloc& a, size_type n):
// Calls a.allocate(n)
static pointer allocate(Alloc& a,  // NOLINT(runtime/references)
                        size_type n) {
  return a.allocate(n);
}

// allocate(Alloc& a, size_type n, const_void_pointer hint):
// Calls a.allocate(n, hint) if possible.
// If not possible, calls a.allocate(n)
static pointer allocate(Alloc& a, size_type n,  // NOLINT(runtime/references)
                        const_void_pointer hint) {
  return allocate_impl(0, a, n, hint);
}

// deallocate(Alloc& a, pointer p, size_type n):
// Calls a.deallocate(p, n)
static void deallocate(Alloc& a, pointer p,  // NOLINT(runtime/references)
                       size_type n) {
  a.deallocate(p, n);
}

// construct(Alloc& a, T* p, Args&&... args):
// Calls a.construct(p, std::forward<Args>(args)...) if possible.
// If not possible, calls
//   ::new (static_cast<void*>(p)) T(std::forward<Args>(args)...)
template <typename T, typename... Args>
static void construct(Alloc& a, T* p,  // NOLINT(runtime/references)
                      Args&&... args) {
  construct_impl(0, a, p, std::forward<Args>(args)...);
}

// destroy(Alloc& a, T* p):
// Calls a.destroy(p) if possible. If not possible, calls p->~T().
template <typename T>
static void destroy(Alloc& a, T* p) {  // NOLINT(runtime/references)
  destroy_impl(0, a, p);
}

// max_size(const Alloc& a):
// Returns a.max_size() if possible. If not possible, returns
//   std::numeric_limits<size_type>::max() / sizeof(value_type)
static size_type max_size(const Alloc& a) { return max_size_impl(0, a); }
```

```
  // select_on_container_copy_construction(const Alloc& a):
  // Returns a.select_on_container_copy_construction() if possible.
  // If not possible, returns a.
  static Alloc select_on_container_copy_construction(const Alloc& a) {
    return select_on_container_copy_construction_impl(0, a);
  }

 private:
  template <typename A>
  static auto allocate_impl(int, A& a,  // NOLINT(runtime/references)
                            size_type n, const_void_pointer hint)
      -> decltype(a.allocate(n, hint)) {
    return a.allocate(n, hint);
  }
  static pointer allocate_impl(char, Alloc& a,  // NOLINT(runtime/references)
                               size_type n, const_void_pointer) {
    return a.allocate(n);
  }

  template <typename A, typename... Args>
  static auto construct_impl(int, A& a,  // NOLINT(runtime/references)
                             Args&&... args)
      -> decltype(a.construct(std::forward<Args>(args)...)) {
    a.construct(std::forward<Args>(args)...);
  }

  template <typename T, typename... Args>
  static void construct_impl(char, Alloc&, T* p, Args&&... args) {
    ::new (static_cast<void*>(p)) T(std::forward<Args>(args)...);
  }

  template <typename A, typename T>
  static auto destroy_impl(int, A& a,  // NOLINT(runtime/references)
                           T* p) -> decltype(a.destroy(p)) {
    a.destroy(p);
  }
  template <typename T>
  static void destroy_impl(char, Alloc&, T* p) {
    p->~T();
  }

  template <typename A>
  static auto max_size_impl(int, const A& a) -> decltype(a.max_size()) {
    return a.max_size();
  }
  static size_type max_size_impl(char, const Alloc&) {
    return (std::numeric_limits<size_type>::max)() / sizeof(value_type);
  }

  template <typename A>
  static auto select_on_container_copy_construction_impl(int, const A& a)
      -> decltype(a.select_on_container_copy_construction()) {
    return a.select_on_container_copy_construction();
  }
  static Alloc select_on_container_copy_construction_impl(char,
                                                           const Alloc& a) {
    return a;
  }
};
```

Code Snippet: Template alias transforming Alloc::is_nothrow into metafunction

```
namespace memory_internal {

// This template alias transforms Alloc::is_nothrow into a metafunction with
// Alloc as a parameter so it can be used with ExtractOrT<>.
template <typename Alloc>
using GetIsNothrow = typename Alloc::is_nothrow;

}  // namespace memory_internal

// ABSL_ALLOCATOR_NOTHROW is a build time configuration macro for user to
// specify whether the default allocation function can throw or never throws.
// If the allocation function never throws, user should define it to a non-zero
// value (e.g. via `-DABSL_ALLOCATOR_NOTHROW`).
// If the allocation function can throw, user should leave it undefined or
// define it to zero.
//
// allocator_is_nothrow<Alloc> is a traits class that derives from
// Alloc::is_nothrow if present, otherwise std::false_type. It's specialized
// for Alloc = std::allocator<T> for any type T according to the state of
// ABSL_ALLOCATOR_NOTHROW.
//
// default_allocator_is_nothrow is a class that derives from std::true_type
// when the default allocator (global operator new) never throws, and
// std::false_type when it can throw. It is a convenience shorthand for writing
// allocator_is_nothrow<std::allocator<T>> (T can be any type).
// NOTE: allocator_is_nothrow<std::allocator<T>> is guaranteed to derive from
// the same type for all T, because users should specialize neither
// allocator_is_nothrow nor std::allocator.
template <typename Alloc>
struct allocator_is_nothrow
    : memory_internal::ExtractOrT<memory_internal::GetIsNothrow, Alloc,
                                  std::false_type> {};

#if defined(ABSL_ALLOCATOR_NOTHROW) && ABSL_ALLOCATOR_NOTHROW
template <typename T>
struct allocator_is_nothrow<std::allocator<T>> : std::true_type {};
struct default_allocator_is_nothrow : std::true_type {};
#else
struct default_allocator_is_nothrow : std::false_type {};
#endif

namespace memory_internal {
template <typename Allocator, typename Iterator, typename... Args>
void ConstructRange(Allocator& alloc, Iterator first, Iterator last,
                    const Args&... args) {
  for (Iterator cur = first; cur != last; ++cur) {
    ABSL_INTERNAL_TRY {
      std::allocator_traits<Allocator>::construct(alloc, std::addressof(*cur),
                                                  args...);
    }
    ABSL_INTERNAL_CATCH_ANY {
      while (cur != first) {
        --cur;
        std::allocator_traits<Allocator>::destroy(alloc, std::addressof(*cur));
      }
      ABSL_INTERNAL_RETHROW;
    }
  }
}

template <typename Allocator, typename Iterator, typename InputIterator>
void CopyRange(Allocator& alloc, Iterator destination, InputIterator first,
               InputIterator last) {
  for (Iterator cur = destination; first != last;
       static_cast<void>(++cur), static_cast<void>(++first)) {
    ABSL_INTERNAL_TRY {
      std::allocator_traits<Allocator>::construct(alloc, std::addressof(*cur),
                                                  *first);
    }
    ABSL_INTERNAL_CATCH_ANY {
      while (cur != destination) {
        --cur;
        std::allocator_traits<Allocator>::destroy(alloc, std::addressof(*cur));
      }
      ABSL_INTERNAL_RETHROW;
    }
  }
}
}  // namespace memory_internal
```

The Abseil InlinedVector container

In this Section we will get into template metaprogramming, revisit relevant concepts on templates and traits. Finally, we will discuss various trait class implementations which are used in Abseil's InlinedVector container implementation.

InlinedVector main interface

```
// ---------------------------------------------------------------------------
// File: inlined_vector.h
// ---------------------------------------------------------------------------
//
// This header file contains the declaration and definition of an "inlined
// vector" which behaves in an equivalent fashion to a `std::vector`, except
// that storage for small sequences of the vector are provided inline without
// requiring any heap allocation.
//
// An `absl::InlinedVector<T, N>` specifies the default capacity `N` as one of
// its template parameters. Instances where `size() <= N` hold contained
// elements in inline space. Typically `N` is very small so that sequences that
// are expected to be short do not require allocations.
//
// An `absl::InlinedVector` does not usually require a specific allocator. If
// the inlined vector grows beyond its initial constraints, it will need to
// allocate (as any normal `std::vector` would). This is usually performed with
// the default allocator (defined as `std::allocator<T>`). Optionally, a custom
// allocator type may be specified as `A` in `absl::InlinedVector<T, N, A>`.

. . .
#include "absl/algorithm/algorithm.h"
#include "absl/base/internal/throw_delegate.h"
#include "absl/base/optimization.h"
#include "absl/base/port.h"
#include "absl/container/internal/inlined_vector.h"
#include "absl/memory/memory.h"

namespace absl {
ABSL_NAMESPACE_BEGIN
// ---------------------------------------------------------------------------
// InlinedVector
// ---------------------------------------------------------------------------
//
```

```
// An `absl::InlinedVector` is designed to be a drop-in replacement for
// `std::vector` for use cases where the vector's size is sufficiently small
// that it can be inlined. If the inlined vector does grow beyond its estimated
// capacity, it will trigger an initial allocation on the heap, and will behave
// as a `std:vector`. The API of the `absl::InlinedVector` within this file is
// designed to cover the same API footprint as covered by `std::vector`.
template <typename T, size_t N, typename A = std::allocator<T>>
class InlinedVector {
  static_assert(N > 0, "`absl::InlinedVector` requires an inlined capacity.");

  using Storage = inlined_vector_internal::Storage<T, N, A>;

  using AllocatorTraits = typename Storage::AllocatorTraits;
  using RValueReference = typename Storage::RValueReference;
  using MoveIterator = typename Storage::MoveIterator;
  using IsMemcpyOk = typename Storage::IsMemcpyOk;

  template <typename Iterator>
  using IteratorValueAdapter =
      typename Storage::template IteratorValueAdapter<Iterator>;
  using CopyValueAdapter = typename Storage::CopyValueAdapter;
  using DefaultValueAdapter = typename Storage::DefaultValueAdapter;

  template <typename Iterator>
  using EnableIfAtLeastForwardIterator = absl::enable_if_t<
      inlined_vector_internal::IsAtLeastForwardIterator<Iterator>::value>;
  template <typename Iterator>
  using DisableIfAtLeastForwardIterator = absl::enable_if_t<
      !inlined_vector_internal::IsAtLeastForwardIterator<Iterator>::value>;

 public:
  using allocator_type = typename Storage::allocator_type;
  using value_type = typename Storage::value_type;
  using pointer = typename Storage::pointer;
  using const_pointer = typename Storage::const_pointer;
  using size_type = typename Storage::size_type;
  using difference_type = typename Storage::difference_type;
  using reference = typename Storage::reference;
  using const_reference = typename Storage::const_reference;
  using iterator = typename Storage::iterator;
  using const_iterator = typename Storage::const_iterator;
  using reverse_iterator = typename Storage::reverse_iterator;
  using const_reverse_iterator = typename Storage::const_reverse_iterator;

  // --------------------------------------------------------------------------
  // InlinedVector Constructors and Destructor
  // --------------------------------------------------------------------------
  // Creates an empty inlined vector with a value-initialized allocator.
  InlinedVector() noexcept(noexcept(allocator_type())) : storage_() {}

  // Creates an empty inlined vector with a copy of `alloc`.
  explicit InlinedVector(const allocator_type& alloc) noexcept
      : storage_(alloc) {}

  // Creates an inlined vector with `n` copies of `value_type()`.
  explicit InlinedVector(size_type n,
                         const allocator_type& alloc = allocator_type())
      : storage_(alloc) {
    storage_.Initialize(DefaultValueAdapter(), n);
  }

  // Creates an inlined vector with `n` copies of `v`.
  InlinedVector(size_type n, const_reference v,
                const allocator_type& alloc = allocator_type())
      : storage_(alloc) {
    storage_.Initialize(CopyValueAdapter(v), n);
  }

  // Creates an inlined vector with copies of the elements of `list`.
  InlinedVector(std::initializer_list<value_type> list,
                const allocator_type& alloc = allocator_type())
      : InlinedVector(list.begin(), list.end(), alloc) {}

  // Creates an inlined vector with elements constructed from the provided
  // forward iterator range [`first`, `last`).
  //
  // NOTE: the `enable_if` prevents ambiguous interpretation between a call to
  // this constructor with two integral arguments and a call to the above
  // `InlinedVector(size_type, const_reference)` constructor.
  template <typename ForwardIterator,
            EnableIfAtLeastForwardIterator<ForwardIterator>* = nullptr>
  InlinedVector(ForwardIterator first, ForwardIterator last,
                const allocator_type& alloc = allocator_type())
      : storage_(alloc) {
    storage_.Initialize(IteratorValueAdapter<ForwardIterator>(first),
                        std::distance(first, last));
  }

  // Creates an inlined vector with elements constructed from the provided input
  // iterator range [`first`, `last`).
  template <typename InputIterator,
            DisableIfAtLeastForwardIterator<InputIterator>* = nullptr>
  InlinedVector(InputIterator first, InputIterator last,
                const allocator_type& alloc = allocator_type())
      : storage_(alloc) {
```

```
  std::copy(first, last, std::back_inserter(*this));
}

// Creates an inlined vector by copying the contents of `other` using
// `other`'s allocator.
InlinedVector(const InlinedVector& other)
    : InlinedVector(other, *other.storage_.GetAllocPtr()) {}

// Creates an inlined vector by copying the contents of `other` using `alloc`.
InlinedVector(const InlinedVector& other, const allocator_type& alloc)
    : storage_(alloc) {
  if (IsMemcpyOk::value && !other.storage_.GetIsAllocated()) {
    storage_.MemcpyFrom(other.storage_);
  } else {
    storage_.Initialize(IteratorValueAdapter<const_pointer>(other.data()),
                        other.size());
  }
}

// Creates an inlined vector by moving in the contents of `other` without
// allocating. If `other` contains allocated memory, the newly-created inlined
// vector will take ownership of that memory. However, if `other` does not
// contain allocated memory, the newly-created inlined vector will perform
// element-wise move construction of the contents of `other`.
//
// NOTE: since no allocation is performed for the inlined vector in either
// case, the `noexcept(...)` specification depends on whether moving the
// underlying objects can throw. It is assumed assumed that...
//  a) move constructors should only throw due to allocation failure.
//  b) if `value_type`'s move constructor allocates, it uses the same
//     allocation function as the inlined vector's allocator.
// Thus, the move constructor is non-throwing if the allocator is non-throwing
// or `value_type`'s move constructor is specified as `noexcept`.
InlinedVector(InlinedVector&& other) noexcept(
    absl::allocator_is_nothrow<allocator_type>::value ||
    std::is_nothrow_move_constructible<value_type>::value)
    : storage_(*other.storage_.GetAllocPtr()) {
  if (IsMemcpyOk::value) {
    storage_.MemcpyFrom(other.storage_);

    other.storage_.SetInlinedSize(0);
  } else if (other.storage_.GetIsAllocated()) {
    storage_.SetAllocatedData(other.storage_.GetAllocatedData(),
                              other.storage_.GetAllocatedCapacity());
    storage_.SetAllocatedSize(other.storage_.GetSize());

    other.storage_.SetInlinedSize(0);
  } else {
    IteratorValueAdapter<MoveIterator> other_values(
        MoveIterator(other.storage_.GetInlinedData()));

    inlined_vector_internal::ConstructElements(
        storage_.GetAllocPtr(), storage_.GetInlinedData(), &other_values,
        other.storage_.GetSize());

    storage_.SetInlinedSize(other.storage_.GetSize());
  }
}

// Creates an inlined vector by moving in the contents of `other` with a copy
// of `alloc`.
//
// NOTE: if `other`'s allocator is not equal to `alloc`, even if `other`
// contains allocated memory, this move constructor will still allocate. Since
// allocation is performed, this constructor can only be `noexcept` if the
// specified allocator is also `noexcept`.
InlinedVector(InlinedVector&& other, const allocator_type& alloc) noexcept(
    absl::allocator_is_nothrow<allocator_type>::value)
    : storage_(alloc) {
  if (IsMemcpyOk::value) {
    storage_.MemcpyFrom(other.storage_);

    other.storage_.SetInlinedSize(0);
  } else if ((*storage_.GetAllocPtr() == *other.storage_.GetAllocPtr()) &&
             other.storage_.GetIsAllocated()) {
    storage_.SetAllocatedData(other.storage_.GetAllocatedData(),
                              other.storage_.GetAllocatedCapacity());
    storage_.SetAllocatedSize(other.storage_.GetSize());

    other.storage_.SetInlinedSize(0);
  } else {
    storage_.Initialize(
        IteratorValueAdapter<MoveIterator>(MoveIterator(other.data())),
        other.size());
  }
}

~InlinedVector() {}

// ---------------------------------------------------------------------------
// InlinedVector Member Accessors
// ---------------------------------------------------------------------------

// `InlinedVector::empty()`
//
// Returns whether the inlined vector contains no elements.
```

```
bool empty() const noexcept { return !size(); }

// `InlinedVector::size()`
//
// Returns the number of elements in the inlined vector.
size_type size() const noexcept { return storage_.GetSize(); }

// `InlinedVector::max_size()`
//
// Returns the maximum number of elements the inlined vector can hold.
size_type max_size() const noexcept {
  // One bit of the size storage is used to indicate whether the inlined
  // vector contains allocated memory. As a result, the maximum size that the
  // inlined vector can express is half of the max for `size_type`.
  return (std::numeric_limits<size_type>::max)() / 2;
}

// `InlinedVector::capacity()`
//
// Returns the number of elements that could be stored in the inlined vector
// without requiring a reallocation.
//
// NOTE: for most inlined vectors, `capacity()` should be equal to the
// template parameter `N`. For inlined vectors which exceed this capacity,
// they will no longer be inlined and `capacity()` will equal the capactity of
// the allocated memory.
size_type capacity() const noexcept {
  return storage_.GetIsAllocated() ? storage_.GetAllocatedCapacity()
                                   : storage_.GetInlinedCapacity();
}

// `InlinedVector::data()`
//
// Returns a `pointer` to the elements of the inlined vector. This pointer
// can be used to access and modify the contained elements.
//
// NOTE: only elements within [`data()`, `data() + size()`) are valid.
pointer data() noexcept {
  return storage_.GetIsAllocated() ? storage_.GetAllocatedData()
                                   : storage_.GetInlinedData();
}

// Overload of `InlinedVector::data()` that returns a `const_pointer` to the
// elements of the inlined vector. This pointer can be used to access but not
// modify the contained elements.
//
// NOTE: only elements within [`data()`, `data() + size()`) are valid.
const_pointer data() const noexcept {
  return storage_.GetIsAllocated() ? storage_.GetAllocatedData()
                                   : storage_.GetInlinedData();
}

// `InlinedVector::operator[](...)`
//
// Returns a `reference` to the `i`th element of the inlined vector.
reference operator[](size_type i) {
  assert(i < size());

  return data()[i];
}

// Overload of `InlinedVector::operator[](...)` that returns a
// `const_reference` to the `i`th element of the inlined vector.
const_reference operator[](size_type i) const {
  assert(i < size());

  return data()[i];
}

// `InlinedVector::at(...)`
//
// Returns a `reference` to the `i`th element of the inlined vector.
//
// NOTE: if `i` is not within the required range of `InlinedVector::at(...)`,
// in both debug and non-debug builds, `std::out_of_range` will be thrown.
reference at(size_type i) {
  if (ABSL_PREDICT_FALSE(i >= size())) {
    base_internal::ThrowStdOutOfRange(
        "`InlinedVector::at(size_type)` failed bounds check");
  }

  return data()[i];
}

// Overload of `InlinedVector::at(...)` that returns a `const_reference` to
// the `i`th element of the inlined vector.
//
// NOTE: if `i` is not within the required range of `InlinedVector::at(...)`,
// in both debug and non-debug builds, `std::out_of_range` will be thrown.
const_reference at(size_type i) const {
  if (ABSL_PREDICT_FALSE(i >= size())) {
    base_internal::ThrowStdOutOfRange(
        "`InlinedVector::at(size_type) const` failed bounds check");
  }

  return data()[i];
```

```
}

// `InlinedVector::front()`
//
// Returns a `reference` to the first element of the inlined vector.
reference front() {
  assert(!empty());

  return at(0);
}

// Overload of `InlinedVector::front()` that returns a `const_reference` to
// the first element of the inlined vector.
const_reference front() const {
  assert(!empty());

  return at(0);
}

// `InlinedVector::back()`
//
// Returns a `reference` to the last element of the inlined vector.
reference back() {
  assert(!empty());

  return at(size() - 1);
}

// Overload of `InlinedVector::back()` that returns a `const_reference` to the
// last element of the inlined vector.
const_reference back() const {
  assert(!empty());

  return at(size() - 1);
}

// `InlinedVector::begin()`
//
// Returns an `iterator` to the beginning of the inlined vector.
iterator begin() noexcept { return data(); }

// Overload of `InlinedVector::begin()` that returns a `const_iterator` to
// the beginning of the inlined vector.
const_iterator begin() const noexcept { return data(); }

// `InlinedVector::end()`
//
// Returns an `iterator` to the end of the inlined vector.
```

```
iterator end() noexcept { return data() + size(); }

// Overload of `InlinedVector::end()` that returns a `const_iterator` to the
// end of the inlined vector.
const_iterator end() const noexcept { return data() + size(); }

// `InlinedVector::cbegin()`
//
// Returns a `const_iterator` to the beginning of the inlined vector.
const_iterator cbegin() const noexcept { return begin(); }

// `InlinedVector::cend()`
//
// Returns a `const_iterator` to the end of the inlined vector.
const_iterator cend() const noexcept { return end(); }

// `InlinedVector::rbegin()`
//
// Returns a `reverse_iterator` from the end of the inlined vector.
reverse_iterator rbegin() noexcept { return reverse_iterator(end()); }

// Overload of `InlinedVector::rbegin()` that returns a
// `const_reverse_iterator` from the end of the inlined vector.
const_reverse_iterator rbegin() const noexcept {
  return const_reverse_iterator(end());
}

// `InlinedVector::rend()`
//
// Returns a `reverse_iterator` from the beginning of the inlined vector.
reverse_iterator rend() noexcept { return reverse_iterator(begin()); }

// Overload of `InlinedVector::rend()` that returns a `const_reverse_iterator`
// from the beginning of the inlined vector.
const_reverse_iterator rend() const noexcept {
  return const_reverse_iterator(begin());
}

// `InlinedVector::crbegin()`
//
// Returns a `const_reverse_iterator` from the end of the inlined vector.
const_reverse_iterator crbegin() const noexcept { return rbegin(); }

// `InlinedVector::crend()`
//
// Returns a `const_reverse_iterator` from the beginning of the inlined
// vector.
const_reverse_iterator crend() const noexcept { return rend(); }
```

```
// `InlinedVector::get_allocator()`
//
// Returns a copy of the inlined vector's allocator.
allocator_type get_allocator() const { return *storage_.GetAllocPtr(); }

// --------------------------------------------------------------------------
// InlinedVector Member Mutators
// --------------------------------------------------------------------------

// `InlinedVector::operator=(...)`
//
// Replaces the elements of the inlined vector with copies of the elements of
// `list`.
InlinedVector& operator=(std::initializer_list<value_type> list) {
  assign(list.begin(), list.end());

  return *this;
}

// Overload of `InlinedVector::operator=(...)` that replaces the elements of
// the inlined vector with copies of the elements of `other`.
InlinedVector& operator=(const InlinedVector& other) {
  if (ABSL_PREDICT_TRUE(this != std::addressof(other))) {
    const_pointer other_data = other.data();
    assign(other_data, other_data + other.size());
  }

  return *this;
}

// Overload of `InlinedVector::operator=(...)` that moves the elements of
// `other` into the inlined vector.
//
// NOTE: as a result of calling this overload, `other` is left in a valid but
// unspecified state.
InlinedVector& operator=(InlinedVector&& other) {
  if (ABSL_PREDICT_TRUE(this != std::addressof(other))) {
    if (IsMemcpyOk::value || other.storage_.GetIsAllocated()) {
      inlined_vector_internal::DestroyElements(storage_.GetAllocPtr(), data(),
                                               size());
      storage_.DeallocateIfAllocated();
      storage_.MemcpyFrom(other.storage_);

      other.storage_.SetInlinedSize(0);
    } else {
      storage_.Assign(IteratorValueAdapter<MoveIterator>(
                          MoveIterator(other.storage_.GetInlinedData())),
          other.size());
    }
  }

  return *this;
}

// `InlinedVector::assign(...)`
//
// Replaces the contents of the inlined vector with `n` copies of `v`.
void assign(size_type n, const_reference v) {
  storage_.Assign(CopyValueAdapter(v), n);
}

// Overload of `InlinedVector::assign(...)` that replaces the contents of the
// inlined vector with copies of the elements of `list`.
void assign(std::initializer_list<value_type> list) {
  assign(list.begin(), list.end());
}

// Overload of `InlinedVector::assign(...)` to replace the contents of the
// inlined vector with the range [`first`, `last`).
//
// NOTE: this overload is for iterators that are "forward" category or better.
template <typename ForwardIterator,
        EnableIfAtLeastForwardIterator<ForwardIterator>* = nullptr>
void assign(ForwardIterator first, ForwardIterator last) {
  storage_.Assign(IteratorValueAdapter<ForwardIterator>(first),
          std::distance(first, last));
}

// Overload of `InlinedVector::assign(...)` to replace the contents of the
// inlined vector with the range [`first`, `last`).
//
// NOTE: this overload is for iterators that are "input" category.
template <typename InputIterator,
        DisableIfAtLeastForwardIterator<InputIterator>* = nullptr>
void assign(InputIterator first, InputIterator last) {
  size_type i = 0;
  for (; i < size() && first != last; ++i, static_cast<void>(++first)) {
    at(i) = *first;
  }

  erase(data() + i, data() + size());
  std::copy(first, last, std::back_inserter(*this));
}

// `InlinedVector::resize(...)`
```

```
//
// Resizes the inlined vector to contain `n` elements.
//
// NOTE: if `n` is smaller than `size()`, extra elements are destroyed. If `n`
// is larger than `size()`, new elements are value-initialized.
void resize(size_type n) { storage_.Resize(DefaultValueAdapter(), n); }

// Overload of `InlinedVector::resize(...)` that resizes the inlined vector to
// contain `n` elements.
//
// NOTE: if `n` is smaller than `size()`, extra elements are destroyed. If `n`
// is larger than `size()`, new elements are copied-constructed from `v`.
void resize(size_type n, const_reference v) {
  storage_.Resize(CopyValueAdapter(v), n);
}

// `InlinedVector::insert(...)`
//
// Inserts a copy of `v` at `pos`, returning an `iterator` to the newly
// inserted element.
iterator insert(const_iterator pos, const_reference v) {
  return emplace(pos, v);
}

// Overload of `InlinedVector::insert(...)` that inserts `v` at `pos` using
// move semantics, returning an `iterator` to the newly inserted element.
iterator insert(const_iterator pos, RValueReference v) {
  return emplace(pos, std::move(v));
}

// Overload of `InlinedVector::insert(...)` that inserts `n` contiguous copies
// of `v` starting at `pos`, returning an `iterator` pointing to the first of
// the newly inserted elements.
iterator insert(const_iterator pos, size_type n, const_reference v) {
  assert(pos >= begin());
  assert(pos <= end());

  if (ABSL_PREDICT_TRUE(n != 0)) {
    value_type dealias = v;
    return storage_.Insert(pos, CopyValueAdapter(dealias), n);
  } else {
    return const_cast<iterator>(pos);
  }
}

// Overload of `InlinedVector::insert(...)` that inserts copies of the
// elements of `list` starting at `pos`, returning an `iterator` pointing to
// the first of the newly inserted elements.
```

```
iterator insert(const_iterator pos, std::initializer_list<value_type> list) {
  return insert(pos, list.begin(), list.end());
}

// Overload of `InlinedVector::insert(...)` that inserts the range [`first`,
// `last`) starting at `pos`, returning an `iterator` pointing to the first
// of the newly inserted elements.
//
// NOTE: this overload is for iterators that are "forward" category or better.
template <typename ForwardIterator,
          EnableIfAtLeastForwardIterator<ForwardIterator>* = nullptr>
iterator insert(const_iterator pos, ForwardIterator first,
                ForwardIterator last) {
  assert(pos >= begin());
  assert(pos <= end());

  if (ABSL_PREDICT_TRUE(first != last)) {
    return storage_.Insert(pos, IteratorValueAdapter<ForwardIterator>(first),
                           std::distance(first, last));
  } else {
    return const_cast<iterator>(pos);
  }
}

// Overload of `InlinedVector::insert(...)` that inserts the range [`first`,
// `last`) starting at `pos`, returning an `iterator` pointing to the first
// of the newly inserted elements.
//
// NOTE: this overload is for iterators that are "input" category.
template <typename InputIterator,
          DisableIfAtLeastForwardIterator<InputIterator>* = nullptr>
iterator insert(const_iterator pos, InputIterator first, InputIterator last) {
  assert(pos >= begin());
  assert(pos <= end());

  size_type index = std::distance(cbegin(), pos);
  for (size_type i = index; first != last; ++i, static_cast<void>(++first)) {
    insert(data() + i, *first);
  }

  return iterator(data() + index);
}

// `InlinedVector::emplace(...)`
//
// Constructs and inserts an element using `args...` in the inlined vector at
// `pos`, returning an `iterator` pointing to the newly emplaced element.
template <typename... Args>
```

```
iterator emplace(const_iterator pos, Args&&... args) {
  assert(pos >= begin());
  assert(pos <= end());

  value_type dealias(std::forward<Args>(args)...);
  return storage_.Insert(pos,
                       IteratorValueAdapter<MoveIterator>(
                           MoveIterator(std::addressof(dealias))),
                       1);
}

// `InlinedVector::emplace_back(...)`
//
// Constructs and inserts an element using `args...` in the inlined vector at
// `end()`, returning a `reference` to the newly emplaced element.
template <typename... Args>
reference emplace_back(Args&&... args) {
  return storage_.EmplaceBack(std::forward<Args>(args)...);
}

// `InlinedVector::push_back(...)`
//
// Inserts a copy of `v` in the inlined vector at `end()`.
void push_back(const_reference v) { static_cast<void>(emplace_back(v)); }

// Overload of `InlinedVector::push_back(...)` for inserting `v` at `end()`
// using move semantics.
void push_back(RValueReference v) {
  static_cast<void>(emplace_back(std::move(v)));
}

// `InlinedVector::pop_back()`
//
// Destroys the element at `back()`, reducing the size by `1`.
void pop_back() noexcept {
  assert(!empty());

  AllocatorTraits::destroy(*storage_.GetAllocPtr(), data() + (size() - 1));
  storage_.SubtractSize(1);
}

// `InlinedVector::erase(...)`
//
// Erases the element at `pos`, returning an `iterator` pointing to where the
// erased element was located.
//
// NOTE: may return `end()`, which is not dereferencable.
iterator erase(const_iterator pos) {
  assert(pos >= begin());
  assert(pos < end());

  return storage_.Erase(pos, pos + 1);
}

// Overload of `InlinedVector::erase(...)` that erases every element in the
// range [`from`, `to`), returning an `iterator` pointing to where the first
// erased element was located.
//
// NOTE: may return `end()`, which is not dereferencable.
iterator erase(const_iterator from, const_iterator to) {
  assert(from >= begin());
  assert(from <= to);
  assert(to <= end());

  if (ABSL_PREDICT_TRUE(from != to)) {
    return storage_.Erase(from, to);
  } else {
    return const_cast<iterator>(from);
  }
}

// `InlinedVector::clear()`
//
// Destroys all elements in the inlined vector, setting the size to `0` and
// deallocating any held memory.
void clear() noexcept {
  inlined_vector_internal::DestroyElements(storage_.GetAllocPtr(), data(),
                                           size());
  storage_.DeallocateIfAllocated();

  storage_.SetInlinedSize(0);
}

// `InlinedVector::reserve(...)`
//
// Ensures that there is enough room for at least `n` elements.
void reserve(size_type n) { storage_.Reserve(n); }

// `InlinedVector::shrink_to_fit()`
//
// Reduces memory usage by freeing unused memory. After being called, calls to
// `capacity()` will be equal to `max(N, size())`.
//
// If `size() <= N` and the inlined vector contains allocated memory, the
// elements will all be moved to the inlined space and the allocated memory
// will be deallocated.
```

```
  //
  // If `size() > N` and `size() < capacity()`, the elements will be moved to a
  // smaller allocation.
  void shrink_to_fit() {
    if (storage_.GetIsAllocated()) {
      storage_.ShrinkToFit();
    }
  }

  // `InlinedVector::swap(...)`
  //
  // Swaps the contents of the inlined vector with `other`.
  void swap(InlinedVector& other) {
    if (ABSL_PREDICT_TRUE(this != std::addressof(other))) {
      storage_.Swap(std::addressof(other.storage_));
    }
  }

 private:
  template <typename H, typename TheT, size_t TheN, typename TheA>
  friend H AbslHashValue(H h, const absl::InlinedVector<TheT, TheN, TheA>& a);

  Storage storage_;
};

// --------------------------------------------------------------------------
// InlinedVector Non-Member Functions
// --------------------------------------------------------------------------

// `swap(...)`
//
// Swaps the contents of two inlined vectors.
template <typename T, size_t N, typename A>
void swap(absl::InlinedVector<T, N, A>& a,
          absl::InlinedVector<T, N, A>& b) noexcept(noexcept(a.swap(b))) {
  a.swap(b);
}

// `operator==(...)`
//
// Tests for value-equality of two inlined vectors.
template <typename T, size_t N, typename A>
bool operator==(const absl::InlinedVector<T, N, A>& a,
                const absl::InlinedVector<T, N, A>& b) {
  auto a_data = a.data();
  auto b_data = b.data();
  return absl::equal(a_data, a_data + a.size(), b_data, b_data + b.size());
}
```

```
// `operator!=(...)`
//
// Tests for value-inequality of two inlined vectors.
template <typename T, size_t N, typename A>
bool operator!=(const absl::InlinedVector<T, N, A>& a,
                const absl::InlinedVector<T, N, A>& b) {
  return !(a == b);
}

// `operator<(...)`
//
// Tests whether the value of an inlined vector is less than the value of
// another inlined vector using a lexicographical comparison algorithm.
template <typename T, size_t N, typename A>
bool operator<(const absl::InlinedVector<T, N, A>& a,
               const absl::InlinedVector<T, N, A>& b) {
  auto a_data = a.data();
  auto b_data = b.data();
  return std::lexicographical_compare(a_data, a_data + a.size(), b_data,
                                      b_data + b.size());
}

// `operator>(...)`
//
// Tests whether the value of an inlined vector is greater than the value of
// another inlined vector using a lexicographical comparison algorithm.
template <typename T, size_t N, typename A>
bool operator>(const absl::InlinedVector<T, N, A>& a,
               const absl::InlinedVector<T, N, A>& b) {
  return b < a;
}

// `operator<=(...)`
//
// Tests whether the value of an inlined vector is less than or equal to the
// value of another inlined vector using a lexicographical comparison algorithm.
template <typename T, size_t N, typename A>
bool operator<=(const absl::InlinedVector<T, N, A>& a,
                const absl::InlinedVector<T, N, A>& b) {
  return !(b < a);
}

// `operator>=(...)`
//
// Tests whether the value of an inlined vector is greater than or equal to the
// value of another inlined vector using a lexicographical comparison algorithm.
template <typename T, size_t N, typename A>
```

```
bool operator>=(const absl::InlinedVector<T, N, A>& a,
                const absl::InlinedVector<T, N, A>& b) {
  return !(a < b);
}

// `AbslHashValue(...)`
//
// Provides `absl::Hash` support for `absl::InlinedVector`. It is uncommon to
// call this directly.
template <typename H, typename T, size_t N, typename A>
H AbslHashValue(H h, const absl::InlinedVector<T, N, A>& a) {
  auto size = a.size();
  return H::combine(H::combine_contiguous(std::move(h), a.data(), size), size);
}

ABSL_NAMESPACE_END
}  // namespace absl
```

The internal implementation details of InlinedVector

Here some bits and pieces from absl/container/internal/inlined_vector.h :

The helper trait class IsAtLeastForwardIterator is defined as:

```
template <typename Iterator>
using IsAtLeastForwardIterator = std::is_convertible<
    typename std::iterator_traits<Iterator>::iterator_category,
    std::forward_iterator_tag>;
```

and evaluates to true if the supplied type Iterator is a ForwardIterator.

The trait class isMemcpyOK checks at compile time the following:

- The specified type AllocatorType is std::allocator<ValueType>. Note that if AllocatorType specializes std::allocator<ValueType> this is not satisfied.
- There exists a trivial copy constructor for the specified type ValueType
- There exists a trivial copy assignment operator for the specified type ValueType
- There exists a trivial destructor for the specified ValueType

If all of the above are satisfied then isMemcpyOK evaluates to true; otherwise it evaluates to false.
 Accordingly, isMemcpyOK is defined as:

```
template <typename AllocatorType,
          typename ValueType =
              typename absl::allocator_traits<AllocatorType>::value_type>
using IsMemcpyOk =
    absl::conjunction<std::is_same<AllocatorType, std::allocator<ValueType>>,
   absl::is_trivially_copy_constructible<ValueType>,
   absl::is_trivially_copy_assignable<ValueType>,
   absl::is_trivially_destructible<ValueType>>;
```

Here is the source of absl/container/internal/inlined_vector.h

```
namespace absl {
ABSL_NAMESPACE_BEGIN
namespace inlined_vector_internal {

template <typename Iterator>
using IsAtLeastForwardIterator = std::is_convertible<
    typename std::iterator_traits<Iterator>::iterator_category,
    std::forward_iterator_tag>;

template <typename AllocatorType,
          typename ValueType =
              typename absl::allocator_traits<AllocatorType>::value_type>
using IsMemcpyOk =
    absl::conjunction<std::is_same<AllocatorType, std::allocator<ValueType>>,
                      absl::is_trivially_copy_constructible<ValueType>,
                      absl::is_trivially_copy_assignable<ValueType>,
                      absl::is_trivially_destructible<ValueType>>;

template <typename AllocatorType, typename Pointer, typename SizeType>
void DestroyElements(AllocatorType* alloc_ptr, Pointer destroy_first,
                     SizeType destroy_size) {
  using AllocatorTraits = absl::allocator_traits<AllocatorType>;

  if (destroy_first != nullptr) {
    for (auto i = destroy_size; i != 0;) {
      --i;
      AllocatorTraits::destroy(*alloc_ptr, destroy_first + i);
    }

#if !defined(NDEBUG)
    {
      using ValueType = typename AllocatorTraits::value_type;

      // Overwrite unused memory with `0xab` so we can catch uninitialized
      // usage.
      //
      // Cast to `void*` to tell the compiler that we don't care that we might
      // be scribbling on a vtable pointer.
      void* memory_ptr = destroy_first;
      auto memory_size = destroy_size * sizeof(ValueType);
      std::memset(memory_ptr, 0xab, memory_size);
    }
#endif  // !defined(NDEBUG)
  }
}

template <typename AllocatorType, typename Pointer, typename ValueAdapter,
          typename SizeType>
```

```
void ConstructElements(AllocatorType* alloc_ptr, Pointer construct_first,
                       ValueAdapter* values_ptr, SizeType construct_size) {
  for (SizeType i = 0; i < construct_size; ++i) {
    ABSL_INTERNAL_TRY {
      values_ptr->ConstructNext(alloc_ptr, construct_first + i);
    }
    ABSL_INTERNAL_CATCH_ANY {
      inlined_vector_internal::DestroyElements(alloc_ptr, construct_first, i);
      ABSL_INTERNAL_RETHROW;
    }
  }
}

template <typename Pointer, typename ValueAdapter, typename SizeType>
void AssignElements(Pointer assign_first, ValueAdapter* values_ptr,
                    SizeType assign_size) {
  for (SizeType i = 0; i < assign_size; ++i) {
    values_ptr->AssignNext(assign_first + i);
  }
}

template <typename AllocatorType>
struct StorageView {
  using AllocatorTraits = absl::allocator_traits<AllocatorType>;
  using Pointer = typename AllocatorTraits::pointer;
  using SizeType = typename AllocatorTraits::size_type;

  Pointer data;
  SizeType size;
  SizeType capacity;
};

template <typename AllocatorType, typename Iterator>
class IteratorValueAdapter {
  using AllocatorTraits = absl::allocator_traits<AllocatorType>;
  using Pointer = typename AllocatorTraits::pointer;

 public:
  explicit IteratorValueAdapter(const Iterator& it) : it_(it) {}

  void ConstructNext(AllocatorType* alloc_ptr, Pointer construct_at) {
    AllocatorTraits::construct(*alloc_ptr, construct_at, *it_);
    ++it_;
  }

  void AssignNext(Pointer assign_at) {
    *assign_at = *it_;
    ++it_;
  }

 private:
  Iterator it_;
};

template <typename AllocatorType>
class CopyValueAdapter {
  using AllocatorTraits = absl::allocator_traits<AllocatorType>;
  using ValueType = typename AllocatorTraits::value_type;
  using Pointer = typename AllocatorTraits::pointer;
  using ConstPointer = typename AllocatorTraits::const_pointer;

 public:
  explicit CopyValueAdapter(const ValueType& v) : ptr_(std::addressof(v)) {}

  void ConstructNext(AllocatorType* alloc_ptr, Pointer construct_at) {
    AllocatorTraits::construct(*alloc_ptr, construct_at, *ptr_);
  }

  void AssignNext(Pointer assign_at) { *assign_at = *ptr_; }

 private:
  ConstPointer ptr_;
};

template <typename AllocatorType>
class DefaultValueAdapter {
  using AllocatorTraits = absl::allocator_traits<AllocatorType>;
  using ValueType = typename AllocatorTraits::value_type;
  using Pointer = typename AllocatorTraits::pointer;

 public:
  explicit DefaultValueAdapter() {}

  void ConstructNext(AllocatorType* alloc_ptr, Pointer construct_at) {
    AllocatorTraits::construct(*alloc_ptr, construct_at);
  }

  void AssignNext(Pointer assign_at) { *assign_at = ValueType(); }
};

template <typename AllocatorType>
class AllocationTransaction {
  using AllocatorTraits = absl::allocator_traits<AllocatorType>;
  using Pointer = typename AllocatorTraits::pointer;
  using SizeType = typename AllocatorTraits::size_type;
 public:
  explicit AllocationTransaction(AllocatorType* alloc_ptr)
      : alloc_data_(*alloc_ptr, nullptr) {}

  ~AllocationTransaction() {
    if (DidAllocate()) {
      AllocatorTraits::deallocate(GetAllocator(), GetData(), GetCapacity());
    }
  }

  AllocationTransaction(const AllocationTransaction&) = delete;
  void operator=(const AllocationTransaction&) = delete;

  AllocatorType& GetAllocator() { return alloc_data_.template get<0>(); }
  Pointer& GetData() { return alloc_data_.template get<1>(); }
  SizeType& GetCapacity() { return capacity_; }

  bool DidAllocate() { return GetData() != nullptr; }
  Pointer Allocate(SizeType capacity) {
    GetData() = AllocatorTraits::allocate(GetAllocator(), capacity);
    GetCapacity() = capacity;
    return GetData();
  }

  void Reset() {
    GetData() = nullptr;
    GetCapacity() = 0;
  }

 private:
  container_internal::CompressedTuple<AllocatorType, Pointer> alloc_data_;
  SizeType capacity_ = 0;
};

template <typename AllocatorType>
class ConstructionTransaction {
  using AllocatorTraits = absl::allocator_traits<AllocatorType>;
  using Pointer = typename AllocatorTraits::pointer;
  using SizeType = typename AllocatorTraits::size_type;

 public:
  explicit ConstructionTransaction(AllocatorType* alloc_ptr)
      : alloc_data_(*alloc_ptr, nullptr) {}

  ~ConstructionTransaction() {
    if (DidConstruct()) {
      inlined_vector_internal::DestroyElements(std::addressof(GetAllocator()),
                                               GetData(), GetSize());
    }
  }

  ConstructionTransaction(const ConstructionTransaction&) = delete;
  void operator=(const ConstructionTransaction&) = delete;

  AllocatorType& GetAllocator() { return alloc_data_.template get<0>(); }
  Pointer& GetData() { return alloc_data_.template get<1>(); }
  SizeType& GetSize() { return size_; }

  bool DidConstruct() { return GetData() != nullptr; }
  template <typename ValueAdapter>
  void Construct(Pointer data, ValueAdapter* values_ptr, SizeType size) {
    inlined_vector_internal::ConstructElements(std::addressof(GetAllocator()),
                                               data, values_ptr, size);
    GetData() = data;
    GetSize() = size;
  }
  void Commit() {
    GetData() = nullptr;
    GetSize() = 0;
  }

 private:
  container_internal::CompressedTuple<AllocatorType, Pointer> alloc_data_;
  SizeType size_ = 0;
};

template <typename T, size_t N, typename A>
class Storage {
 public:
  using AllocatorTraits = absl::allocator_traits<A>;
  using allocator_type = typename AllocatorTraits::allocator_type;
  using value_type = typename AllocatorTraits::value_type;
  using pointer = typename AllocatorTraits::pointer;
  using const_pointer = typename AllocatorTraits::const_pointer;
  using size_type = typename AllocatorTraits::size_type;
  using difference_type = typename AllocatorTraits::difference_type;

  using reference = value_type&;
  using const_reference = const value_type&;
  using RValueReference = value_type&&;
  using iterator = pointer;
  using const_iterator = const_pointer;
  using reverse_iterator = std::reverse_iterator<iterator>;
  using const_reverse_iterator = std::reverse_iterator<const_iterator>;
  using MoveIterator = std::move_iterator<iterator>;
  using IsMemcpyOk = inlined_vector_internal::IsMemcpyOk<allocator_type>;
  using StorageView = inlined_vector_internal::StorageView<allocator_type>;

  template <typename Iterator>
  using IteratorValueAdapter =
     inlined_vector_internal::IteratorValueAdapter<allocator_type, Iterator>;
  using CopyValueAdapter =
     inlined_vector_internal::CopyValueAdapter<allocator_type>;
  using DefaultValueAdapter =
     inlined_vector_internal::DefaultValueAdapter<allocator_type>;

  using AllocationTransaction =
     inlined_vector_internal::AllocationTransaction<allocator_type>;
  using ConstructionTransaction =
     inlined_vector_internal::ConstructionTransaction<allocator_type>;

  static size_type NextCapacity(size_type current_capacity) {
    return current_capacity * 2;
  }

  static size_type ComputeCapacity(size_type current_capacity,
                                   size_type requested_capacity) {
    return (std::max)(NextCapacity(current_capacity), requested_capacity);
  }

  // --------------------------------------------------------------------------
  // Storage Constructors and Destructor
  // --------------------------------------------------------------------------

  Storage() : metadata_() {}

  explicit Storage(const allocator_type& alloc) : metadata_(alloc, {}) {}

  ~Storage() {
    pointer data = GetIsAllocated() ? GetAllocatedData() : GetInlinedData();
    inlined_vector_internal::DestroyElements(GetAllocPtr(), data, GetSize());
    DeallocateIfAllocated();
  }

  // --------------------------------------------------------------------------
  // Storage Member Accessors
  // --------------------------------------------------------------------------

  size_type& GetSizeAndIsAllocated() { return metadata_.template get<1>(); }

  const size_type& GetSizeAndIsAllocated() const {
    return metadata_.template get<1>();
  }
```

```
size_type GetSize() const { return GetSizeAndIsAllocated() >> 1; }

bool GetIsAllocated() const { return GetSizeAndIsAllocated() & 1; }

pointer GetAllocatedData() { return data_.allocated.allocated_data; }

const_pointer GetAllocatedData() const {
  return data_.allocated.allocated_data;
}

pointer GetInlinedData() {
  return reinterpret_cast<pointer>(
      std::addressof(data_.inlined.inlined_data[0]));
}

const_pointer GetInlinedData() const {
  return reinterpret_cast<const_pointer>(
      std::addressof(data_.inlined.inlined_data[0]));
}

size_type GetAllocatedCapacity() const {
  return data_.allocated.allocated_capacity;
}

size_type GetInlinedCapacity() const { return static_cast<size_type>(N); }

StorageView MakeStorageView() {
  return GetIsAllocated()
             ? StorageView{GetAllocatedData(), GetSize(),
                           GetAllocatedCapacity()}
             : StorageView{GetInlinedData(), GetSize(), GetInlinedCapacity()};
}

allocator_type* GetAllocPtr() {
  return std::addressof(metadata_.template get<0>());
}

const allocator_type* GetAllocPtr() const {
  return std::addressof(metadata_.template get<0>());
}

// --------------------------------------------------------------------------
// Storage Member Mutators
// --------------------------------------------------------------------------

template <typename ValueAdapter>
void Initialize(ValueAdapter values, size_type new_size);
```

```
template <typename ValueAdapter>
void Assign(ValueAdapter values, size_type new_size);

template <typename ValueAdapter>
void Resize(ValueAdapter values, size_type new_size);

template <typename ValueAdapter>
iterator Insert(const_iterator pos, ValueAdapter values,
               size_type insert_count);

template <typename... Args>
reference EmplaceBack(Args&&... args);

iterator Erase(const_iterator from, const_iterator to);

void Reserve(size_type requested_capacity);

void ShrinkToFit();

void Swap(Storage* other_storage_ptr);

void SetIsAllocated() {
  GetSizeAndIsAllocated() |= static_cast<size_type>(1);
}

void UnsetIsAllocated() {
  GetSizeAndIsAllocated() &= ((std::numeric_limits<size_type>::max)() - 1);
}

void SetSize(size_type size) {
  GetSizeAndIsAllocated() =
      (size << 1) | static_cast<size_type>(GetIsAllocated());
}

void SetAllocatedSize(size_type size) {
  GetSizeAndIsAllocated() = (size << 1) | static_cast<size_type>(1);
}

void SetInlinedSize(size_type size) {
  GetSizeAndIsAllocated() = size << static_cast<size_type>(1);
}

void AddSize(size_type count) {
  GetSizeAndIsAllocated() += count << static_cast<size_type>(1);
}

void SubtractSize(size_type count) {
    assert(count <= GetSize());

    GetSizeAndIsAllocated() -= count << static_cast<size_type>(1);
  }

  void SetAllocatedData(pointer data, size_type capacity) {
    data_.allocated.allocated_data = data;
    data_.allocated.allocated_capacity = capacity;
  }

  void AcquireAllocatedData(AllocationTransaction* allocation_tx_ptr) {
    SetAllocatedData(allocation_tx_ptr->GetData(),
                     allocation_tx_ptr->GetCapacity());

    allocation_tx_ptr->Reset();
  }

  void MemcpyFrom(const Storage& other_storage) {
    assert(IsMemcpyOk::value || other_storage.GetIsAllocated());

    GetSizeAndIsAllocated() = other_storage.GetSizeAndIsAllocated();
    data_ = other_storage.data_;
  }

  void DeallocateIfAllocated() {
    if (GetIsAllocated()) {
      AllocatorTraits::deallocate(*GetAllocPtr(), GetAllocatedData(),
                                  GetAllocatedCapacity());
    }
  }

 private:
  using Metadata =
      container_internal::CompressedTuple<allocator_type, size_type>;

  struct Allocated {
    pointer allocated_data;
    size_type allocated_capacity;
  };

  struct Inlined {
    alignas(value_type) char inlined_data[sizeof(value_type[N])];
  };

  union Data {
    Allocated allocated;
    Inlined inlined;
  };
  Metadata metadata_;
  Data data_;
};

template <typename T, size_t N, typename A>
template <typename ValueAdapter>
auto Storage<T, N, A>::Initialize(ValueAdapter values, size_type new_size)
    -> void {
  // Only callable from constructors!
  assert(!GetIsAllocated());
  assert(GetSize() == 0);

  pointer construct_data;
  if (new_size > GetInlinedCapacity()) {
    // Because this is only called from the `InlinedVector` constructors, it's
    // safe to take on the allocation with size `0`. If `ConstructElements(...)`
    // throws, deallocation will be automatically handled by `~Storage()`.
    size_type new_capacity = ComputeCapacity(GetInlinedCapacity(), new_size);
    construct_data = AllocatorTraits::allocate(*GetAllocPtr(), new_capacity);
    SetAllocatedData(construct_data, new_capacity);
    SetIsAllocated();
  } else {
    construct_data = GetInlinedData();
  }

  inlined_vector_internal::ConstructElements(GetAllocPtr(), construct_data,
                                             &values, new_size);

  // Since the initial size was guaranteed to be `0` and the allocated bit is
  // already correct for either case, *adding* `new_size` gives us the correct
  // result faster than setting it directly.
  AddSize(new_size);
}

template <typename T, size_t N, typename A>
template <typename ValueAdapter>
auto Storage<T, N, A>::Assign(ValueAdapter values, size_type new_size) -> void {
  StorageView storage_view = MakeStorageView();

  AllocationTransaction allocation_tx(GetAllocPtr());

  absl::Span<value_type> assign_loop;
  absl::Span<value_type> construct_loop;
  absl::Span<value_type> destroy_loop;

  if (new_size > storage_view.capacity) {
    size_type new_capacity = ComputeCapacity(storage_view.capacity, new_size);
    construct_loop = {allocation_tx.Allocate(new_capacity), new_size};
    destroy_loop = {storage_view.data, storage_view.size};
  } else if (new_size > storage_view.size) {
    assign_loop = {storage_view.data, storage_view.size};
    construct_loop = {storage_view.data + storage_view.size,
                      new_size - storage_view.size};
  } else {
    assign_loop = {storage_view.data, new_size};
    destroy_loop = {storage_view.data + new_size, storage_view.size - new_size};
  }

  inlined_vector_internal::AssignElements(assign_loop.data(), &values,
                                          assign_loop.size());

  inlined_vector_internal::ConstructElements(
      GetAllocPtr(), construct_loop.data(), &values, construct_loop.size());

  inlined_vector_internal::DestroyElements(GetAllocPtr(), destroy_loop.data(),
                                           destroy_loop.size());

  if (allocation_tx.DidAllocate()) {
    DeallocateIfAllocated();
    AcquireAllocatedData(&allocation_tx);
    SetIsAllocated();
  }

  SetSize(new_size);
}

template <typename T, size_t N, typename A>
template <typename ValueAdapter>
auto Storage<T, N, A>::Resize(ValueAdapter values, size_type new_size) -> void {
  StorageView storage_view = MakeStorageView();

  IteratorValueAdapter<MoveIterator> move_values(
      MoveIterator(storage_view.data));

  AllocationTransaction allocation_tx(GetAllocPtr());
  ConstructionTransaction construction_tx(GetAllocPtr());

  absl::Span<value_type> construct_loop;
  absl::Span<value_type> move_construct_loop;
  absl::Span<value_type> destroy_loop;

  if (new_size > storage_view.capacity) {
    size_type new_capacity = ComputeCapacity(storage_view.capacity, new_size);
    pointer new_data = allocation_tx.Allocate(new_capacity);
    construct_loop = {new_data + storage_view.size,
                  new_size - storage_view.size};
    move_construct_loop = {new_data, storage_view.size};
    destroy_loop = {storage_view.data, storage_view.size};
  } else if (new_size > storage_view.size) {
    construct_loop = {storage_view.data + storage_view.size,
                  new_size - storage_view.size};
  } else {
    destroy_loop = {storage_view.data + new_size, storage_view.size - new_size};
  }

  construction_tx.Construct(construct_loop.data(), &values,
                        construct_loop.size());

  inlined_vector_internal::ConstructElements(
      GetAllocPtr(), move_construct_loop.data(), &move_values,
      move_construct_loop.size());

  inlined_vector_internal::DestroyElements(GetAllocPtr(), destroy_loop.data(),
                                       destroy_loop.size());

  construction_tx.Commit();
  if (allocation_tx.DidAllocate()) {
    DeallocateIfAllocated();
    AcquireAllocatedData(&allocation_tx);
    SetIsAllocated();
  }

  SetSize(new_size);
}

template <typename T, size_t N, typename A>
template <typename ValueAdapter>
auto Storage<T, N, A>::Insert(const_iterator pos, ValueAdapter values,
                          size_type insert_count) -> iterator {
  StorageView storage_view = MakeStorageView();

  size_type insert_index =
      std::distance(const_iterator(storage_view.data), pos);
  size_type insert_end_index = insert_index + insert_count;
  size_type new_size = storage_view.size + insert_count;

  if (new_size > storage_view.capacity) {
    AllocationTransaction allocation_tx(GetAllocPtr());
    ConstructionTransaction construction_tx(GetAllocPtr());
    ConstructionTransaction move_construciton_tx(GetAllocPtr());

    IteratorValueAdapter<MoveIterator> move_values(
        MoveIterator(storage_view.data));
    size_type new_capacity = ComputeCapacity(storage_view.capacity, new_size);
    pointer new_data = allocation_tx.Allocate(new_capacity);

    construction_tx.Construct(new_data + insert_index, &values, insert_count);

    move_construciton_tx.Construct(new_data, &move_values, insert_index);

    inlined_vector_internal::ConstructElements(
        GetAllocPtr(), new_data + insert_end_index, &move_values,
        storage_view.size - insert_index);

    inlined_vector_internal::DestroyElements(GetAllocPtr(), storage_view.data,
                                             storage_view.size);

    construction_tx.Commit();
    move_construciton_tx.Commit();
    DeallocateIfAllocated();
    AcquireAllocatedData(&allocation_tx);

    SetAllocatedSize(new_size);
    return iterator(new_data + insert_index);
  } else {
    size_type move_construction_destination_index =
        (std::max)(insert_end_index, storage_view.size);

    ConstructionTransaction move_construction_tx(GetAllocPtr());

    IteratorValueAdapter<MoveIterator> move_construction_values(
        MoveIterator(storage_view.data +
                     (move_construction_destination_index - insert_count)));
    absl::Span<value_type> move_construction = {
        storage_view.data + move_construction_destination_index,
        new_size - move_construction_destination_index};

    pointer move_assignment_values = storage_view.data + insert_index;
    absl::Span<value_type> move_assignment = {
        storage_view.data + insert_end_index,
        move_construction_destination_index - insert_end_index};

    absl::Span<value_type> insert_assignment = {move_assignment_values,
                                                move_construction.size()};

    absl::Span<value_type> insert_construction = {
        insert_assignment.data() + insert_assignment.size(),
        insert_count - insert_assignment.size()};

    move_construction_tx.Construct(move_construction.data(),
                              &move_construction_values,
                              move_construction.size());

    for (pointer destination = move_assignment.data() + move_assignment.size(),
                 last_destination = move_assignment.data(),
                 source = move_assignment_values + move_assignment.size();
         ;) {
      --destination;
      --source;
      if (destination < last_destination) break;
      *destination = std::move(*source);
    }

    inlined_vector_internal::AssignElements(insert_assignment.data(), &values,
                                            insert_assignment.size());

    inlined_vector_internal::ConstructElements(
        GetAllocPtr(), insert_construction.data(), &values,
        insert_construction.size());

    move_construction_tx.Commit();

    AddSize(insert_count);
    return iterator(storage_view.data + insert_index);
  }
}

template <typename T, size_t N, typename A>
template <typename... Args>
auto Storage<T, N, A>::EmplaceBack(Args&&... args) -> reference {
  StorageView storage_view = MakeStorageView();

  AllocationTransaction allocation_tx(GetAllocPtr());

  IteratorValueAdapter<MoveIterator> move_values(
      MoveIterator(storage_view.data));

  pointer construct_data;
  if (storage_view.size == storage_view.capacity) {
    size_type new_capacity = NextCapacity(storage_view.capacity);
    construct_data = allocation_tx.Allocate(new_capacity);
  } else {
    construct_data = storage_view.data;
  }

  pointer last_ptr = construct_data + storage_view.size;

  AllocatorTraits::construct(*GetAllocPtr(), last_ptr,
                      std::forward<Args>(args)...);

  if (allocation_tx.DidAllocate()) {
    ABSL_INTERNAL_TRY {
      inlined_vector_internal::ConstructElements(
          GetAllocPtr(), allocation_tx.GetData(), &move_values,
          storage_view.size);
    }
    ABSL_INTERNAL_CATCH_ANY {
      AllocatorTraits::destroy(*GetAllocPtr(), last_ptr);
      ABSL_INTERNAL_RETHROW;
    }

    inlined_vector_internal::DestroyElements(GetAllocPtr(), storage_view.data,
                                             storage_view.size);

    DeallocateIfAllocated();
    AcquireAllocatedData(&allocation_tx);
    SetIsAllocated();
  }

  AddSize(1);
  return *last_ptr;
}

template <typename T, size_t N, typename A>
auto Storage<T, N, A>::Erase(const_iterator from, const_iterator to)
    -> iterator {
  StorageView storage_view = MakeStorageView();

  size_type erase_size = std::distance(from, to);
  size_type erase_index =
      std::distance(const_iterator(storage_view.data), from);
  size_type erase_end_index = erase_index + erase_size;

  IteratorValueAdapter<MoveIterator> move_values(
      MoveIterator(storage_view.data + erase_end_index));

  inlined_vector_internal::AssignElements(storage_view.data + erase_index,
                                          &move_values,
                                          storage_view.size - erase_end_index);

  inlined_vector_internal::DestroyElements(
      GetAllocPtr(), storage_view.data + (storage_view.size - erase_size),
      erase_size);

  SubtractSize(erase_size);
  return iterator(storage_view.data + erase_index);
```

```
}

template <typename T, size_t N, typename A>
auto Storage<T, N, A>::Reserve(size_type requested_capacity) -> void {
  StorageView storage_view = MakeStorageView();

  if (ABSL_PREDICT_FALSE(requested_capacity <= storage_view.capacity)) return;

  AllocationTransaction allocation_tx(GetAllocPtr());

  IteratorValueAdapter<MoveIterator> move_values(
      MoveIterator(storage_view.data));

  size_type new_capacity =
      ComputeCapacity(storage_view.capacity, requested_capacity);
  pointer new_data = allocation_tx.Allocate(new_capacity);

  inlined_vector_internal::ConstructElements(GetAllocPtr(), new_data,
                                             &move_values, storage_view.size);

  inlined_vector_internal::DestroyElements(GetAllocPtr(), storage_view.data,
                                           storage_view.size);

  DeallocateIfAllocated();
  AcquireAllocatedData(&allocation_tx);
  SetIsAllocated();
}

template <typename T, size_t N, typename A>
auto Storage<T, N, A>::ShrinkToFit() -> void {
  // May only be called on allocated instances!
  assert(GetIsAllocated());

  StorageView storage_view{GetAllocatedData(), GetSize(),
                           GetAllocatedCapacity()};

  if (ABSL_PREDICT_FALSE(storage_view.size == storage_view.capacity)) return;

  AllocationTransaction allocation_tx(GetAllocPtr());

  IteratorValueAdapter<MoveIterator> move_values(
      MoveIterator(storage_view.data));

  pointer construct_data;
  if (storage_view.size > GetInlinedCapacity()) {
    size_type new_capacity = storage_view.size;
    construct_data = allocation_tx.Allocate(new_capacity);
  } else {
    construct_data = GetInlinedData();
  }

  ABSL_INTERNAL_TRY {
    inlined_vector_internal::ConstructElements(GetAllocPtr(), construct_data,
                                               &move_values, storage_view.size);
  }
  ABSL_INTERNAL_CATCH_ANY {
    SetAllocatedData(storage_view.data, storage_view.capacity);
    ABSL_INTERNAL_RETHROW;
  }

  inlined_vector_internal::DestroyElements(GetAllocPtr(), storage_view.data,
                                           storage_view.size);

  AllocatorTraits::deallocate(*GetAllocPtr(), storage_view.data,
                              storage_view.capacity);

  if (allocation_tx.DidAllocate()) {
    AcquireAllocatedData(&allocation_tx);
  } else {
    UnsetIsAllocated();
  }
}

template <typename T, size_t N, typename A>
auto Storage<T, N, A>::Swap(Storage* other_storage_ptr) -> void {
  using std::swap;
  assert(this != other_storage_ptr);

  if (GetIsAllocated() && other_storage_ptr->GetIsAllocated()) {
    swap(data_.allocated, other_storage_ptr->data_.allocated);
  } else if (!GetIsAllocated() && !other_storage_ptr->GetIsAllocated()) {
    Storage* small_ptr = this;
    Storage* large_ptr = other_storage_ptr;
    if (small_ptr->GetSize() > large_ptr->GetSize()) swap(small_ptr, large_ptr);

    for (size_type i = 0; i < small_ptr->GetSize(); ++i) {
      swap(small_ptr->GetInlinedData()[i], large_ptr->GetInlinedData()[i]);
    }

    IteratorValueAdapter<MoveIterator> move_values(
        MoveIterator(large_ptr->GetInlinedData() + small_ptr->GetSize()));

    inlined_vector_internal::ConstructElements(
        large_ptr->GetAllocPtr(),
        small_ptr->GetInlinedData() + small_ptr->GetSize(), &move_values,
        large_ptr->GetSize() - small_ptr->GetSize());
    inlined_vector_internal::DestroyElements(
        large_ptr->GetAllocPtr(),
        large_ptr->GetInlinedData() + small_ptr->GetSize(),
        large_ptr->GetSize() - small_ptr->GetSize());
  } else {
    Storage* allocated_ptr = this;
    Storage* inlined_ptr = other_storage_ptr;
    if (!allocated_ptr->GetIsAllocated()) swap(allocated_ptr, inlined_ptr);

    StorageView allocated_storage_view{allocated_ptr->GetAllocatedData(),
                                       allocated_ptr->GetSize(),
                                       allocated_ptr->GetAllocatedCapacity()};

    IteratorValueAdapter<MoveIterator> move_values(
        MoveIterator(inlined_ptr->GetInlinedData()));

    ABSL_INTERNAL_TRY {
      inlined_vector_internal::ConstructElements(
          inlined_ptr->GetAllocPtr(), allocated_ptr->GetInlinedData(),
          &move_values, inlined_ptr->GetSize());
    }
    ABSL_INTERNAL_CATCH_ANY {
      allocated_ptr->SetAllocatedData(allocated_storage_view.data,
                                      allocated_storage_view.capacity);
      ABSL_INTERNAL_RETHROW;
    }

    inlined_vector_internal::DestroyElements(inlined_ptr->GetAllocPtr(),
                                             inlined_ptr->GetInlinedData(),
                                             inlined_ptr->GetSize());

    inlined_ptr->SetAllocatedData(allocated_storage_view.data,
                                  allocated_storage_view.capacity);
  }

  swap(GetSizeAndIsAllocated(), other_storage_ptr->GetSizeAndIsAllocated());
  swap(*GetAllocPtr(), *other_storage_ptr->GetAllocPtr());
}

}  // namespace inlined_vector_internal
ABSL_NAMESPACE_END
}  // namespace absl
```

The Abseil Variant container

The main interface for the Variant container

The variant container used in tensorflow v2.x as of April 2020 can be found here.
Code Snippet: The variant container

```
// -----------------------------------------------------------------------------
// variant.h
// -----------------------------------------------------------------------------
//
// This header file defines an `absl::variant` type for holding a type-safe
// value of some prescribed set of types (noted as alternative types), and
// associated functions for managing variants.
//
// The `absl::variant` type is a form of type-safe union. An `absl::variant`
// should always hold a value of one of its alternative types (except in the
// "valueless by exception state" -- see below). A default-constructed
// `absl::variant` will hold the value of its first alternative type, provided
// it is default-constructible.
//
// In exceptional cases due to error, an `absl::variant` can hold no
// value (known as a "valueless by exception" state), though this is not the
// norm.
//
// As with `absl::optional`, an `absl::variant` -- when it holds a value --
// allocates a value of that type directly within the `variant` itself; it
// cannot hold a reference, array, or the type `void`; it can, however, hold a
// pointer to externally managed memory.
//
// `absl::variant` is a C++11 compatible version of the C++17 `std::variant`
// abstraction and is designed to be a drop-in replacement for code compliant
// with C++17.


#include "absl/meta/type_traits.h"
#include "absl/types/internal/variant.h"

namespace absl {
ABSL_NAMESPACE_BEGIN

// -----------------------------------------------------------------------------
// absl::variant
// -----------------------------------------------------------------------------
//
// An `absl::variant` type is a form of type-safe union. An `absl::variant` --
// except in exceptional cases -- always holds a value of one of its alternative
// types.
//
// Example:
//
```

```
//   // Construct a variant that holds either an integer or a std::string and
//   // assign it to a std::string.
//   absl::variant<int, std::string> v = std::string("abc");
//
//   // A default-constructed variant will hold a value-initialized value of
//   // the first alternative type.
//   auto a = absl::variant<int, std::string>();   // Holds an int of value '0'.
//
//   // variants are assignable.
//
//   // copy assignment
//   auto v1 = absl::variant<int, std::string>("abc");
//   auto v2 = absl::variant<int, std::string>(10);
//   v2 = v1;  // copy assign
//
//   // move assignment
//   auto v1 = absl::variant<int, std::string>("abc");
//   v1 = absl::variant<int, std::string>(10);
//
//   // assignment through type conversion
//   a = 128;         // variant contains int
//   a = "128";       // variant contains std::string
//
// An `absl::variant` holding a value of one of its alternative types `T` holds
// an allocation of `T` directly within the variant itself. An `absl::variant`
// is not allowed to allocate additional storage, such as dynamic memory, to
// allocate the contained value. The contained value shall be allocated in a
// region of the variant storage suitably aligned for all alternative types.
template <typename... Ts>
class variant;

// swap()
//
// Swaps two `absl::variant` values. This function is equivalent to `v.swap(w)`
// where `v` and `w` are `absl::variant` types.
//
// Note that this function requires all alternative types to be both swappable
// and move-constructible, because any two variants may refer to either the same
// type (in which case, they will be swapped) or to two different types (in
// which case the values will need to be moved).
//
template <
    typename... Ts,
    absl::enable_if_t<
        absl::conjunction<std::is_move_constructible<Ts>...,
                          type_traits_internal::IsSwappable<Ts>...>::value,
        int> = 0>
void swap(variant<Ts...>& v, variant<Ts...>& w) noexcept(noexcept(v.swap(w))) {
  v.swap(w);
}

// variant_size
//
// Returns the number of alternative types available for a given `absl::variant`
// type as a compile-time constant expression. As this is a class template, it
// is not generally useful for accessing the number of alternative types of
// any given `absl::variant` instance.
//
// Example:
//
//   auto a = absl::variant<int, std::string>;
//   constexpr int num_types =
//       absl::variant_size<absl::variant<int, std::string>>();
//
//   // You can also use the member constant `value`.
//   constexpr int num_types =
//       absl::variant_size<absl::variant<int, std::string>>::value;
//
//   // `absl::variant_size` is more valuable for use in generic code:
//   template <typename Variant>
//   constexpr bool IsVariantMultivalue() {
//       return absl::variant_size<Variant>() > 1;
//   }
//
// Note that the set of cv-qualified specializations of `variant_size` are
// provided to ensure that those specializations compile (especially when passed
// within template logic).
template <class T>
struct variant_size;

template <class... Ts>
struct variant_size<variant<Ts...>>
    : std::integral_constant<std::size_t, sizeof...(Ts)> {};

// Specialization of `variant_size` for const qualified variants.
template <class T>
struct variant_size<const T> : variant_size<T>::type {};

// Specialization of `variant_size` for volatile qualified variants.
template <class T>
struct variant_size<volatile T> : variant_size<T>::type {};

// Specialization of `variant_size` for const volatile qualified variants.
template <class T>
struct variant_size<const volatile T> : variant_size<T>::type {};
```

```
// variant_alternative
//
// Returns the alternative type for a given `absl::variant` at the passed
// index value as a compile-time constant expression. As this is a class
// template resulting in a type, it is not useful for access of the run-time
// value of any given `absl::variant` variable.
//
// Example:
//
//   // The type of the 0th alternative is "int".
//   using alternative_type_0
//     = absl::variant_alternative<0, absl::variant<int, std::string>>::type;
//
//   static_assert(std::is_same<alternative_type_0, int>::value, "");
//
//   // `absl::variant_alternative` is more valuable for use in generic code:
//   template <typename Variant>
//   constexpr bool IsFirstElementTrivial() {
//     return std::is_trivial_v<variant_alternative<0, Variant>::type>;
//   }
//
// Note that the set of cv-qualified specializations of `variant_alternative`
// are provided to ensure that those specializations compile (especially when
// passed within template logic).
template <std::size_t I, class T>
struct variant_alternative;

template <std::size_t I, class... Types>
struct variant_alternative<I, variant<Types...>> {
  using type =
      variant_internal::VariantAlternativeSfinaeT<I, variant<Types...>>;
};

// Specialization of `variant_alternative` for const qualified variants.
template <std::size_t I, class T>
struct variant_alternative<I, const T> {
  using type = const typename variant_alternative<I, T>::type;
};

// Specialization of `variant_alternative` for volatile qualified variants.
template <std::size_t I, class T>
struct variant_alternative<I, volatile T> {
  using type = volatile typename variant_alternative<I, T>::type;
};

// Specialization of `variant_alternative` for const volatile qualified
// variants.
template <std::size_t I, class T>
```

```
struct variant_alternative<I, const volatile T> {
  using type = const volatile typename variant_alternative<I, T>::type;
};

// Template type alias for variant_alternative<I, T>::type.
//
// Example:
//
//   using alternative_type_0
//     = absl::variant_alternative_t<0, absl::variant<int, std::string>>;
//   static_assert(std::is_same<alternative_type_0, int>::value, "");
template <std::size_t I, class T>
using variant_alternative_t = typename variant_alternative<I, T>::type;

// holds_alternative()
//
// Checks whether the given variant currently holds a given alternative type,
// returning `true` if so.
//
// Example:
//
//   absl::variant<int, std::string> foo = 42;
//   if (absl::holds_alternative<int>(foo)) {
//       std::cout << "The variant holds an integer";
//   }
template <class T, class... Types>
constexpr bool holds_alternative(const variant<Types...>& v) noexcept {
  static_assert(
      variant_internal::UnambiguousIndexOfImpl<variant<Types...>, T,
                                               0>::value != sizeof...(Types),
      "The type T must occur exactly once in Types...");
  return v.index() ==
         variant_internal::UnambiguousIndexOf<variant<Types...>, T>::value;
}

// get()
//
// Returns a reference to the value currently within a given variant, using
// either a unique alternative type amongst the variant's set of alternative
// types, or the variant's index value. Attempting to get a variant's value
// using a type that is not unique within the variant's set of alternative types
// is a compile-time error. If the index of the alternative being specified is
// different from the index of the alternative that is currently stored, throws
// `absl::bad_variant_access`.
//
// Example:
//
//   auto a = absl::variant<int, std::string>;
```

```
//
//   // Get the value by type (if unique).
//   int i = absl::get<int>(a);
//
//   auto b = absl::variant<int, int>;
//
//   // Getting the value by a type that is not unique is ill-formed.
//   int j = absl::get<int>(b);     // Compile Error!
//
//   // Getting value by index not ambiguous and allowed.
//   int k = absl::get<1>(b);

// Overload for getting a variant's lvalue by type.
template <class T, class... Types>
constexpr T& get(variant<Types...>& v) {  // NOLINT
  return variant_internal::VariantCoreAccess::CheckedAccess<
      variant_internal::IndexOf<T, Types...>::value>(v);
}

// Overload for getting a variant's rvalue by type.
// Note: `absl::move()` is required to allow use of constexpr in C++11.
template <class T, class... Types>
constexpr T&& get(variant<Types...>&& v) {
  return variant_internal::VariantCoreAccess::CheckedAccess<
      variant_internal::IndexOf<T, Types...>::value>(absl::move(v));
}

// Overload for getting a variant's const lvalue by type.
template <class T, class... Types>
constexpr const T& get(const variant<Types...>& v) {
  return variant_internal::VariantCoreAccess::CheckedAccess<
      variant_internal::IndexOf<T, Types...>::value>(v);
}

// Overload for getting a variant's const rvalue by type.
// Note: `absl::move()` is required to allow use of constexpr in C++11.
template <class T, class... Types>
constexpr const T&& get(const variant<Types...>&& v) {
  return variant_internal::VariantCoreAccess::CheckedAccess<
      variant_internal::IndexOf<T, Types...>::value>(absl::move(v));
}

// Overload for getting a variant's lvalue by index.
template <std::size_t I, class... Types>
constexpr variant_alternative_t<I, variant<Types...>>& get(
    variant<Types...>& v) {  // NOLINT
  return variant_internal::VariantCoreAccess::CheckedAccess<I>(v);
}
```

```
// Overload for getting a variant's rvalue by index.
// Note: `absl::move()` is required to allow use of constexpr in C++11.
template <std::size_t I, class... Types>
constexpr variant_alternative_t<I, variant<Types...>>&& get(
    variant<Types...>&& v) {
  return variant_internal::VariantCoreAccess::CheckedAccess<I>(absl::move(v));
}

// Overload for getting a variant's const lvalue by index.
template <std::size_t I, class... Types>
constexpr const variant_alternative_t<I, variant<Types...>>& get(
    const variant<Types...>& v) {
  return variant_internal::VariantCoreAccess::CheckedAccess<I>(v);
}

// Overload for getting a variant's const rvalue by index.
// Note: `absl::move()` is required to allow use of constexpr in C++11.
template <std::size_t I, class... Types>
constexpr const variant_alternative_t<I, variant<Types...>>&& get(
    const variant<Types...>&& v) {
  return variant_internal::VariantCoreAccess::CheckedAccess<I>(absl::move(v));
}

// get_if()
//
// Returns a pointer to the value currently stored within a given variant, if
// present, using either a unique alternative type amongst the variant's set of
// alternative types, or the variant's index value. If such a value does not
// exist, returns `nullptr`.
//
// As with `get`, attempting to get a variant's value using a type that is not
// unique within the variant's set of alternative types is a compile-time error.

// Overload for getting a pointer to the value stored in the given variant by
// index.
template <std::size_t I, class... Types>
constexpr absl::add_pointer_t<variant_alternative_t<I, variant<Types...>>>
get_if(variant<Types...>* v) noexcept {
  return (v != nullptr && v->index() == I)
             ? std::addressof(
                   variant_internal::VariantCoreAccess::Access<I>(*v))
             : nullptr;
}

// Overload for getting a pointer to the const value stored in the given
// variant by index.
template <std::size_t I, class... Types>
```

```
constexpr absl::add_pointer_t<const variant_alternative_t<I, variant<Types...>>>
get_if(const variant<Types...>* v) noexcept {
  return (v != nullptr && v->index() == I)
             ? std::addressof(
                   variant_internal::VariantCoreAccess::Access<I>(*v))
             : nullptr;
}

// Overload for getting a pointer to the value stored in the given variant by
// type.
template <class T, class... Types>
constexpr absl::add_pointer_t<T> get_if(variant<Types...>* v) noexcept {
  return absl::get_if<variant_internal::IndexOf<T, Types...>::value>(v);
}

// Overload for getting a pointer to the const value stored in the given variant
// by type.
template <class T, class... Types>
constexpr absl::add_pointer_t<const T> get_if(
    const variant<Types...>* v) noexcept {
  return absl::get_if<variant_internal::IndexOf<T, Types...>::value>(v);
}

// visit()
//
// Calls a provided functor on a given set of variants. `absl::visit()` is
// commonly used to conditionally inspect the state of a given variant (or set
// of variants).
//
// The functor must return the same type when called with any of the variants'
// alternatives.
//
// Example:
//
//   // Define a visitor functor
//   struct GetVariant {
//       template<typename T>
//       void operator()(const T& i) const {
//         std::cout << "The variant's value is: " << i;
//       }
//   };
//
//   // Declare our variant, and call `absl::visit()` on it.
//   // Note that `GetVariant()` returns void in either case.
//   absl::variant<int, std::string> foo = std::string("foo");
//   GetVariant visitor;
//   absl::visit(visitor, foo);  // Prints `The variant's value is: foo'
template <typename Visitor, typename... Variants>
```

```
variant_internal::VisitResult<Visitor, Variants...> visit(Visitor&& vis,
                                                          Variants&&... vars) {
  return variant_internal::
      VisitIndices<variant_size<absl::decay_t<Variants> >::value...>::Run(
          variant_internal::PerformVisitation<Visitor, Variants...>{
              std::forward_as_tuple(absl::forward<Variants>(vars)...),
              absl::forward<Visitor>(vis)},
          vars.index()...);
}

// monostate
//
// The monostate class serves as a first alternative type for a variant for
// which the first variant type is otherwise not default-constructible.
struct monostate {};

// `absl::monostate` Relational Operators

constexpr bool operator<(monostate, monostate) noexcept { return false; }
constexpr bool operator>(monostate, monostate) noexcept { return false; }
constexpr bool operator<=(monostate, monostate) noexcept { return true; }
constexpr bool operator>=(monostate, monostate) noexcept { return true; }
constexpr bool operator==(monostate, monostate) noexcept { return true; }
constexpr bool operator!=(monostate, monostate) noexcept { return false; }


//------------------------------------------------------------------------------
// `absl::variant` Template Definition
//------------------------------------------------------------------------------
template <typename T0, typename... Tn>
class variant<T0, Tn...> : private variant_internal::VariantBase<T0, Tn...> {
  static_assert(absl::conjunction<std::is_object<T0>,
                                  std::is_object<Tn>...>::value,
                "Attempted to instantiate a variant containing a non-object "
                "type.");
  // Intentionally not qualifying `negation` with `absl::` to work around a bug
  // in MSVC 2015 with inline namespace and variadic template.
  static_assert(absl::conjunction<negation<std::is_array<T0> >,
                                  negation<std::is_array<Tn> >...>::value,
                "Attempted to instantiate a variant containing an array type.");
  static_assert(absl::conjunction<std::is_nothrow_destructible<T0>,
                                  std::is_nothrow_destructible<Tn>...>::value,
                "Attempted to instantiate a variant containing a non-nothrow "
                "destructible type.");

  friend struct variant_internal::VariantCoreAccess;

 private:
```

```
 using Base = variant_internal::VariantBase<T0, Tn...>;

public:
 // Constructors

 // Constructs a variant holding a default-initialized value of the first
 // alternative type.
 constexpr variant() /*noexcept(see 111above)*/ = default;

 // Copy constructor, standard semantics
 variant(const variant& other) = default;

 // Move constructor, standard semantics
 variant(variant&& other) /*noexcept(see above)*/ = default;

 // Constructs a variant of an alternative type specified by overload
 // resolution of the provided forwarding arguments through
 // direct-initialization.
 //
 // Note: If the selected constructor is a constexpr constructor, this
 // constructor shall be a constexpr constructor.
 //
 // NOTE: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0608r1.html
 // has been voted passed the design phase in the C++ standard meeting in Mar
 // 2018. It will be implemented and integrated into `absl::variant`.
 template <
    class T,
    std::size_t I = std::enable_if<
       variant_internal::IsNeitherSelfNorInPlace<variant,
                                                 absl::decay_t<T>>::value,
       variant_internal::IndexOfConstructedType<variant, T>>::type::value,
    class Tj = absl::variant_alternative_t<I, variant>,
    absl::enable_if_t<std::is_constructible<Tj, T>::value>* =
       nullptr>
 constexpr variant(T&& t) noexcept(std::is_nothrow_constructible<Tj, T>::value)
    : Base(variant_internal::EmplaceTag<I>(), absl::forward<T>(t)) {}

 // Constructs a variant of an alternative type from the arguments through
 // direct-initialization.
 //
 // Note: If the selected constructor is a constexpr constructor, this
 // constructor shall be a constexpr constructor.
 template <class T, class... Args,
          typename std::enable_if<std::is_constructible<
             variant_internal::UnambiguousTypeOfT<variant, T>,
             Args...>::value>::type* = nullptr>
 constexpr explicit variant(in_place_type_t<T>, Args&&... args)
    : Base(variant_internal::EmplaceTag<
          variant_internal::UnambiguousIndexOf<variant, T>::value>(),
        absl::forward<Args>(args)...) {}

// Constructs a variant of an alternative type from an initializer list
// and other arguments through direct-initialization.
//
// Note: If the selected constructor is a constexpr constructor, this
// constructor shall be a constexpr constructor.
template <class T, class U, class... Args,
          typename std::enable_if<std::is_constructible<
              variant_internal::UnambiguousTypeOfT<variant, T>,
              std::initializer_list<U>&, Args...>::value>::type* = nullptr>
constexpr explicit variant(in_place_type_t<T>, std::initializer_list<U> il,
                           Args&&... args)
    : Base(variant_internal::EmplaceTag<
          variant_internal::UnambiguousIndexOf<variant, T>::value>(),
        il, absl::forward<Args>(args)...) {}

// Constructs a variant of an alternative type from a provided index,
// through value-initialization using the provided forwarded arguments.
template <std::size_t I, class... Args,
          typename std::enable_if<std::is_constructible<
              variant_internal::VariantAlternativeSfinaeT<I, variant>,
              Args...>::value>::type* = nullptr>
constexpr explicit variant(in_place_index_t<I>, Args&&... args)
    : Base(variant_internal::EmplaceTag<I>(), absl::forward<Args>(args)...) {}

// Constructs a variant of an alternative type from a provided index,
// through value-initialization of an initializer list and the provided
// forwarded arguments.
template <std::size_t I, class U, class... Args,
          typename std::enable_if<std::is_constructible<
              variant_internal::VariantAlternativeSfinaeT<I, variant>,
              std::initializer_list<U>&, Args...>::value>::type* = nullptr>
constexpr explicit variant(in_place_index_t<I>, std::initializer_list<U> il,
                           Args&&... args)
    : Base(variant_internal::EmplaceTag<I>(), il,
        absl::forward<Args>(args)...) {}

// Destructors

// Destroys the variant's currently contained value, provided that
// `absl::valueless_by_exception()` is false.
~variant() = default;

// Assignment Operators

// Copy assignment operator
```

```
variant& operator=(const variant& other) = default;

// Move assignment operator
variant& operator=(variant&& other) /*noexcept(see above)*/ = default;

// Converting assignment operator
//
// NOTE: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0608r1.html
// has been voted passed the design phase in the C++ standard meeting in Mar
// 2018. It will be implemented and integrated into `absl::variant`.
template <
    class T,
    std::size_t I = std::enable_if<
        !std::is_same<absl::decay_t<T>, variant>::value,
        variant_internal::IndexOfConstructedType<variant, T>>::type::value,
    class Tj = absl::variant_alternative_t<I, variant>,
    typename std::enable_if<std::is_assignable<Tj&, T>::value &&
                            std::is_constructible<Tj, T>::value>::type* =
        nullptr>
variant& operator=(T&& t) noexcept(
    std::is_nothrow_assignable<Tj&, T>::value&&
        std::is_nothrow_constructible<Tj, T>::value) {
  variant_internal::VisitIndices<sizeof...(Tn) + 1>::Run(
      variant_internal::VariantCoreAccess::MakeConversionAssignVisitor(
          this, absl::forward<T>(t)),
      index());

  return *this;
}


// emplace() Functions

// Constructs a value of the given alternative type T within the variant.
//
// Example:
//
//   absl::variant<std::vector<int>, int, std::string> v;
//   v.emplace<int>(99);
//   v.emplace<std::string>("abc");
template <
    class T, class... Args,
    typename std::enable_if<std::is_constructible<
        absl::variant_alternative_t<
            variant_internal::UnambiguousIndexOf<variant, T>::value, variant>,
        Args...>::value>::type* = nullptr>
T& emplace(Args&&... args) {
  return variant_internal::VariantCoreAccess::Replace<
```

```
      variant_internal::UnambiguousIndexOf<variant, T>::value>(
      this, absl::forward<Args>(args)...);
}

// Constructs a value of the given alternative type T within the variant using
// an initializer list.
//
// Example:
//
//   absl::variant<std::vector<int>, int, std::string> v;
//   v.emplace<std::vector<int>>({0, 1, 2});
template <
    class T, class U, class... Args,
    typename std::enable_if<std::is_constructible<
        absl::variant_alternative_t<
            variant_internal::UnambiguousIndexOf<variant, T>::value, variant>,
        std::initializer_list<U>&, Args...>::value>::type* = nullptr>
T& emplace(std::initializer_list<U> il, Args&&... args) {
  return variant_internal::VariantCoreAccess::Replace<
      variant_internal::UnambiguousIndexOf<variant, T>::value>(
      this, il, absl::forward<Args>(args)...);
}

// Destroys the current value of the variant (provided that
// `absl::valueless_by_exception()` is false, and constructs a new value at
// the given index.
//
// Example:
//
//   absl::variant<std::vector<int>, int, int> v;
//   v.emplace<1>(99);
//   v.emplace<2>(98);
//   v.emplace<int>(99);  // Won't compile. 'int' isn't a unique type.
template <std::size_t I, class... Args,
          typename std::enable_if<
              std::is_constructible<absl::variant_alternative_t<I, variant>,
                                    Args...>::value>::type* = nullptr>
absl::variant_alternative_t<I, variant>& emplace(Args&&... args) {
  return variant_internal::VariantCoreAccess::Replace<I>(
      this, absl::forward<Args>(args)...);
}

// Destroys the current value of the variant (provided that
// `absl::valueless_by_exception()` is false, and constructs a new value at
// the given index using an initializer list and the provided arguments.
//
// Example:
//
```

```
  //   absl::variant<std::vector<int>, int, int> v;
  //   v.emplace<0>({0, 1, 2});
  template <std::size_t I, class U, class... Args,
            typename std::enable_if<std::is_constructible<
                absl::variant_alternative_t<I, variant>,
                std::initializer_list<U>&, Args...>::value>::type* = nullptr>
  absl::variant_alternative_t<I, variant>& emplace(std::initializer_list<U> il,
                                                   Args&&... args) {
    return variant_internal::VariantCoreAccess::Replace<I>(
        this, il, absl::forward<Args>(args)...);
  }

  // variant::valueless_by_exception()
  //
  // Returns false if and only if the variant currently holds a valid value.
  constexpr bool valueless_by_exception() const noexcept {
    return this->index_ == absl::variant_npos;
  }

  // variant::index()
  //
  // Returns the index value of the variant's currently selected alternative
  // type.
  constexpr std::size_t index() const noexcept { return this->index_; }

  // variant::swap()
  //
  // Swaps the values of two variant objects.
  //
  void swap(variant& rhs) noexcept(
      absl::conjunction<
          std::is_nothrow_move_constructible<T0>,
          std::is_nothrow_move_constructible<Tn>...,
          type_traits_internal::IsNothrowSwappable<T0>,
          type_traits_internal::IsNothrowSwappable<Tn>...>::value) {
    return variant_internal::VisitIndices<sizeof...(Tn) + 1>::Run(
        variant_internal::Swap<T0, Tn...>{this, &rhs}, rhs.index());
  }
};

// We need a valid declaration of variant<> for SFINAE and overload resolution
// to work properly above, but we don't need a full declaration since this type
// will never be constructed. This declaration, though incomplete, suffices.
template <>
class variant<>;

//------------------------------------------------------------------------------
// Relational Operators
```

```
//------------------------------------------------------------------------------
//
// If neither operand is in the `variant::valueless_by_exception` state:
//
//   * If the index of both variants is the same, the relational operator
//     returns the result of the corresponding relational operator for the
//     corresponding alternative type.
//   * If the index of both variants is not the same, the relational operator
//     returns the result of that operation applied to the value of the left
//     operand's index and the value of the right operand's index.
//   * If at least one operand is in the valueless_by_exception state:
//     - A variant in the valueless_by_exception state is only considered equal
//       to another variant in the valueless_by_exception state.
//     - If exactly one operand is in the valueless_by_exception state, the
//       variant in the valueless_by_exception state is less than the variant
//       that is not in the valueless_by_exception state.
//
// Note: The value 1 is added to each index in the relational comparisons such
// that the index corresponding to the valueless_by_exception state wraps around
// to 0 (the lowest value for the index type), and the remaining indices stay in
// the same relative order.

// Equal-to operator
template <typename... Types>
constexpr variant_internal::RequireAllHaveEqualT<Types...> operator==(
    const variant<Types...>& a, const variant<Types...>& b) {
  return (a.index() == b.index()) &&
         variant_internal::VisitIndices<sizeof...(Types)>::Run(
             variant_internal::EqualsOp<Types...>{&a, &b}, a.index());
}

// Not equal operator
template <typename... Types>
constexpr variant_internal::RequireAllHaveNotEqualT<Types...> operator!=(
    const variant<Types...>& a, const variant<Types...>& b) {
  return (a.index() != b.index()) ||
         variant_internal::VisitIndices<sizeof...(Types)>::Run(
             variant_internal::NotEqualsOp<Types...>{&a, &b}, a.index());
}

// Less-than operator
template <typename... Types>
constexpr variant_internal::RequireAllHaveLessThanT<Types...> operator<(
    const variant<Types...>& a, const variant<Types...>& b) {
  return (a.index() != b.index())
             ? (a.index() + 1) < (b.index() + 1)
             : variant_internal::VisitIndices<sizeof...(Types)>::Run(
                   variant_internal::LessThanOp<Types...>{&a, &b}, a.index());
}

// Greater-than operator
template <typename... Types>
constexpr variant_internal::RequireAllHaveGreaterThanT<Types...> operator>(
    const variant<Types...>& a, const variant<Types...>& b) {
  return (a.index() != b.index())
             ? (a.index() + 1) > (b.index() + 1)
             : variant_internal::VisitIndices<sizeof...(Types)>::Run(
                   variant_internal::GreaterThanOp<Types...>{&a, &b},
                   a.index());
}

// Less-than or equal-to operator
template <typename... Types>
constexpr variant_internal::RequireAllHaveLessThanOrEqualT<Types...> operator<=(
    const variant<Types...>& a, const variant<Types...>& b) {
  return (a.index() != b.index())
             ? (a.index() + 1) < (b.index() + 1)
             : variant_internal::VisitIndices<sizeof...(Types)>::Run(
                   variant_internal::LessThanOrEqualsOp<Types...>{&a, &b},
                   a.index());
}

// Greater-than or equal-to operator
template <typename... Types>
constexpr variant_internal::RequireAllHaveGreaterThanOrEqualT<Types...>
operator>=(const variant<Types...>& a, const variant<Types...>& b) {
  return (a.index() != b.index())
             ? (a.index() + 1) > (b.index() + 1)
             : variant_internal::VisitIndices<sizeof...(Types)>::Run(
                   variant_internal::GreaterThanOrEqualsOp<Types...>{&a, &b},
                   a.index());
}

ABSL_NAMESPACE_END
}  // namespace absl

namespace std {

// hash()
template <>  // NOLINT
struct hash<absl::monostate> {
  std::size_t operator()(absl::monostate) const { return 0; }
};

template <class... T>  // NOLINT
struct hash<absl::variant<T...>>
```

```
    : absl::variant_internal::VariantHashBase<absl::variant<T...>, void,
                                              absl::remove_const_t<T>...> {};

}  // namespace std

#endif  // ABSL_USES_STD_VARIANT

namespace absl {
ABSL_NAMESPACE_BEGIN
namespace variant_internal {

// Helper visitor for converting a variant<Ts...>` into another type (mostly
// variant) that can be constructed from any type.
template <typename To>
struct ConversionVisitor {
  template <typename T>
  To operator()(T&& v) const {
    return To(std::forward<T>(v));
  }
};

}  // namespace variant_internal

// ConvertVariantTo()
//
// Helper functions to convert an `absl::variant` to a variant of another set of
// types, provided that the alternative type of the new variant type can be
// converted from any type in the source variant.
//
// Example:
//
//   absl::variant<name1, name2, float> InternalReq(const Req&);
//
//   // name1 and name2 are convertible to name
//   absl::variant<name, float> ExternalReq(const Req& req) {
//     return absl::ConvertVariantTo<absl::variant<name, float>>(
//              InternalReq(req));
//   }
template <typename To, typename Variant>
To ConvertVariantTo(Variant&& variant) {
  return absl::visit(variant_internal::ConversionVisitor<To>{},
                     std::forward<Variant>(variant));
}

ABSL_NAMESPACE_END
}  // namespace absl
```

Variant Container implementation details

The following code snippet contains implementation details of the variant container which can be found in absl/types/internal/variant.h.

Code Snippet: Variant container implementation details

```
// Implementation details of absl/types/variant.h, pulled into a
// separate file to avoid cluttering the top of the API header with
// implementation details.

. . .
#include "absl/base/config.h"
#include "absl/base/internal/identity.h"
#include "absl/base/internal/inline_variable.h"
#include "absl/base/internal/invoke.h"
#include "absl/base/macros.h"
#include "absl/base/optimization.h"
#include "absl/meta/type_traits.h"
#include "absl/types/bad_variant_access.h"
#include "absl/utility/utility.h"

#if !defined(ABSL_USES_STD_VARIANT)

namespace absl {
ABSL_NAMESPACE_BEGIN

template <class... Types>
class variant;

ABSL_INTERNAL_INLINE_CONSTEXPR(size_t, variant_npos, -1);

template <class T>
struct variant_size;

template <std::size_t I, class T>
struct variant_alternative;

namespace variant_internal {

// NOTE: See specializations below for details.
template <std::size_t I, class T>
struct VariantAlternativeSfinae {};

// Requires: I < variant_size_v<T>.
//
// Value: The Ith type of Types...
template <std::size_t I, class T0, class... Tn>
struct VariantAlternativeSfinae<I, variant<T0, Tn...>>
```

```
    : VariantAlternativeSfinae<I - 1, variant<Tn...>> {};

// Value: T0
template <class T0, class... Ts>
struct VariantAlternativeSfinae<0, variant<T0, Ts...>> {
  using type = T0;
};

template <std::size_t I, class T>
using VariantAlternativeSfinaeT = typename VariantAlternativeSfinae<I, T>::type;

// NOTE: Requires T to be a reference type.
template <class T, class U>
struct GiveQualsTo;

template <class T, class U>
struct GiveQualsTo<T&, U> {
  using type = U&;
};

template <class T, class U>
struct GiveQualsTo<T&&, U> {
  using type = U&&;
};

template <class T, class U>
struct GiveQualsTo<const T&, U> {
  using type = const U&;
};

template <class T, class U>
struct GiveQualsTo<const T&&, U> {
  using type = const U&&;
};

template <class T, class U>
struct GiveQualsTo<volatile T&, U> {
  using type = volatile U&;
};

template <class T, class U>
struct GiveQualsTo<volatile T&&, U> {
  using type = volatile U&&;
};

template <class T, class U>
struct GiveQualsTo<volatile const T&, U> {
  using type = volatile const U&;
};

template <class T, class U>
struct GiveQualsTo<volatile const T&&, U> {
  using type = volatile const U&&;
};

template <class T, class U>
using GiveQualsToT = typename GiveQualsTo<T, U>::type;

// Convenience alias, since size_t integral_constant is used a lot in this file.
template <std::size_t I>
using SizeT = std::integral_constant<std::size_t, I>;

using NPos = SizeT<variant_npos>;

template <class Variant, class T, class = void>
struct IndexOfConstructedType {};

template <std::size_t I, class Variant>
struct VariantAccessResultImpl;

template <std::size_t I, template <class...> class Variantemplate, class... T>
struct VariantAccessResultImpl<I, Variantemplate<T...>&> {
  using type = typename absl::variant_alternative<I, variant<T...>>::type&;
};

template <std::size_t I, template <class...> class Variantemplate, class... T>
struct VariantAccessResultImpl<I, const Variantemplate<T...>&> {
  using type =
      const typename absl::variant_alternative<I, variant<T...>>::type&;
};

template <std::size_t I, template <class...> class Variantemplate, class... T>
struct VariantAccessResultImpl<I, Variantemplate<T...>&&> {
  using type = typename absl::variant_alternative<I, variant<T...>>::type&&;
};

template <std::size_t I, template <class...> class Variantemplate, class... T>
struct VariantAccessResultImpl<I, const Variantemplate<T...>&&> {
  using type =
      const typename absl::variant_alternative<I, variant<T...>>::type&&;
};

template <std::size_t I, class Variant>
using VariantAccessResult =
    typename VariantAccessResultImpl<I, Variant&&>::type;
```

```
// NOTE: This is used instead of std::array to reduce instantiation overhead.
template <class T, std::size_t Size>
struct SimpleArray {
  static_assert(Size != 0, "");
  T value[Size];
};

template <class T>
struct AccessedType {
  using type = T;
};

template <class T>
using AccessedTypeT = typename AccessedType<T>::type;

template <class T, std::size_t Size>
struct AccessedType<SimpleArray<T, Size>> {
  using type = AccessedTypeT<T>;
};

template <class T>
constexpr T AccessSimpleArray(const T& value) {
  return value;
}

template <class T, std::size_t Size, class... SizeT>
constexpr AccessedTypeT<T> AccessSimpleArray(const SimpleArray<T, Size>& table,
                                             std::size_t head_index,
                                             SizeT... tail_indices) {
  return AccessSimpleArray(table.value[head_index], tail_indices...);
}

// Note: Intentionally is an alias.
template <class T>
using AlwaysZero = SizeT<0>;

template <class Op, class... Vs>
struct VisitIndicesResultImpl {
  using type = absl::result_of_t<Op(AlwaysZero<Vs>...)>;
};

template <class Op, class... Vs>
using VisitIndicesResultT = typename VisitIndicesResultImpl<Op, Vs...>::type;

template <class ReturnType, class FunctionObject, class EndIndices,
          class BoundIndices>
struct MakeVisitationMatrix;
```

```
}

template <class ReturnType, class FunctionObject, std::size_t... Indices>
constexpr ReturnType call_with_indices(FunctionObject&& function) {
  static_assert(
      std::is_same<ReturnType, decltype(std::declval<FunctionObject>()(
                              SizeT<Indices>()...))>::value,
      "Not all visitation overloads have the same return type.");
  return absl::forward<FunctionObject>(function)(SizeT<Indices>()...);
}

template <class ReturnType, class FunctionObject, std::size_t... BoundIndices>
struct MakeVisitationMatrix<ReturnType, FunctionObject, index_sequence<>,
                            index_sequence<BoundIndices...>> {
  using ResultType = ReturnType (*)(FunctionObject&&);
  static constexpr ResultType Run() {
    return &call_with_indices<ReturnType, FunctionObject,
                              (BoundIndices - 1)...>;
  }
};

template <typename Is, std::size_t J>
struct AppendToIndexSequence;

template <typename Is, std::size_t J>
using AppendToIndexSequenceT = typename AppendToIndexSequence<Is, J>::type;

template <std::size_t... Is, std::size_t J>
struct AppendToIndexSequence<index_sequence<Is...>, J> {
  using type = index_sequence<Is..., J>;
};

template <class ReturnType, class FunctionObject, class EndIndices,
          class CurrIndices, class BoundIndices>
struct MakeVisitationMatrixImpl;

template <class ReturnType, class FunctionObject, class EndIndices,
          std::size_t... CurrIndices, class BoundIndices>
struct MakeVisitationMatrixImpl<ReturnType, FunctionObject, EndIndices,
                                index_sequence<CurrIndices...>, BoundIndices> {
  using ResultType = SimpleArray<
      typename MakeVisitationMatrix<ReturnType, FunctionObject, EndIndices,
                                    index_sequence<>>::ResultType,
      sizeof...(CurrIndices)>;

  static constexpr ResultType Run() {
    return {{MakeVisitationMatrix<
        ReturnType, FunctionObject, EndIndices,
        AppendToIndexSequenceT<BoundIndices, CurrIndices>>::Run()...}};
  }
```

```
};

template <class ReturnType, class FunctionObject, std::size_t HeadEndIndex,
          std::size_t... TailEndIndices, std::size_t... BoundIndices>
struct MakeVisitationMatrix<ReturnType, FunctionObject,
                            index_sequence<HeadEndIndex, TailEndIndices...>,
                            index_sequence<BoundIndices...>>
    : MakeVisitationMatrixImpl<ReturnType, FunctionObject,
                               index_sequence<TailEndIndices...>,
                               absl::make_index_sequence<HeadEndIndex>,
                               index_sequence<BoundIndices...>> {};

struct UnreachableSwitchCase {
  template <class Op>
  [[noreturn]] static VisitIndicesResultT<Op, std::size_t> Run(
      Op&& /*ignored*/) {
#if ABSL_HAVE_BUILTIN(__builtin_unreachable) || \
    (defined(__GNUC__) && !defined(__clang__))
    __builtin_unreachable();
#elif defined(_MSC_VER)
    __assume(false);
#else
    // Try to use assert of false being identified as an unreachable intrinsic.
    // NOTE: We use assert directly to increase chances of exploiting an assume
    //       intrinsic.
    assert(false);  // NOLINT

    // Hack to silence potential no return warning -- cause an infinite loop.
    return Run(absl::forward<Op>(op));
#endif  // Checks for __builtin_unreachable
  }
};

template <class Op, std::size_t I>
struct ReachableSwitchCase {
  static VisitIndicesResultT<Op, std::size_t> Run(Op&& op) {
    return absl::base_internal::Invoke(absl::forward<Op>(op), SizeT<I>());
  }
};

// The number 33 is just a guess at a reasonable maximum to our switch. It is
// not based on any analysis. The reason it is a power of 2 plus 1 instead of a
// power of 2 is because the number was picked to correspond to a power of 2
// amount of "normal" alternatives, plus one for the possibility of the user
// providing "monostate" in addition to the more natural alternatives.
ABSL_INTERNAL_INLINE_CONSTEXPR(std::size_t, MaxUnrolledVisitCases, 33);

// Note: The default-definition is for unreachable cases.
```

```
template <bool IsReachable>
struct PickCaseImpl {
  template <class Op, std::size_t I>
  using Apply = UnreachableSwitchCase;
};

template <>
struct PickCaseImpl</*IsReachable =*/true> {
  template <class Op, std::size_t I>
  using Apply = ReachableSwitchCase<Op, I>;
};

// Note: This form of dance with template aliases is to make sure that we
//       instantiate a number of templates proportional to the number of variant
//       alternatives rather than a number of templates proportional to our
//       maximum unrolled amount of visitation cases (aliases are effectively
//       "free" whereas other template instantiations are costly).
template <class Op, std::size_t I, std::size_t EndIndex>
using PickCase = typename PickCaseImpl<(I < EndIndex)>::template Apply<Op, I>;

template <class ReturnType>
[[noreturn]] ReturnType TypedThrowBadVariantAccess() {
  absl::variant_internal::ThrowBadVariantAccess();
}

// Given N variant sizes, determine the number of cases there would need to be
// in a single switch-statement that would cover every possibility in the
// corresponding N-ary visit operation.
template <std::size_t... NumAlternatives>
struct NumCasesOfSwitch;

template <std::size_t HeadNumAlternatives, std::size_t... TailNumAlternatives>
struct NumCasesOfSwitch<HeadNumAlternatives, TailNumAlternatives...> {
  static constexpr std::size_t value =
      (HeadNumAlternatives + 1) *
      NumCasesOfSwitch<TailNumAlternatives...>::value;
};

template <>
struct NumCasesOfSwitch<> {
  static constexpr std::size_t value = 1;
};

// A switch statement optimizes better than the table of function pointers.
template <std::size_t EndIndex>
struct VisitIndicesSwitch {
  static_assert(EndIndex <= MaxUnrolledVisitCases,
                "Maximum unrolled switch size exceeded.");
```

```
template <class Op>
static VisitIndicesResultT<Op, std::size_t> Run(Op&& op, std::size_t i) {
 switch (i) {
  case 0:
   return PickCase<Op, 0, EndIndex>::Run(absl::forward<Op>(op));
  case 1:
   return PickCase<Op, 1, EndIndex>::Run(absl::forward<Op>(op));
  case 2:
   return PickCase<Op, 2, EndIndex>::Run(absl::forward<Op>(op));
  case 3:
   return PickCase<Op, 3, EndIndex>::Run(absl::forward<Op>(op));
  case 4:
   return PickCase<Op, 4, EndIndex>::Run(absl::forward<Op>(op));
  case 5:
   return PickCase<Op, 5, EndIndex>::Run(absl::forward<Op>(op));
  case 6:
   return PickCase<Op, 6, EndIndex>::Run(absl::forward<Op>(op));
  case 7:
   return PickCase<Op, 7, EndIndex>::Run(absl::forward<Op>(op));
  case 8:
   return PickCase<Op, 8, EndIndex>::Run(absl::forward<Op>(op));
  case 9:
   return PickCase<Op, 9, EndIndex>::Run(absl::forward<Op>(op));
  case 10:
   return PickCase<Op, 10, EndIndex>::Run(absl::forward<Op>(op));
  case 11:
   return PickCase<Op, 11, EndIndex>::Run(absl::forward<Op>(op));
  case 12:
   return PickCase<Op, 12, EndIndex>::Run(absl::forward<Op>(op));
  case 13:
   return PickCase<Op, 13, EndIndex>::Run(absl::forward<Op>(op));
  case 14:
   return PickCase<Op, 14, EndIndex>::Run(absl::forward<Op>(op));
  case 15:
   return PickCase<Op, 15, EndIndex>::Run(absl::forward<Op>(op));
  case 16:
   return PickCase<Op, 16, EndIndex>::Run(absl::forward<Op>(op));
  case 17:
   return PickCase<Op, 17, EndIndex>::Run(absl::forward<Op>(op));
  case 18:
   return PickCase<Op, 18, EndIndex>::Run(absl::forward<Op>(op));
  case 19:
   return PickCase<Op, 19, EndIndex>::Run(absl::forward<Op>(op));
  case 20:
   return PickCase<Op, 20, EndIndex>::Run(absl::forward<Op>(op));
  case 21:
   return PickCase<Op, 21, EndIndex>::Run(absl::forward<Op>(op));
```

```
      case 22:
        return PickCase<Op, 22, EndIndex>::Run(absl::forward<Op>(op));
      case 23:
        return PickCase<Op, 23, EndIndex>::Run(absl::forward<Op>(op));
      case 24:
        return PickCase<Op, 24, EndIndex>::Run(absl::forward<Op>(op));
      case 25:
        return PickCase<Op, 25, EndIndex>::Run(absl::forward<Op>(op));
      case 26:
        return PickCase<Op, 26, EndIndex>::Run(absl::forward<Op>(op));
      case 27:
        return PickCase<Op, 27, EndIndex>::Run(absl::forward<Op>(op));
      case 28:
        return PickCase<Op, 28, EndIndex>::Run(absl::forward<Op>(op));
      case 29:
        return PickCase<Op, 29, EndIndex>::Run(absl::forward<Op>(op));
      case 30:
        return PickCase<Op, 30, EndIndex>::Run(absl::forward<Op>(op));
      case 31:
        return PickCase<Op, 31, EndIndex>::Run(absl::forward<Op>(op));
      case 32:
        return PickCase<Op, 32, EndIndex>::Run(absl::forward<Op>(op));
      default:
        ABSL_ASSERT(i == variant_npos);
        return absl::base_internal::Invoke(absl::forward<Op>(op), NPos());
    }
  }
};

template <std::size_t... EndIndices>
struct VisitIndicesFallback {
  template <class Op, class... SizeT>
  static VisitIndicesResultT<Op, SizeT...> Run(Op&& op, SizeT... indices) {
    return AccessSimpleArray(
        MakeVisitationMatrix<VisitIndicesResultT<Op, SizeT...>, Op,
                             index_sequence<(EndIndices + 1)...>,
                             index_sequence<>>::Run(),
        (indices + 1)...)(absl::forward<Op>(op));
  }
};

// Take an N-dimensional series of indices and convert them into a single index
// without loss of information. The purpose of this is to be able to convert an
// N-ary visit operation into a single switch statement.
template <std::size_t...>
struct FlattenIndices;

template <std::size_t HeadSize, std::size_t... TailSize>
```

```
struct FlattenIndices<HeadSize, TailSize...> {
  template<class... SizeType>
  static constexpr std::size_t Run(std::size_t head, SizeType... tail) {
    return head + HeadSize * FlattenIndices<TailSize...>::Run(tail...);
  }
};

template <>
struct FlattenIndices<> {
  static constexpr std::size_t Run() { return 0; }
};

// Take a single "flattened" index (flattened by FlattenIndices) and determine
// the value of the index of one of the logically represented dimensions.
template <std::size_t I, std::size_t IndexToGet, std::size_t HeadSize,
          std::size_t... TailSize>
struct UnflattenIndex {
  static constexpr std::size_t value =
      UnflattenIndex<I / HeadSize, IndexToGet - 1, TailSize...>::value;
};

template <std::size_t I, std::size_t HeadSize, std::size_t... TailSize>
struct UnflattenIndex<I, 0, HeadSize, TailSize...> {
  static constexpr std::size_t value = (I % HeadSize);
};

// The backend for converting an N-ary visit operation into a unary visit.
template <class IndexSequence, std::size_t... EndIndices>
struct VisitIndicesVariadicImpl;

template <std::size_t... N, std::size_t... EndIndices>
struct VisitIndicesVariadicImpl<absl::index_sequence<N...>, EndIndices...> {
  // A type that can take an N-ary function object and converts it to a unary
  // function object that takes a single, flattened index, and "unflattens" it
  // into its individual dimensions when forwarding to the wrapped object.
  template <class Op>
  struct FlattenedOp {
    template <std::size_t I>
    VisitIndicesResultT<Op, decltype(EndIndices)...> operator()(
        SizeT<I> /*index*/) && {
      return base_internal::Invoke(
          absl::forward<Op>(op),
          SizeT<UnflattenIndex<I, N, (EndIndices + 1)...>::value -
                std::size_t{1}>()...);
    }

    Op&& op;
  };
  template <class Op, class... SizeType>
  static VisitIndicesResultT<Op, decltype(EndIndices)...> Run(
      Op&& op, SizeType... i) {
    return VisitIndicesSwitch<NumCasesOfSwitch<EndIndices...>::value>::Run(
        FlattenedOp<Op>{absl::forward<Op>(op)},
        FlattenIndices<(EndIndices + std::size_t{1})...>::Run(
            (i + std::size_t{1})...));
  }
};

template <std::size_t... EndIndices>
struct VisitIndicesVariadic
    : VisitIndicesVariadicImpl<absl::make_index_sequence<sizeof...(EndIndices)>,
                               EndIndices...> {};

// This implementation will flatten N-ary visit operations into a single switch
// statement when the number of cases would be less than our maximum specified
// switch-statement size.
// TODO(calabrese)
//   Based on benchmarks, determine whether the function table approach actually
//   does optimize better than a chain of switch statements and possibly update
//   the implementation accordingly. Also consider increasing the maximum switch
//   size.
template <std::size_t... EndIndices>
struct VisitIndices
    : absl::conditional_t<(NumCasesOfSwitch<EndIndices...>::value <=
                           MaxUnrolledVisitCases),
                          VisitIndicesVariadic<EndIndices...>,
                          VisitIndicesFallback<EndIndices...>> {};

template <std::size_t EndIndex>
struct VisitIndices<EndIndex>
    : absl::conditional_t<(EndIndex <= MaxUnrolledVisitCases),
                          VisitIndicesSwitch<EndIndex>,
                          VisitIndicesFallback<EndIndex>> {};

// Suppress bogus warning on MSVC: MSVC complains that the `reinterpret_cast`
// below is returning the address of a temporary or local object.
#ifdef _MSC_VER
#pragma warning(push)
#pragma warning(disable : 4172)
#endif  // _MSC_VER

// TODO(calabrese) std::launder
// TODO(calabrese) constexpr
// NOTE: DO NOT REMOVE the `inline` keyword as it is necessary to work around a
// MSVC bug. See https://github.com/abseil/abseil-cpp/issues/129 for details.
```

```
template <class Self, std::size_t I>
inline VariantAccessResult<I, Self> AccessUnion(Self&& self, SizeT<I> /*i*/) {
  return reinterpret_cast<VariantAccessResult<I, Self>>(self);
}

#ifdef _MSC_VER
#pragma warning(pop)
#endif  // _MSC_VER

template <class T>
void DeducedDestroy(T& self) {  // NOLINT
  self.~T();
}

// NOTE: This type exists as a single entity for variant and its bases to
// befriend. It contains helper functionality that manipulates the state of the
// variant, such as the implementation of things like assignment and emplace
// operations.
struct VariantCoreAccess {
  template <class VariantType>
  static typename VariantType::Variant& Derived(VariantType& self) {  // NOLINT
    return static_cast<typename VariantType::Variant&>(self);
  }

  template <class VariantType>
  static const typename VariantType::Variant& Derived(
      const VariantType& self) {  // NOLINT
    return static_cast<const typename VariantType::Variant&>(self);
  }

  template <class VariantType>
  static void Destroy(VariantType& self) {  // NOLINT
    Derived(self).destroy();
    self.index_ = absl::variant_npos;
  }

  template <class Variant>
  static void SetIndex(Variant& self, std::size_t i) {  // NOLINT
    self.index_ = i;
  }

  template <class Variant>
  static void InitFrom(Variant& self, Variant&& other) {  // NOLINT
    VisitIndices<absl::variant_size<Variant>::value>::Run(
        InitFromVisitor<Variant, Variant&&>{&self,
                                            std::forward<Variant>(other)},
        other.index());
    self.index_ = other.index();
```

```
}

// Access a variant alternative, assuming the index is correct.
template <std::size_t I, class Variant>
static VariantAccessResult<I, Variant> Access(Variant&& self) {
  // This cast instead of invocation of AccessUnion with an rvalue is a
  // workaround for msvc. Without this there is a runtime failure when dealing
  // with rvalues.
  // TODO(calabrese) Reduce test case and find a simpler workaround.
  return static_cast<VariantAccessResult<I, Variant>>(
      variant_internal::AccessUnion(self.state_, SizeT<I>()));
}

// Access a variant alternative, throwing if the index is incorrect.
template <std::size_t I, class Variant>
static VariantAccessResult<I, Variant> CheckedAccess(Variant&& self) {
  if (ABSL_PREDICT_FALSE(self.index_ != I)) {
    TypedThrowBadVariantAccess<VariantAccessResult<I, Variant>>();
  }

  return Access<I>(absl::forward<Variant>(self));
}

// The implementation of the move-assignment operation for a variant.
template <class VType>
struct MoveAssignVisitor {
  using DerivedType = typename VType::Variant;
  template <std::size_t NewIndex>
  void operator()(SizeT<NewIndex> /*new_i*/) const {
    if (left->index_ == NewIndex) {
      Access<NewIndex>(*left) = std::move(Access<NewIndex>(*right));
    } else {
      Derived(*left).template emplace<NewIndex>(
          std::move(Access<NewIndex>(*right)));
    }
  }

  void operator()(SizeT<absl::variant_npos> /*new_i*/) const {
    Destroy(*left);
  }

  VType* left;
  VType* right;
};

template <class VType>
static MoveAssignVisitor<VType> MakeMoveAssignVisitor(VType* left,
                                                      VType* other) {
```

```
  return {left, other};
}

// The implementation of the assignment operation for a variant.
template <class VType>
struct CopyAssignVisitor {
  using DerivedType = typename VType::Variant;
  template <std::size_t NewIndex>
  void operator()(SizeT<NewIndex> /*new_i*/) const {
    using New =
        typename absl::variant_alternative<NewIndex, DerivedType>::type;

    if (left->index_ == NewIndex) {
      Access<NewIndex>(*left) = Access<NewIndex>(*right);
    } else if (std::is_nothrow_copy_constructible<New>::value ||
               !std::is_nothrow_move_constructible<New>::value) {
      Derived(*left).template emplace<NewIndex>(Access<NewIndex>(*right));
    } else {
      Derived(*left) = DerivedType(Derived(*right));
    }
  }

  void operator()(SizeT<absl::variant_npos> /*new_i*/) const {
    Destroy(*left);
  }

  VType* left;
  const VType* right;
};

template <class VType>
static CopyAssignVisitor<VType> MakeCopyAssignVisitor(VType* left,
                                                      const VType& other) {
  return {left, &other};
}

// The implementation of conversion-assignment operations for variant.
template <class Left, class QualifiedNew>
struct ConversionAssignVisitor {
  using NewIndex =
      variant_internal::IndexOfConstructedType<Left, QualifiedNew>;

  void operator()(SizeT<NewIndex::value> /*old_i*/
                  ) const {
    Access<NewIndex::value>(*left) = absl::forward<QualifiedNew>(other);
  }

  template <std::size_t OldIndex>
  void operator()(SizeT<OldIndex> /*old_i*/
                  ) const {
    using New =
        typename absl::variant_alternative<NewIndex::value, Left>::type;
    if (std::is_nothrow_constructible<New, QualifiedNew>::value ||
        !std::is_nothrow_move_constructible<New>::value) {
      left->template emplace<NewIndex::value>(
          absl::forward<QualifiedNew>(other));
    } else {
      // the standard says "equivalent to
      // operator=(variant(std::forward<T>(t)))", but we use `emplace` here
      // because the variant's move assignment operator could be deleted.
      left->template emplace<NewIndex::value>(
          New(absl::forward<QualifiedNew>(other)));
    }
  }

  Left* left;
  QualifiedNew&& other;
};

template <class Left, class QualifiedNew>
static ConversionAssignVisitor<Left, QualifiedNew>
MakeConversionAssignVisitor(Left* left, QualifiedNew&& qual) {
  return {left, absl::forward<QualifiedNew>(qual)};
}

// Backend for operations for `emplace()` which destructs `*self` then
// construct a new alternative with `Args...`.
template <std::size_t NewIndex, class Self, class... Args>
static typename absl::variant_alternative<NewIndex, Self>::type& Replace(
    Self* self, Args&&... args) {
  Destroy(*self);
  using New = typename absl::variant_alternative<NewIndex, Self>::type;
  New* const result = ::new (static_cast<void*>(&self->state_))
      New(absl::forward<Args>(args)...);
  self->index_ = NewIndex;
  return *result;
}

template <class LeftVariant, class QualifiedRightVariant>
struct InitFromVisitor {
  template <std::size_t NewIndex>
  void operator()(SizeT<NewIndex> /*new_i*/) const {
    using Alternative =
        typename variant_alternative<NewIndex, LeftVariant>::type;
    ::new (static_cast<void*>(&left->state_)) Alternative(
        Access<NewIndex>(std::forward<QualifiedRightVariant>(right)));
    }

    void operator()(SizeT<absl::variant_npos> /*new_i*/) const {
      // This space intentionally left blank.
    }
    LeftVariant* left;
    QualifiedRightVariant&& right;
  };
};

template <class Expected, class... T>
struct IndexOfImpl;

template <class Expected>
struct IndexOfImpl<Expected> {
  using IndexFromEnd = SizeT<0>;
  using MatchedIndexFromEnd = IndexFromEnd;
  using MultipleMatches = std::false_type;
};

template <class Expected, class Head, class... Tail>
struct IndexOfImpl<Expected, Head, Tail...> : IndexOfImpl<Expected, Tail...> {
  using IndexFromEnd =
      SizeT<IndexOfImpl<Expected, Tail...>::IndexFromEnd::value + 1>;
};

template <class Expected, class... Tail>
struct IndexOfImpl<Expected, Expected, Tail...>
    : IndexOfImpl<Expected, Tail...> {
  using IndexFromEnd =
      SizeT<IndexOfImpl<Expected, Tail...>::IndexFromEnd::value + 1>;
  using MatchedIndexFromEnd = IndexFromEnd;
  using MultipleMatches = std::integral_constant<
      bool, IndexOfImpl<Expected, Tail...>::MatchedIndexFromEnd::value != 0>;
};

template <class Expected, class... Types>
struct IndexOfMeta {
  using Results = IndexOfImpl<Expected, Types...>;
  static_assert(!Results::MultipleMatches::value,
                "Attempted to access a variant by specifying a type that "
                "matches more than one alternative.");
  static_assert(Results::MatchedIndexFromEnd::value != 0,
                "Attempted to access a variant by specifying a type that does "
                "not match any alternative.");
  using type = SizeT<sizeof...(Types) - Results::MatchedIndexFromEnd::value>;
};
```

```
template <class Expected, class... Types>
using IndexOf = typename IndexOfMeta<Expected, Types...>::type;

template <class Variant, class T, std::size_t CurrIndex>
struct UnambiguousIndexOfImpl;

// Terminating case encountered once we've checked all of the alternatives
template <class T, std::size_t CurrIndex>
struct UnambiguousIndexOfImpl<variant<>, T, CurrIndex> : SizeT<CurrIndex> {};

// Case where T is not Head
template <class Head, class... Tail, class T, std::size_t CurrIndex>
struct UnambiguousIndexOfImpl<variant<Head, Tail...>, T, CurrIndex>
    : UnambiguousIndexOfImpl<variant<Tail...>, T, CurrIndex + 1>::type {};

// Case where T is Head
template <class Head, class... Tail, std::size_t CurrIndex>
struct UnambiguousIndexOfImpl<variant<Head, Tail...>, Head, CurrIndex>
    : SizeT<UnambiguousIndexOfImpl<variant<Tail...>, Head, 0>::value ==
                sizeof...(Tail)
            ? CurrIndex
            : CurrIndex + sizeof...(Tail) + 1> {};

template <class Variant, class T>
struct UnambiguousIndexOf;

struct NoMatch {
  struct type {};
};

template <class... Alts, class T>
struct UnambiguousIndexOf<variant<Alts...>, T>
    : std::conditional<UnambiguousIndexOfImpl<variant<Alts...>, T, 0>::value !=
                           sizeof...(Alts),
                       UnambiguousIndexOfImpl<variant<Alts...>, T, 0>,
                       NoMatch>::type::type {};

template <class T, std::size_t /*Dummy*/>
using UnambiguousTypeOfImpl = T;

template <class Variant, class T>
using UnambiguousTypeOfT =
    UnambiguousTypeOfImpl<T, UnambiguousIndexOf<Variant, T>::value>;

template <class H, class... T>
class VariantStateBase;

// This is an implementation of the "imaginary function" that is described in
// [variant.ctor]
// It is used in order to determine which alternative to construct during
// initialization from some type T.
template <class Variant, std::size_t I = 0>
struct ImaginaryFun;

template <std::size_t I>
struct ImaginaryFun<variant<>, I> {
  static void Run() = delete;
};

template <class H, class... T, std::size_t I>
struct ImaginaryFun<variant<H, T...>, I> : ImaginaryFun<variant<T...>, I + 1> {
  using ImaginaryFun<variant<T...>, I + 1>::Run;

  // NOTE: const& and && are used instead of by-value due to lack of guaranteed
  // move elision of C++17. This may have other minor differences, but tests
  // pass.
  static SizeT<I> Run(const H&, SizeT<I>);
  static SizeT<I> Run(H&&, SizeT<I>);
};

// The following metafunctions are used in constructor and assignment
// constraints.
template <class Self, class T>
struct IsNeitherSelfNorInPlace : std::true_type {};

template <class Self>
struct IsNeitherSelfNorInPlace<Self, Self> : std::false_type {};

template <class Self, class T>
struct IsNeitherSelfNorInPlace<Self, in_place_type_t<T>> : std::false_type {};

template <class Self, std::size_t I>
struct IsNeitherSelfNorInPlace<Self, in_place_index_t<I>> : std::false_type {};

template <class Variant, class T, class = void>
struct ConversionIsPossibleImpl : std::false_type {};

template <class Variant, class T>
struct ConversionIsPossibleImpl<
    Variant, T,
    void_t<decltype(ImaginaryFun<Variant>::Run(std::declval<T>(), {}))>>
    : std::true_type {};

template <class Variant, class T>
struct ConversionIsPossible : ConversionIsPossibleImpl<Variant, T>::type {};
```

```
template <class Variant, class T>
struct IndexOfConstructedType<
    Variant, T,
    void_t<decltype(ImaginaryFun<Variant>::Run(std::declval<T>(), {}))>>
    : decltype(ImaginaryFun<Variant>::Run(std::declval<T>(), {})) {};

template <std::size_t... Is>
struct ContainsVariantNPos
    : absl::negation<std::is_same<  // NOLINT
          absl::integer_sequence<bool, 0 <= Is...>,
          absl::integer_sequence<bool, Is != absl::variant_npos...>>> {};

template <class Op, class... QualifiedVariants>
using RawVisitResult =
    absl::result_of_t<Op(VariantAccessResult<0, QualifiedVariants>...)>;

// NOTE: The spec requires that all return-paths yield the same type and is not
// SFINAE-friendly, so we can deduce the return type by examining the first
// result. If it's not callable, then we get an error, but are compliant and
// fast to compile.
// TODO(calabrese) Possibly rewrite in a way that yields better compile errors
// at the cost of longer compile-times.
template <class Op, class... QualifiedVariants>
struct VisitResultImpl {
  using type =
      absl::result_of_t<Op(VariantAccessResult<0, QualifiedVariants>...)>;
};

// Done in two steps intentionally so that we don't cause substitution to fail.
template <class Op, class... QualifiedVariants>
using VisitResult = typename VisitResultImpl<Op, QualifiedVariants...>::type;

template <class Op, class... QualifiedVariants>
struct PerformVisitation {
  using ReturnType = VisitResult<Op, QualifiedVariants...>;

  template <std::size_t... Is>
  constexpr ReturnType operator()(SizeT<Is>... indices) const {
    return Run(typename ContainsVariantNPos<Is...>::type{},
               absl::index_sequence_for<QualifiedVariants...>(), indices...);
  }

  template <std::size_t... TupIs, std::size_t... Is>
  constexpr ReturnType Run(std::false_type /*has_valueless*/,
                           index_sequence<TupIs...>, SizeT<Is>...) const {
    static_assert(
        std::is_same<ReturnType,
                     absl::result_of_t<Op(VariantAccessResult<
                                     Is, QualifiedVariants>...)>>::value,
        "All visitation overloads must have the same return type.");
    return absl::base_internal::Invoke(
        absl::forward<Op>(op),
        VariantCoreAccess::Access<Is>(
            absl::forward<QualifiedVariants>(std::get<TupIs>(variant_tup)))...);
  }

  template <std::size_t... TupIs, std::size_t... Is>
  [[noreturn]] ReturnType Run(std::true_type /*has_valueless*/,
                              index_sequence<TupIs...>, SizeT<Is>...) const {
    absl::variant_internal::ThrowBadVariantAccess();
  }

  // TODO(calabrese) Avoid using a tuple, which causes lots of instantiations
  // Attempts using lambda variadic captures fail on current GCC.
  std::tuple<QualifiedVariants&&...> variant_tup;
  Op&& op;
};

template <class... T>
union Union;

// We want to allow for variant<> to be trivial. For that, we need the default
// constructor to be trivial, which means we can't define it ourselves.
// Instead, we use a non-default constructor that takes NoopConstructorTag
// that doesn't affect the triviality of the types.
struct NoopConstructorTag {};

template <std::size_t I>
struct EmplaceTag {};

template <>
union Union<> {
  constexpr explicit Union(NoopConstructorTag) noexcept {}
};

// Suppress bogus warning on MSVC: MSVC complains that Union<T...> has a defined
// deleted destructor from the `std::is_destructible` check below.
#ifdef _MSC_VER
#pragma warning(push)
#pragma warning(disable : 4624)
#endif  // _MSC_VER

template <class Head, class... Tail>
union Union<Head, Tail...> {
  using TailUnion = Union<Tail...>;
  explicit constexpr Union(NoopConstructorTag /*tag*/) noexcept
      : tail(NoopConstructorTag()) {}

  template <class... P>
  explicit constexpr Union(EmplaceTag<0>, P&&... args)
      : head(absl::forward<P>(args)...) {}

  template <std::size_t I, class... P>
  explicit constexpr Union(EmplaceTag<I>, P&&... args)
      : tail(EmplaceTag<I - 1>{}, absl::forward<P>(args)...) {}

  Head head;
  TailUnion tail;
};

#ifdef _MSC_VER
#pragma warning(pop)
#endif  // _MSC_VER

// TODO(calabrese) Just contain a Union in this union (certain configs fail).
template <class... T>
union DestructibleUnionImpl;

template <>
union DestructibleUnionImpl<> {
  constexpr explicit DestructibleUnionImpl(NoopConstructorTag) noexcept {}
};

template <class Head, class... Tail>
union DestructibleUnionImpl<Head, Tail...> {
  using TailUnion = DestructibleUnionImpl<Tail...>;

  explicit constexpr DestructibleUnionImpl(NoopConstructorTag /*tag*/) noexcept
      : tail(NoopConstructorTag()) {}

  template <class... P>
  explicit constexpr DestructibleUnionImpl(EmplaceTag<0>, P&&... args)
      : head(absl::forward<P>(args)...) {}

  template <std::size_t I, class... P>
  explicit constexpr DestructibleUnionImpl(EmplaceTag<I>, P&&... args)
      : tail(EmplaceTag<I - 1>{}, absl::forward<P>(args)...) {}

  ~DestructibleUnionImpl() {}

  Head head;
  TailUnion tail;
};
```

```
// This union type is destructible even if one or more T are not trivially
// destructible. In the case that all T are trivially destructible, then so is
// this resultant type.
template <class... T>
using DestructibleUnion =
    absl::conditional_t<std::is_destructible<Union<T...>>::value, Union<T...>,
                        DestructibleUnionImpl<T...>>;

// Deepest base, containing the actual union and the discriminator
template <class H, class... T>
class VariantStateBase {
 protected:
  using Variant = variant<H, T...>;

  template <class LazyH = H,
            class ConstructibleH = absl::enable_if_t<
                std::is_default_constructible<LazyH>::value, LazyH>>
  constexpr VariantStateBase() noexcept(
      std::is_nothrow_default_constructible<ConstructibleH>::value)
      : state_(EmplaceTag<0>()), index_(0) {}

  template <std::size_t I, class... P>
  explicit constexpr VariantStateBase(EmplaceTag<I> tag, P&&... args)
      : state_(tag, absl::forward<P>(args)...), index_(I) {}

  explicit constexpr VariantStateBase(NoopConstructorTag)
      : state_(NoopConstructorTag()), index_(variant_npos) {}

  void destroy() {}  // Does nothing (shadowed in child if non-trivial)

  DestructibleUnion<H, T...> state_;
  std::size_t index_;
};

using absl::internal::identity;

// OverloadSet::Overload() is a unary function which is overloaded to
// take any of the element types of the variant, by reference-to-const.
// The return type of the overload on T is identity<T>, so that you
// can statically determine which overload was called.
//
// Overload() is not defined, so it can only be called in unevaluated
// contexts.
template <typename... Ts>
struct OverloadSet;

template <typename T, typename... Ts>
```

```
struct OverloadSet<T, Ts...> : OverloadSet<Ts...> {
  using Base = OverloadSet<Ts...>;
  static identity<T> Overload(const T&);
  using Base::Overload;
};

template <>
struct OverloadSet<> {
  // For any case not handled above.
  static void Overload(...);
};

template <class T>
using LessThanResult = decltype(std::declval<T>() < std::declval<T>());

template <class T>
using GreaterThanResult = decltype(std::declval<T>() > std::declval<T>());

template <class T>
using LessThanOrEqualResult = decltype(std::declval<T>() <= std::declval<T>());

template <class T>
using GreaterThanOrEqualResult =
    decltype(std::declval<T>() >= std::declval<T>());

template <class T>
using EqualResult = decltype(std::declval<T>() == std::declval<T>());

template <class T>
using NotEqualResult = decltype(std::declval<T>() != std::declval<T>());

using type_traits_internal::is_detected_convertible;

template <class... T>
using RequireAllHaveEqualT = absl::enable_if_t<
    absl::conjunction<is_detected_convertible<bool, EqualResult, T>...>::value,
    bool>;

template <class... T>
using RequireAllHaveNotEqualT =
    absl::enable_if_t<absl::conjunction<is_detected_convertible<
                          bool, NotEqualResult, T>...>::value,
                      bool>;

template <class... T>
using RequireAllHaveLessThanT =
    absl::enable_if_t<absl::conjunction<is_detected_convertible<
                          bool, LessThanResult, T>...>::value,
                  bool>;

template <class... T>
using RequireAllHaveLessThanOrEqualT =
   absl::enable_if_t<absl::conjunction<is_detected_convertible<
                     bool, LessThanOrEqualResult, T>...>::value,
                  bool>;

template <class... T>
using RequireAllHaveGreaterThanOrEqualT =
   absl::enable_if_t<absl::conjunction<is_detected_convertible<
                     bool, GreaterThanOrEqualResult, T>...>::value,
                  bool>;

template <class... T>
using RequireAllHaveGreaterThanT =
   absl::enable_if_t<absl::conjunction<is_detected_convertible<
                     bool, GreaterThanResult, T>...>::value,
                  bool>;

// Helper template containing implementations details of variant that can't go
// in the private section. For convenience, this takes the variant type as a
// single template parameter.
template <typename T>
struct VariantHelper;

template <typename... Ts>
struct VariantHelper<variant<Ts...>> {
  // Type metafunction which returns the element type selected if
  // OverloadSet::Overload() is well-formed when called with argument type U.
  template <typename U>
  using BestMatch = decltype(
     variant_internal::OverloadSet<Ts...>::Overload(std::declval<U>()));

  // Type metafunction which returns true if OverloadSet::Overload() is
  // well-formed when called with argument type U.
  // CanAccept can't be just an alias because there is a MSVC bug on parameter
  // pack expansion involving decltype.
  template <typename U>
  struct CanAccept :
     std::integral_constant<bool, !std::is_void<BestMatch<U>>::value> {};

  // Type metafunction which returns true if Other is an instantiation of
  // variant, and variants's converting constructor from Other will be
  // well-formed. We will use this to remove constructors that would be
  // ill-formed from the overload set.
  template <typename Other>
  struct CanConvertFrom;
  template <typename... Us>
  struct CanConvertFrom<variant<Us...>>
      : public absl::conjunction<CanAccept<Us>...> {};
};

// A type with nontrivial copy ctor and trivial move ctor.
struct TrivialMoveOnly {
  TrivialMoveOnly(TrivialMoveOnly&&) = default;
};

// Trait class to detect whether a type is trivially move constructible.
// A union's defaulted copy/move constructor is deleted if any variant member's
// copy/move constructor is nontrivial.
template <typename T>
struct IsTriviallyMoveConstructible:
  std::is_move_constructible<Union<T, TrivialMoveOnly>> {};

// To guarantee triviality of all special-member functions that can be trivial,
// we use a chain of conditional bases for each one.
// The order of inheritance of bases from child to base are logically:
//
// variant
// VariantCopyAssignBase
// VariantMoveAssignBase
// VariantCopyBase
// VariantMoveBase
// VariantStateBaseDestructor
// VariantStateBase
//
// Note that there is a separate branch at each base that is dependent on
// whether or not that corresponding special-member-function can be trivial in
// the resultant variant type.

template <class... T>
class VariantStateBaseDestructorNontrivial;

template <class... T>
class VariantMoveBaseNontrivial;

template <class... T>
class VariantCopyBaseNontrivial;

template <class... T>
class VariantMoveAssignBaseNontrivial;

template <class... T>
class VariantCopyAssignBaseNontrivial;
```

```
// Base that is dependent on whether or not the destructor can be trivial.
template <class... T>
using VariantStateBaseDestructor =
  absl::conditional_t<std::is_destructible<Union<T...>>::value,
                      VariantStateBase<T...>,
                      VariantStateBaseDestructorNontrivial<T...>>;

// Base that is dependent on whether or not the move-constructor can be
// implicitly generated by the compiler (trivial or deleted).
// Previously we were using `std::is_move_constructible<Union<T...>>` to check
// whether all Ts have trivial move constructor, but it ran into a GCC bug:
// https://gcc.gnu.org/bugzilla/show_bug.cgi?id=84866
// So we have to use a different approach (i.e. `HasTrivialMoveConstructor`) to
// work around the bug.
template <class... T>
using VariantMoveBase = absl::conditional_t<
  absl::disjunction<
    absl::negation<absl::conjunction<std::is_move_constructible<T>...>>,
    absl::conjunction<IsTriviallyMoveConstructible<T>...>>::value,
  VariantStateBaseDestructor<T...>, VariantMoveBaseNontrivial<T...>>;

// Base that is dependent on whether or not the copy-constructor can be trivial.
template <class... T>
using VariantCopyBase = absl::conditional_t<
  absl::disjunction<
    absl::negation<absl::conjunction<std::is_copy_constructible<T>...>>,
    std::is_copy_constructible<Union<T...>>>::value,
  VariantMoveBase<T...>, VariantCopyBaseNontrivial<T...>>;

// Base that is dependent on whether or not the move-assign can be trivial.
template <class... T>
using VariantMoveAssignBase = absl::conditional_t<
  absl::disjunction<
    absl::conjunction<absl::is_move_assignable<Union<T...>>,
                      std::is_move_constructible<Union<T...>>,
                      std::is_destructible<Union<T...>>>,
    absl::negation<absl::conjunction<std::is_move_constructible<T>...,
                                     // Note: We're not qualifying this with
                                     // absl:: because it doesn't compile
                                     // under MSVC.
                                     is_move_assignable<T>...>>>::value,
  VariantCopyBase<T...>, VariantMoveAssignBaseNontrivial<T...>>;

// Base that is dependent on whether or not the copy-assign can be trivial.
template <class... T>
using VariantCopyAssignBase = absl::conditional_t<
  absl::disjunction<
```

```
        absl::conjunction<absl::is_copy_assignable<Union<T...>>,
                          std::is_copy_constructible<Union<T...>>,
                          std::is_destructible<Union<T...>>>,
        absl::negation<absl::conjunction<std::is_copy_constructible<T>...,
                                         // Note: We're not qualifying this with
                                         // absl:: because it doesn't compile
                                         // under MSVC.
                                         is_copy_assignable<T>...>>>::value,
    VariantMoveAssignBase<T...>, VariantCopyAssignBaseNontrivial<T...>>;

template <class... T>
using VariantBase = VariantCopyAssignBase<T...>;

template <class... T>
class VariantStateBaseDestructorNontrivial : protected VariantStateBase<T...> {
 private:
  using Base = VariantStateBase<T...>;

 protected:
  using Base::Base;

  VariantStateBaseDestructorNontrivial() = default;
  VariantStateBaseDestructorNontrivial(VariantStateBaseDestructorNontrivial&&) =
      default;
  VariantStateBaseDestructorNontrivial(
      const VariantStateBaseDestructorNontrivial&) = default;
  VariantStateBaseDestructorNontrivial& operator=(
      VariantStateBaseDestructorNontrivial&&) = default;
  VariantStateBaseDestructorNontrivial& operator=(
      const VariantStateBaseDestructorNontrivial&) = default;

  struct Destroyer {
    template <std::size_t I>
    void operator()(SizeT<I> i) const {
      using Alternative =
          typename absl::variant_alternative<I, variant<T...>>::type;
      variant_internal::AccessUnion(self->state_, i).~Alternative();
    }

    void operator()(SizeT<absl::variant_npos> /*i*/) const {
      // This space intentionally left blank
    }

    VariantStateBaseDestructorNontrivial* self;
  };

  void destroy() { VisitIndices<sizeof...(T)>::Run(Destroyer{this}, index_); }
  ~VariantStateBaseDestructorNontrivial() { destroy(); }

 protected:
  using Base::index_;
  using Base::state_;
};

template <class... T>
class VariantMoveBaseNontrivial : protected VariantStateBaseDestructor<T...> {
 private:
  using Base = VariantStateBaseDestructor<T...>;

 protected:
  using Base::Base;

  struct Construct {
    template <std::size_t I>
    void operator()(SizeT<I> i) const {
      using Alternative =
          typename absl::variant_alternative<I, variant<T...>>::type;
      ::new (static_cast<void*>(&self->state_)) Alternative(
          variant_internal::AccessUnion(absl::move(other->state_), i));
    }

    void operator()(SizeT<absl::variant_npos> /*i*/) const {}

    VariantMoveBaseNontrivial* self;
    VariantMoveBaseNontrivial* other;
  };

  VariantMoveBaseNontrivial() = default;
  VariantMoveBaseNontrivial(VariantMoveBaseNontrivial&& other) noexcept(
      absl::conjunction<std::is_nothrow_move_constructible<T>...>::value)
      : Base(NoopConstructorTag()) {
    VisitIndices<sizeof...(T)>::Run(Construct{this, &other}, other.index_);
    index_ = other.index_;
  }

  VariantMoveBaseNontrivial(VariantMoveBaseNontrivial const&) = default;

  VariantMoveBaseNontrivial& operator=(VariantMoveBaseNontrivial&&) = default;
  VariantMoveBaseNontrivial& operator=(VariantMoveBaseNontrivial const&) =
      default;

 protected:
  using Base::index_;
  using Base::state_;
};
```

```
template <class... T>
class VariantCopyBaseNontrivial : protected VariantMoveBase<T...> {
 private:
  using Base = VariantMoveBase<T...>;

 protected:
  using Base::Base;

  VariantCopyBaseNontrivial() = default;
  VariantCopyBaseNontrivial(VariantCopyBaseNontrivial&&) = default;

  struct Construct {
    template <std::size_t I>
    void operator()(SizeT<I> i) const {
      using Alternative =
          typename absl::variant_alternative<I, variant<T...>>::type;
      ::new (static_cast<void*>(&self->state_))
          Alternative(variant_internal::AccessUnion(other->state_, i));
    }

    void operator()(SizeT<absl::variant_npos> /*i*/) const {}

    VariantCopyBaseNontrivial* self;
    const VariantCopyBaseNontrivial* other;
  };

  VariantCopyBaseNontrivial(VariantCopyBaseNontrivial const& other)
      : Base(NoopConstructorTag()) {
    VisitIndices<sizeof...(T)>::Run(Construct{this, &other}, other.index_);
    index_ = other.index_;
  }

  VariantCopyBaseNontrivial& operator=(VariantCopyBaseNontrivial&&) = default;
  VariantCopyBaseNontrivial& operator=(VariantCopyBaseNontrivial const&) =
      default;

 protected:
  using Base::index_;
  using Base::state_;
};

template <class... T>
class VariantMoveAssignBaseNontrivial : protected VariantCopyBase<T...> {
  friend struct VariantCoreAccess;

 private:
  using Base = VariantCopyBase<T...>;
 protected:
  using Base::Base;

  VariantMoveAssignBaseNontrivial() = default;
  VariantMoveAssignBaseNontrivial(VariantMoveAssignBaseNontrivial&&) = default;
  VariantMoveAssignBaseNontrivial(const VariantMoveAssignBaseNontrivial&) =
      default;
  VariantMoveAssignBaseNontrivial& operator=(
      VariantMoveAssignBaseNontrivial const&) = default;

    VariantMoveAssignBaseNontrivial&
    operator=(VariantMoveAssignBaseNontrivial&& other) noexcept(
        absl::conjunction<std::is_nothrow_move_constructible<T>...,
                          std::is_nothrow_move_assignable<T>...>::value) {
      VisitIndices<sizeof...(T)>::Run(
          VariantCoreAccess::MakeMoveAssignVisitor(this, &other), other.index_);
      return *this;
    }

 protected:
  using Base::index_;
  using Base::state_;
};

template <class... T>
class VariantCopyAssignBaseNontrivial : protected VariantMoveAssignBase<T...> {
  friend struct VariantCoreAccess;

 private:
  using Base = VariantMoveAssignBase<T...>;

 protected:
  using Base::Base;

  VariantCopyAssignBaseNontrivial() = default;
  VariantCopyAssignBaseNontrivial(VariantCopyAssignBaseNontrivial&&) = default;
  VariantCopyAssignBaseNontrivial(const VariantCopyAssignBaseNontrivial&) =
      default;
  VariantCopyAssignBaseNontrivial& operator=(
      VariantCopyAssignBaseNontrivial&&) = default;

    VariantCopyAssignBaseNontrivial& operator=(
        const VariantCopyAssignBaseNontrivial& other) {
      VisitIndices<sizeof...(T)>::Run(
          VariantCoreAccess::MakeCopyAssignVisitor(this, other), other.index_);
      return *this;
    }
 protected:
  using Base::index_;
  using Base::state_;
};

////////////////////////////////////////
// Visitors for Comparison Operations //
////////////////////////////////////////

template <class... Types>
struct EqualsOp {
  const variant<Types...>* v;
  const variant<Types...>* w;

  constexpr bool operator()(SizeT<absl::variant_npos> /*v_i*/) const {
    return true;
  }

  template <std::size_t I>
  constexpr bool operator()(SizeT<I> /*v_i*/) const {
    return VariantCoreAccess::Access<I>(*v) == VariantCoreAccess::Access<I>(*w);
  }
};

template <class... Types>
struct NotEqualsOp {
  const variant<Types...>* v;
  const variant<Types...>* w;

  constexpr bool operator()(SizeT<absl::variant_npos> /*v_i*/) const {
    return false;
  }

  template <std::size_t I>
  constexpr bool operator()(SizeT<I> /*v_i*/) const {
    return VariantCoreAccess::Access<I>(*v) != VariantCoreAccess::Access<I>(*w);
  }
};

template <class... Types>
struct LessThanOp {
  const variant<Types...>* v;
  const variant<Types...>* w;

  constexpr bool operator()(SizeT<absl::variant_npos> /*v_i*/) const {
    return false;
  }
  template <std::size_t I>
  constexpr bool operator()(SizeT<I> /*v_i*/) const {
    return VariantCoreAccess::Access<I>(*v) < VariantCoreAccess::Access<I>(*w);
  }
};

template <class... Types>
struct GreaterThanOp {
  const variant<Types...>* v;
  const variant<Types...>* w;

  constexpr bool operator()(SizeT<absl::variant_npos> /*v_i*/) const {
    return false;
  }

  template <std::size_t I>
  constexpr bool operator()(SizeT<I> /*v_i*/) const {
    return VariantCoreAccess::Access<I>(*v) > VariantCoreAccess::Access<I>(*w);
  }
};

template <class... Types>
struct LessThanOrEqualsOp {
  const variant<Types...>* v;
  const variant<Types...>* w;

  constexpr bool operator()(SizeT<absl::variant_npos> /*v_i*/) const {
    return true;
  }

  template <std::size_t I>
  constexpr bool operator()(SizeT<I> /*v_i*/) const {
    return VariantCoreAccess::Access<I>(*v) <= VariantCoreAccess::Access<I>(*w);
  }
};

template <class... Types>
struct GreaterThanOrEqualsOp {
  const variant<Types...>* v;
  const variant<Types...>* w;

  constexpr bool operator()(SizeT<absl::variant_npos> /*v_i*/) const {
    return true;
  }

  template <std::size_t I>
  constexpr bool operator()(SizeT<I> /*v_i*/) const {
    return VariantCoreAccess::Access<I>(*v) >= VariantCoreAccess::Access<I>(*w);
  }
};

// Precondition: v.index() == w.index();
template <class... Types>
struct SwapSameIndex {
  variant<Types...>* v;
  variant<Types...>* w;
  template <std::size_t I>
  void operator()(SizeT<I>) const {
    type_traits_internal::Swap(VariantCoreAccess::Access<I>(*v),
                               VariantCoreAccess::Access<I>(*w));
  }

  void operator()(SizeT<variant_npos>) const {}
};

// TODO(calabrese) do this from a different namespace for proper adl usage
template <class... Types>
struct Swap {
  variant<Types...>* v;
  variant<Types...>* w;

  void generic_swap() const {
    variant<Types...> tmp(std::move(*w));
    VariantCoreAccess::Destroy(*w);
    VariantCoreAccess::InitFrom(*w, std::move(*v));
    VariantCoreAccess::Destroy(*v);
    VariantCoreAccess::InitFrom(*v, std::move(tmp));
  }

  void operator()(SizeT<absl::variant_npos> /*w_i*/) const {
    if (!v->valueless_by_exception()) {
      generic_swap();
    }
  }

  template <std::size_t Wi>
  void operator()(SizeT<Wi> /*w_i*/) {
    if (v->index() == Wi) {
      VisitIndices<sizeof...(Types)>::Run(SwapSameIndex<Types...>{v, w}, Wi);
    } else {
      generic_swap();
    }
  }
};
```

```
template <typename Variant, typename = void, typename... Ts>
struct VariantHashBase {
  VariantHashBase() = delete;
  VariantHashBase(const VariantHashBase&) = delete;
  VariantHashBase(VariantHashBase&&) = delete;
  VariantHashBase& operator=(const VariantHashBase&) = delete;
  VariantHashBase& operator=(VariantHashBase&&) = delete;
};

struct VariantHashVisitor {
  template <typename T>
  size_t operator()(const T& t) {
    return std::hash<T>{}(t);
  }
};

template <typename Variant, typename... Ts>
struct VariantHashBase<Variant,
                       absl::enable_if_t<absl::conjunction<
                           type_traits_internal::IsHashable<Ts>...>::value>,
                       Ts...> {
  using argument_type = Variant;
  using result_type = size_t;
  size_t operator()(const Variant& var) const {
    type_traits_internal::AssertHashEnabled<Ts...>();
    if (var.valueless_by_exception()) {
      return 239799884;
    }
    size_t result = VisitIndices<variant_size<Variant>::value>::Run(
        PerformVisitation<VariantHashVisitor, const Variant&>{
            std::forward_as_tuple(var), VariantHashVisitor{}},
        var.index());
    // Combine the index and the hash result in order to distinguish
    // std::variant<int, int> holding the same value as different alternative.
    return result ^ var.index();
  }
};

}  // namespace variant_internal
ABSL_NAMESPACE_END
}  // namespace absl
```

The Abseil synchronization library

The Abseil synchronization library contains implementation of various synchronization primitives which we will take a look at: mutex, barrier, notification and blocking counter

Mutex class

Barrier class

Notification class

Blocking counter

Tensorflow Internal structures, containers and interfaces

Internal Data Structures and Synchronization Primitives

The internal data structures which we will look into in this section are gtl::FlatMap and gtl::FlatSet

Internal helper structs FlatMap::Bucket and FlatSet::Bucket

Both gtl::FlatMap and gtl::FlatSet declare and define their own nested container Bucket storing a single element of the corresponding structure. Both FlatMap::Bucket and FlatSet::Bucket are referenced in common internal container gtl::internal::FlatRep which is discussed in the next parapgraph. As we need to be familiar with both FlatMap::Bucket and FlatSet::Bucket before we move to gtl::internal::FlatRep, gtl::FlatMap and ftl::FlatSet we start our journey in the Tensorflow's internal structures with Code Snippet: FlatMap::Bucket and Code Snippet: FlatSet::Bucket.

The first thing to notice is that the type of the key and the type of the value are template parameters of the enclosing class gtl::FlatMap, and that the arrays which hold the markers, keys and values are of size 8 which is defined in the constexpr FlatRep::kWidth initializer. The internal storage of Bucket is a struct member wrapped in a union which facilities lazy on-first-access construction. As the comment below points the Bucket stores kWidth triplets <maker, key, value> and provides simple interface for manipulating those. Bucket::InitVal(uint32 i, V&& v) initializes the Value in i-th position by using perfect forwarding. Bucket::CopyFrom(uint32 i, Bucket* src, uint32 src_index) copies the element on the src_index position of the internal storage array of the specified Bucket instance (which has FlatRep::kWidth elements) to the element on the i-th position in the current Bucket instance using placement new expression. Similarly Bucket::MoveFrom(uint32 i, Bucket* src, uint32 src_index) moves the element on the src_index position in the Bucket instance *src* to the element in the i-th position of the current Bucket instance. The marker has to be >= 2 because the values 0 and 1 are reserved for empty and deleted elements as the private gtl::FlatRep anonymous enum indicates

```
enum { kEmpty = 0, kDeleted = 1 };  // Special markers for an entry.
```

The class gtl::FlatSet::Bucket is organized similarly to gtl::FlatMap::Bucket except that it is tailored to represent a set instead of a map which means the Value from the triplet <marker, key, value> is missing which now it becomes the tuple <marker, key>.

Code Snippet: FlatMap::Bucket

```
template <typename Key, typename Val, class Hash = hash<Key>,
        class Eq = std::equal_to<Key>>
class FlatMap {
  . . .
// Bucket stores kWidth <marker, key, value> triples.
// The data is organized as three parallel arrays to reduce padding.
struct Bucket {
   uint8 marker[Rep::kWidth];

   // Wrap keys and values in union to control construction and destruction.
   union Storage {
     struct {
       Key key[Rep::kWidth];
       Val val[Rep::kWidth];
     };
     Storage() {}
     ~Storage() {}
   } storage;

   Key& key(uint32 i) {
     DCHECK_GE(marker[i], 2);
     return storage.key[i];
   }
   Val& val(uint32 i) {
     DCHECK_GE(marker[i], 2);
     return storage.val[i];
   }
   template <typename V>
   void InitVal(uint32 i, V&& v) {
     new (&storage.val[i]) Val(std::forward<V>(v));
   }
   void Destroy(uint32 i) {
     storage.key[i].Key::~Key();
     storage.val[i].Val::~Val();
   }
   void MoveFrom(uint32 i, Bucket* src, uint32 src_index) {
     new (&storage.key[i]) Key(std::move(src->storage.key[src_index]));
     new (&storage.val[i]) Val(std::move(src->storage.val[src_index]));
   }
   void CopyFrom(uint32 i, Bucket* src, uint32 src_index) {
     new (&storage.key[i]) Key(src->storage.key[src_index]);
     new (&storage.val[i]) Val(src->storage.val[src_index]);
   }
};  // nested struct FlatMap::Bucket declaration
 . . .
};  // class FlatMap declaration
```

Code Snippet: FlatSet::Bucket

```
template <typename Key, class Hash = hash<Key>, class Eq = std::equal_to<Key>>
class FlatSet {
  . . .

 // Bucket stores kWidth <marker, key> tuples.
 // The data is organized as three parallel arrays to reduce padding.
 struct Bucket {
  uint8 marker[Rep::kWidth];

  // Wrap keys in union to control construction and destruction.
  union Storage {
   Key key[Rep::kWidth];
   Storage() {}
   ~Storage() {}
  } storage;

  Key& key(uint32 i) {
   DCHECK_GE(marker[i], 2);
   return storage.key[i];
  }
  void Destroy(uint32 i) { storage.key[i].Key::~Key(); }
  void MoveFrom(uint32 i, Bucket* src, uint32 src_index) {
   new (&storage.key[i]) Key(std::move(src->storage.key[src_index]));
  }
  void CopyFrom(uint32 i, Bucket* src, uint32 src_index) {
   new (&storage.key[i]) Key(src->storage.key[src_index]);
  }
 };  // nested struct FlatSet::Bucket declaration
 . . .
};  // class FlatSet declaration
```

Class gtl::internal::FlatRep - the internal representation for FlatMap and FlatSet

FlatRep is an internal representation container introduced in FlatMap and FlatSet by the using declaration

```
 using Rep = internal::FlatRep<Key, Bucket, Hash, Eq>;
```

it is declared in tensorflow/core/lib/gtl/flatrep.h. In order to understand FlatMap and FlatSet we will go over FlatRep first. The three template type parameters of class FlatRep are Key, Bucket, Hash, and Eq. Their usage will become clear as we go through the code. FlatRep is the internal representation of FlatSet and FlatMap which is a flat array of entries split into Buckets. FlatMap passes a Bucket which contains Key and Value entries while FlatSet passes a Bucket which contains Key only.

First we will look at the member void FlatRep::Init(size_t N) shown on the snippet below. The code below initializes the required number of Bucket instances on the heap which will accommodate N elements, noting that each Bucket stores kWidth=8 elements. For each of the n new Buckets it sets the marker values to kEmpty indicating an empty element. Here are few interesting members initialized inside FlatRep::Init – FlatRep::lglen_ is the binary logarithm of the number of Buckets calculated to hold N elements. FlatRep::mask_ is set to the index of the last Bucket instance: capacity – 1. There are two more factors: FlatRep::grow_ = capacity * 0.8 and FlatRep::shrink_ = grow_ * 0.4 which control the growth rate and the shrink rate of the array of Buckets.

```
 void Init(size_t N) {
  // Make enough room for N elements.
  size_t lg = 0;  // Smallest table is just one bucket.
  while (N >= 0.8 * ((1 << lg) * kWidth)) {
    lg++;
  }
  const size_t n = (1 << lg);
  Bucket* array = new Bucket[n];
  for (size_t i = 0; i < n; i++) {
    Bucket* b = &array[i];
    memset(b->marker, kEmpty, kWidth);
  }
  const size_t capacity = (1 << lg) * kWidth;
  lglen_ = lg;
  mask_ = capacity - 1;
  array_ = array;
  end_ = array + n;
  not_empty_ = 0;
  deleted_ = 0;
  grow_ = static_cast<size_t>(capacity * 0.8);
  if (lg == 0) {
    // Already down to one bucket; no more shrinking.
    shrink_ = 0;
  } else {
    shrink_ = static_cast<size_t>(grow_ * 0.4);  // Must be less than 0.5
  }
 }
```

There are some interesting methods in the class FlatRep. One of those is SearchResult Find(const Key& k) const. It finds a bucket/index for key k. The process of finding a bucket starts with hashing the key by applying the hashing function FlatRep::hash_ supplied as constructor parameter with type specified by the given template type. Then we take the lower 8 bits of the result from applying hash_ and make sure it is at least with value 2 thus avoiding kEmpty and kDeleted marker values. Those modified 8 bits will be stored in the field Bucket::marker to help speed up comparisons. Recall from FlatRep::Init(size_t) that mask_ was set to hold capacity – 1 of elements where capacity is given with (1 << lg) * kWidth. Here lg is the smallest integer such that N < 0.8 * ((1 << lg) * kWidth). The value index = (h >> 8) & mask_ which corresponds to the bits higher than the lower 8 bits hold the bucket number and the element index-in-bucket. The element index-in-bucket is obtained as the lowest 3 bits from the higher 24 bits of the hash by using the equation b = index & (kWidth - 1). And finally the bucket number is obtained by shifting right the lower 3 bits as index >> kBase. If the marker of the obtained Bucket is equal to the marker of the calculated hash and the key of that Bucket is the same as the key specified as an argument then this Bucket is what we are searching for. Otherwise, if the marker of the referenced Bucket is set to kEmpty we return nullptr and the found flag set to false indicating that the search with this key failed. If the Bucket is not empty and the marker or the key does not match we move to the next bucket index by using NextIndex. Note that contrary to the code comment NextIndex does not use quadratic probing but rather use the masked increment formula (index + num_probes) & mask_ to obtain the index of the next Bucket candidate.

```
 // Avoid kEmpty and kDeleted markers when computing hash values to
 // store in Bucket::marker[].
 static uint32 Marker(uint32 hb) { return hb + (hb < 2 ? 2 : 0); }


// Hash value is partitioned as follows:
// 1. Bottom 8 bits are stored in bucket to help speed up comparisons.
// 2. Next 3 bits give index inside bucket.
// 3. Remaining bits give bucket number.

// Find bucket/index for key k.
SearchResult Find(const Key& k) const {
   size_t h = hash_(k);
   const uint32 marker = Marker(h & 0xff);
   size_t index = (h >> 8) & mask_;  // Holds bucket num and index-in-bucket
   uint32 num_probes = 1;           // Needed for quadratic probing
   while (true) {
     uint32 bi = index & (kWidth - 1);
     Bucket* b = &array_[index >> kBase];
     const uint32 x = b->marker[bi];
     if (x == marker && equal_(b->key(bi), k)) {
       return {true, b, bi};
     } else if (x == kEmpty) {
       return {false, nullptr, 0};
     }
     index = NextIndex(index, num_probes);
     num_probes++;
   }
 }
```

The next interesting method SearchResult FindOrInsert(KeyType&& k) is shown below.

First notice the templetized key type and the rvalue reference for the corresponding argument which facilitate perfect forwarding i.e. lvalue references of the key are forwarded as lvalue references and rvalue references are forwarded as rvalue references. In the first lines of this method the hash value , the bucket number and the index-in-bucket are obtained in a similar fashion as in SearchResult Find(const Key& k) const. Next we start probing in a loop for Bucket with matching marker and key and if one is found we return immediately that instance. If a Bucket instance is found with in-bucket-index marker set to kDeleted we store a reference to it and its number and continue probing. If the next probed Bucket and in-bucket-index render marker set to kEmpty then we move the given key to the Key field of the last kDeleted element found from the previous probing iteration. If there is not kDeleted element found in a previous iteration then mark the current element as not empty, transfer the given Key to it and return it with the found flag set to false.

```
// Find bucket/index for key k, creating a new one if necessary.
//
// KeyType is a template parameter so that k's type is deduced and it
// becomes a universal reference which allows the key initialization
// below to use an rvalue constructor if available.
template <typename KeyType>
SearchResult FindOrInsert(KeyType&& k) {
  size_t h = hash_(k);
  const uint32 marker = Marker(h & 0xff);
  size_t index = (h >> 8) & mask_;  // Holds bucket num and index-in-bucket
  uint32 num_probes = 1;            // Needed for quadratic probing
  Bucket* del = nullptr;            // First encountered deletion for kInsert
  uint32 di = 0;
  while (true) {
    uint32 bi = index & (kWidth - 1);
    Bucket* b = &array_[index >> kBase];
    const uint32 x = b->marker[bi];
    if (x == marker && equal_(b->key(bi), k)) {
      return {true, b, bi};
    } else if (!del && x == kDeleted) {
      // Remember deleted index to use for insertion.
      del = b;
      di = bi;
    } else if (x == kEmpty) {
      if (del) {
        // Store in the first deleted slot we encountered
        b = del;
        bi = di;
        deleted_--;  // not_empty_ does not change
      } else {
        not_empty_++;
      }
      b->marker[bi] = marker;
      new (&b->key(bi)) Key(std::forward<KeyType>(k));
      return {false, b, bi};
    }
    index = NextIndex(index, num_probes);
    num_probes++;
  }
}
```

An interesting set of FlatRep methods which need mention are Code Snippet: CopyEntries and FreshInsert shown below. FreshInsert creates an entry for the key corresponding to an element index src_index in Bucket *src. So this is a helper method for populating empty hashtable. Uses the templatized Copier instance to perform the copy of the source Bucket instance into the found empty Bucket.

Code Snippet: CopyEntries and FreshInsert

```
template <typename Copier>
 void CopyEntries(Bucket* start, Bucket* end, Copier copier) {
  for (Bucket* b = start; b != end; b++) {
   for (uint32 i = 0; i < kWidth; i++) {
    if (b->marker[i] >= 2) {
     FreshInsert(b, i, copier);
    }
   }
  }
 }

 // Create an entry for the key numbered src_index in *src and return
 // its bucket/index.  Used for insertion into a fresh table.  We
 // assume that there are no deletions, and k does not already exist
 // in the table.
 template <typename Copier>
 void FreshInsert(Bucket* src, uint32 src_index, Copier copier) {
  size_t h = hash_(src->key(src_index));
  const uint32 marker = Marker(h & 0xff);
  size_t index = (h >> 8) & mask_;  // Holds bucket num and index-in-bucket
  uint32 num_probes = 1;           // Needed for quadratic probing
  while (true) {
   uint32 bi = index & (kWidth - 1);
   Bucket* b = &array_[index >> kBase];
   const uint32 x = b->marker[bi];
   if (x == 0) {
    b->marker[bi] = marker;
    not_empty_++;
```

```
        copier(b, bi, src, src_index);
        return;
      }
      index = NextIndex(index, num_probes);
      num_probes++;
    }
  }
```

Another set of related FlatRep methods are clear_no_resize(), CopyEntries(), MoveEntry(), Resize(), MaybeResize(), Prefetch(), Erase(), clear(). clear_no_resize() traverses the internal FlatRep array of Bucket instances. If the marker of the current Bucket instance is different than kEmpty and kDeleted it invokes the destructor of the Key instance referenced by corresponding index-in-bucket and sets the corresponding marker to kEmpty. CopyEntries() copies all entries from specified starting Bucket instance until the specified ending Bucket instance where the supplied copier is applied only to those entries which are marked neither as kEmpty nor as kDeleted. MoveEntry() moves the entry specified with the in-bucket-index srci from the source Bucket src to the destination entry given with its in-bucket-index dsti for the specified destination Bucket dst. Resize(N) preserves the old beginning and old end of the internal array of Bucket instances , then it invokes Init(N) for the new number N of entries, then it invokes CopyEntries for all elements of the old array, and finally it destroys all entries of the old array. Note that no actual copy of the old entries to the newly allocated array of Buckets takes place. The reason why is that the copier instance MoveEntry supplied to CopyEntries() actually uses the move semantics (see Bucket::MoveFrom() discussed previously) followed by Bucket::Destroy() which invokes the destructor of the specified with in-bucket-index Key entry. The final thing which CopyEntries() does is to set the marker of the specified in-bucket entry to kDeleted.

```
  struct MoveEntry {
    inline void operator()(Bucket* dst, uint32 dsti, Bucket* src, uint32 srci) {
      dst->MoveFrom(dsti, src, srci);
      src->Destroy(srci);
      src->marker[srci] = kDeleted;
    }
  };
```

MaybeResize() first checks if the member FlatRep::not_empty_ is smaller than FlatRep::grow_ and if it is it returns immediately since the resize threshold has not been reached. Next there is a check for FlatRep::grow_ is equal to 0 which is a special value set by Erase() causing a shrink on next insert. The shrink will happen only if the current entry count given with not_empty_ - deleted_ is larger or equal to the value of FlatRep::shrink_ and if not_empty_ is larger or equal to bucket_count() * 0.8. Otherwise Resize(not_empty_ - deleted_+1) is executed.

Code Snippet: Portion of MaybeResize()

```
if (grow_ == 0) {
    // Special value set by erase to cause shrink on next insert.
    if (size() >= shrink_) {
      // Not small enough to shrink.
      grow_ = static_cast<size_t>(bucket_count() * 0.8);
      if (not_empty_ < grow_) return;
    }
  }
  Resize(size() + 1);
}
```

The method Prefetch(const Key& k) issues __builtin_prefetch() for the entry marker and key for the specified Key k. __builtin_prefetch(x,T0) issues the x86 PREFETCH machine instruction which leads to an entire 64 byte cache line to be read *into all levels* of cache hierarchy.

```
void Prefetch(const Key& k) const {
  size_t h = hash_(k);
  size_t index = (h >> 8) & mask_;  // Holds bucket num and index-in-bucket
  uint32 bi = index & (kWidth - 1);
  Bucket* b = &array_[index >> kBase];
  port::prefetch<port::PREFETCH_HINT_T0>(&b->marker[bi]);
  port::prefetch<port::PREFETCH_HINT_T0>(&b->storage.key[bi]);
}
```

The method Erase() destroys the entry specified by the supplied in-bucket-index i and sets the corresponding marker to kDeleted. It increase the count of FlatRep::deleted_ and sets FlatRep::grow_ to 0 which would trigger shrink on a new insert.

```
void Erase(Bucket* b, uint32 i) {
  b->Destroy(i);
  b->marker[i] = kDeleted;
  deleted_++;
  grow_ = 0;  // Consider shrinking on next insert
}
```

Code Snippet: The entire code of class FlatRep

```
// Internal representation for FlatMap and FlatSet.
//
// The representation is an open-addressed hash table.  Conceptually,
// the representation is a flat array of entries.  However, we
// structure it as an array of buckets where each bucket holds
// kWidth entries along with metadata for the kWidth entries.  The
// metadata marker is
//
//  (a) kEmpty: the entry is empty
//  (b) kDeleted: the entry has been deleted
//  (c) other: the entry is occupied and has low-8 bits of its hash.
//      These hash bits can be used to avoid potentially expensive
//      key comparisons.
//
// FlatMap passes in a bucket that contains keys and values, FlatSet
// passes in a bucket that does not contain values.
template <typename Key, typename Bucket, class Hash, class Eq>
class FlatRep {
 public:
  // kWidth is the number of entries stored in a bucket.
  static constexpr uint32 kBase = 3;
  static constexpr uint32 kWidth = (1 << kBase);

  FlatRep(size_t N, const Hash& hf, const Eq& eq) : hash_(hf), equal_(eq) {
    Init(N);
  }
  FlatRep(const FlatRep& src) : hash_(src.hash_), equal_(src.equal_) {
    Init(src.size());
    CopyEntries(src.array_, src.end_, CopyEntry());
  }

  FlatRep(FlatRep&& src)
      // Copy rather than move src.hash_ and src.equal_.  This is necessary to
      // leave src in a valid state -- otherwise e.g. if hash_ is an
      // std::function, moving it would null it out.
      : hash_(src.hash_), equal_(src.equal_) {
    // TODO(jlebar): Init(1) still allocates some memory, so this isn't as cheap
    // as it could be.  The fundamental problem is that we need to leave src in
    // a valid state, and FlatRep *always* owns a nonzero amount of memory.
    Init(1);
    swap(src);
  }

  ~FlatRep() {
    clear_no_resize();
    delete[] array_;
  }

  // Simple accessors.
  size_t size() const { return not_empty_ - deleted_; }
  size_t bucket_count() const { return mask_ + 1; }
  Bucket* start() const { return array_; }
  Bucket* limit() const { return end_; }
  const Hash& hash_function() const { return hash_; }
  const Eq& key_eq() const { return equal_; }

  // Overwrite contents of *this with contents of src.
  void CopyFrom(const FlatRep& src) {
    if (this != &src) {
      clear_no_resize();
      delete[] array_;
      Init(src.size());
      CopyEntries(src.array_, src.end_, CopyEntry());
    }
  }

  void MoveFrom(FlatRep&& src) {
    if (this != &src) {
      swap(src);
    }
  }

  void clear_no_resize() {
    for (Bucket* b = array_; b != end_; b++) {
      for (uint32 i = 0; i < kWidth; i++) {
        if (b->marker[i] >= 2) {
          b->Destroy(i);
          b->marker[i] = kEmpty;
        }
      }
    }
    not_empty_ = 0;
    deleted_ = 0;
  }

  void clear() {
    clear_no_resize();
    grow_ = 0;  // Consider shrinking in MaybeResize()
    MaybeResize();
  }

  void swap(FlatRep& x) {
    using std::swap;
    swap(array_, x.array_);
    swap(end_, x.end_);
    swap(lglen_, x.lglen_);
    swap(mask_, x.mask_);
    swap(not_empty_, x.not_empty_);
    swap(deleted_, x.deleted_);
    swap(grow_, x.grow_);
    swap(shrink_, x.shrink_);
  }

  struct SearchResult {
    bool found;
    Bucket* b;
    uint32 index;
  };

  // Hash value is partitioned as follows:
  // 1. Bottom 8 bits are stored in bucket to help speed up comparisons.
  // 2. Next 3 bits give index inside bucket.
  // 3. Remaining bits give bucket number.

  // Find bucket/index for key k.
  SearchResult Find(const Key& k) const {
    size_t h = hash_(k);
    const uint32 marker = Marker(h & 0xff);
    size_t index = (h >> 8) & mask_;  // Holds bucket num and index-in-bucket
    uint32 num_probes = 1;            // Needed for quadratic probing
    while (true) {
      uint32 bi = index & (kWidth - 1);
      Bucket* b = &array_[index >> kBase];
      const uint32 x = b->marker[bi];
      if (x == marker && equal_(b->key(bi), k)) {
        return {true, b, bi};
      } else if (x == kEmpty) {
        return {false, nullptr, 0};
      }
      index = NextIndex(index, num_probes);
      num_probes++;
    }
  }

  // Find bucket/index for key k, creating a new one if necessary.
  //
  // KeyType is a template parameter so that k's type is deduced and it
  // becomes a universal reference which allows the key initialization
  // below to use an rvalue constructor if available.
  template <typename KeyType>
  SearchResult FindOrInsert(KeyType&& k) {
    size_t h = hash_(k);
    const uint32 marker = Marker(h & 0xff);
    size_t index = (h >> 8) & mask_;  // Holds bucket num and index-in-bucket
    uint32 num_probes = 1;            // Needed for quadratic probing
    Bucket* del = nullptr;            // First encountered deletion for kInsert
    uint32 di = 0;
    while (true) {
      uint32 bi = index & (kWidth - 1);
      Bucket* b = &array_[index >> kBase];
      const uint32 x = b->marker[bi];
      if (x == marker && equal_(b->key(bi), k)) {
        return {true, b, bi};
      } else if (!del && x == kDeleted) {
        // Remember deleted index to use for insertion.
        del = b;
        di = bi;
      } else if (x == kEmpty) {
        if (del) {
          // Store in the first deleted slot we encountered
          b = del;
          bi = di;
          deleted_--;  // not_empty_ does not change
        } else {
          not_empty_++;
        }
        b->marker[bi] = marker;
        new (&b->key(bi)) Key(std::forward<KeyType>(k));
        return {false, b, bi};
      }
      index = NextIndex(index, num_probes);
      num_probes++;
    }
  }

  void Erase(Bucket* b, uint32 i) {
    b->Destroy(i);
    b->marker[i] = kDeleted;
    deleted_++;
    grow_ = 0;  // Consider shrinking on next insert
  }

  void Prefetch(const Key& k) const {
    size_t h = hash_(k);
    size_t index = (h >> 8) & mask_;  // Holds bucket num and index-in-bucket
    uint32 bi = index & (kWidth - 1);
    Bucket* b = &array_[index >> kBase];
    port::prefetch<port::PREFETCH_HINT_T0>(&b->marker[bi]);
    port::prefetch<port::PREFETCH_HINT_T0>(&b->storage.key[bi]);
   }

   inline void MaybeResize() {
    if (not_empty_ < grow_) {
     return;  // Nothing to do
    }
    if (grow_ == 0) {
     // Special value set by erase to cause shrink on next insert.
     if (size() >= shrink_) {
      // Not small enough to shrink.
      grow_ = static_cast<size_t>(bucket_count() * 0.8);
      if (not_empty_ < grow_) return;
     }
    }
    Resize(size() + 1);
   }

   void Resize(size_t N) {
    Bucket* old = array_;
    Bucket* old_end = end_;
    Init(N);
    CopyEntries(old, old_end, MoveEntry());
    delete[] old;
   }

  private:
   enum { kEmpty = 0, kDeleted = 1 };  // Special markers for an entry.

   Hash hash_;         // User-supplied hasher
   Eq equal_;          // User-supplied comparator
   uint8 lglen_;       // lg(#buckets)
   Bucket* array_;     // array of length (1 << lglen_)
   Bucket* end_;       // Points just past last bucket in array_
   size_t mask_;       // (# of entries in table) - 1
   size_t not_empty_;  // Count of entries with marker != kEmpty
   size_t deleted_;    // Count of entries with marker == kDeleted
   size_t grow_;       // Grow array when not_empty_ >= grow_
   size_t shrink_;     // Shrink array when size() < shrink_

   // Avoid kEmpty and kDeleted markers when computing hash values to
   // store in Bucket::marker[].
```

```
static uint32 Marker(uint32 hb) { return hb + (hb < 2 ? 2 : 0); }

void Init(size_t N) {
  // Make enough room for N elements.
  size_t lg = 0;  // Smallest table is just one bucket.
  while (N >= 0.8 * ((1 << lg) * kWidth)) {
    lg++;
  }
  const size_t n = (1 << lg);
  Bucket* array = new Bucket[n];
  for (size_t i = 0; i < n; i++) {
    Bucket* b = &array[i];
    memset(b->marker, kEmpty, kWidth);
  }
  const size_t capacity = (1 << lg) * kWidth;
  lglen_ = lg;
  mask_ = capacity - 1;
  array_ = array;
  end_ = array + n;
  not_empty_ = 0;
  deleted_ = 0;
  grow_ = static_cast<size_t>(capacity * 0.8);
  if (lg == 0) {
    // Already down to one bucket; no more shrinking.
    shrink_ = 0;
  } else {
    shrink_ = static_cast<size_t>(grow_ * 0.4);  // Must be less than 0.5
  }
}

// Used by FreshInsert when we should copy from source.
struct CopyEntry {
  inline void operator()(Bucket* dst, uint32 dsti, Bucket* src, uint32 srci) {
    dst->CopyFrom(dsti, src, srci);
  }
};

// Used by FreshInsert when we should move from source.
struct MoveEntry {
  inline void operator()(Bucket* dst, uint32 dsti, Bucket* src, uint32 srci) {
    dst->MoveFrom(dsti, src, srci);
    src->Destroy(srci);
    src->marker[srci] = kDeleted;
  }
```

```
 };

 template <typename Copier>
 void CopyEntries(Bucket* start, Bucket* end, Copier copier) {
   for (Bucket* b = start; b != end; b++) {
     for (uint32 i = 0; i < kWidth; i++) {
       if (b->marker[i] >= 2) {
         FreshInsert(b, i, copier);
       }
     }
   }
 }

 // Create an entry for the key numbered src_index in *src and return
 // its bucket/index.  Used for insertion into a fresh table.  We
 // assume that there are no deletions, and k does not already exist
 // in the table.
 template <typename Copier>
 void FreshInsert(Bucket* src, uint32 src_index, Copier copier) {
   size_t h = hash_(src->key(src_index));
   const uint32 marker = Marker(h & 0xff);
   size_t index = (h >> 8) & mask_;  // Holds bucket num and index-in-bucket
   uint32 num_probes = 1;           // Needed for quadratic probing
   while (true) {
     uint32 bi = index & (kWidth - 1);
     Bucket* b = &array_[index >> kBase];
     const uint32 x = b->marker[bi];
     if (x == 0) {
       b->marker[bi] = marker;
       not_empty_++;
       copier(b, bi, src, src_index);
       return;
     }
     index = NextIndex(index, num_probes);
     num_probes++;
   }
 }

 inline size_t NextIndex(size_t i, uint32 num_probes) const {
   // Quadratic probing.
   return (i + num_probes) & mask_;
 }
};
```

Class gtl::FlatMap

gtl::FlatMap is a templetized container class used throughout the tensorflow core functionality. The FlatMap is not implemented like balanced binary search tree but instead it is implemented as a hash table with O(1) insertion, deletion, and search times.

Class gtl::FlatMap is declared in tensorflow/core/lib/gtl/flatmap.h as shown below.
The class gtl::FlatMap declares and defines the following internals in its class declaration:
struct Bucket, struct ValueType, class iterator, class const_iterator,

```
// FlatMap<K,V,...> provides a map from K to V.
//
// The map is implemented using an open-addressed hash table.  A
// single array holds entire map contents and collisions are resolved
// by probing at a sequence of locations in the array.
template <typename Key, typename Val, class Hash = hash<Key>,
          class Eq = std::equal_to<Key>>
class FlatMap {
 private:
  // Forward declare some internal types needed in public section.
  struct Bucket;

  // We cannot use std::pair<> since internal representation stores
  // keys and values in separate arrays, so we make a custom struct
  // that holds references to the internal key, value elements.
  //
  // We define the struct as private ValueType, and typedef it as public
  // value_type, to work around a gcc bug when compiling the iterators.
  struct ValueType {
    typedef Key first_type;
    typedef Val second_type;

    const Key& first;
    Val& second;
    ValueType(const Key& k, Val& v) : first(k), second(v) {}
  };

 public:
  typedef Key key_type;
  typedef Val mapped_type;
  typedef Hash hasher;
  typedef Eq key_equal;
  typedef size_t size_type;
  typedef ptrdiff_t difference_type;
  typedef ValueType value_type;
  typedef value_type* pointer;
  typedef const value_type* const_pointer;
typedef value_type& reference;
typedef const value_type& const_reference;

FlatMap() : FlatMap(1) {}

explicit FlatMap(size_t N, const Hash& hf = Hash(), const Eq& eq = Eq())
    : rep_(N, hf, eq) {}

FlatMap(const FlatMap& src) : rep_(src.rep_) {}

// Move constructor leaves src in a valid but unspecified state (same as
// std::unordered_map).
FlatMap(FlatMap&& src) : rep_(std::move(src.rep_)) {}

template <typename InputIter>
FlatMap(InputIter first, InputIter last, size_t N = 1,
        const Hash& hf = Hash(), const Eq& eq = Eq())
    : FlatMap(N, hf, eq) {
  insert(first, last);
}

FlatMap(std::initializer_list<std::pair<const Key, Val>> init, size_t N = 1,
        const Hash& hf = Hash(), const Eq& eq = Eq())
    : FlatMap(init.begin(), init.end(), N, hf, eq) {}

FlatMap& operator=(const FlatMap& src) {
  rep_.CopyFrom(src.rep_);
  return *this;
}

// Move-assignment operator leaves src in a valid but unspecified state (same
// as std::unordered_map).
FlatMap& operator=(FlatMap&& src) {
  rep_.MoveFrom(std::move(src.rep_));
  return *this;
}

~FlatMap() {}

void swap(FlatMap& x) { rep_.swap(x.rep_); }
void clear_no_resize() { rep_.clear_no_resize(); }
void clear() { rep_.clear(); }
void reserve(size_t N) { rep_.Resize(std::max(N, size())); }
void rehash(size_t N) { rep_.Resize(std::max(N, size())); }
void resize(size_t N) { rep_.Resize(std::max(N, size())); }
size_t size() const { return rep_.size(); }
bool empty() const { return size() == 0; }
size_t bucket_count() const { return rep_.bucket_count(); }
hasher hash_function() const { return rep_.hash_function(); }
key_equal key_eq() const { return rep_.key_eq(); }

class iterator {
 public:
  typedef typename FlatMap::difference_type difference_type;
  typedef typename FlatMap::value_type value_type;
  typedef typename FlatMap::pointer pointer;
  typedef typename FlatMap::reference reference;
  typedef ::std::forward_iterator_tag iterator_category;

  iterator() : b_(nullptr), end_(nullptr), i_(0) {}

  // Make iterator pointing at first element at or after b.
  iterator(Bucket* b, Bucket* end) : b_(b), end_(end), i_(0) { SkipUnused(); }

  // Make iterator pointing exactly at ith element in b, which must exist.
  iterator(Bucket* b, Bucket* end, uint32 i) : b_(b), end_(end), i_(i) {
    FillValue();
  }

  reference operator*() { return *val(); }
  pointer operator->() { return val(); }
  bool operator==(const iterator& x) const {
    return b_ == x.b_ && i_ == x.i_;
  }
  bool operator!=(const iterator& x) const { return !(*this == x); }
  iterator& operator++() {
    DCHECK(b_ != end_);
    i_++;
    SkipUnused();
    return *this;
  }
  iterator operator++(int /*indicates postfix*/) {
    iterator tmp(*this);
    ++*this;
    return tmp;
  }

 private:
  friend class FlatMap;
  Bucket* b_;
  Bucket* end_;
  char space_ alignas(value_type)[sizeof(value_type)];
  uint32 i_;

  pointer val() { return reinterpret_cast<pointer>(space_); }
  void FillValue() { new (space_) value_type(b_->key(i_), b_->val(i_)); }
  void SkipUnused() {
    while (b_ < end_) {
      if (i_ >= Rep::kWidth) {
        i_ = 0;
        b_++;
      } else if (b_->marker[i_] < 2) {
        i_++;
      } else {
        FillValue();
        break;
      }
    }
  }
};

class const_iterator {
 private:
  mutable iterator rep_;  // Share state and logic with non-const iterator.
 public:
  typedef typename FlatMap::difference_type difference_type;
  typedef typename FlatMap::value_type value_type;
  typedef typename FlatMap::const_pointer pointer;
  typedef typename FlatMap::const_reference reference;
  typedef ::std::forward_iterator_tag iterator_category;

  const_iterator() : rep_() {}
  const_iterator(Bucket* start, Bucket* end) : rep_(start, end) {}
  const_iterator(Bucket* b, Bucket* end, uint32 i) : rep_(b, end, i) {}

  reference operator*() const { return *rep_.val(); }
  pointer operator->() const { return rep_.val(); }
  bool operator==(const const_iterator& x) const { return rep_ == x.rep_; }
  bool operator!=(const const_iterator& x) const { return rep_ != x.rep_; }
  const_iterator& operator++() {
    ++rep_;
    return *this;
  }
  const_iterator operator++(int /*indicates postfix*/) {
    const_iterator tmp(*this);
    ++*this;
    return tmp;
  }
};

iterator begin() { return iterator(rep_.start(), rep_.limit()); }
iterator end() { return iterator(rep_.limit(), rep_.limit()); }
const_iterator begin() const {
  return const_iterator(rep_.start(), rep_.limit());
}
const_iterator end() const {
  return const_iterator(rep_.limit(), rep_.limit());
}

size_t count(const Key& k) const { return rep_.Find(k).found ? 1 : 0; }
iterator find(const Key& k) {
  auto r = rep_.Find(k);
  return r.found ? iterator(r.b, rep_.limit(), r.index) : end();
}
const_iterator find(const Key& k) const {
  auto r = rep_.Find(k);
  return r.found ? const_iterator(r.b, rep_.limit(), r.index) : end();
}

Val& at(const Key& k) {
  auto r = rep_.Find(k);
  DCHECK(r.found);
  return r.b->val(r.index);
}
const Val& at(const Key& k) const {
  auto r = rep_.Find(k);
  DCHECK(r.found);
  return r.b->val(r.index);
}

template <typename P>
std::pair<iterator, bool> insert(const P& p) {
  return Insert(p.first, p.second);
}
std::pair<iterator, bool> insert(const std::pair<const Key, Val>& p) {
  return Insert(p.first, p.second);
```

```
}
template <typename InputIter>
void insert(InputIter first, InputIter last) {
  for (; first != last; ++first) {
    insert(*first);
  }
}

Val& operator[](const Key& k) { return IndexOp(k); }
Val& operator[](Key&& k) { return IndexOp(std::forward<Key>(k)); }

template <typename... Args>
std::pair<iterator, bool> emplace(Args&&... args) {
  return InsertPair(std::make_pair(std::forward<Args>(args)...));
}

size_t erase(const Key& k) {
  auto r = rep_.Find(k);
  if (!r.found) return 0;
  rep_.Erase(r.b, r.index);
  return 1;
}
iterator erase(iterator pos) {
  rep_.Erase(pos.b_, pos.i_);
  ++pos;
  return pos;
}
iterator erase(iterator pos, iterator last) {
  for (; pos != last; ++pos) {
    rep_.Erase(pos.b_, pos.i_);
  }
  return pos;
}

std::pair<iterator, iterator> equal_range(const Key& k) {
  auto pos = find(k);
  if (pos == end()) {
    return std::make_pair(pos, pos);
  } else {
    auto next = pos;
    ++next;
    return std::make_pair(pos, next);
  }
}
```

```
std::pair<const_iterator, const_iterator> equal_range(const Key& k) const {
  auto pos = find(k);
  if (pos == end()) {
    return std::make_pair(pos, pos);
  } else {
    auto next = pos;
    ++next;
    return std::make_pair(pos, next);
  }
}

bool operator==(const FlatMap& x) const {
  if (size() != x.size()) return false;
  for (auto& p : x) {
    auto i = find(p.first);
    if (i == end()) return false;
    if (i->second != p.second) return false;
  }
  return true;
}
bool operator!=(const FlatMap& x) const { return !(*this == x); }

// If key exists in the table, prefetch the associated value.  This
// is a hint, and may have no effect.
void prefetch_value(const Key& key) const { rep_.Prefetch(key); }

private:
using Rep = internal::FlatRep<Key, Bucket, Hash, Eq>;

// Bucket stores kWidth <marker, key, value> triples.
// The data is organized as three parallel arrays to reduce padding.
struct Bucket {
  uint8 marker[Rep::kWidth];

  // Wrap keys and values in union to control construction and destruction.
  union Storage {
    struct {
      Key key[Rep::kWidth];
      Val val[Rep::kWidth];
    };
    Storage() {}
    ~Storage() {}
  }  storage;
  Key& key(uint32 i) {
    DCHECK_GE(marker[i], 2);
    return storage.key[i];
  }
  Val& val(uint32 i) {
    DCHECK_GE(marker[i], 2);
    return storage.val[i];
  }
  template <typename V>
  void InitVal(uint32 i, V&& v) {
    new (&storage.val[i]) Val(std::forward<V>(v));
  }
  void Destroy(uint32 i) {
    storage.key[i].Key::~Key();
    storage.val[i].Val::~Val();
  }
  void MoveFrom(uint32 i, Bucket* src, uint32 src_index) {
    new (&storage.key[i]) Key(std::move(src->storage.key[src_index]));
    new (&storage.val[i]) Val(std::move(src->storage.val[src_index]));
  }
  void CopyFrom(uint32 i, Bucket* src, uint32 src_index) {
    new (&storage.key[i]) Key(src->storage.key[src_index]);
    new (&storage.val[i]) Val(src->storage.val[src_index]);
  }
};

template <typename Pair>
std::pair<iterator, bool> InsertPair(Pair&& p) {
  return Insert(std::forward<decltype(p.first)>(p.first),
                std::forward<decltype(p.second)>(p.second));
}

template <typename K, typename V>
std::pair<iterator, bool> Insert(K&& k, V&& v) {
  rep_.MaybeResize();
  auto r = rep_.FindOrInsert(std::forward<K>(k));
  const bool inserted = !r.found;
  if (inserted) {
    r.b->InitVal(r.index, std::forward<V>(v));
  }
  return {iterator(r.b, rep_.limit(), r.index), inserted};
}

template <typename K>
```

```
  Val& IndexOp(K&& k) {
    rep_.MaybeResize();
    auto r = rep_.FindOrInsert(std::forward<K>(k));
    Val* vptr = &r.b->val(r.index);
    if (!r.found) {
      new (vptr) Val();  // Initialize value in new slot.
    }
    return *vptr;
  }

  Rep rep_;
};
```

Class gtl::FlatSet

```
// FlatSet<K,...> provides a set of K.
//
// The map is implemented using an open-addressed hash table.  A
// single array holds entire map contents and collisions are resolved
// by probing at a sequence of locations in the array.
template <typename Key, class Hash = hash<Key>, class Eq = std::equal_to<Key>>
class FlatSet {
 private:
  // Forward declare some internal types needed in public section.
  struct Bucket;

 public:
  typedef Key key_type;
  typedef Key value_type;
  typedef Hash hasher;
  typedef Eq key_equal;
  typedef size_t size_type;
  typedef ptrdiff_t difference_type;
  typedef value_type* pointer;
  typedef const value_type* const_pointer;
  typedef value_type& reference;
  typedef const value_type& const_reference;

  FlatSet() : FlatSet(1) {}

  explicit FlatSet(size_t N, const Hash& hf = Hash(), const Eq& eq = Eq())
      : rep_(N, hf, eq) {}

  FlatSet(const FlatSet& src) : rep_(src.rep_) {}
```

```
// Move constructor leaves src in a valid but unspecified state (same as
// std::unordered_set).
FlatSet(FlatSet&& src) : rep_(std::move(src.rep_)) {}

template <typename InputIter>
FlatSet(InputIter first, InputIter last, size_t N = 1,
        const Hash& hf = Hash(), const Eq& eq = Eq())
    : FlatSet(N, hf, eq) {
  insert(first, last);
}

FlatSet(std::initializer_list<value_type> init, size_t N = 1,
        const Hash& hf = Hash(), const Eq& eq = Eq())
    : FlatSet(init.begin(), init.end(), N, hf, eq) {}

FlatSet& operator=(const FlatSet& src) {
  rep_.CopyFrom(src.rep_);
  return *this;
}

// Move-assignment operator leaves src in a valid but unspecified state (same
// as std::unordered_set).
FlatSet& operator=(FlatSet&& src) {
  rep_.MoveFrom(std::move(src.rep_));
  return *this;
}

~FlatSet() {}

void swap(FlatSet& x) { rep_.swap(x.rep_); }
void clear_no_resize() { rep_.clear_no_resize(); }
void clear() { rep_.clear(); }
void reserve(size_t N) { rep_.Resize(std::max(N, size())); }
void rehash(size_t N) { rep_.Resize(std::max(N, size())); }
void resize(size_t N) { rep_.Resize(std::max(N, size())); }
size_t size() const { return rep_.size(); }
bool empty() const { return size() == 0; }
size_t bucket_count() const { return rep_.bucket_count(); }
hasher hash_function() const { return rep_.hash_function(); }
key_equal key_eq() const { return rep_.key_eq(); }

class const_iterator {
 public:
  typedef typename FlatSet::difference_type difference_type;
 typedef typename FlatSet::value_type value_type;
 typedef typename FlatSet::const_pointer pointer;
 typedef typename FlatSet::const_reference reference;
 typedef ::std::forward_iterator_tag iterator_category;

 const_iterator() : b_(nullptr), end_(nullptr), i_(0) {}

 // Make iterator pointing at first element at or after b.
 const_iterator(Bucket* b, Bucket* end) : b_(b), end_(end), i_(0) {
   SkipUnused();
 }

 // Make iterator pointing exactly at ith element in b, which must exist.
 const_iterator(Bucket* b, Bucket* end, uint32 i)
     : b_(b), end_(end), i_(i) {}

 reference operator*() const { return key(); }
 pointer operator->() const { return &key(); }
 bool operator==(const const_iterator& x) const {
   return b_ == x.b_ && i_ == x.i_;
 }
 bool operator!=(const const_iterator& x) const { return !(*this == x); }
 const_iterator& operator++() {
   DCHECK(b_ != end_);
   i_++;
   SkipUnused();
   return *this;
 }
 const_iterator operator++(int /*indicates postfix*/) {
   const_iterator tmp(*this);
   ++*this;
   return tmp;
 }

private:
 friend class FlatSet;
 Bucket* b_;
 Bucket* end_;
 uint32 i_;

 reference key() const { return b_->key(i_); }
 void SkipUnused() {
   while (b_ < end_) {
     if (i_ >= Rep::kWidth) {
        i_ = 0;
        b_++;
      } else if (b_->marker[i_] < 2) {
        i_++;
      } else {
        break;
      }
    }
  }
};

typedef const_iterator iterator;

iterator begin() { return iterator(rep_.start(), rep_.limit()); }
iterator end() { return iterator(rep_.limit(), rep_.limit()); }
const_iterator begin() const {
  return const_iterator(rep_.start(), rep_.limit());
}
const_iterator end() const {
  return const_iterator(rep_.limit(), rep_.limit());
}

size_t count(const Key& k) const { return rep_.Find(k).found ? 1 : 0; }
iterator find(const Key& k) {
  auto r = rep_.Find(k);
  return r.found ? iterator(r.b, rep_.limit(), r.index) : end();
}
const_iterator find(const Key& k) const {
  auto r = rep_.Find(k);
  return r.found ? const_iterator(r.b, rep_.limit(), r.index) : end();
}

std::pair<iterator, bool> insert(const Key& k) { return Insert(k); }
std::pair<iterator, bool> insert(Key&& k) { return Insert(std::move(k)); }
template <typename InputIter>
void insert(InputIter first, InputIter last) {
  for (; first != last; ++first) {
    insert(*first);
  }
}

template <typename... Args>
std::pair<iterator, bool> emplace(Args&&... args) {
  rep_.MaybeResize();
  auto r = rep_.FindOrInsert(std::forward<Args>(args)...);
  const bool inserted = !r.found;
  return {iterator(r.b, rep_.limit(), r.index), inserted};
}

size_t erase(const Key& k) {
  auto r = rep_.Find(k);
  if (!r.found) return 0;
  rep_.Erase(r.b, r.index);
  return 1;
}
iterator erase(iterator pos) {
  rep_.Erase(pos.b_, pos.i_);
  ++pos;
  return pos;
}
iterator erase(iterator pos, iterator last) {
  for (; pos != last; ++pos) {
    rep_.Erase(pos.b_, pos.i_);
  }
  return pos;
}

std::pair<iterator, iterator> equal_range(const Key& k) {
  auto pos = find(k);
  if (pos == end()) {
    return std::make_pair(pos, pos);
  } else {
    auto next = pos;
    ++next;
    return std::make_pair(pos, next);
  }
}
std::pair<const_iterator, const_iterator> equal_range(const Key& k) const {
  auto pos = find(k);
  if (pos == end()) {
    return std::make_pair(pos, pos);
  } else {
    auto next = pos;
    ++next;
    return std::make_pair(pos, next);
  }
}
```

```
 bool operator==(const FlatSet& x) const {
   if (size() != x.size()) return false;
   for (const auto& elem : x) {
     auto i = find(elem);
     if (i == end()) return false;
   }
   return true;
 }
 bool operator!=(const FlatSet& x) const { return !(*this == x); }

 // If key exists in the table, prefetch it.  This is a hint, and may
 // have no effect.
 void prefetch_value(const Key& key) const { rep_.Prefetch(key); }

private:
 using Rep = internal::FlatRep<Key, Bucket, Hash, Eq>;

 // Bucket stores kWidth <marker, key, value> triples.
 // The data is organized as three parallel arrays to reduce padding.
 struct Bucket {
   uint8 marker[Rep::kWidth];

   // Wrap keys in union to control construction and destruction.
   union Storage {
     Key key[Rep::kWidth];
     Storage() {}
     ~Storage() {}
   } storage;

   Key& key(uint32 i) {
     DCHECK_GE(marker[i], 2);
     return storage.key[i];
   }
   void Destroy(uint32 i) { storage.key[i].Key::~Key(); }
   void MoveFrom(uint32 i, Bucket* src, uint32 src_index) {
     new (&storage.key[i]) Key(std::move(src->storage.key[src_index]));
   }
   void CopyFrom(uint32 i, Bucket* src, uint32 src_index) {
     new (&storage.key[i]) Key(src->storage.key[src_index]);
   }
 };

 template <typename K>
 std::pair<iterator, bool> Insert(K&& k) {
    rep_.MaybeResize();
    auto r = rep_.FindOrInsert(std::forward<K>(k));
    const bool inserted = !r.found;
    return {iterator(r.b, rep_.limit(), r.index), inserted};
  }

  Rep rep_;
};
```

Class CompactPointerSet<T>

```
// CompactPointerSet<T> is like a std::unordered_set<T> but is optimized
// for small sets (<= 1 element).  T must be a pointer type.
template <typename T>
class CompactPointerSet {
 private:
  using BigRep = FlatSet<T>;

 public:
  using value_type = T;

  CompactPointerSet() : rep_(0) {}

  ~CompactPointerSet() {
    static_assert(
        std::is_pointer<T>::value,
        "CompactPointerSet<T> can only be used with T's that are pointers");
    if (isbig()) delete big();
  }

  CompactPointerSet(const CompactPointerSet& other) : rep_(0) { *this = other; }

  CompactPointerSet& operator=(const CompactPointerSet& other) {
    if (this == &other) return *this;
    if (other.isbig()) {
      // big => any
      if (!isbig()) MakeBig();
      *big() = *other.big();
    } else if (isbig()) {
      // !big => big
      big()->clear();
      if (other.rep_ != 0) {
        big()->insert(reinterpret_cast<T>(other.rep_));
      }
    } else {
```

```
    // !big => !big
    rep_ = other.rep_;
  }
  return *this;
}

class iterator {
 public:
  typedef ssize_t difference_type;
  typedef T value_type;
  typedef const T* pointer;
  typedef const T& reference;
  typedef ::std::forward_iterator_tag iterator_category;

  explicit iterator(uintptr_t rep)
      : bigrep_(false), single_(reinterpret_cast<T>(rep)) {}
  explicit iterator(typename BigRep::iterator iter)
      : bigrep_(true), single_(nullptr), iter_(iter) {}

  iterator& operator++() {
    if (bigrep_) {
      ++iter_;
    } else {
      DCHECK(single_ != nullptr);
      single_ = nullptr;
    }
    return *this;
  }
  // maybe post-increment?

  bool operator==(const iterator& other) const {
    if (bigrep_) {
      return iter_ == other.iter_;
    } else {
      return single_ == other.single_;
    }
  }
  bool operator!=(const iterator& other) const { return !(*this == other); }

  const T& operator*() const {
    if (bigrep_) {
      return *iter_;
    } else {
      DCHECK(single_ != nullptr);
      return single_;
    }
  }

 private:
  friend class CompactPointerSet;
  bool bigrep_;
  T single_;
  typename BigRep::iterator iter_;
};
using const_iterator = iterator;

bool empty() const { return isbig() ? big()->empty() : (rep_ == 0); }
size_t size() const { return isbig() ? big()->size() : (rep_ == 0 ? 0 : 1); }

void clear() {
  if (isbig()) {
    delete big();
  }
  rep_ = 0;
}

std::pair<iterator, bool> insert(T elem) {
  if (!isbig()) {
    if (rep_ == 0) {
      uintptr_t v = reinterpret_cast<uintptr_t>(elem);
      if (v == 0 || ((v & 0x3) != 0)) {
        // Cannot use small representation for nullptr.  Fall through.
      } else {
        rep_ = v;
        return {iterator(v), true};
      }
    }
    MakeBig();
  }
  auto p = big()->insert(elem);
  return {iterator(p.first), p.second};
}

template <typename InputIter>
void insert(InputIter begin, InputIter end) {
  for (; begin != end; ++begin) {
    insert(*begin);
  }
```

```
}

const_iterator begin() const {
  return isbig() ? iterator(big()->begin()) : iterator(rep_);
}
const_iterator end() const {
  return isbig() ? iterator(big()->end()) : iterator(0);
}

iterator find(T elem) const {
  if (rep_ == reinterpret_cast<uintptr_t>(elem)) {
    return iterator(rep_);
  } else if (!isbig()) {
    return iterator(0);
  } else {
    return iterator(big()->find(elem));
  }
}

size_t count(T elem) const { return find(elem) != end() ? 1 : 0; }

size_t erase(T elem) {
  if (!isbig()) {
    if (rep_ == reinterpret_cast<uintptr_t>(elem)) {
      rep_ = 0;
      return 1;
    } else {
      return 0;
    }
  } else {
    return big()->erase(elem);
  }
}

private:
// Size       rep_
// ----------------------------------------------------------------------
// 0          0
// 1          The pointer itself (bottom bits == 00)
// large      Pointer to a BigRep (bottom bits == 01)
uintptr_t rep_;

bool isbig() const { return (rep_ & 0x3) == 1; }
BigRep* big() const {
    DCHECK(isbig());
    return reinterpret_cast<BigRep*>(rep_ - 1);
  }

  void MakeBig() {
    DCHECK(!isbig());
    BigRep* big = new BigRep;
    if (rep_ != 0) {
      big->insert(reinterpret_cast<T>(rep_));
    }
    rep_ = reinterpret_cast<uintptr_t>(big) + 0x1;
  }
};
```

Class ArraySlice

tensorflow::gtl::ArraySlice is abseil::Span as shown below. Let's take a look in the abseil::Span design. The header file absl/types/span.h is pasted below.

```
#include "absl/types/span.h"
// TODO(timshen): This is kept only because lots of targets transitively depend
// on it. Remove all targets' dependencies.
#include "tensorflow/core/lib/gtl/inlined_vector.h"

namespace tensorflow {
namespace gtl {

template <typename T>
using ArraySlice = absl::Span<const T>;

template <typename T>
using MutableArraySlice = absl::Span<T>;

}  // namespace gtl
}  // namespace tensorflow
```

absl::Span is a lightweight templetized container which accepts arrays of contiguous data providing a thin layer of functionality for manipulating the underlying array without needing to manage pointers and array lengths manually. Most of this functionality is designed through constexpr/templates and is available at compile time.

***Note** on Span constructors*: There are two Span constructors, one for immutable view and the other for mutable view, provided in case the template type argument V can be converted into a Span i.e. has data() and size() members.

```
template <typename T>
class Span {
. . .
template <typename V, typename = EnableIfConvertibleFrom<V>,
          typename = EnableIfMutableView<V>>
  explicit Span(V& v) noexcept  // NOLINT(runtime/references)
      : Span(span_internal::GetData(v), v.size()) {}

  // Implicit reference constructor for a read-only `Span<const T>` type
  template <typename V, typename = EnableIfConvertibleFrom<V>,
          typename = EnableIfConstView<V>>
  constexpr Span(const V& v) noexcept  // NOLINT(runtime/explicit)
      : Span(span_internal::GetData(v), v.size()) {}
```

The constructors are enabled through SFINAE and helper template constructs as the ones shown below:

```
template <typename C>
  using EnableIfConvertibleFrom =
      typename std::enable_if<span_internal::HasData<T, C>::value &&
                        span_internal::HasSize<C>::value>::type;
. . .
}
```

The template utilities HasSize<C> and HasData<T,C> rely on type checking in unevaluated context using the pair of std::declval and std::decltype as shown below.

```
// Detection idioms for size() and data().
template <typename C>
using HasSize =
    std::is_integral<absl::decay_t<decltype(std::declval<C&>().size())>>;
```

As the comment below clarifies C is convertible to T if T is the same or more cv-qualified version of C. We try to convert C** to T* const* and if that succeeds then conclude that C is convertible to T; otherwise it is not. For the purpose we use std::is_convertible<A,B>::value which is true if std::declval<A>() can be converted to B using implicit conversion otherwise it is false.

```
// We want to enable conversion from vector<T*> to Span<const T* const> but
// disable conversion from vector<Derived> to Span<Base>. Here we use
// the fact that U** is convertible to Q* const* if and only if Q is the same
// type or a more cv-qualified version of U.  We also decay the result type of
// data() to avoid problems with classes which have a member function data()
// which returns a reference.
template <typename T, typename C>
using HasData =
    std::is_convertible<absl::decay_t<decltype(GetData(std::declval<C&>()))>*,
                  T* const*>;
```

In the above template absl::decay_t is part of the abseil metalibrary and is defined as std::decay<T>::type. std::decay<T> removes cv-qualifier and reference if such is present and if T is neither an array of another type U nor it is function reference; otherwise it returns U*.
The GetData<T> template is defined as:

```
template <typename C>
constexpr auto GetData(C& c) noexcept  // NOLINT(runtime/references)
    -> decltype(GetDataImpl(c, 0)) {
  return GetDataImpl(c, 0);
}
```

GetDataImpl template shown below will be expanded for every C which is not std::string when in the latter case we have a special inline GetDataImpl function which will invoked instead.

```
// Wrappers for access to container data pointers.
template <typename C>
constexpr auto GetDataImpl(C& c, char) noexcept  // NOLINT(runtime/references)
    -> decltype(c.data()) {
  return c.data();
}

// Before C++17, std::string::data returns a const char* in all cases.
inline char* GetDataImpl(std::string& s,  // NOLINT(runtime/references)
                         int) noexcept {
  return &s[0];
}
```

Note *on Abseil version*: the Abseil version used in the TF `master` branch from April 26,20 is `lts_2020_02_25` which is stored in the global ABSL_OPTION_INLINE_NAMESPACE_NAME:

```
#define ABSL_OPTION_INLINE_NAMESPACE_NAME lts_2020_02_25
```

Note *on ABSL_NAMESPACE_BEGIN*: ABSL_NAMESPACE_BEGIN and ABSL_NAMESPACE_END are defined to expand as a namespace which denotes the current abseil version if the global ABSL_OPTION_USE_INLINE_NAMESPACE is defined and set to 1. Otherwise it is a noop. This is shown on the code excerpt below:

```
#if ABSL_OPTION_USE_INLINE_NAMESPACE == 0
#define ABSL_NAMESPACE_BEGIN
#define ABSL_NAMESPACE_END
#elif ABSL_OPTION_USE_INLINE_NAMESPACE == 1
#define ABSL_NAMESPACE_BEGIN \
  inline namespace ABSL_OPTION_INLINE_NAMESPACE_NAME {
#define ABSL_NAMESPACE_END }
#else
#error options.h is misconfigured. // ABSL_OPTION_USE_INLINE_NAMESPACE is specified in abseil's
                                   //  options.h file
```

```
#endif
```

Note *on ExplicitArgumentBarrier*: in the constexpr factory methods MakeSpan(..) a variadic template argument ExplicitArgumentBarrier is used as shown below:

```
template <int&... ExplicitArgumentBarrier, typename T>
constexpr Span<T> MakeSpan(T* ptr, size_t size) noexcept {
  return Span<T>(ptr, size);
}
```

This barrier is introduced to prevent a bug-prone and superfluous use of explicitly typed parameters which in turn could prevent template deduction from taking place. For example:

```
double b[3] = {1.0, 2.0, 3.0};
auto s1 = MakeSpan(b,3); // will compile without an issue
auto s2 = MakeSpan<double>(b,3); // will fail to compile!
```

Note that Span works with absl::InlinedVector and can be created with it.

Remark on *clang-tidy*:
The comment // NOLINT(runtime/explicit) indicates to the cpp linter / clang-tidy diagnostic utility not to emit an warning. While clang-tidy diagnostics are intended to call out code that does not adhere to a coding standard, or is otherwise problematic in some way, there are times when it is more appropriate to silence the diagnostic instead of changing the semantics of the code. In such circumstances, the NOLINT or NOLINTNEXTLINE comments can be used to silence the diagnostic. For example:

```
  // Implicit conversion constructors
  template <size_t N>
  constexpr Span(T (&a)[N]) noexcept  // NOLINT(runtime/explicit)
      : Span(a, N) {}
```

The comment syntax can be described more formally as:

```
*lint-comment:*
  *lint-command* *lint-args~opt~*

*lint-args:*
  `(` *check-name-list* `)`

*check-name-list:*
  *check-name*
  *check-name-list* `,` *check-name*

*lint-command:*
  `NOLINT`
  `NOLINTNEXTLINE`
```

Specific to the prose mentioned above, you should document where the feature is tolerant to whitespace (can there be a space between NOLINT and the parens, what about inside the parens, how about after or before commas, etc).

Note on ABSL_HARDENING_ASSERT:
ABSL_HARDENING_ASSERT is defined as:

```
// ABSL_HARDENING_ASSERT()
//
// `ABSL_HARDENING_ASSERT()` is like `ABSL_ASSERT()`, but used to implement
// runtime assertions that should be enabled in hardened builds even when
// `NDEBUG` is defined.
//
// When `NDEBUG` is not defined, `ABSL_HARDENING_ASSERT()` is identical to
// `ABSL_ASSERT()`.
//
// See `ABSL_OPTION_HARDENED` in `absl/base/options.h` for more information on
// hardened mode.
#if ABSL_OPTION_HARDENED == 1 && defined(NDEBUG)
#define ABSL_HARDENING_ASSERT(expr)                 \
  (ABSL_PREDICT_TRUE((expr)) ? static_cast<void>(0) \
                             : [] { ABSL_INTERNAL_HARDENING_ABORT(); }())
#else
#define ABSL_HARDENING_ASSERT(expr) ABSL_ASSERT(expr)
#endif

// `ABSL_INTERNAL_HARDENING_ABORT()` controls how `ABSL_HARDENING_ASSERT()`
// aborts the program in release mode (when NDEBUG is defined). The
// implementation should abort the program as quickly as possible and ideally it
// should not be possible to ignore the abort request.
#if (ABSL_HAVE_BUILTIN(__builtin_trap) &&         \
     ABSL_HAVE_BUILTIN(__builtin_unreachable)) || \
    (defined(__GNUC__) && !defined(__clang__))
#define ABSL_INTERNAL_HARDENING_ABORT() \
  do {                                  \
    __builtin_trap();                   \
    __builtin_unreachable();            \
  } while (false)
#else
#define ABSL_INTERNAL_HARDENING_ABORT() abort()
#endif

// ABSL_HAVE_BUILTIN()
//
// Checks whether the compiler supports a Clang Feature Checking Macro, and if
// so, checks whether it supports the provided builtin function "x" where x
// is one of the functions noted in
// https://clang.llvm.org/docs/LanguageExtensions.html
//
// Note: Use this macro to avoid an extra level of #ifdef __has_builtin check.
// http://releases.llvm.org/3.3/tools/clang/docs/LanguageExtensions.html
```

```
#ifdef __has_builtin
#define ABSL_HAVE_BUILTIN(x) __has_builtin(x)
#else
#define ABSL_HAVE_BUILTIN(x) 0
#endif
```

Here is the complete code of the templatized class absl::Span with my comments in green italics interspersed between the lines when appropriate:

```
namespace absl {
ABSL_NAMESPACE_BEGIN

//-----------------------------------------------------------------------------
// Span
//-----------------------------------------------------------------------------
//
// A `Span` is an "array reference" type for holding a reference of contiguous
// array data; the `Span` object does not and cannot own such data itself. A
// span provides an easy way to provide overloads for anything operating on
// contiguous sequences without needing to manage pointers and array lengths
// manually.

// A span is conceptually a pointer (ptr) and a length (size) into an already
// existing array of contiguous memory; the array it represents references the
// elements "ptr[0] .. ptr[size-1]". Passing a properly-constructed `Span`
// instead of raw pointers avoids many issues related to index out of bounds
// errors.
//
// Spans may also be constructed from containers holding contiguous sequences.
// Such containers must supply `data()` and `size() const` methods (e.g
// `std::vector<T>`, `absl::InlinedVector<T, N>`). All implicit conversions to
// `absl::Span` from such containers will create spans of type `const T`;
// spans which can mutate their values (of type `T`) must use explicit
// constructors.
//
// A `Span<T>` is somewhat analogous to an `absl::string_view`, but for an array
// of elements of type `T`, and unlike an `absl::string_view`, a span can hold a
// reference to mutable data. A user of `Span` must ensure that the data being
// pointed to outlives the `Span` itself.
//
// You can construct a `Span<T>` in several ways:
//
//   * Explicitly from a reference to a container type
//   * Explicitly from a pointer and size
//   * Implicitly from a container type (but only for spans of type `const T`)
//   * Using the `MakeSpan()` or `MakeConstSpan()` factory functions.
//
// Examples:
//
//   // Construct a Span explicitly from a container:
//   std::vector<int> v = {1, 2, 3, 4, 5};
//   auto span = absl::Span<const int>(v);
//
//   // Construct a Span explicitly from a C-style array:
//   int a[5] =  {1, 2, 3, 4, 5};
//   auto span = absl::Span<const int>(a);
//
//   // Construct a Span implicitly from a container
//   void MyRoutine(absl::Span<const int> a) {
//     ...
//   }
//   std::vector v = {1,2,3,4,5};
//   MyRoutine(v)                     // convert to Span<const T>
//
// Note that `Span` objects, in addition to requiring that the memory they
// point to remains alive, must also ensure that such memory does not get
// reallocated. Therefore, to avoid undefined behavior, containers with
// associated spans should not invoke operations that may reallocate memory
// (such as resizing) or invalidate iterators into the container.
//
// One common use for a `Span` is when passing arguments to a routine that can
// accept a variety of array types (e.g. a `std::vector`, `absl::InlinedVector`,
// a C-style array, etc.). Instead of creating overloads for each case, you
// can simply specify a `Span` as the argument to such a routine.
//
// Example:
//
//   void MyRoutine(absl::Span<const int> a) {
//     ...
//   }
//
//   std::vector v = {1,2,3,4,5};
//   MyRoutine(v);
//
//   absl::InlinedVector<int, 4> my_inline_vector;
//   MyRoutine(my_inline_vector);
//
//   // Explicit constructor from pointer,size
//   int* my_array = new int[10];
//   MyRoutine(absl::Span<const int>(my_array, 10));
template <typename T>
class Span {
 private:
  // Used to determine whether a Span can be constructed from a container of
  // type C.
  template <typename C>
  using EnableIfConvertibleFrom =
      typename std::enable_if<span_internal::HasData<T, C>::value &&
                              span_internal::HasSize<C>::value>::type;

  // Used to SFINAE-enable a function when the slice elements are const.
  template <typename U>
  using EnableIfConstView =
      typename std::enable_if<std::is_const<T>::value, U>::type;

  // Used to SFINAE-enable a function when the slice elements are mutable.
  template <typename U>
  using EnableIfMutableView =
      typename std::enable_if<!std::is_const<T>::value, U>::type;

 public:
  using value_type = absl::remove_cv_t<T>;

// absl::remove_cv_t<T> is defined as
// template <typename T>
// using remove_cv_t = typename std::remove_cv<T>::type;

  using pointer = T*;
  using const_pointer = const T*;
  using reference = T&;
  using const_reference = const T&;
  using iterator = pointer;
  using const_iterator = const_pointer;
  using reverse_iterator = std::reverse_iterator<iterator>;
  using const_reverse_iterator = std::reverse_iterator<const_iterator>;
  using size_type = size_t;
  using difference_type = ptrdiff_t;

  static const size_type npos = ~(size_type(0));

  constexpr Span() noexcept : Span(nullptr, 0) {}
  constexpr Span(pointer array, size_type length) noexcept
```

```
    : ptr_(array), len_(length) {}

// Implicit conversion constructors
template <size_t N>
constexpr Span(T (&a)[N]) noexcept  // NOLINT(runtime/explicit)
    : Span(a, N) {}

// Explicit reference constructor for a mutable `Span<T>` type. Can be
// replaced with MakeSpan() to infer the type parameter.
template <typename V, typename = EnableIfConvertibleFrom<V>,
          typename = EnableIfMutableView<V>>
explicit Span(V& v) noexcept  // NOLINT(runtime/references)
    : Span(span_internal::GetData(v), v.size()) {}

// Implicit reference constructor for a read-only `Span<const T>` type
template <typename V, typename = EnableIfConvertibleFrom<V>,
          typename = EnableIfConstView<V>>
constexpr Span(const V& v) noexcept  // NOLINT(runtime/explicit)
    : Span(span_internal::GetData(v), v.size()) {}

// Implicit constructor from an initializer list, making it possible to pass a
// brace-enclosed initializer list to a function expecting a `Span`. Such
// spans constructed from an initializer list must be of type `Span<const T>`.
//
//   void Process(absl::Span<const int> x);
//   Process({1, 2, 3});
//
// Note that as always the array referenced by the span must outlive the span.
// Since an initializer list constructor acts as if it is fed a temporary
// array (cf. C++ standard [dcl.init.list]/5), it's safe to use this
// constructor only when the `std::initializer_list` itself outlives the span.
// In order to meet this requirement it's sufficient to ensure that neither
// the span nor a copy of it is used outside of the expression in which it's
// created:
//
//   // Assume that this function uses the array directly, not retaining any
//   // copy of the span or pointer to any of its elements.
//   void Process(absl::Span<const int> ints);
//
//   // Okay: the std::initializer_list<int> will reference a temporary array
//   // that isn't destroyed until after the call to Process returns.
//   Process({ 17, 19 });
//
//   // Not okay: the storage used by the std::initializer_list<int> is not
//   // allowed to be referenced after the first line.
//   absl::Span<const int> ints = { 17, 19 };
//   Process(ints);
//
//   // Not okay for the same reason as above: even when the elements of the
//   // initializer list expression are not temporaries the underlying array
//   // is, so the initializer list must still outlive the span.
//   const int foo = 17;
//   absl::Span<const int> ints = { foo };
//   Process(ints);
//
template <typename LazyT = T,
          typename = EnableIfConstView<LazyT>>
Span(
    std::initializer_list<value_type> v) noexcept  // NOLINT(runtime/explicit)
    : Span(v.begin(), v.size()) {}

// Accessors

// Span::data()
//
// Returns a pointer to the span's underlying array of data (which is held
// outside the span).
constexpr pointer data() const noexcept { return ptr_; }

// Span::size()
//
// Returns the size of this span.
constexpr size_type size() const noexcept { return len_; }

// Span::length()
//
// Returns the length (size) of this span.
constexpr size_type length() const noexcept { return size(); }

// Span::empty()
//
// Returns a boolean indicating whether or not this span is considered empty.
constexpr bool empty() const noexcept { return size() == 0; }


// Span::operator[]
//
// Returns a reference to the i'th element of this span.
```

```
constexpr reference operator[](size_type i) const noexcept {
  // MSVC 2015 accepts this as constexpr, but not ptr_[i]
  return ABSL_HARDENING_ASSERT(i < size()), *(data() + i);
}

// Span::at()
//
// Returns a reference to the i'th element of this span.
constexpr reference at(size_type i) const {
  return ABSL_PREDICT_TRUE(i < size())
             ? *(data() + i)
             : (base_internal::ThrowStdOutOfRange(
                    "Span::at failed bounds check"),
                *(data() + i));
}

// Span::front()
//
// Returns a reference to the first element of this span. The span must not
// be empty.
constexpr reference front() const noexcept {
  return ABSL_HARDENING_ASSERT(size() > 0), *data();
}

// Span::back()
//
// Returns a reference to the last element of this span. The span must not
// be empty.
constexpr reference back() const noexcept {
  return ABSL_HARDENING_ASSERT(size() > 0), *(data() + size() - 1);
}

// Span::begin()
//
// Returns an iterator pointing to the first element of this span, or `end()`
// if the span is empty.
constexpr iterator begin() const noexcept { return data(); }

// Span::cbegin()
//
// Returns a const iterator pointing to the first element of this span, or
// `end()` if the span is empty.
constexpr const_iterator cbegin() const noexcept { return begin(); }
```

```
// Span::end()
//
// Returns an iterator pointing just beyond the last element at the
// end of this span. This iterator acts as a placeholder; attempting to
// access it results in undefined behavior.
constexpr iterator end() const noexcept { return data() + size(); }

// Span::cend()
//
// Returns a const iterator pointing just beyond the last element at the
// end of this span. This iterator acts as a placeholder; attempting to
// access it results in undefined behavior.
constexpr const_iterator cend() const noexcept { return end(); }

// Span::rbegin()
//
// Returns a reverse iterator pointing to the last element at the end of this
// span, or `rend()` if the span is empty.
constexpr reverse_iterator rbegin() const noexcept {
  return reverse_iterator(end());
}

// Span::crbegin()
//
// Returns a const reverse iterator pointing to the last element at the end of
// this span, or `crend()` if the span is empty.
constexpr const_reverse_iterator crbegin() const noexcept { return rbegin(); }

// Span::rend()
//
// Returns a reverse iterator pointing just before the first element
// at the beginning of this span. This pointer acts as a placeholder;
// attempting to access its element results in undefined behavior.
constexpr reverse_iterator rend() const noexcept {
  return reverse_iterator(begin());
}

// Span::crend()
//
// Returns a reverse const iterator pointing just before the first element
// at the beginning of this span. This pointer acts as a placeholder;
// attempting to access its element results in undefined behavior.
constexpr const_reverse_iterator crend() const noexcept { return rend(); }
```

```
// Span mutations

// Span::remove_prefix()
//
// Removes the first `n` elements from the span.
void remove_prefix(size_type n) noexcept {
  ABSL_HARDENING_ASSERT(size() >= n);
  ptr_ += n;
  len_ -= n;
}

// Span::remove_suffix()
//
// Removes the last `n` elements from the span.
void remove_suffix(size_type n) noexcept {
  ABSL_HARDENING_ASSERT(size() >= n);
  len_ -= n;
}

// Span::subspan()
//
// Returns a `Span` starting at element `pos` and of length `len`. Both `pos`
// and `len` are of type `size_type` and thus non-negative. Parameter `pos`
// must be <= size(). Any `len` value that points past the end of the span
// will be trimmed to at most size() - `pos`. A default `len` value of `npos`
// ensures the returned subspan continues until the end of the span.
//
// Examples:
//
//   std::vector<int> vec = {10, 11, 12, 13};
//   absl::MakeSpan(vec).subspan(1, 2);  // {11, 12}
//   absl::MakeSpan(vec).subspan(2, 8);  // {12, 13}
//   absl::MakeSpan(vec).subspan(1);     // {11, 12, 13}
//   absl::MakeSpan(vec).subspan(4);     // {}
//   absl::MakeSpan(vec).subspan(5);     // throws std::out_of_range
constexpr Span subspan(size_type pos = 0, size_type len = npos) const {
  return (pos <= size())
           ? Span(data() + pos, span_internal::Min(size() - pos, len))
           : (base_internal::ThrowStdOutOfRange("pos > size()"), Span());
}

// Span::first()
//
// Returns a `Span` containing first `len` elements. Parameter `len` is of
```

```
  // type `size_type` and thus non-negative. `len` value must be <= size().
  //
  // Examples:
  //
  //   std::vector<int> vec = {10, 11, 12, 13};
  //   absl::MakeSpan(vec).first(1);  // {10}
  //   absl::MakeSpan(vec).first(3);  // {10, 11, 12}
  //   absl::MakeSpan(vec).first(5);  // throws std::out_of_range
  constexpr Span first(size_type len) const {
    return (len <= size())
               ? Span(data(), len)
               : (base_internal::ThrowStdOutOfRange("len > size()"), Span());
  }

  // Span::last()
  //
  // Returns a `Span` containing last `len` elements. Parameter `len` is of
  // type `size_type` and thus non-negative. `len` value must be <= size().
  //
  // Examples:
  //
  //   std::vector<int> vec = {10, 11, 12, 13};
  //   absl::MakeSpan(vec).last(1);  // {13}
  //   absl::MakeSpan(vec).last(3);  // {11, 12, 13}
  //   absl::MakeSpan(vec).last(5);  // throws std::out_of_range
  constexpr Span last(size_type len) const {
    return (len <= size())
               ? Span(size() - len + data(), len)
               : (base_internal::ThrowStdOutOfRange("len > size()"), Span());
  }

  // Support for absl::Hash.
  template <typename H>
  friend H AbslHashValue(H h, Span v) {
    return H::combine(H::combine_contiguous(std::move(h), v.data(), v.size()),
                      v.size());
  }

 private:
  pointer ptr_;
  size_type len_;
};

template <typename T>
```

```
const typename Span<T>::size_type Span<T>::npos;

// Span relationals

// Equality is compared element-by-element, while ordering is lexicographical.
// We provide three overloads for each operator to cover any combination on the
// left or right hand side of mutable Span<T>, read-only Span<const T>, and
// convertible-to-read-only Span<T>.
// TODO(zhangxy): Due to MSVC overload resolution bug with partial ordering
// template functions, 5 overloads per operator is needed as a workaround. We
// should update them to 3 overloads per operator using non-deduced context like
// string_view, i.e.
// - (Span<T>, Span<T>)
// - (Span<T>, non_deduced<Span<const T>>)
// - (non_deduced<Span<const T>>, Span<T>)

// operator==
template <typename T>
bool operator==(Span<T> a, Span<T> b) {
  return span_internal::EqualImpl<Span, const T>(a, b);
}
template <typename T>
bool operator==(Span<const T> a, Span<T> b) {
  return span_internal::EqualImpl<Span, const T>(a, b);
}
template <typename T>
bool operator==(Span<T> a, Span<const T> b) {
  return span_internal::EqualImpl<Span, const T>(a, b);
}
template <
    typename T, typename U,
    typename = span_internal::EnableIfConvertibleTo<U, absl::Span<const T>>>
bool operator==(const U& a, Span<T> b) {
  return span_internal::EqualImpl<Span, const T>(a, b);
}
template <
    typename T, typename U,
    typename = span_internal::EnableIfConvertibleTo<U, absl::Span<const T>>>
bool operator==(Span<T> a, const U& b) {
  return span_internal::EqualImpl<Span, const T>(a, b);
}

// operator!=
template <typename T>
```

```
bool operator!=(Span<T> a, Span<T> b) {
  return !(a == b);
}
template <typename T>
bool operator!=(Span<const T> a, Span<T> b) {
  return !(a == b);
}
template <typename T>
bool operator!=(Span<T> a, Span<const T> b) {
  return !(a == b);
}
template <
    typename T, typename U,
    typename = span_internal::EnableIfConvertibleTo<U, absl::Span<const T>>>
bool operator!=(const U& a, Span<T> b) {
  return !(a == b);
}
template <
    typename T, typename U,
    typename = span_internal::EnableIfConvertibleTo<U, absl::Span<const T>>>
bool operator!=(Span<T> a, const U& b) {
  return !(a == b);
}

// operator<
template <typename T>
bool operator<(Span<T> a, Span<T> b) {
  return span_internal::LessThanImpl<Span, const T>(a, b);
}
template <typename T>
bool operator<(Span<const T> a, Span<T> b) {
  return span_internal::LessThanImpl<Span, const T>(a, b);
}
template <typename T>
bool operator<(Span<T> a, Span<const T> b) {
  return span_internal::LessThanImpl<Span, const T>(a, b);
}
template <
    typename T, typename U,
    typename = span_internal::EnableIfConvertibleTo<U, absl::Span<const T>>>
bool operator<(const U& a, Span<T> b) {
  return span_internal::LessThanImpl<Span, const T>(a, b);
}
template <
```

```
    typename T, typename U,
    typename = span_internal::EnableIfConvertibleTo<U, absl::Span<const T>>>
bool operator<(Span<T> a, const U& b) {
  return span_internal::LessThanImpl<Span, const T>(a, b);
}

// operator>
template <typename T>
bool operator>(Span<T> a, Span<T> b) {
  return b < a;
}
template <typename T>
bool operator>(Span<const T> a, Span<T> b) {
  return b < a;
}
template <typename T>
bool operator>(Span<T> a, Span<const T> b) {
  return b < a;
}
template <
    typename T, typename U,
    typename = span_internal::EnableIfConvertibleTo<U, absl::Span<const T>>>
bool operator>(const U& a, Span<T> b) {
  return b < a;
}
template <
    typename T, typename U,
    typename = span_internal::EnableIfConvertibleTo<U, absl::Span<const T>>>
bool operator>(Span<T> a, const U& b) {
  return b < a;
}

// operator<=
template <typename T>
bool operator<=(Span<T> a, Span<T> b) {
  return !(b < a);
}
template <typename T>
bool operator<=(Span<const T> a, Span<T> b) {
  return !(b < a);
}
template <typename T>
bool operator<=(Span<T> a, Span<const T> b) {
  return !(b < a);
}
template <
    typename T, typename U,
    typename = span_internal::EnableIfConvertibleTo<U, absl::Span<const T>>>
bool operator<=(const U& a, Span<T> b) {
  return !(b < a);
}
template <
    typename T, typename U,
    typename = span_internal::EnableIfConvertibleTo<U, absl::Span<const T>>>
bool operator<=(Span<T> a, const U& b) {
  return !(b < a);
}

// operator>=
template <typename T>
bool operator>=(Span<T> a, Span<T> b) {
  return !(a < b);
}
template <typename T>
bool operator>=(Span<const T> a, Span<T> b) {
  return !(a < b);
}
template <typename T>
bool operator>=(Span<T> a, Span<const T> b) {
  return !(a < b);
}
template <
    typename T, typename U,
    typename = span_internal::EnableIfConvertibleTo<U, absl::Span<const T>>>
bool operator>=(const U& a, Span<T> b) {
  return !(a < b);
}
template <
    typename T, typename U,
    typename = span_internal::EnableIfConvertibleTo<U, absl::Span<const T>>>
bool operator>=(Span<T> a, const U& b) {
  return !(a < b);
}

// MakeSpan()
//
// Constructs a mutable `Span<T>`, deducing `T` automatically from either a
// container or pointer+size.
```

```
//
// Because a read-only `Span<const T>` is implicitly constructed from container
// types regardless of whether the container itself is a const container,
// constructing mutable spans of type `Span<T>` from containers requires
// explicit constructors. The container-accepting version of `MakeSpan()`
// deduces the type of `T` by the constness of the pointer received from the
// container's `data()` member. Similarly, the pointer-accepting version returns
// a `Span<const T>` if `T` is `const`, and a `Span<T>` otherwise.
//
// Examples:
//
//   void MyRoutine(absl::Span<MyComplicatedType> a) {
//     ...
//   };
//   // my_vector is a container of non-const types
//   std::vector<MyComplicatedType> my_vector;
//
//   // Constructing a Span implicitly attempts to create a Span of type
//   // `Span<const T>`
//   MyRoutine(my_vector);                // error, type mismatch
//
//   // Explicitly constructing the Span is verbose
//   MyRoutine(absl::Span<MyComplicatedType>(my_vector));
//
//   // Use MakeSpan() to make an absl::Span<T>
//   MyRoutine(absl::MakeSpan(my_vector));
//
//   // Construct a span from an array ptr+size
//   absl::Span<T> my_span() {
//     return absl::MakeSpan(&array[0], num_elements_);
//   }
//
template <int&... ExplicitArgumentBarrier, typename T>
constexpr Span<T> MakeSpan(T* ptr, size_t size) noexcept {
  return Span<T>(ptr, size);
}

template <int&... ExplicitArgumentBarrier, typename T>
Span<T> MakeSpan(T* begin, T* end) noexcept {
  return ABSL_HARDENING_ASSERT(begin <= end), Span<T>(begin, end - begin);
}

template <int&... ExplicitArgumentBarrier, typename C>
constexpr auto MakeSpan(C& c) noexcept  // NOLINT(runtime/references)
```

```
    -> decltype(absl::MakeSpan(span_internal::GetData(c), c.size())) {
  return MakeSpan(span_internal::GetData(c), c.size());
}

template <int&... ExplicitArgumentBarrier, typename T, size_t N>
constexpr Span<T> MakeSpan(T (&array)[N]) noexcept {
  return Span<T>(array, N);
}

// MakeConstSpan()
//
// Constructs a `Span<const T>` as with `MakeSpan`, deducing `T` automatically,
// but always returning a `Span<const T>`.
//
// Examples:
//
//   void ProcessInts(absl::Span<const int> some_ints);
//
//   // Call with a pointer and size.
//   int array[3] = { 0, 0, 0 };
//   ProcessInts(absl::MakeConstSpan(&array[0], 3));
//
//   // Call with a [begin, end) pair.
//   ProcessInts(absl::MakeConstSpan(&array[0], &array[3]));
//
//   // Call directly with an array.
//   ProcessInts(absl::MakeConstSpan(array));
//
//   // Call with a contiguous container.
//   std::vector<int> some_ints = ...;
//   ProcessInts(absl::MakeConstSpan(some_ints));
//   ProcessInts(absl::MakeConstSpan(std::vector<int>{ 0, 0, 0 }));
//
template <int&... ExplicitArgumentBarrier, typename T>
constexpr Span<const T> MakeConstSpan(T* ptr, size_t size) noexcept {
  return Span<const T>(ptr, size);
}

template <int&... ExplicitArgumentBarrier, typename T>
Span<const T> MakeConstSpan(T* begin, T* end) noexcept {
  return ABSL_HARDENING_ASSERT(begin <= end), Span<const T>(begin, end - begin);
}

template <int&... ExplicitArgumentBarrier, typename C>
```

```
constexpr auto MakeConstSpan(const C& c) noexcept -> decltype(MakeSpan(c)) {
  return MakeSpan(c);
}

template <int&... ExplicitArgumentBarrier, typename T, size_t N>
constexpr Span<const T> MakeConstSpan(const T (&array)[N]) noexcept {
  return Span<const T>(array, N);
}
ABSL_NAMESPACE_END
}  // namespace absl

namespace span_internal {
namespace absl {
ABSL_NAMESPACE_BEGIN

namespace span_internal {
// A constexpr min function
constexpr size_t Min(size_t a, size_t b) noexcept { return a < b ? a : b; }

// Wrappers for access to container data pointers.
template <typename C>
constexpr auto GetDataImpl(C& c, char) noexcept  // NOLINT(runtime/references)
    -> decltype(c.data()) {
  return c.data();
}

// Before C++17, std::string::data returns a const char* in all cases.
inline char* GetDataImpl(std::string& s,  // NOLINT(runtime/references)
                         int) noexcept {
  return &s[0];
}

template <typename C>
constexpr auto GetData(C& c) noexcept  // NOLINT(runtime/references)
    -> decltype(GetDataImpl(c, 0)) {
  return GetDataImpl(c, 0);
}

// Detection idioms for size() and data().
template <typename C>
using HasSize =
    std::is_integral<absl::decay_t<decltype(std::declval<C&>().size())>>;

// We want to enable conversion from vector<T*> to Span<const T* const> but
// disable conversion from vector<Derived> to Span<Base>. Here we use
// the fact that U** is convertible to Q* const* if and only if Q is the same
// type or a more cv-qualified version of U.  We also decay the result type of
// data() to avoid problems with classes which have a member function data()
// which returns a reference.
template <typename T, typename C>
using HasData =
    std::is_convertible<absl::decay_t<decltype(GetData(std::declval<C&>()))>*,
                        T* const*>;

// Extracts value type from a Container
template <typename C>
struct ElementType {
  using type = typename absl::remove_reference_t<C>::value_type;
};

template <typename T, size_t N>
struct ElementType<T (&)[N]> {
  using type = T;
};

template <typename C>
using ElementT = typename ElementType<C>::type;

template <typename T>
using EnableIfMutable =
    typename std::enable_if<!std::is_const<T>::value, int>::type;

template <template <typename> class SpanT, typename T>
bool EqualImpl(SpanT<T> a, SpanT<T> b) {
  static_assert(std::is_const<T>::value, "");
  return absl::equal(a.begin(), a.end(), b.begin(), b.end());
}

template <template <typename> class SpanT, typename T>
bool LessThanImpl(SpanT<T> a, SpanT<T> b) {
  // We can't use value_type since that is remove_cv_t<T>, so we go the long way
  // around.
  static_assert(std::is_const<T>::value, "");
  return std::lexicographical_compare(a.begin(), a.end(), b.begin(), b.end());
}

// The `IsConvertible` classes here are needed because of the
// `std::is_convertible` bug in libcxx when compiled with GCC. This build
// configuration is used by Android NDK toolchain. Reference link:
// https://bugs.llvm.org/show_bug.cgi?id=27538.
template <typename From, typename To>
struct IsConvertibleHelper {
 private:
  static std::true_type testval(To);
  static std::false_type testval(...);

 public:
  using type = decltype(testval(std::declval<From>()));
};

template <typename From, typename To>
struct IsConvertible : IsConvertibleHelper<From, To>::type {};

// TODO(zhangxy): replace `IsConvertible` with `std::is_convertible` once the
// older version of libcxx is not supported.
template <typename From, typename To>
using EnableIfConvertibleTo =
    typename std::enable_if<IsConvertible<From, To>::value>::type;
}  // namespace span_internal
ABSL_NAMESPACE_END
}  // namespace absl
```

Class InlinedVector

```
#include "absl/container/inlined_vector.h"  // IWYU pragma: export
// TODO(kramerb): This is kept only because lots of targets transitively depend
// on it. Remove all targets' dependencies.
#include "tensorflow/core/platform/macros.h"
#include "tensorflow/core/platform/types.h"

namespace tensorflow {
namespace gtl {

using absl::InlinedVector;

}  // namespace gtl
}  // namespace tensorflow
```

Absl/container/inlined_vector.h:

```
// This header file contains the declaration and definition of an "inlined
// vector" which behaves in an equivalent fashion to a `std::vector`, except
// that storage for small sequences of the vector are provided inline without
// requiring any heap allocation.
//
// An `absl::InlinedVector<T, N>` specifies the default capacity `N` as one of
// its template parameters. Instances where `size() <= N` hold contained
// elements in inline space. Typically `N` is very small so that sequences that
// are expected to be short do not require allocations.
//
// An `absl::InlinedVector` does not usually require a specific allocator. If
// the inlined vector grows beyond its initial constraints, it will need to
// allocate (as any normal `std::vector` would). This is usually performed with
// the default allocator (defined as `std::allocator<T>`). Optionally, a custom
// allocator type may be specified as `A` in `absl::InlinedVector<T, N, A>`.

#include "absl/container/internal/inlined_vector.h"
…
(more includes)

namespace absl {
ABSL_NAMESPACE_BEGIN
// -----------------------------------------------------------------------------
// InlinedVector
// -----------------------------------------------------------------------------
//
// An `absl::InlinedVector` is designed to be a drop-in replacement for
// `std::vector` for use cases where the vector's size is sufficiently small
// that it can be inlined. If the inlined vector does grow beyond its estimated
// capacity, it will trigger an initial allocation on the heap, and will behave
// as a `std:vector`. The API of the `absl::InlinedVector` within this file is
// designed to cover the same API footprint as covered by `std::vector`.
template <typename T, size_t N, typename A = std::allocator<T>>
class InlinedVector {
  static_assert(N > 0, "`absl::InlinedVector` requires an inlined capacity.");

  using Storage = inlined_vector_internal::Storage<T, N, A>;

  using AllocatorTraits = typename Storage::AllocatorTraits;
  using RValueReference = typename Storage::RValueReference;
  using MoveIterator = typename Storage::MoveIterator;
  using IsMemcpyOk = typename Storage::IsMemcpyOk;

  template <typename Iterator>
  using IteratorValueAdapter =
      typename Storage::template IteratorValueAdapter<Iterator>;
  using CopyValueAdapter = typename Storage::CopyValueAdapter;
  using DefaultValueAdapter = typename Storage::DefaultValueAdapter;

  template <typename Iterator>
  using EnableIfAtLeastForwardIterator = absl::enable_if_t<
      inlined_vector_internal::IsAtLeastForwardIterator<Iterator>::value>;
  template <typename Iterator>
  using DisableIfAtLeastForwardIterator = absl::enable_if_t<
      !inlined_vector_internal::IsAtLeastForwardIterator<Iterator>::value>;

 public:
  using allocator_type = typename Storage::allocator_type;
  using value_type = typename Storage::value_type;
  using pointer = typename Storage::pointer;
  using const_pointer = typename Storage::const_pointer;
  using size_type = typename Storage::size_type;
  using difference_type = typename Storage::difference_type;
  using reference = typename Storage::reference;
  using const_reference = typename Storage::const_reference;
  using iterator = typename Storage::iterator;
  using const_iterator = typename Storage::const_iterator;
  using reverse_iterator = typename Storage::reverse_iterator;
  using const_reverse_iterator = typename Storage::const_reverse_iterator;

  // ---------------------------------------------------------------------------
  // InlinedVector Constructors and Destructor
  // ---------------------------------------------------------------------------

  // Creates an empty inlined vector with a value-initialized allocator.
  InlinedVector() noexcept(noexcept(allocator_type())) : storage_() {}

  // Creates an empty inlined vector with a copy of `alloc`.
  explicit InlinedVector(const allocator_type& alloc) noexcept
      : storage_(alloc) {}

  // Creates an inlined vector with `n` copies of `value_type()`.
  explicit InlinedVector(size_type n,
                         const allocator_type& alloc = allocator_type())
      : storage_(alloc) {
    storage_.Initialize(DefaultValueAdapter(), n);
  }

  // Creates an inlined vector with `n` copies of `v`.
  InlinedVector(size_type n, const_reference v,
            const allocator_type& alloc = allocator_type())
    : storage_(alloc) {
  storage_.Initialize(CopyValueAdapter(v), n);
}

// Creates an inlined vector with copies of the elements of `list`.
InlinedVector(std::initializer_list<value_type> list,
            const allocator_type& alloc = allocator_type())
    : InlinedVector(list.begin(), list.end(), alloc) {}

// Creates an inlined vector with elements constructed from the provided
// forward iterator range [`first`, `last`).
//
// NOTE: the `enable_if` prevents ambiguous interpretation between a call to
// this constructor with two integral arguments and a call to the above
// `InlinedVector(size_type, const_reference)` constructor.
template <typename ForwardIterator,
         EnableIfAtLeastForwardIterator<ForwardIterator>* = nullptr>
InlinedVector(ForwardIterator first, ForwardIterator last,
            const allocator_type& alloc = allocator_type())
    : storage_(alloc) {
  storage_.Initialize(IteratorValueAdapter<ForwardIterator>(first),
                  std::distance(first, last));
}

// Creates an inlined vector with elements constructed from the provided input
// iterator range [`first`, `last`).
template <typename InputIterator,
         DisableIfAtLeastForwardIterator<InputIterator>* = nullptr>
InlinedVector(InputIterator first, InputIterator last,
            const allocator_type& alloc = allocator_type())
    : storage_(alloc) {
  std::copy(first, last, std::back_inserter(*this));
}

// Creates an inlined vector by copying the contents of `other` using
// `other`'s allocator.
InlinedVector(const InlinedVector& other)
    : InlinedVector(other, *other.storage_.GetAllocPtr()) {}

// Creates an inlined vector by copying the contents of `other` using `alloc`.
InlinedVector(const InlinedVector& other, const allocator_type& alloc)
    : storage_(alloc) {
  if (IsMemcpyOk::value && !other.storage_.GetIsAllocated()) {
    storage_.MemcpyFrom(other.storage_);
  } else {
    storage_.Initialize(IteratorValueAdapter<const_pointer>(other.data()),
                        other.size());
  }
}

// Creates an inlined vector by moving in the contents of `other` without
// allocating. If `other` contains allocated memory, the newly-created inlined
// vector will take ownership of that memory. However, if `other` does not
// contain allocated memory, the newly-created inlined vector will perform
// element-wise move construction of the contents of `other`.
//
// NOTE: since no allocation is performed for the inlined vector in either
// case, the `noexcept(...)` specification depends on whether moving the
// underlying objects can throw. It is assumed assumed that...
//  a) move constructors should only throw due to allocation failure.
//  b) if `value_type`'s move constructor allocates, it uses the same
//     allocation function as the inlined vector's allocator.
// Thus, the move constructor is non-throwing if the allocator is non-throwing
// or `value_type`'s move constructor is specified as `noexcept`.
InlinedVector(InlinedVector&& other) noexcept(
    absl::allocator_is_nothrow<allocator_type>::value ||
    std::is_nothrow_move_constructible<value_type>::value)
    : storage_(*other.storage_.GetAllocPtr()) {
  if (IsMemcpyOk::value) {
    storage_.MemcpyFrom(other.storage_);

    other.storage_.SetInlinedSize(0);
  } else if (other.storage_.GetIsAllocated()) {
    storage_.SetAllocatedData(other.storage_.GetAllocatedData(),
                              other.storage_.GetAllocatedCapacity());
    storage_.SetAllocatedSize(other.storage_.GetSize());

    other.storage_.SetInlinedSize(0);
  } else {
    IteratorValueAdapter<MoveIterator> other_values(
        MoveIterator(other.storage_.GetInlinedData()));

    inlined_vector_internal::ConstructElements(
        storage_.GetAllocPtr(), storage_.GetInlinedData(), &other_values,
        other.storage_.GetSize());

    storage_.SetInlinedSize(other.storage_.GetSize());
  }
}

// Creates an inlined vector by moving in the contents of `other` with a copy
// of `alloc`.
//
// NOTE: if `other`'s allocator is not equal to `alloc`, even if `other`
// contains allocated memory, this move constructor will still allocate. Since
// allocation is performed, this constructor can only be `noexcept` if the
// specified allocator is also `noexcept`.
InlinedVector(InlinedVector&& other, const allocator_type& alloc) noexcept(
    absl::allocator_is_nothrow<allocator_type>::value)
    : storage_(alloc) {
  if (IsMemcpyOk::value) {
    storage_.MemcpyFrom(other.storage_);

    other.storage_.SetInlinedSize(0);
  } else if ((*storage_.GetAllocPtr() == *other.storage_.GetAllocPtr()) &&
             other.storage_.GetIsAllocated()) {
    storage_.SetAllocatedData(other.storage_.GetAllocatedData(),
                              other.storage_.GetAllocatedCapacity());
    storage_.SetAllocatedSize(other.storage_.GetSize());

    other.storage_.SetInlinedSize(0);
  } else {
    storage_.Initialize(
        IteratorValueAdapter<MoveIterator>(MoveIterator(other.data())),
        other.size());
  }
}

~InlinedVector() {}

// ---------------------------------------------------------------------------
// InlinedVector Member Accessors
// ---------------------------------------------------------------------------

// `InlinedVector::empty()`
//
// Returns whether the inlined vector contains no elements.
bool empty() const noexcept { return !size(); }

// `InlinedVector::size()`
//
```

```
// Returns the number of elements in the inlined vector.
size_type size() const noexcept { return storage_.GetSize(); }

// `InlinedVector::max_size()`
//
// Returns the maximum number of elements the inlined vector can hold.
size_type max_size() const noexcept {
  // One bit of the size storage is used to indicate whether the inlined
  // vector contains allocated memory. As a result, the maximum size that the
  // inlined vector can express is half of the max for `size_type`.
  return (std::numeric_limits<size_type>::max)() / 2;
}

// `InlinedVector::capacity()`
//
// Returns the number of elements that could be stored in the inlined vector
// without requiring a reallocation.
//
// NOTE: for most inlined vectors, `capacity()` should be equal to the
// template parameter `N`. For inlined vectors which exceed this capacity,
// they will no longer be inlined and `capacity()` will equal the capactity of
// the allocated memory.
size_type capacity() const noexcept {
  return storage_.GetIsAllocated() ? storage_.GetAllocatedCapacity()
                                   : storage_.GetInlinedCapacity();
}

// `InlinedVector::data()`
//
// Returns a `pointer` to the elements of the inlined vector. This pointer
// can be used to access and modify the contained elements.
//
// NOTE: only elements within [`data()`, `data() + size()`) are valid.
pointer data() noexcept {
  return storage_.GetIsAllocated() ? storage_.GetAllocatedData()
                                   : storage_.GetInlinedData();
}

// Overload of `InlinedVector::data()` that returns a `const_pointer` to the
// elements of the inlined vector. This pointer can be used to access but not
// modify the contained elements.
//
// NOTE: only elements within [`data()`, `data() + size()`) are valid.
const_pointer data() const noexcept {
  return storage_.GetIsAllocated() ? storage_.GetAllocatedData()
                                   : storage_.GetInlinedData();
}

// `InlinedVector::operator[](...)`
//
// Returns a `reference` to the `i`th element of the inlined vector.
reference operator[](size_type i) {
  assert(i < size());

  return data()[i];
}

// Overload of `InlinedVector::operator[](...)` that returns a
// `const_reference` to the `i`th element of the inlined vector.
const_reference operator[](size_type i) const {
  assert(i < size());

  return data()[i];
}

// `InlinedVector::at(...)`
//
// Returns a `reference` to the `i`th element of the inlined vector.
//
// NOTE: if `i` is not within the required range of `InlinedVector::at(...)`,
// in both debug and non-debug builds, `std::out_of_range` will be thrown.
reference at(size_type i) {
  if (ABSL_PREDICT_FALSE(i >= size())) {
    base_internal::ThrowStdOutOfRange(
        "`InlinedVector::at(size_type)` failed bounds check");
  }

  return data()[i];
}

// Overload of `InlinedVector::at(...)` that returns a `const_reference` to
// the `i`th element of the inlined vector.
//
// NOTE: if `i` is not within the required range of `InlinedVector::at(...)`,
// in both debug and non-debug builds, `std::out_of_range` will be thrown.
const_reference at(size_type i) const {
  if (ABSL_PREDICT_FALSE(i >= size())) {
    base_internal::ThrowStdOutOfRange(
        "`InlinedVector::at(size_type) const` failed bounds check");
  }

  return data()[i];
}

// `InlinedVector::front()`
//
// Returns a `reference` to the first element of the inlined vector.
reference front() {
  assert(!empty());

  return at(0);
}

// Overload of `InlinedVector::front()` that returns a `const_reference` to
// the first element of the inlined vector.
const_reference front() const {
  assert(!empty());

  return at(0);
}

// `InlinedVector::back()`
//
// Returns a `reference` to the last element of the inlined vector.
reference back() {
  assert(!empty());

  return at(size() - 1);
}

// Overload of `InlinedVector::back()` that returns a `const_reference` to the
// last element of the inlined vector.
const_reference back() const {
  assert(!empty());

  return at(size() - 1);
}

// `InlinedVector::begin()`
//
// Returns an `iterator` to the beginning of the inlined vector.
iterator begin() noexcept { return data(); }
```

```
// Overload of `InlinedVector::begin()` that returns a `const_iterator` to
// the beginning of the inlined vector.
const_iterator begin() const noexcept { return data(); }

// `InlinedVector::end()`
//
// Returns an `iterator` to the end of the inlined vector.
iterator end() noexcept { return data() + size(); }

// Overload of `InlinedVector::end()` that returns a `const_iterator` to the
// end of the inlined vector.
const_iterator end() const noexcept { return data() + size(); }

// `InlinedVector::cbegin()`
//
// Returns a `const_iterator` to the beginning of the inlined vector.
const_iterator cbegin() const noexcept { return begin(); }

// `InlinedVector::cend()`
//
// Returns a `const_iterator` to the end of the inlined vector.
const_iterator cend() const noexcept { return end(); }

// `InlinedVector::rbegin()`
//
// Returns a `reverse_iterator` from the end of the inlined vector.
reverse_iterator rbegin() noexcept { return reverse_iterator(end()); }

// Overload of `InlinedVector::rbegin()` that returns a
// `const_reverse_iterator` from the end of the inlined vector.
const_reverse_iterator rbegin() const noexcept {
  return const_reverse_iterator(end());
}

// `InlinedVector::rend()`
//
// Returns a `reverse_iterator` from the beginning of the inlined vector.
reverse_iterator rend() noexcept { return reverse_iterator(begin()); }

// Overload of `InlinedVector::rend()` that returns a `const_reverse_iterator`
// from the beginning of the inlined vector.
const_reverse_iterator rend() const noexcept {
  return const_reverse_iterator(begin());
}

// `InlinedVector::crbegin()`
//
// Returns a `const_reverse_iterator` from the end of the inlined vector.
const_reverse_iterator crbegin() const noexcept { return rbegin(); }

// `InlinedVector::crend()`
//
// Returns a `const_reverse_iterator` from the beginning of the inlined
// vector.
const_reverse_iterator crend() const noexcept { return rend(); }

// `InlinedVector::get_allocator()`
//
// Returns a copy of the inlined vector's allocator.
allocator_type get_allocator() const { return *storage_.GetAllocPtr(); }

// --------------------------------------------------------------------------
// InlinedVector Member Mutators
// --------------------------------------------------------------------------

// `InlinedVector::operator=(...)`
//
// Replaces the elements of the inlined vector with copies of the elements of
// `list`.
InlinedVector& operator=(std::initializer_list<value_type> list) {
  assign(list.begin(), list.end());

  return *this;
}

// Overload of `InlinedVector::operator=(...)` that replaces the elements of
// the inlined vector with copies of the elements of `other`.
InlinedVector& operator=(const InlinedVector& other) {
  if (ABSL_PREDICT_TRUE(this != std::addressof(other))) {
    const_pointer other_data = other.data();
    assign(other_data, other_data + other.size());
  }

  return *this;
}

// Overload of `InlinedVector::operator=(...)` that moves the elements of
// `other` into the inlined vector.
//
// NOTE: as a result of calling this overload, `other` is left in a valid but
// unspecified state.
InlinedVector& operator=(InlinedVector&& other) {
  if (ABSL_PREDICT_TRUE(this != std::addressof(other))) {
    if (IsMemcpyOk::value || other.storage_.GetIsAllocated()) {
      inlined_vector_internal::DestroyElements(storage_.GetAllocPtr(), data(),
                                               size());
      storage_.DeallocateIfAllocated();
      storage_.MemcpyFrom(other.storage_);

      other.storage_.SetInlinedSize(0);
    } else {
      storage_.Assign(IteratorValueAdapter<MoveIterator>(
                          MoveIterator(other.storage_.GetInlinedData())),
                      other.size());
    }
  }

  return *this;
}

// `InlinedVector::assign(...)`
//
// Replaces the contents of the inlined vector with `n` copies of `v`.
void assign(size_type n, const_reference v) {
  storage_.Assign(CopyValueAdapter(v), n);
}

// Overload of `InlinedVector::assign(...)` that replaces the contents of the
// inlined vector with copies of the elements of `list`.
void assign(std::initializer_list<value_type> list) {
  assign(list.begin(), list.end());
}

// Overload of `InlinedVector::assign(...)` to replace the contents of the
// inlined vector with the range [`first`, `last`).
//
// NOTE: this overload is for iterators that are "forward" category or better.
template <typename ForwardIterator,
          EnableIfAtLeastForwardIterator<ForwardIterator>* = nullptr>
void assign(ForwardIterator first, ForwardIterator last) {
  storage_.Assign(IteratorValueAdapter<ForwardIterator>(first),
                  std::distance(first, last));
}

// Overload of `InlinedVector::assign(...)` to replace the contents of the
// inlined vector with the range [`first`, `last`).
//
// NOTE: this overload is for iterators that are "input" category.
template <typename InputIterator,
          DisableIfAtLeastForwardIterator<InputIterator>* = nullptr>
void assign(InputIterator first, InputIterator last) {
  size_type i = 0;
  for (; i < size() && first != last; ++i, static_cast<void>(++first)) {
    at(i) = *first;
  }

  erase(data() + i, data() + size());
  std::copy(first, last, std::back_inserter(*this));
}

// `InlinedVector::resize(...)`
//
// Resizes the inlined vector to contain `n` elements.
//
// NOTE: if `n` is smaller than `size()`, extra elements are destroyed. If `n`
// is larger than `size()`, new elements are value-initialized.
void resize(size_type n) { storage_.Resize(DefaultValueAdapter(), n); }

// Overload of `InlinedVector::resize(...)` that resizes the inlined vector to
// contain `n` elements.
//
// NOTE: if `n` is smaller than `size()`, extra elements are destroyed. If `n`
// is larger than `size()`, new elements are copied-constructed from `v`.
void resize(size_type n, const_reference v) {
  storage_.Resize(CopyValueAdapter(v), n);
}

// `InlinedVector::insert(...)`
//
// Inserts a copy of `v` at `pos`, returning an `iterator` to the newly
// inserted element.
iterator insert(const_iterator pos, const_reference v) {
  return emplace(pos, v);
}
```

```
// Overload of `InlinedVector::insert(...)` that inserts `v` at `pos` using
// move semantics, returning an `iterator` to the newly inserted element.
iterator insert(const_iterator pos, RValueReference v) {
  return emplace(pos, std::move(v));
}

// Overload of `InlinedVector::insert(...)` that inserts `n` contiguous copies
// of `v` starting at `pos`, returning an `iterator` pointing to the first of
// the newly inserted elements.
iterator insert(const_iterator pos, size_type n, const_reference v) {
  assert(pos >= begin());
  assert(pos <= end());

  if (ABSL_PREDICT_TRUE(n != 0)) {
    value_type dealias = v;
    return storage_.Insert(pos, CopyValueAdapter(dealias), n);
  } else {
    return const_cast<iterator>(pos);
  }
}

// Overload of `InlinedVector::insert(...)` that inserts copies of the
// elements of `list` starting at `pos`, returning an `iterator` pointing to
// the first of the newly inserted elements.
iterator insert(const_iterator pos, std::initializer_list<value_type> list) {
  return insert(pos, list.begin(), list.end());
}

// Overload of `InlinedVector::insert(...)` that inserts the range [`first`,
// `last`) starting at `pos`, returning an `iterator` pointing to the first
// of the newly inserted elements.
//
// NOTE: this overload is for iterators that are "forward" category or better.
template <typename ForwardIterator,
          EnableIfAtLeastForwardIterator<ForwardIterator>* = nullptr>
iterator insert(const_iterator pos, ForwardIterator first,
                ForwardIterator last) {
  assert(pos >= begin());
  assert(pos <= end());

  if (ABSL_PREDICT_TRUE(first != last)) {
    return storage_.Insert(pos, IteratorValueAdapter<ForwardIterator>(first),
                           std::distance(first, last));
  } else {
    return const_cast<iterator>(pos);
  }
}

// Overload of `InlinedVector::insert(...)` that inserts the range [`first`,
// `last`) starting at `pos`, returning an `iterator` pointing to the first
// of the newly inserted elements.
//
// NOTE: this overload is for iterators that are "input" category.
template <typename InputIterator,
          DisableIfAtLeastForwardIterator<InputIterator>* = nullptr>
iterator insert(const_iterator pos, InputIterator first, InputIterator last) {
  assert(pos >= begin());
  assert(pos <= end());

  size_type index = std::distance(cbegin(), pos);
  for (size_type i = index; first != last; ++i, static_cast<void>(++first)) {
    insert(data() + i, *first);
  }

  return iterator(data() + index);
}

// `InlinedVector::emplace(...)`
//
// Constructs and inserts an element using `args...` in the inlined vector at
// `pos`, returning an `iterator` pointing to the newly emplaced element.
template <typename... Args>
iterator emplace(const_iterator pos, Args&&... args) {
  assert(pos >= begin());
  assert(pos <= end());

  value_type dealias(std::forward<Args>(args)...);
  return storage_.Insert(pos,
                         IteratorValueAdapter<MoveIterator>(
                             MoveIterator(std::addressof(dealias))),
                         1);
}

// `InlinedVector::emplace_back(...)`
//
// Constructs and inserts an element using `args...` in the inlined vector at
// `end()`, returning a `reference` to the newly emplaced element.
template <typename... Args>
```

```
reference emplace_back(Args&&... args) {
  return storage_.EmplaceBack(std::forward<Args>(args)...);
}

// `InlinedVector::push_back(...)`
//
// Inserts a copy of `v` in the inlined vector at `end()`.
void push_back(const_reference v) { static_cast<void>(emplace_back(v)); }

// Overload of `InlinedVector::push_back(...)` for inserting `v` at `end()`
// using move semantics.
void push_back(RValueReference v) {
  static_cast<void>(emplace_back(std::move(v)));
}

// `InlinedVector::pop_back()`
//
// Destroys the element at `back()`, reducing the size by `1`.
void pop_back() noexcept {
  assert(!empty());

  AllocatorTraits::destroy(*storage_.GetAllocPtr(), data() + (size() - 1));
  storage_.SubtractSize(1);
}

// `InlinedVector::erase(...)`
//
// Erases the element at `pos`, returning an `iterator` pointing to where the
// erased element was located.
//
// NOTE: may return `end()`, which is not dereferencable.
iterator erase(const_iterator pos) {
  assert(pos >= begin());
  assert(pos < end());

  return storage_.Erase(pos, pos + 1);
}

// Overload of `InlinedVector::erase(...)` that erases every element in the
// range [`from`, `to`), returning an `iterator` pointing to where the first
// erased element was located.
//
// NOTE: may return `end()`, which is not dereferencable.
iterator erase(const_iterator from, const_iterator to) {
  assert(from >= begin());
  assert(from <= to);
  assert(to <= end());

  if (ABSL_PREDICT_TRUE(from != to)) {
    return storage_.Erase(from, to);
  } else {
    return const_cast<iterator>(from);
  }
}

// `InlinedVector::clear()`
//
// Destroys all elements in the inlined vector, setting the size to `0` and
// deallocating any held memory.
void clear() noexcept {
  inlined_vector_internal::DestroyElements(storage_.GetAllocPtr(), data(),
                                           size());
  storage_.DeallocateIfAllocated();

  storage_.SetInlinedSize(0);
}

// `InlinedVector::reserve(...)`
//
// Ensures that there is enough room for at least `n` elements.
void reserve(size_type n) { storage_.Reserve(n); }

// `InlinedVector::shrink_to_fit()`
//
// Reduces memory usage by freeing unused memory. After being called, calls to
// `capacity()` will be equal to `max(N, size())`.
//
// If `size() <= N` and the inlined vector contains allocated memory, the
// elements will all be moved to the inlined space and the allocated memory
// will be deallocated.
//
// If `size() > N` and `size() < capacity()`, the elements will be moved to a
// smaller allocation.
void shrink_to_fit() {
  if (storage_.GetIsAllocated()) {
    storage_.ShrinkToFit();
  }
}
```

```
  // `InlinedVector::swap(...)`
  //
  // Swaps the contents of the inlined vector with `other`.
  void swap(InlinedVector& other) {
    if (ABSL_PREDICT_TRUE(this != std::addressof(other))) {
      storage_.Swap(std::addressof(other.storage_));
    }
  }

 private:
  template <typename H, typename TheT, size_t TheN, typename TheA>
  friend H AbslHashValue(H h, const absl::InlinedVector<TheT, TheN, TheA>& a);

  Storage storage_;
};

// -----------------------------------------------------------------------------
// InlinedVector Non-Member Functions
// -----------------------------------------------------------------------------

// `swap(...)`
//
// Swaps the contents of two inlined vectors.
template <typename T, size_t N, typename A>
void swap(absl::InlinedVector<T, N, A>& a,
          absl::InlinedVector<T, N, A>& b) noexcept(noexcept(a.swap(b))) {
  a.swap(b);
}

// `operator==(...)`
//
// Tests for value-equality of two inlined vectors.
template <typename T, size_t N, typename A>
bool operator==(const absl::InlinedVector<T, N, A>& a,
                const absl::InlinedVector<T, N, A>& b) {
  auto a_data = a.data();
  auto b_data = b.data();
  return absl::equal(a_data, a_data + a.size(), b_data, b_data + b.size());
}

// `operator!=(...)`
//
// Tests for value-inequality of two inlined vectors.
```

```
template <typename T, size_t N, typename A>
bool operator!=(const absl::InlinedVector<T, N, A>& a,
                const absl::InlinedVector<T, N, A>& b) {
  return !(a == b);
}

// `operator<(...)`
//
// Tests whether the value of an inlined vector is less than the value of
// another inlined vector using a lexicographical comparison algorithm.
template <typename T, size_t N, typename A>
bool operator<(const absl::InlinedVector<T, N, A>& a,
               const absl::InlinedVector<T, N, A>& b) {
  auto a_data = a.data();
  auto b_data = b.data();
  return std::lexicographical_compare(a_data, a_data + a.size(), b_data,
                                      b_data + b.size());
}

// `operator>(...)`
//
// Tests whether the value of an inlined vector is greater than the value of
// another inlined vector using a lexicographical comparison algorithm.
template <typename T, size_t N, typename A>
bool operator>(const absl::InlinedVector<T, N, A>& a,
               const absl::InlinedVector<T, N, A>& b) {
  return b < a;
}

// `operator<=(...)`
//
// Tests whether the value of an inlined vector is less than or equal to the
// value of another inlined vector using a lexicographical comparison algorithm.
template <typename T, size_t N, typename A>
bool operator<=(const absl::InlinedVector<T, N, A>& a,
                const absl::InlinedVector<T, N, A>& b) {
  return !(b < a);
}

// `operator>=(...)`
//
// Tests whether the value of an inlined vector is greater than or equal to the
// value of another inlined vector using a lexicographical comparison algorithm.
template <typename T, size_t N, typename A>
```

```
bool operator>=(const absl::InlinedVector<T, N, A>& a,
                const absl::InlinedVector<T, N, A>& b) {
  return !(a < b);
}

// `AbslHashValue(...)`
//
// Provides `absl::Hash` support for `absl::InlinedVector`. It is uncommon to
// call this directly.
template <typename H, typename T, size_t N, typename A>
H AbslHashValue(H h, const absl::InlinedVector<T, N, A>& a) {
  auto size = a.size();
  return H::combine(H::combine_contiguous(std::move(h), a.data(), size), size);
}

ABSL_NAMESPACE_END
}  // namespace absl
```

Absl/container/internal/inlined_vector.h

The excerpt from the source code of the internal/inlined_vector.h is shown below. It contains template utility code used to build the InlinedVector container:

```
namespace absl {
ABSL_NAMESPACE_BEGIN
namespace inlined_vector_internal {

template <typename Iterator>
using IsAtLeastForwardIterator = std::is_convertible<
    typename std::iterator_traits<Iterator>::iterator_category,
    std::forward_iterator_tag>;

template <typename AllocatorType,
          typename ValueType =
              typename absl::allocator_traits<AllocatorType>::value_type>
using IsMemcpyOk =
    absl::conjunction<std::is_same<AllocatorType, std::allocator<ValueType>>,
                      absl::is_trivially_copy_constructible<ValueType>,
                      absl::is_trivially_copy_assignable<ValueType>,
                      absl::is_trivially_destructible<ValueType>>;

template <typename AllocatorType, typename Pointer, typename SizeType>
void DestroyElements(AllocatorType* alloc_ptr, Pointer destroy_first,
                     SizeType destroy_size) {
  using AllocatorTraits = absl::allocator_traits<AllocatorType>;
  if (destroy_first != nullptr) {
    for (auto i = destroy_size; i != 0;) {
      --i;
      AllocatorTraits::destroy(*alloc_ptr, destroy_first + i);
    }

#if !defined(NDEBUG)
    {
      using ValueType = typename AllocatorTraits::value_type;

      // Overwrite unused memory with `0xab` so we can catch uninitialized
      // usage.
      //
      // Cast to `void*` to tell the compiler that we don't care that we might
      // be scribbling on a vtable pointer.
      void* memory_ptr = destroy_first;
      auto memory_size = destroy_size * sizeof(ValueType);
      std::memset(memory_ptr, 0xab, memory_size);
    }
#endif  // !defined(NDEBUG)
  }
}

template <typename AllocatorType, typename Pointer, typename ValueAdapter,
          typename SizeType>
void ConstructElements(AllocatorType* alloc_ptr, Pointer construct_first,
                       ValueAdapter* values_ptr, SizeType construct_size) {
  for (SizeType i = 0; i < construct_size; ++i) {
    ABSL_INTERNAL_TRY {
      values_ptr->ConstructNext(alloc_ptr, construct_first + i);
    }
    ABSL_INTERNAL_CATCH_ANY {
      inlined_vector_internal::DestroyElements(alloc_ptr, construct_first, i);
      ABSL_INTERNAL_RETHROW;
    }
  }
}

template <typename Pointer, typename ValueAdapter, typename SizeType>
void AssignElements(Pointer assign_first, ValueAdapter* values_ptr,
                    SizeType assign_size) {
  for (SizeType i = 0; i < assign_size; ++i) {
    values_ptr->AssignNext(assign_first + i);
  }
}

template <typename AllocatorType>
struct StorageView {
  using AllocatorTraits = absl::allocator_traits<AllocatorType>;
  using Pointer = typename AllocatorTraits::pointer;
  using SizeType = typename AllocatorTraits::size_type;

  Pointer data;
  SizeType size;
  SizeType capacity;
};

template <typename AllocatorType, typename Iterator>
class IteratorValueAdapter {
  using AllocatorTraits = absl::allocator_traits<AllocatorType>;
  using Pointer = typename AllocatorTraits::pointer;

 public:
  explicit IteratorValueAdapter(const Iterator& it) : it_(it) {}

  void ConstructNext(AllocatorType* alloc_ptr, Pointer construct_at) {
    AllocatorTraits::construct(*alloc_ptr, construct_at, *it_);
    ++it_;
  }

  void AssignNext(Pointer assign_at) {
    *assign_at = *it_;
    ++it_;
  }

 private:
  Iterator it_;
};

template <typename AllocatorType>
class CopyValueAdapter {
  using AllocatorTraits = absl::allocator_traits<AllocatorType>;
  using ValueType = typename AllocatorTraits::value_type;
  using Pointer = typename AllocatorTraits::pointer;
  using ConstPointer = typename AllocatorTraits::const_pointer;

 public:
  explicit CopyValueAdapter(const ValueType& v) : ptr_(std::addressof(v)) {}

  void ConstructNext(AllocatorType* alloc_ptr, Pointer construct_at) {
    AllocatorTraits::construct(*alloc_ptr, construct_at, *ptr_);
  }

  void AssignNext(Pointer assign_at) { *assign_at = *ptr_; }

 private:
  ConstPointer ptr_;
};

template <typename AllocatorType>
class DefaultValueAdapter {
  using AllocatorTraits = absl::allocator_traits<AllocatorType>;
  using ValueType = typename AllocatorTraits::value_type;
  using Pointer = typename AllocatorTraits::pointer;

 public:
  explicit DefaultValueAdapter() {}

  void ConstructNext(AllocatorType* alloc_ptr, Pointer construct_at) {
    AllocatorTraits::construct(*alloc_ptr, construct_at);
  }

  void AssignNext(Pointer assign_at) { *assign_at = ValueType(); }
};

template <typename AllocatorType>
class AllocationTransaction {
  using AllocatorTraits = absl::allocator_traits<AllocatorType>;
  using Pointer = typename AllocatorTraits::pointer;
  using SizeType = typename AllocatorTraits::size_type;

 public:
  explicit AllocationTransaction(AllocatorType* alloc_ptr)
      : alloc_data_(*alloc_ptr, nullptr) {}

  ~AllocationTransaction() {
    if (DidAllocate()) {
      AllocatorTraits::deallocate(GetAllocator(), GetData(), GetCapacity());
    }
  }
  AllocationTransaction(const AllocationTransaction&) = delete;
  void operator=(const AllocationTransaction&) = delete;

  AllocatorType& GetAllocator() { return alloc_data_.template get<0>(); }
  Pointer& GetData() { return alloc_data_.template get<1>(); }
  SizeType& GetCapacity() { return capacity_; }

  bool DidAllocate() { return GetData() != nullptr; }
  Pointer Allocate(SizeType capacity) {
    GetData() = AllocatorTraits::allocate(GetAllocator(), capacity);
    GetCapacity() = capacity;
    return GetData();
  }

  void Reset() {
    GetData() = nullptr;
    GetCapacity() = 0;
  }

 private:
  container_internal::CompressedTuple<AllocatorType, Pointer> alloc_data_;
  SizeType capacity_ = 0;
};

template <typename AllocatorType>
class ConstructionTransaction {
  using AllocatorTraits = absl::allocator_traits<AllocatorType>;
  using Pointer = typename AllocatorTraits::pointer;
  using SizeType = typename AllocatorTraits::size_type;

 public:
  explicit ConstructionTransaction(AllocatorType* alloc_ptr)
      : alloc_data_(*alloc_ptr, nullptr) {}

  ~ConstructionTransaction() {
    if (DidConstruct()) {
      inlined_vector_internal::DestroyElements(std::addressof(GetAllocator()),
                                               GetData(), GetSize());
    }
  }

  ConstructionTransaction(const ConstructionTransaction&) = delete;
  void operator=(const ConstructionTransaction&) = delete;
  AllocatorType& GetAllocator() { return alloc_data_.template get<0>(); }
  Pointer& GetData() { return alloc_data_.template get<1>(); }
  SizeType& GetSize() { return size_; }

  bool DidConstruct() { return GetData() != nullptr; }
  template <typename ValueAdapter>
  void Construct(Pointer data, ValueAdapter* values_ptr, SizeType size) {
    inlined_vector_internal::ConstructElements(std::addressof(GetAllocator()),
                                               data, values_ptr, size);
    GetData() = data;
    GetSize() = size;
  }
  void Commit() {
    GetData() = nullptr;
    GetSize() = 0;
  }

 private:
  container_internal::CompressedTuple<AllocatorType, Pointer> alloc_data_;
  SizeType size_ = 0;
};

template <typename T, size_t N, typename A>
class Storage {
 public:
  using AllocatorTraits = absl::allocator_traits<A>;
  using allocator_type = typename AllocatorTraits::allocator_type;
  using value_type = typename AllocatorTraits::value_type;
  using pointer = typename AllocatorTraits::pointer;
  using const_pointer = typename AllocatorTraits::const_pointer;
  using size_type = typename AllocatorTraits::size_type;
  using difference_type = typename AllocatorTraits::difference_type;

  using reference = value_type&;
  using const_reference = const value_type&;
  using RValueReference = value_type&&;
  using iterator = pointer;
  using const_iterator = const_pointer;
  using reverse_iterator = std::reverse_iterator<iterator>;
  using const_reverse_iterator = std::reverse_iterator<const_iterator>;
  using MoveIterator = std::move_iterator<iterator>;
  using IsMemcpyOk = inlined_vector_internal::IsMemcpyOk<allocator_type>;

  using StorageView = inlined_vector_internal::StorageView<allocator_type>;
```

```
template <typename Iterator>
using IteratorValueAdapter =
   inlined_vector_internal::IteratorValueAdapter<allocator_type, Iterator>;
using CopyValueAdapter =
   inlined_vector_internal::CopyValueAdapter<allocator_type>;
using DefaultValueAdapter =
   inlined_vector_internal::DefaultValueAdapter<allocator_type>;

using AllocationTransaction =
   inlined_vector_internal::AllocationTransaction<allocator_type>;
using ConstructionTransaction =
   inlined_vector_internal::ConstructionTransaction<allocator_type>;

static size_type NextCapacity(size_type current_capacity) {
  return current_capacity * 2;
}

static size_type ComputeCapacity(size_type current_capacity,
                                 size_type requested_capacity) {
  return (std::max)(NextCapacity(current_capacity), requested_capacity);
}

// --------------------------------------------------------------------------
// Storage Constructors and Destructor
// --------------------------------------------------------------------------

Storage() : metadata_() {}

explicit Storage(const allocator_type& alloc) : metadata_(alloc, {}) {}

~Storage() {
  pointer data = GetIsAllocated() ? GetAllocatedData() : GetInlinedData();
  inlined_vector_internal::DestroyElements(GetAllocPtr(), data, GetSize());
  DeallocateIfAllocated();
}

// --------------------------------------------------------------------------
// Storage Member Accessors
// --------------------------------------------------------------------------

size_type& GetSizeAndIsAllocated() { return metadata_.template get<1>(); }

const size_type& GetSizeAndIsAllocated() const {
  return metadata_.template get<1>();
}

size_type GetSize() const { return GetSizeAndIsAllocated() >> 1; }

bool GetIsAllocated() const { return GetSizeAndIsAllocated() & 1; }

pointer GetAllocatedData() { return data_.allocated.allocated_data; }

const_pointer GetAllocatedData() const {
  return data_.allocated.allocated_data;
}

pointer GetInlinedData() {
  return reinterpret_cast<pointer>(
      std::addressof(data_.inlined.inlined_data[0]));
}

const_pointer GetInlinedData() const {
  return reinterpret_cast<const_pointer>(
      std::addressof(data_.inlined.inlined_data[0]));
}

size_type GetAllocatedCapacity() const {
  return data_.allocated.allocated_capacity;
}

size_type GetInlinedCapacity() const { return static_cast<size_type>(N); }

StorageView MakeStorageView() {
  return GetIsAllocated()
             ? StorageView{GetAllocatedData(), GetSize(),
                           GetAllocatedCapacity()}
             : StorageView{GetInlinedData(), GetSize(), GetInlinedCapacity()};
}

allocator_type* GetAllocPtr() {
  return std::addressof(metadata_.template get<0>());
}

const allocator_type* GetAllocPtr() const {
  return std::addressof(metadata_.template get<0>());
}
```

```
// --------------------------------------------------------------------------
// Storage Member Mutators
// --------------------------------------------------------------------------

template <typename ValueAdapter>
void Initialize(ValueAdapter values, size_type new_size);

template <typename ValueAdapter>
void Assign(ValueAdapter values, size_type new_size);

template <typename ValueAdapter>
void Resize(ValueAdapter values, size_type new_size);

template <typename ValueAdapter>
iterator Insert(const_iterator pos, ValueAdapter values,
                size_type insert_count);

template <typename... Args>
reference EmplaceBack(Args&&... args);

iterator Erase(const_iterator from, const_iterator to);

void Reserve(size_type requested_capacity);

void ShrinkToFit();

void Swap(Storage* other_storage_ptr);

void SetIsAllocated() {
  GetSizeAndIsAllocated() |= static_cast<size_type>(1);
}

void UnsetIsAllocated() {
  GetSizeAndIsAllocated() &= ((std::numeric_limits<size_type>::max)() - 1);
}

void SetSize(size_type size) {
  GetSizeAndIsAllocated() =
      (size << 1) | static_cast<size_type>(GetIsAllocated());
}

void SetAllocatedSize(size_type size) {
  GetSizeAndIsAllocated() = (size << 1) | static_cast<size_type>(1);
}
```

```
 void SetInlinedSize(size_type size) {
  GetSizeAndIsAllocated() = size << static_cast<size_type>(1);
 }

 void AddSize(size_type count) {
  GetSizeAndIsAllocated() += count << static_cast<size_type>(1);
 }

 void SubtractSize(size_type count) {
  assert(count <= GetSize());

  GetSizeAndIsAllocated() -= count << static_cast<size_type>(1);
 }

 void SetAllocatedData(pointer data, size_type capacity) {
  data_.allocated.allocated_data = data;
  data_.allocated.allocated_capacity = capacity;
 }

 void AcquireAllocatedData(AllocationTransaction* allocation_tx_ptr) {
  SetAllocatedData(allocation_tx_ptr->GetData(),
                   allocation_tx_ptr->GetCapacity());

  allocation_tx_ptr->Reset();
 }

 void MemcpyFrom(const Storage& other_storage) {
  assert(IsMemcpyOk::value || other_storage.GetIsAllocated());

  GetSizeAndIsAllocated() = other_storage.GetSizeAndIsAllocated();
  data_ = other_storage.data_;
 }

 void DeallocateIfAllocated() {
  if (GetIsAllocated()) {
   AllocatorTraits::deallocate(*GetAllocPtr(), GetAllocatedData(),
                               GetAllocatedCapacity());
  }
 }

private:
 using Metadata =
   container_internal::CompressedTuple<allocator_type, size_type>;
  struct Allocated {
    pointer allocated_data;
    size_type allocated_capacity;
  };

  struct Inlined {
    alignas(value_type) char inlined_data[sizeof(value_type[N])];
  };

  union Data {
    Allocated allocated;
    Inlined inlined;
  };

  Metadata metadata_;
  Data data_;
};

template <typename T, size_t N, typename A>
template <typename ValueAdapter>
auto Storage<T, N, A>::Initialize(ValueAdapter values, size_type new_size)
    -> void {
  // Only callable from constructors!
  assert(!GetIsAllocated());
  assert(GetSize() == 0);

  pointer construct_data;
  if (new_size > GetInlinedCapacity()) {
    // Because this is only called from the `InlinedVector` constructors, it's
    // safe to take on the allocation with size `0`. If `ConstructElements(...)`
    // throws, deallocation will be automatically handled by `~Storage()`.
    size_type new_capacity = ComputeCapacity(GetInlinedCapacity(), new_size);
    construct_data = AllocatorTraits::allocate(*GetAllocPtr(), new_capacity);
    SetAllocatedData(construct_data, new_capacity);
    SetIsAllocated();
  } else {
    construct_data = GetInlinedData();
  }

  inlined_vector_internal::ConstructElements(GetAllocPtr(), construct_data,
                                             &values, new_size);

  // Since the initial size was guaranteed to be `0` and the allocated bit is
  // already correct for either case, *adding* `new_size` gives us the correct
  // result faster than setting it directly.
  AddSize(new_size);
}

template <typename T, size_t N, typename A>
template <typename ValueAdapter>
auto Storage<T, N, A>::Assign(ValueAdapter values, size_type new_size) -> void {
  StorageView storage_view = MakeStorageView();

  AllocationTransaction allocation_tx(GetAllocPtr());

  absl::Span<value_type> assign_loop;
  absl::Span<value_type> construct_loop;
  absl::Span<value_type> destroy_loop;

  if (new_size > storage_view.capacity) {
    size_type new_capacity = ComputeCapacity(storage_view.capacity, new_size);
    construct_loop = {allocation_tx.Allocate(new_capacity), new_size};
    destroy_loop = {storage_view.data, storage_view.size};
  } else if (new_size > storage_view.size) {
    assign_loop = {storage_view.data, storage_view.size};
    construct_loop = {storage_view.data + storage_view.size,
                      new_size - storage_view.size};
  } else {
    assign_loop = {storage_view.data, new_size};
    destroy_loop = {storage_view.data + new_size, storage_view.size - new_size};
  }

  inlined_vector_internal::AssignElements(assign_loop.data(), &values,
                                          assign_loop.size());

  inlined_vector_internal::ConstructElements(
      GetAllocPtr(), construct_loop.data(), &values, construct_loop.size());

  inlined_vector_internal::DestroyElements(GetAllocPtr(), destroy_loop.data(),
                                           destroy_loop.size());

  if (allocation_tx.DidAllocate()) {
    DeallocateIfAllocated();
    AcquireAllocatedData(&allocation_tx);
    SetIsAllocated();
  }
  SetSize(new_size);
}

template <typename T, size_t N, typename A>
template <typename ValueAdapter>
auto Storage<T, N, A>::Resize(ValueAdapter values, size_type new_size) -> void {
  StorageView storage_view = MakeStorageView();

  IteratorValueAdapter<MoveIterator> move_values(
      MoveIterator(storage_view.data));

  AllocationTransaction allocation_tx(GetAllocPtr());
  ConstructionTransaction construction_tx(GetAllocPtr());

  absl::Span<value_type> construct_loop;
  absl::Span<value_type> move_construct_loop;
  absl::Span<value_type> destroy_loop;

  if (new_size > storage_view.capacity) {
    size_type new_capacity = ComputeCapacity(storage_view.capacity, new_size);
    pointer new_data = allocation_tx.Allocate(new_capacity);
    construct_loop = {new_data + storage_view.size,
                      new_size - storage_view.size};
    move_construct_loop = {new_data, storage_view.size};
    destroy_loop = {storage_view.data, storage_view.size};
  } else if (new_size > storage_view.size) {
    construct_loop = {storage_view.data + storage_view.size,
                      new_size - storage_view.size};
  } else {
    destroy_loop = {storage_view.data + new_size, storage_view.size - new_size};
  }

  construction_tx.Construct(construct_loop.data(), &values,
                            construct_loop.size());

  inlined_vector_internal::ConstructElements(
      GetAllocPtr(), move_construct_loop.data(), &move_values,
      move_construct_loop.size());

  inlined_vector_internal::DestroyElements(GetAllocPtr(), destroy_loop.data(),
                                           destroy_loop.size());

  construction_tx.Commit();
  if (allocation_tx.DidAllocate()) {
    DeallocateIfAllocated();
    AcquireAllocatedData(&allocation_tx);
    SetIsAllocated();
  }

  SetSize(new_size);
}

template <typename T, size_t N, typename A>
template <typename ValueAdapter>
auto Storage<T, N, A>::Insert(const_iterator pos, ValueAdapter values,
                              size_type insert_count) -> iterator {
  StorageView storage_view = MakeStorageView();

  size_type insert_index =
      std::distance(const_iterator(storage_view.data), pos);
  size_type insert_end_index = insert_index + insert_count;
  size_type new_size = storage_view.size + insert_count;

  if (new_size > storage_view.capacity) {
    AllocationTransaction allocation_tx(GetAllocPtr());
    ConstructionTransaction construction_tx(GetAllocPtr());
    ConstructionTransaction move_construciton_tx(GetAllocPtr());

    IteratorValueAdapter<MoveIterator> move_values(
        MoveIterator(storage_view.data));

    size_type new_capacity = ComputeCapacity(storage_view.capacity, new_size);
    pointer new_data = allocation_tx.Allocate(new_capacity);

    construction_tx.Construct(new_data + insert_index, &values, insert_count);

    move_construciton_tx.Construct(new_data, &move_values, insert_index);

    inlined_vector_internal::ConstructElements(
        GetAllocPtr(), new_data + insert_end_index, &move_values,
        storage_view.size - insert_index);

    inlined_vector_internal::DestroyElements(GetAllocPtr(), storage_view.data,
                                             storage_view.size);

    construction_tx.Commit();
    move_construciton_tx.Commit();
    DeallocateIfAllocated();
```

```
      AcquireAllocatedData(&allocation_tx);

      SetAllocatedSize(new_size);
      return iterator(new_data + insert_index);
    } else {
      size_type move_construction_destination_index =
          (std::max)(insert_end_index, storage_view.size);

      ConstructionTransaction move_construction_tx(GetAllocPtr());

      IteratorValueAdapter<MoveIterator> move_construction_values(
          MoveIterator(storage_view.data +
                       (move_construction_destination_index - insert_count)));
      absl::Span<value_type> move_construction = {
          storage_view.data + move_construction_destination_index,
          new_size - move_construction_destination_index};

      pointer move_assignment_values = storage_view.data + insert_index;
      absl::Span<value_type> move_assignment = {
          storage_view.data + insert_end_index,
          move_construction_destination_index - insert_end_index};

      absl::Span<value_type> insert_assignment = {move_assignment_values,
                                                  move_construction.size()};

      absl::Span<value_type> insert_construction = {
          insert_assignment.data() + insert_assignment.size(),
          insert_count - insert_assignment.size()};

      move_construction_tx.Construct(move_construction.data(),
                                     &move_construction_values,
                                     move_construction.size());

      for (pointer destination = move_assignment.data() + move_assignment.size(),
                   last_destination = move_assignment.data(),
                   source = move_assignment_values + move_assignment.size();
           ;) {
        --destination;
        --source;
        if (destination < last_destination) break;
        *destination = std::move(*source);
      }

      inlined_vector_internal::AssignElements(insert_assignment.data(), &values,
                                       insert_assignment.size());

      inlined_vector_internal::ConstructElements(
          GetAllocPtr(), insert_construction.data(), &values,
          insert_construction.size());

      move_construction_tx.Commit();

      AddSize(insert_count);
      return iterator(storage_view.data + insert_index);
    }
  }

  template <typename T, size_t N, typename A>
  template <typename... Args>
  auto Storage<T, N, A>::EmplaceBack(Args&&... args) -> reference {
    StorageView storage_view = MakeStorageView();

    AllocationTransaction allocation_tx(GetAllocPtr());

    IteratorValueAdapter<MoveIterator> move_values(
        MoveIterator(storage_view.data));

    pointer construct_data;
    if (storage_view.size == storage_view.capacity) {
      size_type new_capacity = NextCapacity(storage_view.capacity);
      construct_data = allocation_tx.Allocate(new_capacity);
    } else {
      construct_data = storage_view.data;
    }

    pointer last_ptr = construct_data + storage_view.size;

    AllocatorTraits::construct(*GetAllocPtr(), last_ptr,
                               std::forward<Args>(args)...);

    if (allocation_tx.DidAllocate()) {
      ABSL_INTERNAL_TRY {
        inlined_vector_internal::ConstructElements(
            GetAllocPtr(), allocation_tx.GetData(), &move_values,
            storage_view.size);
      }
      ABSL_INTERNAL_CATCH_ANY {
        AllocatorTraits::destroy(*GetAllocPtr(), last_ptr);
        ABSL_INTERNAL_RETHROW;
      }

      inlined_vector_internal::DestroyElements(GetAllocPtr(), storage_view.data,
                                               storage_view.size);

      DeallocateIfAllocated();
      AcquireAllocatedData(&allocation_tx);
      SetIsAllocated();
    }

    AddSize(1);
    return *last_ptr;
  }

  template <typename T, size_t N, typename A>
  auto Storage<T, N, A>::Erase(const_iterator from, const_iterator to)
      -> iterator {
    StorageView storage_view = MakeStorageView();

    size_type erase_size = std::distance(from, to);
    size_type erase_index =
        std::distance(const_iterator(storage_view.data), from);
    size_type erase_end_index = erase_index + erase_size;

    IteratorValueAdapter<MoveIterator> move_values(
        MoveIterator(storage_view.data + erase_end_index));

    inlined_vector_internal::AssignElements(storage_view.data + erase_index,
                                            &move_values,
                                            storage_view.size - erase_end_index);

    inlined_vector_internal::DestroyElements(
        GetAllocPtr(), storage_view.data + (storage_view.size - erase_size),
        erase_size);

    SubtractSize(erase_size);
    return iterator(storage_view.data + erase_index);
  }

  template <typename T, size_t N, typename A>
  auto Storage<T, N, A>::Reserve(size_type requested_capacity) -> void {
    StorageView storage_view = MakeStorageView();
    if (ABSL_PREDICT_FALSE(requested_capacity <= storage_view.capacity)) return;

    AllocationTransaction allocation_tx(GetAllocPtr());

    IteratorValueAdapter<MoveIterator> move_values(
        MoveIterator(storage_view.data));

    size_type new_capacity =
        ComputeCapacity(storage_view.capacity, requested_capacity);
    pointer new_data = allocation_tx.Allocate(new_capacity);

    inlined_vector_internal::ConstructElements(GetAllocPtr(), new_data,
                                               &move_values, storage_view.size);

    inlined_vector_internal::DestroyElements(GetAllocPtr(), storage_view.data,
                                             storage_view.size);

    DeallocateIfAllocated();
    AcquireAllocatedData(&allocation_tx);
    SetIsAllocated();
  }

  template <typename T, size_t N, typename A>
  auto Storage<T, N, A>::ShrinkToFit() -> void {
    // May only be called on allocated instances!
    assert(GetIsAllocated());

    StorageView storage_view{GetAllocatedData(), GetSize(),
                             GetAllocatedCapacity()};

    if (ABSL_PREDICT_FALSE(storage_view.size == storage_view.capacity)) return;

    AllocationTransaction allocation_tx(GetAllocPtr());

    IteratorValueAdapter<MoveIterator> move_values(
        MoveIterator(storage_view.data));

    pointer construct_data;
    if (storage_view.size > GetInlinedCapacity()) {
      size_type new_capacity = storage_view.size;
      construct_data = allocation_tx.Allocate(new_capacity);
    } else {
      construct_data = GetInlinedData();
    }
    ABSL_INTERNAL_TRY {
      inlined_vector_internal::ConstructElements(GetAllocPtr(), construct_data,
                                                 &move_values, storage_view.size);
    }
    ABSL_INTERNAL_CATCH_ANY {
      SetAllocatedData(storage_view.data, storage_view.capacity);
      ABSL_INTERNAL_RETHROW;
    }

    inlined_vector_internal::DestroyElements(GetAllocPtr(), storage_view.data,
                                             storage_view.size);

    AllocatorTraits::deallocate(*GetAllocPtr(), storage_view.data,
                                storage_view.capacity);

    if (allocation_tx.DidAllocate()) {
      AcquireAllocatedData(&allocation_tx);
    } else {
      UnsetIsAllocated();
    }
  }

  template <typename T, size_t N, typename A>
  auto Storage<T, N, A>::Swap(Storage* other_storage_ptr) -> void {
    using std::swap;
    assert(this != other_storage_ptr);

    if (GetIsAllocated() && other_storage_ptr->GetIsAllocated()) {
      swap(data_.allocated, other_storage_ptr->data_.allocated);
    } else if (!GetIsAllocated() && !other_storage_ptr->GetIsAllocated()) {
      Storage* small_ptr = this;
      Storage* large_ptr = other_storage_ptr;
      if (small_ptr->GetSize() > large_ptr->GetSize()) swap(small_ptr, large_ptr);

      for (size_type i = 0; i < small_ptr->GetSize(); ++i) {
        swap(small_ptr->GetInlinedData()[i], large_ptr->GetInlinedData()[i]);
      }

      IteratorValueAdapter<MoveIterator> move_values(
          MoveIterator(large_ptr->GetInlinedData() + small_ptr->GetSize()));

      inlined_vector_internal::ConstructElements(
          large_ptr->GetAllocPtr(),
        small_ptr->GetInlinedData() + small_ptr->GetSize(), &move_values,
        large_ptr->GetSize() - small_ptr->GetSize());

      inlined_vector_internal::DestroyElements(
        large_ptr->GetAllocPtr(),
        large_ptr->GetInlinedData() + small_ptr->GetSize(),
        large_ptr->GetSize() - small_ptr->GetSize());
    } else {
      Storage* allocated_ptr = this;
      Storage* inlined_ptr = other_storage_ptr;
      if (!allocated_ptr->GetIsAllocated()) swap(allocated_ptr, inlined_ptr);

      StorageView allocated_storage_view{allocated_ptr->GetAllocatedData(),
                                         allocated_ptr->GetSize(),
                                         allocated_ptr->GetAllocatedCapacity()};

      IteratorValueAdapter<MoveIterator> move_values(
        MoveIterator(inlined_ptr->GetInlinedData()));

      ABSL_INTERNAL_TRY {
        inlined_vector_internal::ConstructElements(
          inlined_ptr->GetAllocPtr(), allocated_ptr->GetInlinedData(),
          &move_values, inlined_ptr->GetSize());
      }
      ABSL_INTERNAL_CATCH_ANY {
        allocated_ptr->SetAllocatedData(allocated_storage_view.data,
                                        allocated_storage_view.capacity);
        ABSL_INTERNAL_RETHROW;
      }

      inlined_vector_internal::DestroyElements(inlined_ptr->GetAllocPtr(),
                                               inlined_ptr->GetInlinedData(),
                                               inlined_ptr->GetSize());

      inlined_ptr->SetAllocatedData(allocated_storage_view.data,
                                    allocated_storage_view.capacity);
    }

    swap(GetSizeAndIsAllocated(), other_storage_ptr->GetSizeAndIsAllocated());
    swap(*GetAllocPtr(), *other_storage_ptr->GetAllocPtr());
  }

  }  // namespace inlined_vector_internal
  ABSL_NAMESPACE_END
```

```
}  // namespace absl
```

Class IntType

```
// #status: LEGACY
// #category: Miscellaneous
// #summary: Integral types; prefer util/intops/strong_int.h
// #bugs: Infrastructure > C++ Library Team > util
//
// IntType is a simple template class mechanism for defining "logical"
// integer-like class types that support many of the same functionalities
// as native integer types, but which prevent assignment, construction, and
// other operations from other similar integer-like types.  Essentially, the
// template class IntType<IntTypeName, ValueType> (where ValueType assumes
// valid scalar types such as int, uint, int32, etc) has the additional
// property that it cannot be assigned to or constructed from other IntTypes
// or native integer types of equal or implicitly convertible type.
//
// The class is useful for preventing mingling of integer variables with
// different logical roles or units.  Unfortunately, C++ provides relatively
// good type-safety for user-defined classes but not for integer types.  It is
// essentially up to the user to use nice variable names and comments to prevent
// accidental mismatches, such as confusing a user-index with a group-index or a
// time-in-milliseconds with a time-in-seconds.  The use of typedefs are limited
// in that regard as they do not enforce type-safety.
//
// USAGE -----------------------------------------------------------------------
//
//    DEFINE_INT_TYPE(IntTypeName, ValueType);
//
//  where:
//    IntTypeName: is the desired (unique) name for the "logical" integer type.
//    ValueType: is one of the integral types as defined by base::is_integral
//               (see base/type_traits.h).
//
// DISALLOWED OPERATIONS / TYPE-SAFETY ENFORCEMENT -----------------------------
//
//  Consider these definitions and variable declarations:
//    DEFINE_INT_TYPE(GlobalDocID, int64);
//    DEFINE_INT_TYPE(LocalDocID, int64);
//    GlobalDocID global;
//    LocalDocID local;
//
//  The class IntType prevents:
//
```

```
//  1) Assignments of other IntTypes with different IntTypeNames.
//
//    global = local;                 <-- Fails to compile!
//    local = global;                 <-- Fails to compile!
//
//  2) Explicit/implicit conversion from an IntType to another IntType.
//
//    LocalDocID l(global);           <-- Fails to compile!
//    LocalDocID l = global;          <-- Fails to compile!
//
//    void GetGlobalDoc(GlobalDocID global) { }
//    GetGlobalDoc(global);           <-- Compiles fine, types match!
//    GetGlobalDoc(local);            <-- Fails to compile!
//
//  3) Implicit conversion from an IntType to a native integer type.
//
//    void GetGlobalDoc(int64 global) { ...
//    GetGlobalDoc(global);           <-- Fails to compile!
//    GetGlobalDoc(local);            <-- Fails to compile!
//
//    void GetLocalDoc(int32 local) { ...
//    GetLocalDoc(global);            <-- Fails to compile!
//    GetLocalDoc(local);             <-- Fails to compile!
//
//
// SUPPORTED OPERATIONS --------------------------------------------------------
//
// The following operators are supported: unary: ++ (both prefix and postfix),
// +, -, ! (logical not), ~ (one's complement); comparison: ==, !=, <, <=, >,
// >=; numerical: +, -, *, /; assignment: =, +=, -=, /=, *=; stream: <<. Each
// operator allows the same IntTypeName and the ValueType to be used on
// both left- and right-hand sides.
//
// It also supports an accessor value() returning the stored value as ValueType,
// and a templatized accessor value<T>() method that serves as syntactic sugar
// for static_cast<T>(var.value()).  These accessors are useful when assigning
// the stored value into protocol buffer fields and using it as printf args.
//
// The class also defines a hash functor that allows the IntType to be used
// as key to hashable containers such as std::unordered_map and
// std::unordered_set.
//
// We suggest using the IntTypeIndexedContainer wrapper around FixedArray and
// STL vector (see int-type-indexed-container.h) if an IntType is intended to
// be used as an index into these containers.  These wrappers are indexed in a
// type-safe manner using IntTypes to ensure type-safety.
//
// NB: this implementation does not attempt to abide by or enforce dimensional
// analysis on these scalar types.
//
// EXAMPLES --------------------------------------------------------------------
//
//    DEFINE_INT_TYPE(GlobalDocID, int64);
//    GlobalDocID global = 3;
//    cout << global;                      <-- Prints 3 to stdout.
//
//    for (GlobalDocID i(0); i < global; ++i) {
//      cout << i;
//    }                                    <-- Print(ln)s 0 1 2 to stdout
//
//    DEFINE_INT_TYPE(LocalDocID, int64);
//    LocalDocID local;
//    cout << local;                       <-- Prints 0 to stdout it default
//                                             initializes the value to 0.
//
//    local = 5;
//    local *= 2;
//    LocalDocID l(local);
//    cout << l + local;                   <-- Prints 20 to stdout.
//
//    GenericSearchRequest request;
//    request.set_doc_id(global.value());  <-- Uses value() to extract the value
//                                             from the IntType class.
//
// REMARKS ---------------------------------------------------------------------
//
// The following bad usage is permissible although discouraged.  Essentially, it
// involves using the value*() accessors to extract the native integer type out
// of the IntType class.  Keep in mind that the primary reason for the IntType
// class is to prevent *accidental* mingling of similar logical integer types --
// and not type casting from one type to another.
//
//  DEFINE_INT_TYPE(GlobalDocID, int64);
//  DEFINE_INT_TYPE(LocalDocID, int64);
//  GlobalDocID global;
//  LocalDocID local;
//
//  global = local.value();                       <-- Compiles fine.
```

```
//
//  void GetGlobalDoc(GlobalDocID global) { ...
//  GetGlobalDoc(local.value());              <-- Compiles fine.
//
//  void GetGlobalDoc(int64 global) { ...
//  GetGlobalDoc(local.value());              <-- Compiles fine.

namespace tensorflow {
namespace gtl {

template <typename IntTypeName, typename _ValueType>
class IntType;

// Defines the IntType using value_type and typedefs it to int_type_name.
// The struct int_type_name ## _tag_ trickery is needed to ensure that a new
// type is created per int_type_name.
#define TF_LIB_GTL_DEFINE_INT_TYPE(int_type_name, value_type)        \
  struct int_type_name##_tag_ {};                                    \
  typedef ::tensorflow::gtl::IntType<int_type_name##_tag_, value_type>  \
      int_type_name;

// Holds an integer value (of type ValueType) and behaves as a ValueType by
// exposing assignment, unary, comparison, and arithmetic operators.
//
// The template parameter IntTypeName defines the name for the int type and must
// be unique within a binary (the convenient DEFINE_INT_TYPE macro at the end of
// the file generates a unique IntTypeName).  The parameter ValueType defines
// the integer type value (see supported list above).
//
// This class is NOT thread-safe.
template <typename IntTypeName, typename _ValueType>
class IntType {
 public:
  typedef _ValueType ValueType;                    // for non-member operators
  typedef IntType<IntTypeName, ValueType> ThisType;  // Syntactic sugar.

  // Note that this may change from time to time without notice.
  struct Hasher {
    size_t operator()(const IntType& arg) const {
      return static_cast<size_t>(arg.value());
    }
  };

  template <typename H>
  friend H AbslHashValue(H h, const IntType& i) {
    return H::combine(std::move(h), i.value());
  }

 public:
  // Default c'tor initializing value_ to 0.
  constexpr IntType() : value_(0) {}
  // C'tor explicitly initializing from a ValueType.
  constexpr explicit IntType(ValueType value) : value_(value) {}

  // IntType uses the default copy constructor, destructor and assign operator.
  // The defaults are sufficient and omitting them allows the compiler to add
  // the move constructor/assignment.

  // -- ACCESSORS --------------------------------------------------------------
  // The class provides a value() accessor returning the stored ValueType value_
  // as well as a templatized accessor that is just a syntactic sugar for
  // static_cast<T>(var.value());
  constexpr ValueType value() const { return value_; }

  template <typename ValType>
  constexpr ValType value() const {
    return static_cast<ValType>(value_);
  }

  // -- UNARY OPERATORS --------------------------------------------------------
  ThisType& operator++() {  // prefix ++
    ++value_;
    return *this;
  }
  const ThisType operator++(int v) {  // postfix ++
    ThisType temp(*this);
    ++value_;
    return temp;
  }
  ThisType& operator--() {  // prefix --
    --value_;
    return *this;
  }
  const ThisType operator--(int v) {  // postfix --
    ThisType temp(*this);
    --value_;
    return temp;
  }
```

```
  constexpr bool operator!() const { return value_ == 0; }
  constexpr const ThisType operator+() const { return ThisType(value_); }
  constexpr const ThisType operator-() const { return ThisType(-value_); }
  constexpr const ThisType operator~() const { return ThisType(~value_); }

// -- ASSIGNMENT OPERATORS --------------------------------------------------
// We support the following assignment operators: =, +=, -=, *=, /=, <<=, >>=
// and %= for both ThisType and ValueType.
#define INT_TYPE_ASSIGNMENT_OP(op)                        \
  ThisType& operator op(const ThisType& arg_value) {      \
    value_ op arg_value.value();                          \
    return *this;                                         \
  }                                                       \
  ThisType& operator op(ValueType arg_value) {            \
    value_ op arg_value;                                  \
    return *this;                                         \
  }
  INT_TYPE_ASSIGNMENT_OP(+=);
  INT_TYPE_ASSIGNMENT_OP(-=);
  INT_TYPE_ASSIGNMENT_OP(*=);
  INT_TYPE_ASSIGNMENT_OP(/=);
  INT_TYPE_ASSIGNMENT_OP(<<=);  // NOLINT
  INT_TYPE_ASSIGNMENT_OP(>>=);  // NOLINT
  INT_TYPE_ASSIGNMENT_OP(%=);
#undef INT_TYPE_ASSIGNMENT_OP

  ThisType& operator=(ValueType arg_value) {
    value_ = arg_value;
    return *this;
  }

 private:
  // The integer value of type ValueType.
  ValueType value_;

  static_assert(std::is_integral<ValueType>::value, "invalid integer type");
} TF_PACKED;

// -- NON-MEMBER STREAM OPERATORS ---------------------------------------------
// We provide the << operator, primarily for logging purposes.  Currently, there
// seems to be no need for an >> operator.
template <typename IntTypeName, typename ValueType>
std::ostream& operator<<(std::ostream& os,  // NOLINT
                        IntType<IntTypeName, ValueType> arg) {
  return os << arg.value();
}

// -- NON-MEMBER ARITHMETIC OPERATORS -----------------------------------------
// We support only the +, -, *, and / operators with the same IntType and
// ValueType types.  The reason is to allow simple manipulation on these IDs
// when used as indices in vectors and arrays.
//
// NB: Although it is possible to do IntType * IntType and IntType / IntType,
// it is probably non-sensical from a dimensionality analysis perspective.
#define INT_TYPE_ARITHMETIC_OP(op)                                      \
  template <typename IntTypeName, typename ValueType>                   \
  static inline constexpr IntType<IntTypeName, ValueType> operator op(  \
      IntType<IntTypeName, ValueType> id_1,                             \
      IntType<IntTypeName, ValueType> id_2) {                           \
    return IntType<IntTypeName, ValueType>(id_1.value() op id_2.value()); \
  }                                                                     \
  template <typename IntTypeName, typename ValueType>                   \
  static inline constexpr IntType<IntTypeName, ValueType> operator op(  \
      IntType<IntTypeName, ValueType> id,                               \
      typename IntType<IntTypeName, ValueType>::ValueType arg_val) {    \
    return IntType<IntTypeName, ValueType>(id.value() op arg_val);      \
  }                                                                     \
  template <typename IntTypeName, typename ValueType>                   \
  static inline constexpr IntType<IntTypeName, ValueType> operator op(  \
      typename IntType<IntTypeName, ValueType>::ValueType arg_val,      \
      IntType<IntTypeName, ValueType> id) {                             \
    return IntType<IntTypeName, ValueType>(arg_val op id.value());      \
  }
INT_TYPE_ARITHMETIC_OP(+);
INT_TYPE_ARITHMETIC_OP(-);
INT_TYPE_ARITHMETIC_OP(*);
INT_TYPE_ARITHMETIC_OP(/);
INT_TYPE_ARITHMETIC_OP(<<);  // NOLINT
INT_TYPE_ARITHMETIC_OP(>>);  // NOLINT
INT_TYPE_ARITHMETIC_OP(%);
#undef INT_TYPE_ARITHMETIC_OP

// -- NON-MEMBER COMPARISON OPERATORS -----------------------------------------
// Static inline comparison operators.  We allow all comparison operators among
// the following types (OP \in [==, !=, <, <=, >, >=]:
//   IntType<IntTypeName, ValueType> OP IntType<IntTypeName, ValueType>
//   IntType<IntTypeName, ValueType> OP ValueType
//   ValueType OP IntType<IntTypeName, ValueType>
#define INT_TYPE_COMPARISON_OP(op)                              \
  template <typename IntTypeName, typename ValueType>           \
  static inline constexpr bool operator op(                     \
      IntType<IntTypeName, ValueType> id_1,                     \
      IntType<IntTypeName, ValueType> id_2) {                   \
    return id_1.value() op id_2.value();                        \
  }                                                             \
  template <typename IntTypeName, typename ValueType>           \
  static inline constexpr bool operator op(                     \
      IntType<IntTypeName, ValueType> id,                       \
      typename IntType<IntTypeName, ValueType>::ValueType val) { \
    return id.value() op val;                                   \
  }                                                             \
  template <typename IntTypeName, typename ValueType>           \
  static inline constexpr bool operator op(                     \
      typename IntType<IntTypeName, ValueType>::ValueType val,  \
      IntType<IntTypeName, ValueType> id) {                     \
    return val op id.value();                                   \
  }
INT_TYPE_COMPARISON_OP(==);  // NOLINT
INT_TYPE_COMPARISON_OP(!=);  // NOLINT
INT_TYPE_COMPARISON_OP(<);   // NOLINT
INT_TYPE_COMPARISON_OP(<=);  // NOLINT
INT_TYPE_COMPARISON_OP(>);   // NOLINT
INT_TYPE_COMPARISON_OP(>=);  // NOLINT
#undef INT_TYPE_COMPARISON_OP

}  // namespace gtl
}  // namespace tensorflow
```

Class iterator_range

```
// A range adaptor for a pair of iterators.
//
// This just wraps two iterators into a range-compatible interface. Nothing
// fancy at all.
template <typename IteratorT>
class iterator_range {
 public:
  using value_type = decltype(*std::declval<IteratorT>());
  using iterator = IteratorT;
  using const_iterator = IteratorT;
  iterator_range() : begin_iterator_(), end_iterator_() {}
  iterator_range(IteratorT begin_iterator, IteratorT end_iterator)
      : begin_iterator_(std::move(begin_iterator)),
        end_iterator_(std::move(end_iterator)) {}

  IteratorT begin() const { return begin_iterator_; }
  IteratorT end() const { return end_iterator_; }

 private:
  IteratorT begin_iterator_, end_iterator_;
};

// Convenience function for iterating over sub-ranges.
//
// This provides a bit of syntactic sugar to make using sub-ranges
// in for loops a bit easier. Analogous to std::make_pair().
template <class T>
iterator_range<T> make_range(T x, T y) {
  return iterator_range<T>(std::move(x), std::move(y));
}
```

map_util and map_traits headers

```
// Traits classes for performing uniform lookup on different map value types.
//
// The access is computed as follows:
//
//   1. If T has a `first` or `second` field, use them.
//   2. Otherwise if it has `key()` or `value()` methods, use them.
//   3. Otherwise the program is ill-formed.


namespace tensorflow {
namespace gtl {
namespace subtle {
namespace internal_map_traits {
struct Rank1 {};
struct Rank0 : Rank1 {};

template <class V>
auto GetKey(V&& v, Rank0) -> decltype((std::forward<V>(v).first)) {
  return std::forward<V>(v).first;
}
template <class V>
auto GetKey(V&& v, Rank1) -> decltype(std::forward<V>(v).key()) {
  return std::forward<V>(v).key();
}

template <class V>
auto GetMapped(V&& v, Rank0) -> decltype((std::forward<V>(v).second)) {
  return std::forward<V>(v).second;
}
template <class V>
auto GetMapped(V&& v, Rank1) -> decltype(std::forward<V>(v).value()) {
  return std::forward<V>(v).value();
}

}  // namespace internal_map_traits

// Accesses the `key_type` from a `value_type`.
template <typename V>
auto GetKey(V&& v)
    -> decltype(internal_map_traits::GetKey(std::forward<V>(v),
                                            internal_map_traits::Rank0())) {
  return internal_map_traits::GetKey(std::forward<V>(v),
                                     internal_map_traits::Rank0());
}

// Accesses the `mapped_type` from a `value_type`.
template <typename V>
auto GetMapped(V&& v)
    -> decltype(internal_map_traits::GetMapped(std::forward<V>(v),
                                               internal_map_traits::Rank0())) {
  return internal_map_traits::GetMapped(std::forward<V>(v),
                                        internal_map_traits::Rank0());
}

}  // namespace subtle
}  // namespace gtl
}  // namespace tensorflow


// Returns a pointer to the const value associated with the given key if it
// exists, or NULL otherwise.
template <class Collection>
const typename Collection::value_type::second_type* FindOrNull(
    const Collection& collection,
    const typename Collection::value_type::first_type& key) {
  typename Collection::const_iterator it = collection.find(key);
  if (it == collection.end()) {
    return 0;
  }
  return &it->second;
}

// Same as above but returns a pointer to the non-const value.
template <class Collection>
typename Collection::value_type::second_type* FindOrNull(
    Collection& collection,  // NOLINT
    const typename Collection::value_type::first_type& key) {
  typename Collection::iterator it = collection.find(key);
  if (it == collection.end()) {
    return 0;
  }
  return &it->second;
}

// Returns the pointer value associated with the given key. If none is found,
// NULL is returned. The function is designed to be used with a map of keys to
// pointers.
//
// This function does not distinguish between a missing key and a key mapped
// to a NULL value.
template <class Collection>
typename Collection::value_type::second_type FindPtrOrNull(
    const Collection& collection,
    const typename Collection::value_type::first_type& key) {
  typename Collection::const_iterator it = collection.find(key);
  if (it == collection.end()) {
    return typename Collection::value_type::second_type();
  }
  return it->second;
}

// Returns a const reference to the value associated with the given key if it
// exists, otherwise returns a const reference to the provided default value.
//
// WARNING: If a temporary object is passed as the default "value,"
// this function will return a reference to that temporary object,
// which will be destroyed at the end of the statement. A common
// example: if you have a map with string values, and you pass a char*
// as the default "value," either use the returned value immediately
// or store it in a string (not string&).
template <class Collection>
const typename Collection::value_type::second_type& FindWithDefault(
    const Collection& collection,
    const typename Collection::value_type::first_type& key,
    const typename Collection::value_type::second_type& value) {
  typename Collection::const_iterator it = collection.find(key);
  if (it == collection.end()) {
    return value;
  }
  return it->second;
}

// Inserts the given key-value pair into the collection. Returns true if and
// only if the key from the given pair didn't previously exist. Otherwise, the
// value in the map is replaced with the value from the given pair.
template <class Collection>
bool InsertOrUpdate(Collection* const collection,
                    const typename Collection::value_type& vt) {
  std::pair<typename Collection::iterator, bool> ret = collection->insert(vt);
  if (!ret.second) {
    // update
    ret.first->second = vt.second;
    return false;
  }
  return true;
}

// Same as above, except that the key and value are passed separately.
template <class Collection>
bool InsertOrUpdate(Collection* const collection,
                    const typename Collection::value_type::first_type& key,
                    const typename Collection::value_type::second_type& value) {
  return InsertOrUpdate(collection,
                        typename Collection::value_type(key, value));
}

// Inserts the given key and value into the given collection if and only if the
// given key did NOT already exist in the collection. If the key previously
// existed in the collection, the value is not changed. Returns true if the
// key-value pair was inserted; returns false if the key was already present.
template <class Collection>
```

```
bool InsertIfNotPresent(Collection* const collection,
                        const typename Collection::value_type& vt) {
  return collection->insert(vt).second;
}

// Same as above except the key and value are passed separately.
template <class Collection>
bool InsertIfNotPresent(
    Collection* const collection,
    const typename Collection::value_type::first_type& key,
    const typename Collection::value_type::second_type& value) {
  return InsertIfNotPresent(collection,
                            typename Collection::value_type(key, value));
}

// Looks up a given key and value pair in a collection and inserts the key-value
// pair if it's not already present. Returns a reference to the value associated
// with the key.
template <class Collection>
typename Collection::value_type::second_type& LookupOrInsert(
    Collection* const collection, const typename Collection::value_type& vt) {
  return collection->insert(vt).first->second;
}

// Same as above except the key-value are passed separately.
template <class Collection>
typename Collection::value_type::second_type& LookupOrInsert(
    Collection* const collection,
    const typename Collection::value_type::first_type& key,
    const typename Collection::value_type::second_type& value) {
  return LookupOrInsert(collection,
                        typename Collection::value_type(key, value));
}

// Saves the reverse mapping into reverse. Returns true if values could all be
// inserted.
template <typename M, typename ReverseM>
bool ReverseMap(const M& m, ReverseM* reverse) {
  bool all_unique = true;
  for (const auto& kv : m) {
    if (!InsertOrUpdate(reverse, kv.second, kv.first)) {
      all_unique = false;
    }
  }
  return all_unique;
}

// Like ReverseMap above, but returns its output m. Return type has to
// be specified explicitly. Example:
// M::M(...) : m_(...), r_(ReverseMap<decltype(r_)>(m_)) {}
template <typename ReverseM, typename M>
ReverseM ReverseMap(const M& m) {
  typename std::remove_const<ReverseM>::type reverse;
  ReverseMap(m, &reverse);
  return reverse;
}

// Erases the m item identified by the given key, and returns the value
// associated with that key. It is assumed that the value (i.e., the
// mapped_type) is a pointer. Returns null if the key was not found in the
// m.
//
// Examples:
//   std::map<string, MyType*> my_map;
//
// One line cleanup:
//     delete EraseKeyReturnValuePtr(&my_map, "abc");
//
// Use returned value:
//     std::unique_ptr<MyType> value_ptr(
//         EraseKeyReturnValuePtr(&my_map, "abc"));
//     if (value_ptr.get())
//       value_ptr->DoSomething();
//
template <typename Collection>
typename Collection::value_type::second_type EraseKeyReturnValuePtr(
    Collection* collection,
    const typename Collection::value_type::first_type& key) {
  auto it = collection->find(key);
  if (it == collection->end()) return nullptr;
  auto v = gtl::subtle::GetMapped(*it);
  collection->erase(it);
  return v;
}
```

C++ External and Internal API

The main header file which declares the C++ API interface is:

bazel-tensorflow/tensorflow/c/c_api.h

```
// --------------------------------------------------------------------------
// C API for TensorFlow.
//
// The API leans towards simplicity and uniformity instead of convenience
// since most usage will be by language specific wrappers.
//
// Conventions:
// * We use the prefix TF_ for everything in the API.
// * Objects are always passed around as pointers to opaque structs
//   and these structs are allocated/deallocated via the API.
// * TF_Status holds error information.  It is an object type
//   and therefore is passed around as a pointer to an opaque
//   struct as mentioned above.
// * Every call that has a TF_Status* argument clears it on success
//   and fills it with error info on failure.
// * unsigned char is used for booleans (instead of the 'bool' type).
//   In C++ bool is a keyword while in C99 bool is a macro defined
//   in stdbool.h. It is possible for the two to be inconsistent.
//   For example, neither the C99 nor the C++11 standard force a byte
//   size on the bool type, so the macro defined in stdbool.h could
//   be inconsistent with the bool keyword in C++. Thus, the use
//   of stdbool.h is avoided and unsigned char is used instead.
// * size_t is used to represent byte sizes of objects that are
//   materialized in the address space of the calling process.
// * int is used as an index into arrays.
// * Deletion functions are safe to call on nullptr.
//
// Questions left to address:
// * Might at some point need a way for callers to provide their own Env.
// * Maybe add TF_TensorShape that encapsulates dimension info.
//
// Design decisions made:
// * Backing store for tensor memory has an associated deallocation
//   function.  This deallocation function will point to client code
//   for tensors populated by the client.  So the client can do things
//   like shadowing a numpy array.
// * We do not provide TF_OK since it is not strictly necessary and we
//   are not optimizing for convenience.
// * We make assumption that one session has one graph.  This should be
//   fine since we have the ability to run sub-graphs.
// * We could allow NULL for some arguments (e.g., NULL options arg).
//   However since convenience is not a primary goal, we don't do this.
// * Devices are not in this API.  Instead, they are created/used internally
//   and the API just provides high level controls over the number of
//   devices of each type.
```

For Linux and MacOS the symbol TF_CAPI_EXPORT is defined as:

TF_CAPI_EXPORT directive

```
#define TF_CAPI_EXPORT __attribute__((visibility("default")))
```

GCC-specific details on the visibility attribute can be found here. Basically, by judicious use of the visibility attribute we can decrease dramatically the load times of dynamically shared objects i.e. tensorflow library `libtensorflow_cc.so`. Great article about writing shared libraries by Ulrich Drepper can be found here.

TF_VERSION string

The first member of the API is the TF_VERSION string :

```
// --------------------------------------------------------------------------
// TF_Version returns a string describing version information of the
// TensorFlow library. TensorFlow using semantic versioning.
TF_CAPI_EXPORT extern const char* TF_Version(void);
```

TensorFlow follows Semantic Versioning 2.0 (semver) for its public API. Each release version of TensorFlow has the form MAJOR.MINOR.PATCH. For example, TensorFlow version 1.2.3 has MAJOR version 1, MINOR version 2, and PATCH version 3. Changes to each number have the following meaning:

MAJOR: Potentially backwards incompatible changes. Code and data that worked with a previous major release will not necessarily work with the new release. However, in some cases existing TensorFlow graphs and checkpoints may be migratable to the newer release; see Compatibility of graphs and checkpoints for details on data compatibility.

MINOR: Backwards compatible features, speed improvements, etc. Code and data that worked with a previous minor release and which depends only on the non-experimental public API will continue to work unchanged. For details on what is and is not the public API, see What is covered.

PATCH: Backwards compatible bug fixes.

For example, release 1.0.0 introduced backwards incompatible changes from release 0.12.1. However, release 1.1.1 was backwards compatible with release 1.0.0.

TF_Buffer struct and functionality for manipulating it

The struct **TF_Buffer** is defined next. Its purpose and usage are described in the comment lines:

```
// --------------------------------------------------------------------------
// TF_Buffer holds a pointer to a block of data and its associated length.
// Typically, the data consists of a serialized protocol buffer, but other data
// may also be held in a buffer.
//
// By default, TF_Buffer itself does not do any memory management of the
// pointed-to block.  If need be, users of this struct should specify how to
// deallocate the block by setting the `data_deallocator` function pointer.
```

```
typedef struct TF_Buffer {
  const void* data;
  size_t length;
  void (*data_deallocator)(void* data, size_t length);
} TF_Buffer;
```

The next member is the global function **TF_NewBufferFromString** which instantiates a new **TF_Buffer** from read-only protobuf instances

```
// Makes a copy of the input and sets an appropriate deallocator.  Useful for
// passing in read-only, input protobufs.
TF_CAPI_EXPORT extern TF_Buffer* TF_NewBufferFromString(const void* proto, size_t proto_len);
```

Here is an implementation for the function:

```
TF_Buffer* TF_NewBufferFromString(const void* proto, size_t proto_len) {
  void* copy = tensorflow::port::Malloc(proto_len);
  memcpy(copy, proto, proto_len);

  TF_Buffer* buf = new TF_Buffer;
  buf->data = copy;
  buf->length = proto_len;
  buf->data_deallocator = [](void* data, size_t length) {
    tensorflow::port::Free(data);
  };
  return buf;
}
```

Follow three more global functions for manipulation of **TF_Buffer**:

```
// Useful for passing *out* a protobuf.
TF_CAPI_EXPORT extern TF_Buffer* TF_NewBuffer(void);

TF_CAPI_EXPORT extern void TF_DeleteBuffer(TF_Buffer*);

TF_CAPI_EXPORT extern TF_Buffer TF_GetBuffer(TF_Buffer* buffer);

TF_Buffer* TF_NewBuffer() { return new TF_Buffer{nullptr, 0, nullptr}; }

void TF_DeleteBuffer(TF_Buffer* buffer) {
  if (buffer == nullptr) return;
  if (buffer->data_deallocator != nullptr) {
    (*buffer->data_deallocator)(const_cast<void*>(buffer->data),
                               buffer->length);
  }
  delete buffer;
}

TF_Buffer TF_GetBuffer(TF_Buffer* buffer) { return *buffer; }
```

Global functions for manipulation of TF_SessionOptions

```
// -------------------------------------------------------------------------
// TF_SessionOptions holds options that can be passed during session creation.
typedef struct TF_SessionOptions TF_SessionOptions;

// Return a new options object.
TF_CAPI_EXPORT extern TF_SessionOptions* TF_NewSessionOptions(void);

// Set the target in TF_SessionOptions.options.
// target can be empty, a single entry, or a comma separated list of entries.
// Each entry is in one of the following formats :
// "local"
// ip:port
// host:port
TF_CAPI_EXPORT extern void TF_SetTarget(TF_SessionOptions* options,
                                        const char* target);

// Set the config in TF_SessionOptions.options.
// config should be a serialized tensorflow.ConfigProto proto.
// If config was not parsed successfully as a ConfigProto, record the
// error information in *status.
TF_CAPI_EXPORT extern void TF_SetConfig(TF_SessionOptions* options,
                                        const void* proto, size_t proto_len,
                                        TF_Status* status);

// Destroy an options object.
TF_CAPI_EXPORT extern void TF_DeleteSessionOptions(TF_SessionOptions*);

TF_SessionOptions* TF_NewSessionOptions() { return new TF_SessionOptions; }
void TF_DeleteSessionOptions(TF_SessionOptions* opt) { delete opt; }

void TF_SetTarget(TF_SessionOptions* options, const char* target) {
  options->options.target = target;
}

void TF_SetConfig(TF_SessionOptions* options, const void* proto,
                  size_t proto_len, TF_Status* status) {
  if (!options->options.config.ParseFromArray(proto, proto_len)) {
    status->status = InvalidArgument("Unparseable ConfigProto");
  }
}

/// Configuration information for a Session.
struct TF_SessionOptions {
  tensorflow::SessionOptions options;
};
```

tensorflow/core/public/session_options.h

```
struct SessionOptions {
  /// The environment to use.
  Env* env;

  /// \brief The TensorFlow runtime to connect to.
  ///
  /// If 'target' is empty or unspecified, the local TensorFlow runtime
  /// implementation will be used.  Otherwise, the TensorFlow engine
  /// defined by 'target' will be used to perform all computations.
  ///
  /// "target" can be either a single entry or a comma separated list
  /// of entries. Each entry is a resolvable address of the
  /// following format:
  ///   local
  ///   ip:port
  ///   host:port
  ///   ... other system-specific formats to identify tasks and jobs ...
  ///
  /// NOTE: at the moment 'local' maps to an in-process service-based
  /// runtime.
  ///
  /// Upon creation, a single session affines itself to one of the
  /// remote processes, with possible load balancing choices when the
  /// "target" resolves to a list of possible processes.
  ///
  /// If the session disconnects from the remote process during its
  /// lifetime, session calls may fail immediately.
  std::string target;

  /// Configuration options.
  ConfigProto config;

  SessionOptions();
};
```

New graph construction API (under construction)

```
// Represents a computation graph.  Graphs may be shared between sessions.
// Graphs are thread-safe when used as directed below.
typedef struct TF_Graph TF_Graph;

// Return a new graph object.
TF_CAPI_EXPORT extern TF_Graph* TF_NewGraph(void);

// Destroy an options object.  Graph will be deleted once no more
// TFSession's are referencing it.
TF_CAPI_EXPORT extern void TF_DeleteGraph(TF_Graph*);

// Operation being built. The underlying graph must outlive this.
typedef struct TF_OperationDescription TF_OperationDescription;

// Operation that has been added to the graph. Valid until the graph is
// deleted -- in particular adding a new operation to the graph does not
// invalidate old TF_Operation* pointers.
typedef struct TF_Operation TF_Operation;

// Represents a specific input of an operation.
typedef struct TF_Input {
  TF_Operation* oper;
  int index;  // The index of the input within oper.
} TF_Input;

// Represents a specific output of an operation.
typedef struct TF_Output {
  TF_Operation* oper;
  int index;  // The index of the output within oper.
} TF_Output;

// TF_Function is a grouping of operations with defined inputs and outputs.
// Once created and added to graphs, functions can be invoked by creating an
// operation whose operation type matches the function name.
typedef struct TF_Function TF_Function;

// Function definition options. TODO(iga): Define and implement
typedef struct TF_FunctionOptions TF_FunctionOptions;

// Sets the shape of the Tensor referenced by `output` in `graph` to
// the shape described by `dims` and `num_dims`.
//
// If the number of dimensions is unknown, `num_dims` must be set to
// -1 and `dims` can be null. If a dimension is unknown, the
// corresponding entry in the `dims` array must be -1.
//
// This does not overwrite the existing shape associated with `output`,
// but merges the input shape with the existing shape.  For example,
// setting a shape of [-1, 2] with an existing shape [2, -1] would set
// a final shape of [2, 2] based on shape merging semantics.
//
// Returns an error into `status` if:
//   * `output` is not in `graph`.
//   * An invalid shape is being set (e.g., the shape being set
//     is incompatible with the existing shape).
TF_CAPI_EXPORT extern void TF_GraphSetTensorShape(TF_Graph* graph,
                                                  TF_Output output,
                                                  const int64_t* dims,
                                                  const int num_dims,
                                                  TF_Status* status);

// Returns the number of dimensions of the Tensor referenced by `output`
// in `graph`.
//
// If the number of dimensions in the shape is unknown, returns -1.
//
// Returns an error into `status` if:
//   * `output` is not in `graph`.
TF_CAPI_EXPORT extern int TF_GraphGetTensorNumDims(TF_Graph* graph,
                                                   TF_Output output,
                                                   TF_Status* status);

// Returns the shape of the Tensor referenced by `output` in `graph`
// into `dims`. `dims` must be an array large enough to hold `num_dims`
// entries (e.g., the return value of TF_GraphGetTensorNumDims).
//
// If the number of dimensions in the shape is unknown or the shape is
// a scalar, `dims` will remain untouched. Otherwise, each element of
// `dims` will be set corresponding to the size of the dimension. An
// unknown dimension is represented by `-1`.
//
// Returns an error into `status` if:
//   * `output` is not in `graph`.
//   * `num_dims` does not match the actual number of dimensions.
TF_CAPI_EXPORT extern void TF_GraphGetTensorShape(TF_Graph* graph,
                                                  TF_Output output,
                                                  int64_t* dims, int num_dims,
                                        TF_Status* status);

// Operation will only be added to *graph when TF_FinishOperation() is
// called (assuming TF_FinishOperation() does not return an error).
// *graph must not be deleted until after TF_FinishOperation() is
// called.
TF_CAPI_EXPORT extern TF_OperationDescription* TF_NewOperation(
    TF_Graph* graph, const char* op_type, const char* oper_name);

// Specify the device for `desc`.  Defaults to empty, meaning unconstrained.
TF_CAPI_EXPORT extern void TF_SetDevice(TF_OperationDescription* desc,
                                        const char* device);
```

bazel-tensorflow/tensorflow/c/c_api.cc

bazel-tensorflow/tensorflow/c/c_api_internal.h

bazel-tensorflow/tensorflow/c/c_api_function.cc

bazel-tensorflow/tensorflow/c/eager/c_api_unified_experimental_graph.cc // defines struct
                                                                       // GraphTensor, struct GraphFunction

bazel-tensorflow/tensorflow/c/c_api_experimental.cc

bazel-tensorflow/tensorflow/c/c_api_test.cc

bazel-tensorflow/tensorflow/c/c_api_function_test.cc

bazel-tensorflow/tensorflow/c/while_loop_test.cc

bazel-tensorflow/tensorflow/c/c_test_util.cc

bazel-tensorflow/tensorflow/c/eager/c_api_experimental_test.cc

Classes Graph and GraphDef

The classes Graph (or Computation Graph) is a core concept of tensorflow to present computation. When first using TF, we first will create Computation Graph and pass the Graph to TF.

The Computation Graph is given by class TF_Graph defined in

```
struct TF_Graph {
  TF_Graph();

  tensorflow::mutex mu;
  tensorflow::Graph graph TF_GUARDED_BY(mu);

  // Runs shape inference.
  tensorflow::ShapeRefiner refiner TF_GUARDED_BY(mu);

  // Maps from name of an operation to the Node* in 'graph'.
  std::unordered_map<tensorflow::string, tensorflow::Node*> name_map
      TF_GUARDED_BY(mu);

  // The keys of this map are all the active sessions using this graph. Each
  // value records whether the graph has been mutated since the corresponding
  // session has been run (this is detected in RecordMutation function). If the
  // string is empty, no mutation has occurred. Otherwise the string is a
  // description of the mutation suitable for returning to the user.
  //
  // Sessions are added to this map in TF_NewSession, and removed in
  // TF_DeleteSession.
  // TF_Graph may only / must be deleted when
  //   sessions.size() == 0 && delete_requested == true
  //
  // TODO(b/74949947): mutations currently trigger a warning instead of a bad
  // status, this should be reverted when possible.
  tensorflow::gtl::FlatMap<TF_Session*, tensorflow::string> sessions
      TF_GUARDED_BY(mu);
  bool delete_requested TF_GUARDED_BY(mu);  // set true by TF_DeleteGraph

  // Used to link graphs contained in TF_WhileParams to the parent graph that
  // will eventually contain the full while loop.
  TF_Graph* parent;
  TF_Output* parent_inputs;
};
```

The internal container class tensorflow::Graph is defined in [tensorflow/core/graph/graph.h]() as:

```
// Thread compatible but not thread safe.
class Graph {
 public:
  // Constructs a graph with a single SOURCE (always id kSourceId) and a
  // single SINK (always id kSinkId) node, and an edge from SOURCE->SINK.
  //
  // The graph can hold ops found in the registry. `ops`s lifetime must be at
  // least that of the constructed graph's.
  explicit Graph(const OpRegistryInterface* ops);

  // Constructs a graph with a single SOURCE (always id kSourceId) and a
  // single SINK (always id kSinkId) node, and an edge from SOURCE->SINK.
  //
```

```
// The graph can hold ops found in `flib_def`. Unlike the constructor taking
// an OpRegistryInterface, this constructor copies the function definitions in
// `flib_def` so its lifetime may be shorter than that of the graph's. The
// OpRegistryInterface backing `flib_def` must still have the lifetime of the
// graph though.
explicit Graph(const FunctionLibraryDefinition& flib_def);

~Graph();

static const int kControlSlot;

// The GraphDef version range of this graph (see graph.proto).
const VersionDef& versions() const;
void set_versions(const VersionDef& versions);

// Adds a new node to this graph, and returns it. Infers the Op and
// input/output types for the node. *this owns the returned instance.
// Returns nullptr and sets *status on error.
Node* AddNode(NodeDef node_def, Status* status);

// Copies *node, which may belong to another graph, to a new node,
// which is returned.  Does not copy any edges.  *this owns the
// returned instance.
Node* CopyNode(const Node* node);

// Removes a node from this graph, including all edges from or to it.
// *node should not be accessed after calling this function.
// REQUIRES: node->IsOp()
void RemoveNode(Node* node);

// Adds an edge that connects the xth output of `source` to the yth input of
// `dest` and returns it. Does not update dest's NodeDef.
const Edge* AddEdge(Node* source, int x, Node* dest, int y);

// Adds a control edge (no data flows along this edge) that connects `source`
// to `dest`. If `dest`s NodeDef is missing the corresponding control input,
// adds the control input.
//
// If such a control edge already exists and `allow_duplicates` is false, no
// edge is added and the function returns nullptr. Otherwise the edge is
// unconditionally created and returned. The NodeDef is not updated if
// `allow_duplicates` is true.
// TODO(skyewm): // TODO(skyewm): allow_duplicates is needed only by
// graph_partition.cc. Figure out if we can do away with it.
```

```
const Edge* AddControlEdge(Node* source, Node* dest,
                           bool allow_duplicates = false);

// Removes edge from the graph. Does not update the destination node's
// NodeDef.
// REQUIRES: The edge must exist.
void RemoveEdge(const Edge* edge);

// Removes control edge `edge` from the graph. Note that this also updates
// the corresponding NodeDef to reflect the change.
// REQUIRES: The control edge must exist.
void RemoveControlEdge(const Edge* e);

// Updates the input to a node.  The existing edge to `dst` is removed and an
// edge from `new_src` to `dst` is created. The NodeDef associated with `dst`
// is also updated.
Status UpdateEdge(Node* new_src, int new_src_index, Node* dst, int dst_index);

// Like AddEdge but updates dst's NodeDef. Used to add an input edge to a
// "While" op during gradient construction, see AddInputWhileHack in
// python_api.h for more details.
Status AddWhileInputHack(Node* new_src, int new_src_index, Node* dst);

// Adds the function and gradient definitions in `fdef_lib` to this graph's op
// registry. Ignores duplicate functions, and returns a bad status if an
// imported function differs from an existing function or op with the same
// name.
Status AddFunctionLibrary(const FunctionDefLibrary& fdef_lib);

// The number of live nodes in the graph.
//
// Because nodes can be removed from the graph, num_nodes() is often
// smaller than num_node_ids(). If one needs to create an array of
// nodes indexed by node ids, num_node_ids() should be used as the
// array's size.
int num_nodes() const { return num_nodes_; }

// The number of live nodes in the graph, excluding the Source and Sink nodes.
int num_op_nodes() const {
  DCHECK_GE(num_nodes_, 2);
  return num_nodes_ - 2;
}

// The number of live edges in the graph.
```

```
//
// Because edges can be removed from the graph, num_edges() is often
// smaller than num_edge_ids(). If one needs to create an array of
// edges indexed by edge ids, num_edge_ids() should be used as the
// array's size.
int num_edges() const { return num_edges_; }

// Serialize the nodes starting at `from_node_id` to a GraphDef.
void ToGraphDefSubRange(GraphDef* graph_def, int from_node_id) const;

// Serialize to a GraphDef.
void ToGraphDef(GraphDef* graph_def) const;

// This version can be called from debugger to inspect the graph content.
// Use the previous version outside debug context for efficiency reasons.
//
// Note: We do not expose a DebugString() API, since GraphDef.DebugString() is
// not defined in some TensorFlow builds.
GraphDef ToGraphDefDebug() const;

// Generate new node name with the specified prefix that is unique
// across this graph.
string NewName(StringPiece prefix);

// Access to the list of all nodes.  Example usage:
//   for (Node* node : graph.nodes()) { ... }
gtl::iterator_range<NodeIter> nodes() const;

// Access to the list of all nodes, excluding the Source and Sink nodes.
gtl::iterator_range<NodeIter> op_nodes() const;

// Returns one more than the maximum id assigned to any node.
int num_node_ids() const { return nodes_.size(); }

// Returns the node associated with an id, or nullptr if no node
// with that id (the node with that id was removed and the id has
// not yet been re-used). *this owns the returned instance.
// REQUIRES: 0 <= id < num_node_ids().
Node* FindNodeId(int id) const { return nodes_[id]; }

// Returns one more than the maximum id assigned to any edge.
int num_edge_ids() const { return edges_.size(); }

// Returns the Edge associated with an id, or nullptr if no edge
// with that id (the node with that id was removed and the id has
// not yet been re-used). *this owns the returned instance.
// REQUIRES: 0 <= id < num_node_ids().
const Edge* FindEdgeId(int id) const { return edges_[id]; }

// Access to the set of all edges.  Example usage:
//   for (const Edge* e : graph.edges()) { ... }
GraphEdgesIterable edges() const { return GraphEdgesIterable(edges_); }

// The pre-defined nodes.
enum { kSourceId = 0, kSinkId = 1 };
Node* source_node() const { return FindNodeId(kSourceId); }
Node* sink_node() const { return FindNodeId(kSinkId); }

const OpRegistryInterface* op_registry() const { return &ops_; }
const FunctionLibraryDefinition& flib_def() const { return ops_; }

void CheckDeviceNameIndex(int index) {
  DCHECK_GE(index, 0);
  DCHECK_LT(index, static_cast<int>(device_names_.size()));
}

int InternDeviceName(const string& device_name);

const string& get_assigned_device_name(const Node& node) const {
  return device_names_[node.assigned_device_name_index()];
}

void set_assigned_device_name_index(Node* node, int device_name_index) {
  CheckDeviceNameIndex(device_name_index);
  node->assigned_device_name_index_ = device_name_index;
}

void set_assigned_device_name(Node* node, const string& device_name) {
  node->assigned_device_name_index_ = InternDeviceName(device_name);
}

// Returns OK if `node` is non-null and belongs to this graph
Status IsValidNode(const Node* node) const;

// Returns OK if IsValidNode(`node`) and `idx` is a valid output.  Does not
// accept control outputs.
Status IsValidOutputTensor(const Node* node, int idx) const;
```

```
 // Returns OK if IsValidNode(`node`) and `idx` a valid input.  Does not accept
 // control inputs.
 Status IsValidInputTensor(const Node* node, int idx) const;

 // Create and return a new WhileContext owned by this graph. This is called
 // when a new while loop is created. `frame_name` must be unique among
 // WhileContexts in this graph.
 Status AddWhileContext(StringPiece frame_name, std::vector<Node*> enter_nodes,
                        std::vector<Node*> exit_nodes,
                        OutputTensor cond_output,
                        std::vector<OutputTensor> body_inputs,
                        std::vector<OutputTensor> body_outputs,
                        WhileContext** result);

 // Builds a node name to node pointer index for all nodes in the graph.
 std::unordered_map<string, Node*> BuildNodeNameIndex() const;

 absl::optional<std::vector<bool>>& GetConstArgIndicesCache() const {
   return const_arg_indices_cache_;
 }

 // TODO(josh11b): uint64 hash() const;

private:
 // If cost_node is non-null, then cost accounting (in CostModel)
 // will be associated with that node rather than the new one being
 // created.
 //
 // Ownership of the returned Node is not transferred to caller.
 Node* AllocateNode(std::shared_ptr<NodeProperties> props,
                    const Node* cost_node, Node::NodeClass node_class);
 void ReleaseNode(Node* node);
 // Insert edge in free_edges_ for possible reuse.
 void RecycleEdge(const Edge* edge);
 // Registry of all known ops, including functions.
 FunctionLibraryDefinition ops_;

 // GraphDef versions
 const std::unique_ptr<VersionDef> versions_;

 // Allocator which will give us good locality.
 core::Arena arena_;

 // Map from node ids to allocated nodes.  nodes_[id] may be nullptr if
// the node with that id was removed from the graph.
std::vector<Node*> nodes_;

// Number of nodes alive.
int64 num_nodes_ = 0;

// Map from edge ids to allocated edges.  edges_[id] may be nullptr if
// the edge with that id was removed from the graph.
std::vector<Edge*> edges_;

// The number of entries in edges_ that are not nullptr.
int num_edges_ = 0;

// Allocated but free nodes and edges.
std::vector<Node*> free_nodes_;
std::vector<Edge*> free_edges_;

// For generating unique names.
int name_counter_ = 0;

// In most graphs, the number of unique values used for the
// Node::assigned_device_name() property is quite small.  If the graph is
// large, then this duplication of values can consume a significant amount of
// memory.  Instead, we represent the same information using an interning
// table, which consists of a vector of unique strings (device_names_), as
// well a map (device_names_map_) from unique strings to indices within the
// unique string table.
//
// The InternDeviceName() method handles adding a new entry into the table,
// or locating the index of an existing entry.
//
// The fact that Node::assigned_device_name() is implemented using an
// interning table is intentionally public.  This allows algorithms that
// frequently access this field to do so efficiently, especially for the case
// where the assigned_device_name of one Node is copied directly from that
// of another Node.

// A table of the unique assigned device names.  Indices do NOT correspond
// to node IDs.  Index 0 is always the empty string.
std::vector<string> device_names_;

// Maps unique device names to indices within device_names_[i].
std::unordered_map<string, int> device_names_map_;
```

```
  // All the while contexts owned by this graph, keyed by frame name,
  // corresponding to all the while loops contained in this graph (including
  // nested loops). The stored contexts are usually accessed via
  // AddWhileContext() or Node::while_ctx(), but this manages the lifetime.
  std::map<string, WhileContext> while_ctxs_;

  // Cache of the indices of the arguments which need to be constant for the XLA
  // compilation.
  mutable absl::optional<std::vector<bool>> const_arg_indices_cache_;

  TF_DISALLOW_COPY_AND_ASSIGN(Graph);
};
```

GraphDef is a serialization utility class which binds to Graph. The corresponding Protobuf interface is defined as:

```
// Represents the graph of operations
message GraphDef {
  repeated NodeDef node = 1;

  // Compatibility versions of the graph.  See core/public/version.h for version
  // history.  The GraphDef version is distinct from the TensorFlow version, and
  // each release of TensorFlow will support a range of GraphDef versions.
  VersionDef versions = 4;

  // Deprecated single version field; use versions above instead.  Since all
  // GraphDef changes before "versions" was introduced were forward
  // compatible, this field is entirely ignored.
  int32 version = 3 [deprecated = true];

  // EXPERIMENTAL. DO NOT USE OR DEPEND ON THIS YET.
  //
  // "library" provides user-defined functions.
  //
  // Naming:
  //   * library.function.name are in a flat namespace.
  //     NOTE: We may need to change it to be hierarchical to support
  //     different orgs. E.g.,
  //     { "/google/nn", { ... }},
  //     { "/google/vision", { ... }}
  //     { "/org_foo/module_bar", { ... }}
  //     map<string, FunctionDefLib> named_lib;
  //   * If node[i].op is the name of one function in "library",
  //     node[i] is deemed as a function call. Otherwise, node[i].op
  //     must be a primitive operation supported by the runtime.
  //
  //
  // Function call semantics:
  //
  //   * The callee may start execution as soon as some of its inputs
  //     are ready. The caller may want to use Tuple() mechanism to
  //     ensure all inputs are ready in the same time.
  //
  //   * The consumer of return values may start executing as soon as
  //     the return values the consumer depends on are ready.  The
  //     consumer may want to use Tuple() mechanism to ensure the
  //     consumer does not start until all return values of the callee
  //     function are ready.
  FunctionDefLibrary library = 2;
}
```

Protobuf interfaces and formats

Core/Protobuf/Config.proto

https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/protobuf/config.proto

This interface contains various options:

1) for tuning the resources occupied by the GPU (see message GPUOptions).
   Inside GPUOptions there are various experimental configuration options such as per virtual device memory limit, memory-specific options, kernel-specific timing and memory parameters (see message Experimental).
2) Optimizer tuning parameters (see message OptimizerOptions). For instance there is an option for selecting the level at which the Optimizer works where level L1 denotes common subexpression elimination and constant folding. a special option for turning on the internal Just-in-time compiler and selecting how aggressive the auto-compilation should be.
3) Various graph options (see message GraphOptions). For instance when to build a cost model for the nodes in the graph in terms of memory and cpu resource consumption, parameters controlling various aspects of the graph construction and updates
4) Thread pool tuning parameters and options

Exploring the C++ code examples

Exploring //tensorflow/core/example

Proto file for ***Feature***

This proto file contains protocol messages for describing features for machine learning model training or inference. There are three base ***Feature*** types:

- bytes
- float
- int64

A ***Feature*** contains Lists which may hold zero or more values. These lists are the base values ***BytesList***, ***FloatList***, ***Int64List***.
Features are organized into categories by name. The ***Features*** message contains the mapping from the name to Feature. Here are example Features for a movie recommendation application:

```
Feature {
 key: “age”
 value: { float_list {
     value: 29.0
  }}
}
Feature {
  key: “movie”
  value: { bytes_list {
        value: “The Shawshank Redemption”
        value: “Fight Clubs”
  }}
}
Feature {
    key: “movie_ratings”
    value: { float_list {
              value: 9.0,
              value: 9.7
    }}
}
Feature {
    key: “suggestion”
    value: { byte_list {
             value: “Inception”
    }}
}
Feature {
    key: “suggestion_purchased”
    value: { int64_list {
             value: 1
    }}
}
Feature {
    key: "purchase_price"
    value: { float_list {
              value: 9.99
    }}
}

// containers to hold repeated fundamental values
message ByteList {
  repeated bytes value = 1;
}

message FloatList {
  repeated float value = 1 [packed = true];
}

message Int64List {
  repeated int64 value = 1 [packed = true];
}

// Containers for non-sequential data.
message Feature {
  // Each feature can be exactly one kind.
  oneof kind {
    BytesList bytes_list = 1;
    FloatList float_list = 2;
    Int64List int64_list = 3;
  }
}

message Features {
  // Map from feature name to feature.
  map<string, Feature> feature = 1;
}

// Containers for sequential data.
//
// A FeatureList contains lists of Features.  These may hold zero or more
// Feature values.
//
// FeatureLists are organized into categories by name.  The FeatureLists message
// contains the mapping from name to FeatureList.
```

```
//
message FeatureList {
  repeated Feature feature = 1;
}

message FeatureLists {
  // Map from feature name to feature list.
  map<string, FeatureList> feature_list = 1;
}
```

Proto file for ***Example***:

An ***Example*** is a mostly-normalized data format for storing data for training and inference. It contains key-value store (features); where each key (string) maps to a Feature message which is one of packed BytesList, FloatList, or Int64List. This flexible and compact format allows the storage of large amounts of typed data, but it requires that the data shape and use be determined by the configuration files and parsers that are used to read and write that format. That is the ***Example*** is mostly not self-describing format. In TF, Examples are read in row-major format so any configuration that describes data with rank-2 or above should keep that in mind. For example, to store an M x N Matrix of Bytes, the BytesList must contain M*N bytes with M rows of N contiguous values each. That is, the ByteList value must store the matrix as:

```
//     .... row 0 .... .... row 1 .... // ...........  // ... row M-1 ....
```

An Example for a movie recommendation application:

```
Features {
  Feature {
     key: "age"
        value { float_list {
        value: 29.0
     }}
   }
  Feature {
     key: "movie"
     value { bytes_list {
       value: "The Shawshank Redemption"
       value: "Fight Club"
     }}
   }
  Feature {
     key: "movie_ratings"
     value { float_list {
       value: 9.0
       value: 9.7
     }}
```

```
    }
   Feature {
       key: "suggestion"
       value { bytes_list {
         value: "Inception"
       }}
    }
 # Note that this feature exists to be used as a label in training.
 # E.g., if training a logistic regression model to predict purchase
 # probability in our learning tool we would set the label feature to
 # "suggestion_purchased".
   Feature {
     key: "suggestion_purchased"
     value { float_list {
        value: 1.0
     }}
    }
# Similar to "suggestion_purchased" above this feature exists to be used
# as a label in training.
# E.g., if training a linear regression model to predict purchase
# price in our learning tool we would set the label feature to
# "purchase_price".
//    feature {
//      key: "purchase_price"
//      value { float_list {
//        value: 9.99
//      }}
//    }
//  }
//
```

Building the tensorflow libraries and examples using Bazel

Building tensorflow libraries

Building tensorflow libraries with debug symbols

```
bazel build --config=opt --verbose_failures -c dbg --strip=never
//tensorflow:libtensorflow_cc.so
```

```
bazel build --config=opt --verbose_failures -c dbg --strip=never
//tensorflow:libtensorflow_framework.so
```

Building tensorflow C++ api library using monolithic config

```
bazel build -c opt --config=monolithic //tensorflow:libtensorflow_cc.so
```

Building tensorflow core components

```
bazel build --config=opt //tensorflow/core:lib
```
exports the public non-test headers for:
`//third_party/tensorflow/core/platform:` platform-specific code and external dependencies
`lib/:` Low-level libraries that are not TensorFlow-specific

```
bazel build --config=opt //tensorflow/core:framework
```
exports the public non-test headers for:
`util/:` General low-level TensorFlow-specific libraries
`framework/:` Support for adding new ops & kernels
`example/:` Wrappers to simplify access to Example proto

Building Example Code

Building example code: parser configuration test
```
bazel build --config=opt //tensorflow/core/example:example_parser_configuration_test
```

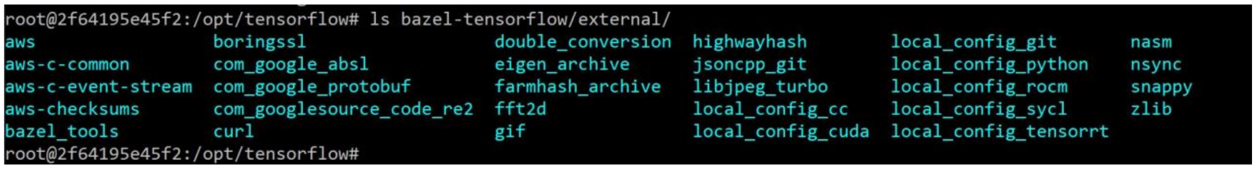
Building label_image example code with debug symbols:
```
bazel build --config=opt --verbose_failures -c dbg --strip=never
tensorflow/examples/label_image/...
```

Building label_image example code with debug symbols and limited RAM resource (<2GB):
```
bazel build --config=opt --verbose_failures -c dbg --strip=never --jobs 1 --
local_ram_resources 2048 tensorflow/examples/label_image/...
```

Build Folder structure

ROOT_FOLDER = /opt/tensorflow

Inside ROOT_FOLDER/bazel-tensroflow/external :

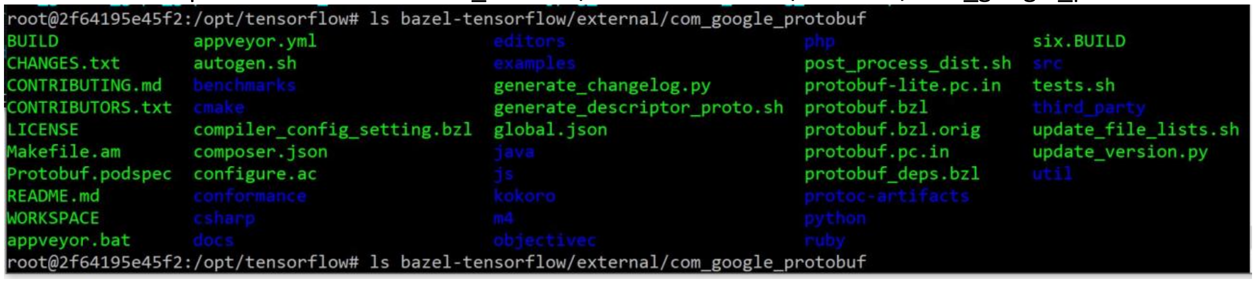

```
root@2f64195e45f2:/opt/tensorflow# ls bazel-tensorflow/external/
aws                  boringssl                  double_conversion  highwayhash        local_config_git       nasm
aws-c-common         com_google_absl            eigen_archive      jsoncpp_git        local_config_python    nsync
aws-c-event-stream   com_google_protobuf        farmhash_archive   libjpeg_turbo      local_config_rocm      snappy
aws-checksums        com_googlesource_code_re2  fft2d              local_config_cc    local_config_sycl      zlib
bazel_tools          curl                       gif                local_config_cuda  local_config_tensorrt
root@2f64195e45f2:/opt/tensorflow#
```

By default tensorflow master as of 4/26/2020 installs the following external package dependencies:

aws	*aws-c-common*	*aws-c-event-stream*
aws-checksums	*bazel_tools*	*boringssl*
com_google_absl	*com_google_protobuf*	*com_googlesource_code_re2*
com_google_grpc_grpc	*curl*	*double_conversion*
eigen_archive	*farmhash_archive*	*fft2d*
gif	*highwayhash*	*jsoncpp_git*
libjpeg_turbo	*local_config_cc*	*local_config_cuda*
local_config_git	*local_config_python*	*local_config_rocm*
local_config_sycl	*local_config_tensorrt*	*nasm*
nsync	*snappy*	*zlib*

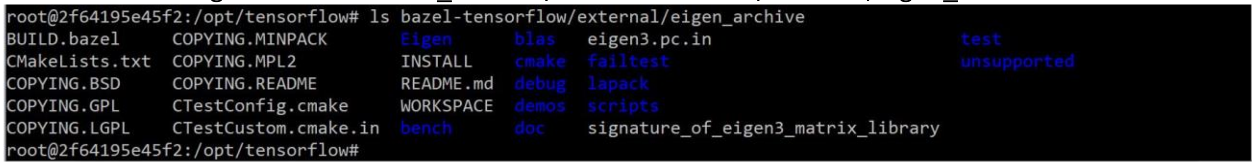
Some of the important libraries to build a C++ tensorflow app using the C++ tensorflow shared library libtensorflow_cc.so:

The location of protobuf source: ROOT_FOLDER/bazel-tensorflow/external/com_google_protobuf



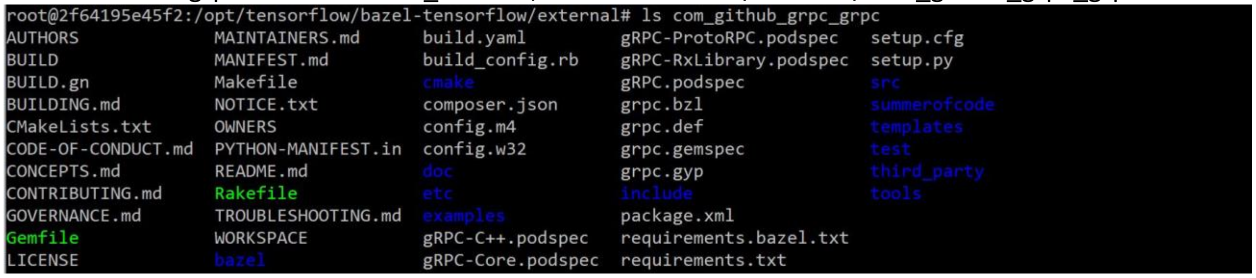
```
root@2f64195e45f2:/opt/tensorflow# ls bazel-tensorflow/external/com_google_protobuf
BUILD            appveyor.yml             editors                       php                     six.BUILD
CHANGES.txt      autogen.sh               examples                      post_process_dist.sh    src
CONTRIBUTING.md  benchmarks               generate_changelog.py         protobuf-lite.pc.in     tests.sh
CONTRIBUTORS.txt cmake                    generate_descriptor_proto.sh  protobuf.bzl            third_party
LICENSE          compiler_config_setting.bzl global.json                protobuf.bzl.orig       update_file_lists.sh
Makefile.am      composer.json            java                          protobuf.pc.in          update_version.py
Protobuf.podspec configure.ac             js                            protobuf_deps.bzl       util
README.md        conformance              kokoro                        protoc-artifacts
WORKSPACE        csharp                   m4                            python
appveyor.bat     docs                     objectivec                    ruby
root@2f64195e45f2:/opt/tensorflow# ls bazel-tensorflow/external/com_google_protobuf
```

The location of eigen source: ROOT_FOLDER/bazel-tensorflow/external/eigen_archive



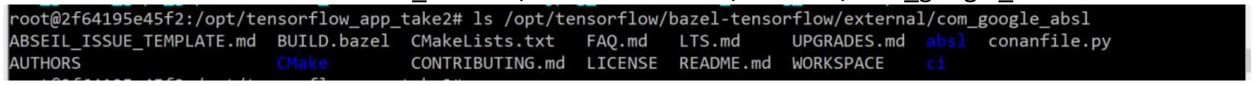
```
root@2f64195e45f2:/opt/tensorflow# ls bazel-tensorflow/external/eigen_archive
BUILD.bazel     COPYING.MINPACK      Eigen      blas   eigen3.pc.in                     test
CMakeLists.txt  COPYING.MPL2         INSTALL    cmake  failtest                         unsupported
COPYING.BSD     COPYING.README       README.md  debug  lapack
COPYING.GPL     CTestConfig.cmake    WORKSPACE  demos  scripts
COPYING.LGPL    CTestCustom.cmake.in bench      doc    signature_of_eigen3_matrix_library
root@2f64195e45f2:/opt/tensorflow#
```

The location of grpc source: ROOT_FOLDER/bazel-tensorflow/external/com_github_grpc_grpc



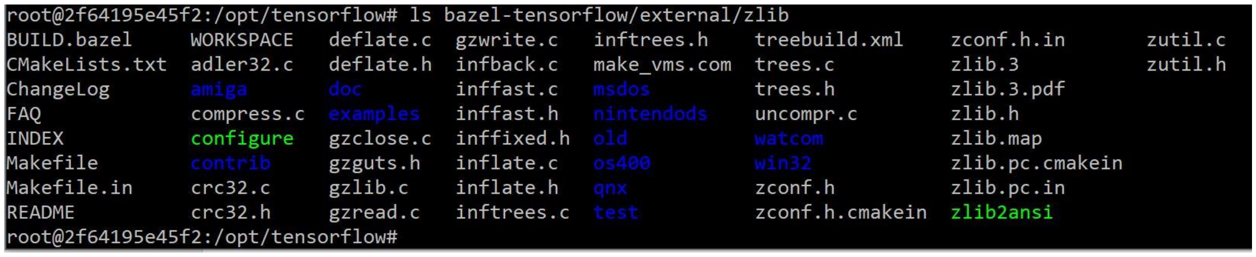
```
root@2f64195e45f2:/opt/tensorflow/bazel-tensorflow/external# ls com_github_grpc_grpc
AUTHORS             MAINTAINERS.md      build.yaml          gRPC-ProtoRPC.podspec   setup.cfg
BUILD               MANIFEST.md         build_config.rb     gRPC-RxLibrary.podspec  setup.py
BUILD.gn            Makefile            cmake               gRPC.podspec            src
BUILDING.md         NOTICE.txt          composer.json       grpc.bzl                summerofcode
CMakeLists.txt      OWNERS              config.m4           grpc.def                templates
CODE-OF-CONDUCT.md  PYTHON-MANIFEST.in  config.w32          grpc.gemspec            test
CONCEPTS.md         README.md           doc                 grpc.gyp                third_party
CONTRIBUTING.md     Rakefile            etc                 include                 tools
GOVERNANCE.md       TROUBLESHOOTING.md  examples            package.xml
Gemfile             WORKSPACE           gRPC-C++.podspec    requirements.bazel.txt
LICENSE             bazel               gRPC-Core.podspec   requirements.txt
```

The location of absl source: ROOT_FOLDER/bazel-tensorflow/external/com_google_absl



```
root@2f64195e45f2:/opt/tensorflow_app_take2# ls /opt/tensorflow/bazel-tensorflow/external/com_google_absl
ABSEIL_ISSUE_TEMPLATE.md  BUILD.bazel  CMakeLists.txt   FAQ.md   LTS.md     UPGRADES.md  absl  conanfile.py
AUTHORS                   CMake        CONTRIBUTING.md  LICENSE  README.md  WORKSPACE    ci
```

The location of zlib source: ROOT_FOLDER/bazel-tensorflow/external/zlib



```
root@2f64195e45f2:/opt/tensorflow# ls bazel-tensorflow/external/zlib
BUILD.bazel     WORKSPACE   deflate.c  gzwrite.c   inftrees.h    treebuild.xml    zconf.h.in       zutil.c
CMakeLists.txt  adler32.c   deflate.h  infback.c   make_vms.com  trees.c          zlib.3           zutil.h
ChangeLog       amiga       doc        inffast.c   msdos         trees.h          zlib.3.pdf
FAQ             compress.c  examples   inffast.h   nintendods    uncompr.c        zlib.h
INDEX           configure   gzclose.c  inffixed.h  old           watcom           zlib.map
Makefile        contrib     gzguts.h   inflate.c   os400         win32            zlib.pc.cmakein
Makefile.in     crc32.c     gzlib.c    inflate.h   qnx           zconf.h          zlib.pc.in
README          crc32.h     gzread.c   inftrees.c  test          zconf.h.cmakein  zlib2ansi
root@2f64195e45f2:/opt/tensorflow#
```

Build first C++ Tensorflow app

1. Build the tensorflow C++ API library with

```
bazel build -c opt --config=monolithic //tensorflow:libtensorflow_cc.so
```

or

```
       bazel build --config=opt --verbose_failures -c dbg --strip=never
//tensorflow:libtensorflow_cc.so
```
if we want to have DEBUG symbols available.

2. Build

Third Party and External Packages

Abseil common library: https://github.com/abseil/abseil-cpp/tree/20200225.2

Abseil is an open-source collection of C++ library code designed to augment the C++ standard library. The Abseil library code is collected from Google's own C++ code base, has been extensively tested and used in production, and is the same code we depend on in our daily coding lives.
In some cases, Abseil provides pieces missing from the C++ standard; in others, Abseil provides alternatives to the standard for special needs we've found through usage in the Google code base. We denote those cases clearly within the library code we provide you.

Abseil contains the following C++ library components:

base Abseil Fundamentals
The base library contains initialization code and other code which all other Abseil code depends on. Code within base may not depend on any other code (other than the C++ standard library).
algorithm
The algorithm library contains additions to the C++ <algorithm> library and container-based versions of such algorithms.
container
The container library contains additional STL-style containers, including Abseil's unordered "Swiss table" containers.
debugging
The debugging library contains code useful for enabling leak checks, and stacktrace and symbolization utilities.
hash
The hash library contains the hashing framework and default hash functor implementations for hashable types in Abseil.
memory
The memory library contains C++11-compatible versions of std::make_unique() and related memory management facilities.
meta
The meta library contains C++11-compatible versions of type checks available within C++14 and C++17 versions of the C++ <type_traits> library.
numeric
The numeric library contains C++11-compatible 128-bit integers.
strings
The strings library contains a variety of strings routines and utilities, including a C++11-compatible version of the C++17 std::string_view type.

synchronization
The synchronization library contains concurrency primitives (Abseil's absl::Mutex class, an alternative to std::mutex) and a variety of synchronization abstractions.
time
The time library contains abstractions for computing with absolute points in time, durations of time, and formatting and parsing time within time zones.
types
The types library contains non-container utility types, like a C++11-compatible version of the C++17 std::optional type.
utility
The utility library contains utility and helper code.
Tensorflow 2 master from April 27th uses Abseil tag 20200225.2.

BoringSSL: https://boringssl.googlesource.com/boringssl/

BoringSSL is a fork of OpenSSL that is designed to meet Google's needs.

Although BoringSSL is an open source project, it is not intended for general use, as OpenSSL is. We don't recommend that third parties depend upon it. Doing so is likely to be frustrating because there are no guarantees of API or ABI stability.

Programs ship their own copies of BoringSSL when they use it and we update everything as needed when deciding to make API changes. This allows us to mostly avoid compromises in the name of compatibility. It works for us, but it may not work for you.

BoringSSL arose because Google used OpenSSL for many years in various ways and, over time, built up a large number of patches that were maintained while tracking upstream OpenSSL. As Google's product portfolio became more complex, more copies of OpenSSL sprung up and the effort involved in maintaining all these patches in multiple places was growing steadily.

Currently BoringSSL is the SSL library in Chrome/Chromium, Android (but it's not part of the NDK) and a number of other apps/programs.

FarmHash: https://github.com/google/farmhash

Introducing FarmHash
Monday, March 31, 2014
We're pleased to announce the new FarmHash family of hash functions for strings. FarmHash is a successor to CityHash, and includes many of the same tricks and techniques, several of them taken from Austin Appleby's MurmurHash.

We're heavily influenced by the types of CPUs that are common in Google's datacenters, but FarmHash's goals don't end there. We want FarmHash to be fast and easy for developers to use in phones, tablets, and PCs too. So, yes, we've improved on CityHash64 and CityHash32 and so on. But we're also catering to the case where you simply want a fast, robust hash function for hash tables, and it need not be the same on every platform. To that end, we provide sample code that has one interface harboring multiple platform-specific implementations.

Over time, we plan to expand FarmHash to include hash functions for integers, tuples, and other data. For now, it provides hash functions for strings, though some of the subroutines could be adapted to other *uses.*

Overall, we believe that FarmHash provides high-performance solutions to some classic problems. Please give it a try! Contributions and bug reports are most welcome.

HighwayHash: https://github.com/google/highwayhash

Hash functions are widely used, so it is desirable to increase their speed and security. This package provides two 'strong' (well-distributed and unpredictable) hash functions: a faster version of SipHash, and an even faster algorithm we call HighwayHash.

SipHash is a fast but 'cryptographically strong' pseudo-random function by Aumasson and Bernstein [https://www.131002.net/siphash/siphash.pdf].

HighwayHash is a new way of mixing inputs which may inspire new cryptographically strong hashes. Large inputs are processed at a rate of 0.24 cycles per byte, and latency remains low even for small inputs. HighwayHash is faster than SipHash for all input sizes, with 5 times higher throughput at 1 KiB. We discuss design choices and provide statistical analysis and preliminary cryptanalysis in https://arxiv.org/abs/1612.06257.

Appendix A: Bazel tutorial for Tensorflow Builds

Appendix B: Hashing in Tensorflow

Understanding Hash Functions intro by Geoff Pike

```
UNDERSTANDING HASH FUNCTIONS
by Geoff Pike

Version 0.2 --- early draft --- comments and questions welcome!
References appear in square brackets.

1 INTRODUCTION

Hashing has proven tremendously useful in constructing various fast
data structures and algorithms.  It is typically possible to simplify
the analysis of hash-based algorithms if one assumes that the relevant
hash functions are high quality.  At the other extreme, if the
relevant hash functions were always to return the same value, many
hash-based algorithms become algorithms that are slower, simpler, but still well-known.
For example, a chaining hash table devolves into a linked list.

There are many possible definitions of hash function quality.  For
example, one might want a list of keys and their hashes to provide no
pattern that would allow an opponent to predict anything about the
hashes of other keys.  Although I cannot prove it, I think I can meet
this and many other definitions of quality with

  f(s) = SHA-3(concatenation of z and s),

where z is some secret string known only to me.  This well-known trick
```

provides, I think, more high-quality hash functions than anyone will need, though greater computational power in the future may push us to replace SHA-3 from time to time.

In short, discussions about choosing a hash function are almost always discussions about speed, energy consumption, or similar. Concerns about hash quality are easy to fix, for a price.

2 ANATOMY OF A HASH FUNCTION

Hash functions that input strings of arbitrary length are written in terms of an internal state, S. In many cases the internal state is a fixed number of bits and will fit in machine registers. One generic sketch of a string hash is:

```
  let S = some initial value
  let c = the length of S in bits
  while (input is not exhausted) {
    let t = the next c bits of input (padded with zeroes if less than c remain)
    S = M(S xor t)
  }
  let n = the number of bytes hashed
  return F(S, n)
```

where M is a hash function that inputs and outputs c bits, and F is a hash function that inputs c bits (plus, say, 64 for its second argument) and outputs however many bits one needs to return. In some sense we have reduced the string-hashing problem to two integer hashing problems.

2.1 INTEGER HASHING TECHNIQUES

A hash function that inputs and outputs the same number of bits, say, 32, can use reversible bit-twiddling operations, each of which is "onto" in the mathematical sense. For example, multiplication by an odd constant is reversible, as all odd numbers are relatively prime to 2^32. Other commonly used reversible operations include:

- o Adding or xoring a constant
- o Bitwise rotation or other bitwise permutations
- o bit j = (bit j) xor (bit k) for unequal constants j and k
- o "Shift mix": S = S xor (S >> k), where k is, say, 17
- o Replacing a fixed-length bit string with its cyclic redundancy checksum, perhaps via _mm_crc32_u32(f, <some constant>) [Pike]

Each of the above is a "bad" hash function that inputs and outputs the same number of bits. One can simply compose two or more of those bad hash functions to construct a higher-quality hash function.

One common quality goal for integer hashing (and string hashing) is that flipping the 19th bit, or any other small change, applied to multiple input keys, causes a seemingly unpredictable difference each time. Similarly, any change to an input should lead to a seemingly unpredictable selection of the output bits to flip.

Therefore, if we want a high-quality hash function that inputs c bits and outputs fewer than c bits, we can simply truncate the output of a high-quality hash function that inputs and outputs c bits.

To give a concrete example, here is Bob Jenkins' mix(), published in 1996 [Jenkins]. Its input is 96 bits in three 32-bit variables, and its output is 96 bits. However, one may use a subset of the output bits, as every output bit is affected by every non-empty subset of the input bits.

```
  Input: a, b, and c
  Algorithm:
    a -= b; a -= c; a ^= (c>>13);
    b -= c; b -= a; b ^= (a<<8);
    c -= a; c -= b; c ^= (b>>13);
    a -= b; a -= c; a ^= (c>>12);
    b -= c; b -= a; b ^= (a<<16);
    c -= a; c -= b; c ^= (b>>5);
    a -= b; a -= c; a ^= (c>>3);
    b -= c; b -= a; b ^= (a<<10);
    c -= a; c -= b; c ^= (b>>15);
  Output: a, b, and c
```

2.2 VARIATIONS ON STRING HASHING

There are three variations on our initial sketch worth noting.

First, for speed, one can special-case short inputs, as the CityHash and FarmHash algorithms do. The number of special cases can be reduced by using loads that may overlap: for example, a hash of a 9- to 16-byte string can be implemented by a hash that inputs two 8-byte values (the first 8 and last 8 bytes of the input string) and the string length [CityHash, FarmHash].

Second, one may choose different means of incorporating input bits into the internal state. One example: the mixing of S and input bits may be interleaved with the mixing of parts of S and other parts of S. Another example: the input bits processed in a loop iteration might be xor'ed into multiple places in S, rather than just one, or might be hashed with each other before touching S [Murmur]. The advantages and disadvantages of these are unclear.

Third, one may repeatedly "squeeze information" from S, by remixing it with itself and then revealing a subset of S. This is convenient when one would like a family of hash functions with different output lengths. A special case of the idea, called the "sponge construction," has been well studied and adopted by the authors of Keccak and others [SHA-3].

3 HASH FUNCTIONS FOR HASH TABLES

It isn't hard to find real-life examples where hash tables or the hash functions for them take more than 5% of a program's CPU time. Improvements to hash tables and their hash functions are therefore a classic example of software performance tuning. Unfortunately, the best choice may be platform-dependent, so to avoid writing your own collection of #ifdefs, please consider selecting something like the FarmHash family of hash functions, that supply decent platform-dependent logic for you.

To tune a program, often one will replace an existing hash function with a faster, lower-quality hash function, despite the increased chance of unlucky or pathological performance problems. Clever algorithms can mitigate this risk. For example, hash tables can start with one hash function and then switch to another if things seem to be going poorly. Therefore, one should rarely plan to spend much CPU time on a secure hash function (such as SHA-3) or a near-universal hash function (such as VHASH) when seeking the best possible performance from a hash table. Against that, those types of hash functions can limit the risk of pathological performance problems when one is designing around typical hash-based algorithms that stick with a single hash function no matter how it behaves on the data at hand.

4 REFERENCES

```
[Murmur] Appleby, Austin. https://code.google.com/p/smhasher,
            https://sites.google.com/site/murmurhash/
[SMHasher] Appleby, Austin. https://code.google.com/p/smhasher
[SHA-3] Bertoni, Guido, et al. http://keccak.noekeon.org/
[Jenkins] Jenkins, Bob. http://burtleburtle.net/bob/hash/doobs.html
[VHASH] Krovetz, Ted. Message authentication on 64-bit architectures. In
            Selected Areas of Cryptography – SAC 2006.  Springer-Verlag, 2006.
[CityHash] Pike, Geoff and Alakuijala, Jyrki. https://code.google.com/p/cityhash
[FarmHash] Pike, Geoff. https://code.google.com/p/farmhash
[Pike] Pike, Geoff. http://www.stanford.edu/class/ee380/Abstracts/121017-slides.pdf
```

CityHash: Fast hash functions for Strings by Geoff Pike, Google

Notation

N – the length of the inputs (bytes)

^ - exponentiation operator

a $\oplus$ b – bitwise exclusive or

a + b – sum (usually mod 2^64)

a * b – product (usually mod 2^64)

$\sigma_n(a)$ – right shift by n bits

$\sigma_{-n}(a)$ – left shift by n bits

$\rho_n(a)$ – right rotate by n bits

$\rho_{-n}(a)$ – left rotate by n bits

$\beta(a)$ – byteswap a

B_i – the i-th byte of the input (0-based)

W_i^b - the i-th b-bit word of the input

W_{-1}^b - the last b-bit word of the input

W_{-2}^b - the second-to-last b-bit word of the input

GF(n) – Galois field of order n

Deg(p) – degree of polynomial p

≡ - equivalence relation

Cyclic Redundancy Check (CRC)

CRC can be viewed as a result of algebraic operations applied to polynomials which coefficients are elements of GF(2). Recall, in GF(2):

0 is additive identity

1 is multiplicative identity

1 + 1 = 0 + 0 = 0

Sample polynomial:

p = x^32 + x^27 + 1

We can use p to define equivalence relation : we say that q and r are equivalent iff they differ by polynomial times p

Observation1: The equivalence relation has 2^Deg(p) elements

Lemma1: If Deg(p) == Deg(q) > 0

  Then Deg(p+q) < Deg(q)

 Else Deg(p+q) = max(Deg(p),Deg(q))

Observation2: There are 2^Deg(p) polynomials with degree less than Deg(p), none equivalent.

Observation3: Any polynomial with degree >= Deg(p) is equivalent to a lower degree polynomial
Example1: What is a degree <= 31 polynomial equivalent to x^50?
Deg(x^50) – Deg(p) = 18. Therefore x^50 – x^18*p has a degree less than 50.
x^50 – x^18*p = x^50 – x^18 * (x^32 + x^27 + 1) = x^45 + x^18
Applying the same idea repeatedly will lead us to the lowest degree polynomial that is equivalent to x^50:
x^50 ≡ x^30 + x^18 + x^13 + x^8 + x^3
More examples for the defined equivalence relation:
x^50 + 1 ≡ x^30 + x^18 + x^13 + x^8 + x^3 + 1
x^51 ≡ x^31 + x^19 + x^14 + x^9 + x^4
x^51 + x^50 ≡ x^31 + x^30 + x^19 + x^18 + x^14 + x^13 + x^9 + x^8 + x^4 + x^3
x^51 + x^31 ≡ x^19 + x^14 + x^9 + x^4
Let us look into the CRC implementation known as _mm_crc32_u64() / crc32q:
The inputs of crc32q are 32-bit number and 64-bit number. The output is 32-bit number.
The polynomial p used in _mm_crc32_u64 is :
x^32+x^28+x^27+x^26+x^25+x^23+x^22+x^20+x^19+x^18+x^14+x^13+x^11+x^10+x^9+x^8+x^6+1
which corresponds to the following normal hex number representation: `0x11EDC6F41`.

Here is summary of the algorithm used for calculating the intrinsic _mm_crc32_u64 on Intel CPUs.
Details can be found here.

```
unsigned __int64 _mm_crc32_u64 (unsigned __int64 crc, unsigned __int64 v)
```

Synopsis

```
unsigned __int64 _mm_crc32_u64 (unsigned __int64 crc, unsigned __int64 v)
#include <nmmintrin.h>
Instruction: crc32 r64, r64
CPUID Flags: SSE4.2
```

Description

Starting with the initial value in crc, accumulates a CRC32 value for unsigned 64-bit integer v, and stores the result in dst.

Operation

```
tmp1[63:0]  := v[0:63] // bit reflection
tmp2[31:0]  := crc[0:31] // bit reflection
tmp3[95:0]  := tmp1[31:0] << 32
tmp4[95:0]  := tmp2[63:0] << 64
tmp5[95:0]  := tmp3[95:0] XOR tmp4[95:0]
tmp6[31:0]  := MOD2(tmp5[95:0], 0x11EDC6F41) // remainder from polynomial division
                                             // modulus 2
dst[31:0] := tmp6[0:31] // bit reflection
```

Traditional String Hashing

Hash function loops over the input. While looping the internal state is kept in registers. In each iteration consume a fix amount of input. Here is a sample loop for traditional byte-at-a-time hash

```
for (int i = 0; i < N; i++) {
  state = Combine(state, $B_i$)
  state = Mix(state)
}
```

Two more concrete loop examples (old):

```
for (int i=0; i < N; i++)
  state = $\rho_{-5}$(state) $\oplus$ $B_i$
```

```
for (int i=0; i < N; i++)
  state = 33 * state + $B_i$
```

A complete byte-at-a-time example (Bob Jenkins circa 1996):

```
int state = 0
for (int i = 0; i < N; i++) {
  state = state + $B_i$
  state = state + $\sigma_{-10}(state)$
  state = state $\oplus$ $\sigma_6(state)$
}
state = state + $\sigma_{-3}(state)$
state = state $\oplus$ $\sigma_{11}(state)$
state = state + $\sigma_{-15}(state)$
```

Jenkins' Mix

Also around 1996 Bob Jenkins published a hash function with 96-bit and 96-bit output.
Pseudocode with three 32-bit registers:

```
a = a – b; a = a – c; a = a $\oplus$ $\sigma_{13}(c)$
b = b – c; b = b – a; b = b $\oplus$ $\sigma_{-8}(a)$
c = c – a; c = c – b; c = c $\oplus$ $\sigma_{13}(b)$
a = a – b; a = a – c; a = a $\oplus$ $\sigma_{12}(c)$
b = b – c; b = b – a; b = b $\oplus$ $\sigma_{-16}(a)$
c = c – a; c = c – b; c = c $\oplus$ $\sigma_5(b)$
a = a – b; a = a – c; a = a $\oplus$ $\sigma_3(c)$
b = b – c; b = b – a; b = b $\oplus$ $\sigma_{-10}(a)$
c = c – a; c = c – b; c = c $\oplus$ $\sigma_{15}(b)$
```

Jenkins' Mix-based string Hash

Given mix(a,b,c) as defined above , pseudo code for string hash:

```
uint32 a = …
uint32 b = …
uint32 c = …
int iters = $\lfloor N/12 \rfloor$
for (int i = 0; i < iters; i++) {
  a = a + $W_{3i}$
  b = b + $W_{3i+1}$
  c = c + $W_{3i+2}$
  mix(a, b, c)
}
```

Until 2009 Google was using 32-bit and 64-bit flavors of Jenkins' hash. Then tested Austin Appleby's 64-bit Murmur hash and found it was faster.

Murmur2 Algorithm Basics

There are four kinds of desirable qualities which one would seek from String Hashing algorithm:

- Quick and dirty
- Suitable for a library
- Suitable for fingerprinting

- Secure

Questions: Is Murmur2 good for library? Is it good for fingerprinting?
First define two subroutines: ShiftMix and TailBytes. Then we write the traditional word-at-time loop with a post-loop hash transformation in terms of ShiftMix and TailBytes:

```
ShiftMix(a) = a $\oplus \sigma_{47}(a)$
TailBytes(N) = $\sum_{i=1}^{N \bmod 8} 256^{(N \bmod 8)-i} B_{N-i}$
uint64 k = 14313749767032793493
int iters = $\lfloor N/8 \rfloor$
uint64 hash = seed $\oplus$ N * k
for (int i = 0; i < iters; i++)
   hash = ( hash $\oplus$ ( ShiftMix($W_i * k$) * k )) * k

if (N mod 8 > 0)
  hash = ( hash $\oplus$ ( TailBytes(N) ) * k

return ShiftMix(ShiftMix(hash) * k)
```

Murmur2 analysis:
Murmur2 strong points:

- Simple
- Fast (assuming multiplication is cheap)
- Quality is good

Questions about Murmur2:
Could its speed be better?
Could its quality be better?
The inner loop of MurmurHash2 is :

```
for (int i = 0; i < iters; i++)
  hash = ( hash $\oplus$ f($W_i$) ) * k
```

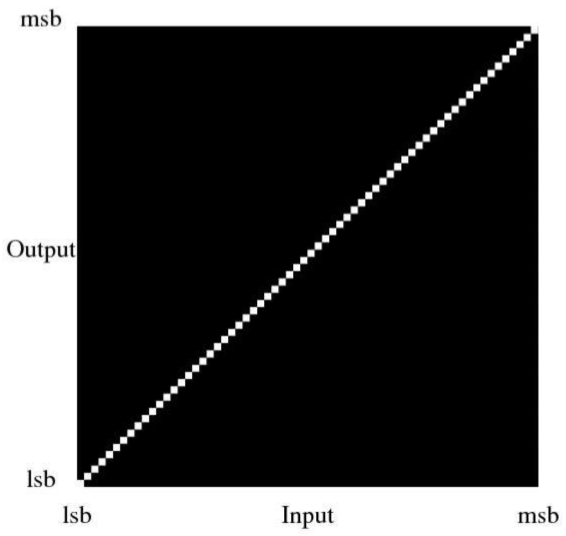
where f is "Mul-ShiftMix-Mul".
Murmur2 speed:

- ➢ Instruction-Level-Parallelism comes from parallel application of f.
- ➢ Cost of TailBytes(N) can be painful for N < 60
- ➢ f is invertible
- ➢ diffusion is not perfect

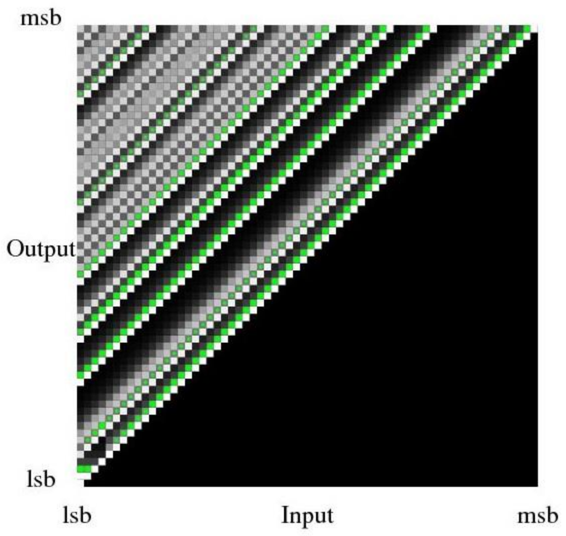
Murmur2 testing:

- ➢ Hash a bunch of words or phrases
- ➢ Hash other real world data sets
- ➢ Hash all strings with edit distance <= d from some string
- ➢ Hash other synthetic data sets
- ➢ Avalanche

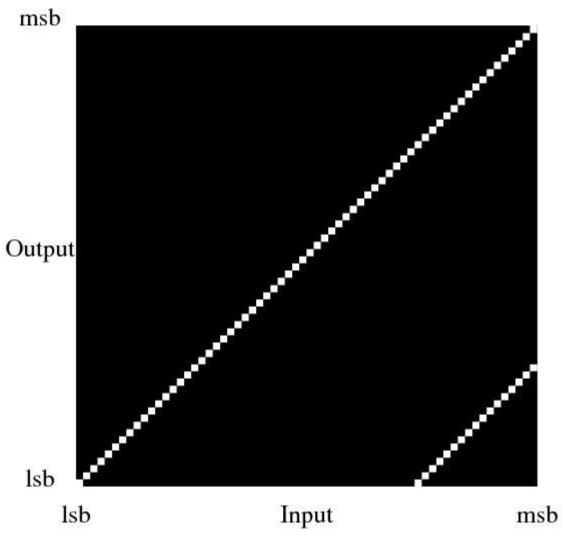
Avalanche by example:
Suppose we have a function which inputs and outputs 32 bits. Find M random input values. Hash each input value with and without its j-th bit flipped. How often do the results differ in their k-th output bit? Ideally, we want "coin flip" behavior, so the relevant distribution has mean M/2 and variance 1/4M.
Here are 6 examples of 64-bit by 64-bit avalanche diagrams for various functions.



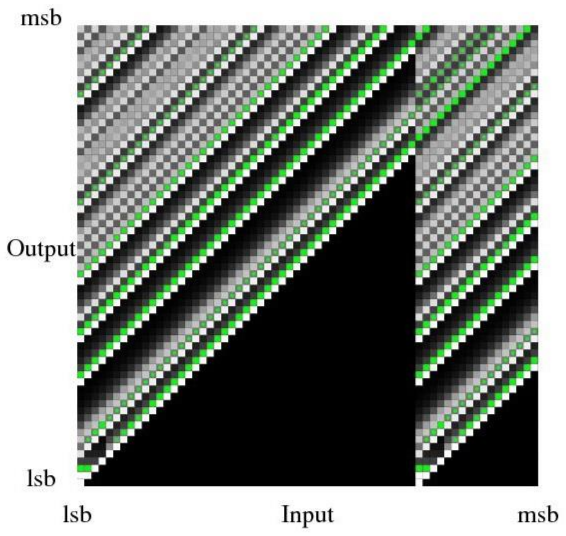
f(x) = x



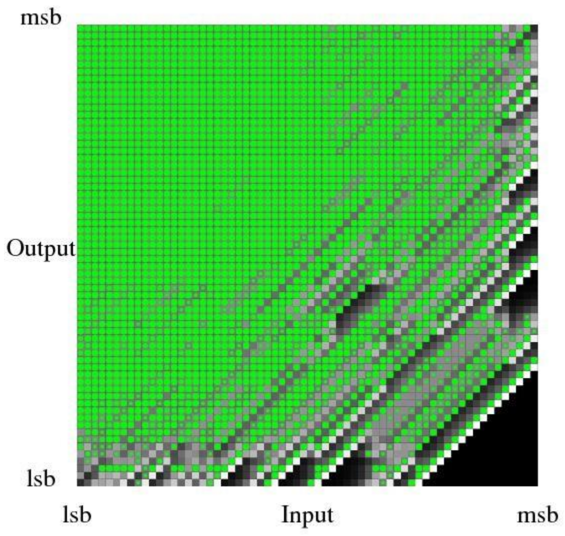
f(x) = k*x



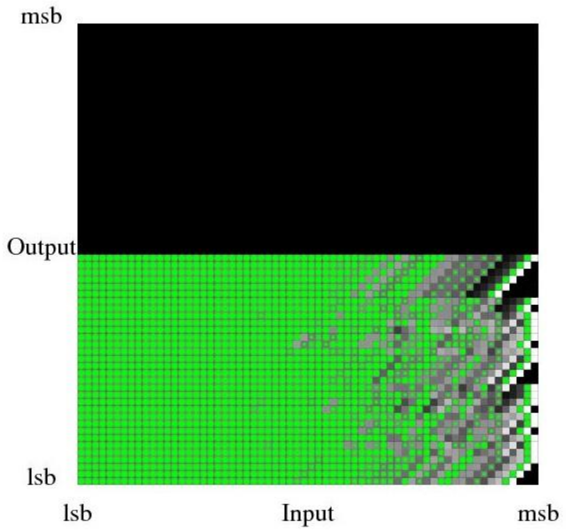
f(x) = ShiftMix(x)



f(x) = ShiftMix(x)*k



f(x) = ShiftMix(k*x)*k



f(x) = CRC(k*x)

The CityHash algorithm

Goals:

- ➢ Speed
- ➢ Quality
  - Excellent diffusion
  - Excellent behavior on all contributed test data
  - Excellent behavior on basic synthetic test data
  - Good internal state diffusion but not too good cf. Rogaway's Bucket Hashing
- ➢ Portability – for speed without total loss of portability assume
  - 64-bit registers
  - Pipelined and superscalar
  - Fairly cheap multiplication
  - Cheap +, -, $\oplus, \sigma, \rho, \beta$
  - Cheap register-to-register moves
  - a + b may be cheaper than a $\oplus$ b
  - a + c*b + 1 may be fairly cheap for $c \in \{0,1,2,4,8\}$
- ➢ Branches are expensive.
  - Is there a better way to handle tails of short strings?
  - How many dynamic branches are reasonable for hashing 12-byte inputs?
  - How many arithmetic operations?

CityHash64 initial design (2010)

- ➢ Focus on short strings
- ➢ Perhaps use Murmur2 on long strings
- ➢ Use overlapping unaligned reads
- ➢ Write the minimum number of loops: 1
- ➢ Focus on speed first; fix quality later

The CityHash64 function: overall structure

```
If (N <= 32)
    If (N <=16)
        If (N <= 8)
            …
        Else
            …
 Else if (N <= 64) {
     // Handle 33 <= N <= 64
     …
 } else {
    // Handle N > 64
    int iters = $\lfloor N/64 \rfloor$
    …
 }
```

The CityHash64 function (2012): preliminaries

Define α(u,v,m):

```
Let a = u$\oplus$v
       $a'$=ShiftMix($a * m$)
       $a''$= $a' \oplus v$
       $a'''$=ShiftMix( $a'' * m$ )
in
       $a'''$ * m
```

Also $k_0, k_1, and\ k_2$ are primes near 2^64 and K is $k_2 + 2N$

CityHash64: the case when $1 \leq N \leq 3$

```
Let $a = B_0$
    $b = B_{\lfloor N/2 \rfloor}$
    $c = B_{N-1}$
    $y = a + 256 * b$
    $z = N + 4 * c$
In
ShiftMix( $(y * k_2) \oplus (z * k_0)$ )
```

CityHash64: the case when $4 \leq N \leq 8$

$\alpha(N + 4W_0^{32}, W_{-1}^{32}, K)$

CityHash64: the case when $9 \leq N \leq 16$

```
Let $a = W_0 + k_2$
    $b = W_{-1}$
    $c = \rho_{37}(b) * K + a$
    $d = (\rho_{25}(a) + b) * K$
In
    $\alpha(c, d, K)$
```

CityHash64: the case when $17 \le N \le 32$

Let $a = W_0 * k_1$

$b = W_1$

$c = W_{-1} * K$

$d = W_{-2} * k_2$

In

$$\alpha(\rho_{43}(a+b) + \rho_{30}(c) + d, a + \rho_{18}(b + k_2) + c, K)$$

CityHash64: the case when $33 \le N \le 64$

Let $a = W_0 * k_2$

$e = W_2 * k_2$

$f = W_3 * 9$

$h = W_{-2} * K$

$u = \rho_{43}(a + W_{-1}) + 9 * (\rho_{30}(W_1) + c)$

$v = a + W_{-1} + f + 1$

$w = h + \beta\big((u+v) * K\big)$

$x = \rho_{42}(e+f) + W_{-3} + \beta(W_{-4})$

$y = \big(\beta\big((v+w) * K\big) + W_{-1}\big) * K$

$z = e + f + W_{-3}$

$r = \beta\big((x+z) * K + y\big) + W_1$

t = ShiftMix($(r+z) * K + W_{-4} + h$)

in

$tK + x$

Conclusions:

- CityHash64 is about 1.5x faster than Murmur2 for $N \le 64$
- Simplyifing it would be nice
- know your machine architecture
- do not loop over bytes
- know when to stop

CityHash64 for N > 64

The one loop in CityHash64:

- 56 bytes of state
- 64 bytes consumed per iteration
- 7 rotates, 4 multiples, 1 xor, about 36 adds
- Influenced by mix and Murmur2

Abseil CityHash Implementation

The algorithm CityHash32 is implemented in the files absl/hash/internal/city.h and absl/hash/internal/city.cc .

Here are the details: first in the header file city.h there are declared the main functions

```
namespace absl {
ABSL_NAMESPACE_BEGIN
namespace hash_internal {

typedef std::pair<uint64_t, uint64_t> uint128;

inline uint64_t Uint128Low64(const uint128 &x) { return x.first; }
inline uint64_t Uint128High64(const uint128 &x) { return x.second; }

// Hash function for a byte array.
uint64_t CityHash64(const char *s, size_t len);

// Hash function for a byte array.  For convenience, a 64-bit seed is also
// hashed into the result.
uint64_t CityHash64WithSeed(const char *s, size_t len, uint64_t seed);

// Hash function for a byte array.  For convenience, two seeds are also
// hashed into the result.
uint64_t CityHash64WithSeeds(const char *s, size_t len, uint64_t seed0,
                             uint64_t seed1);

// Hash function for a byte array.  Most useful in 32-bit binaries.
uint32_t CityHash32(const char *s, size_t len);

// Hash 128 input bits down to 64 bits of output.
// This is intended to be a reasonably good hash function.
inline uint64_t Hash128to64(const uint128 &x) {
  // Murmur-inspired hashing.
  const uint64_t kMul = 0x9ddfea08eb382d69ULL;
  uint64_t a = (Uint128Low64(x) ^ Uint128High64(x)) * kMul;
  a ^= (a >> 47);
  uint64_t b = (Uint128High64(x) ^ a) * kMul;
  b ^= (b >> 47);
  b *= kMul;
  return b;
}

}  // namespace hash_internal
ABSL_NAMESPACE_END
}  // namespace absl
```

And now the implementation file Code of city.cc shown below.
Some useful macros which will take a look before we start looking into city.cc: ABSL_IS_LITTLE_ENDIAN and ABSL_IS_BIG_ENDIAN.

```
// ABSL_IS_LITTLE_ENDIAN
// ABSL_IS_BIG_ENDIAN
//
// Checks the endianness of the platform.
```

```
//
// Notes: uses the built in endian macros provided by GCC (since 4.6) and
// Clang (since 3.2); see
// https://gcc.gnu.org/onlinedocs/cpp/Common-Predefined-Macros.html.
// Otherwise, if _WIN32, assume little endian. Otherwise, bail with an error.
#if defined(ABSL_IS_BIG_ENDIAN)
#error "ABSL_IS_BIG_ENDIAN cannot be directly set."
#endif
#if defined(ABSL_IS_LITTLE_ENDIAN)
#error "ABSL_IS_LITTLE_ENDIAN cannot be directly set."
#endif

#if (defined(__BYTE_ORDER__) && defined(__ORDER_LITTLE_ENDIAN__) && \
     __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__)
#define ABSL_IS_LITTLE_ENDIAN 1
#elif defined(__BYTE_ORDER__) && defined(__ORDER_BIG_ENDIAN__) && \
    __BYTE_ORDER__ == __ORDER_BIG_ENDIAN__
#define ABSL_IS_BIG_ENDIAN 1
#elif defined(_WIN32)
#define ABSL_IS_LITTLE_ENDIAN 1
#else
#error "absl endian detection needs to be set up for your compiler"
#endif
```

From the gnu online docs Common-Predefiend-Macros.html:
__BYTE_ORDER__
__ORDER_LITTLE_ENDIAN__
__ORDER_BIG_ENDIAN__
__ORDER_PDP_ENDIAN__
__BYTE_ORDER__ is defined to one of the values __ORDER_LITTLE_ENDIAN__, __ORDER_BIG_ENDIAN__, or __ORDER_PDP_ENDIAN__ to reflect the layout of multi-byte and multi-word quantities in memory. If __BYTE_ORDER__ is equal to __ORDER_LITTLE_ENDIAN__ or __ORDER_BIG_ENDIAN__, then multi-byte and multi-word quantities are laid out identically: the byte (word) at the lowest address is the least significant or most significant byte (word) of the quantity, respectively. If __BYTE_ORDER__ is equal to __ORDER_PDP_ENDIAN__, then bytes in 16-bit words are laid out in a little-endian fashion, whereas the 16-bit subwords of a 32-bit quantity are laid out in big-endian fashion.

Clang defines those macros as well. Here is an exceprt from Clang's InitPreprocessor.cpp (Clang v12):

```
  // Initialize target-specific preprocessor defines.
  // __BYTE_ORDER__ was added in GCC 4.6. It's analogous
  // to the macro __BYTE_ORDER (no trailing underscores)
  // from glibc's <endian.h> header.
  // We don't support the PDP-11 as a target, but include
  // the define so it can still be compared against.
  Builder.defineMacro("__ORDER_LITTLE_ENDIAN__", "1234");
  Builder.defineMacro("__ORDER_BIG_ENDIAN__",    "4321");
  Builder.defineMacro("__ORDER_PDP_ENDIAN__",    "3412");
  if (TI.isBigEndian()) {
    Builder.defineMacro("__BYTE_ORDER__", "__ORDER_BIG_ENDIAN__");
    Builder.defineMacro("__BIG_ENDIAN__");
  } else {
    Builder.defineMacro("__BYTE_ORDER__", "__ORDER_LITTLE_ENDIAN__");
    Builder.defineMacro("__LITTLE_ENDIAN__");
  }
```

You should use these macros for testing like this:

```
/* Test for a little-endian machine */
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
```

Note on gbswap_32 and gbswap_64: use the built-in compiler intrinsics __builtin_bswap64, __builtin_bswap32, __builtin_bswap16 when available. Those are available in Clang and in GCC 4.8.0 or later. Also available in the Microsoft C++ compiler but under different names - _byteswap_uint64, _byteswap_ulong, _byteswap_ushort. If no intrinsics are available but the compiler is older version of GCC and the uint64_t argument host_int is a compile time constant then use __bswap_constant_64 . __bswap_constant_64 can be found in /usr/include/x86_64-linux-gnu/bits/byteswap.h and is defined as:

```
/* Swap bytes in 64-bit value.  */
#define __bswap_constant_64(x)                \
  ((((x) & 0xff00000000000000ull) >> 56)      \
   | (((x) & 0x00ff000000000000ull) >> 40)    \
   | (((x) & 0x0000ff0000000000ull) >> 24)    \
   | (((x) & 0x000000ff00000000ull) >> 8)     \
   | (((x) & 0x00000000ff000000ull) << 8)     \
   | (((x) & 0x0000000000ff0000ull) << 24)    \
   | (((x) & 0x000000000000ff00ull) << 40)    \
   | (((x) & 0x00000000000000ffull) << 56))
```

Excerpt from base/internal/endian.h:

```
// Use compiler byte-swapping intrinsics if they are available.  32-bit
// and 64-bit versions are available in Clang and GCC as of GCC 4.3.0.
// The 16-bit version is available in Clang and GCC only as of GCC 4.8.0.
// For simplicity, we enable them all only for GCC 4.8.0 or later.
#if defined(__clang__) || \
    (defined(__GNUC__) && \
     ((__GNUC__ == 4 && __GNUC_MINOR__ >= 8) || __GNUC__ >= 5))
inline uint64_t gbswap_64(uint64_t host_int) {
  return __builtin_bswap64(host_int);
}
inline uint32_t gbswap_32(uint32_t host_int) {
  return __builtin_bswap32(host_int);
}
```

```
inline uint16_t gbswap_16(uint16_t host_int) {
  return __builtin_bswap16(host_int);
}

#elif defined(_MSC_VER)
inline uint64_t gbswap_64(uint64_t host_int) {
  return _byteswap_uint64(host_int);
}
inline uint32_t gbswap_32(uint32_t host_int) {
  return _byteswap_ulong(host_int);
}
inline uint16_t gbswap_16(uint16_t host_int) {
  return _byteswap_ushort(host_int);
}

#else // no compiler intrinsics are available
inline uint64_t gbswap_64(uint64_t host_int) {
#if defined(__GNUC__) && defined(__x86_64__) && !defined(__APPLE__)
  // Adapted from /usr/include/byteswap.h.  Not available on Mac.
  if (__builtin_constant_p(host_int)) {
    return __bswap_constant_64(host_int);
  } else {
    uint64_t result;
    __asm__("bswap %0" : "=r"(result) : "0"(host_int));
    return result;
  }
#elif defined(__GLIBC__)
  return bswap_64(host_int);
#else
  return (((host_int & uint64_t{0xFF}) << 56) |
          ((host_int & uint64_t{0xFF00}) << 40) |
          ((host_int & uint64_t{0xFF0000}) << 24) |
          ((host_int & uint64_t{0xFF000000}) << 8) |
          ((host_int & uint64_t{0xFF00000000}) >> 8) |
          ((host_int & uint64_t{0xFF0000000000}) >> 24) |
          ((host_int & uint64_t{0xFF000000000000}) >> 40) |
          ((host_int & uint64_t{0xFF00000000000000}) >> 56));
#endif  // bswap_64
}

inline uint32_t gbswap_32(uint32_t host_int) {
#if defined(__GLIBC__)
  return bswap_32(host_int);
#else
  return (((host_int & uint32_t{0xFF}) << 24) |
          ((host_int & uint32_t{0xFF00}) << 8) |
          ((host_int & uint32_t{0xFF0000}) >> 8) |
          ((host_int & uint32_t{0xFF000000}) >> 24));
#endif
}

inline uint16_t gbswap_16(uint16_t host_int) {
#if defined(__GLIBC__)
  return bswap_16(host_int);
#else
  return (((host_int & uint16_t{0xFF}) << 8) |
          ((host_int & uint16_t{0xFF00}) >> 8));
#endif
}

#endif  // intrinsics available
```

The CityHash32 and CityHash64 are shown below and follow the logic described in the earlier paragraph The CityHash algorithm.

Note on ABSL_INTERNAL_UNALIGNED_LOADXX and ABSL_INTERNAL_UNALIGNED_STOREXX which are used in city.cc. Those are defined in terms of absl::base_internal::UnalignedLoadxx and absl::base_internal::UnalignedStorexx.

```
#define ABSL_INTERNAL_UNALIGNED_LOAD16(_p) \
  (absl::base_internal::UnalignedLoad16(_p))
#define ABSL_INTERNAL_UNALIGNED_LOAD32(_p) \
  (absl::base_internal::UnalignedLoad32(_p))
#define ABSL_INTERNAL_UNALIGNED_LOAD64(_p) \
  (absl::base_internal::UnalignedLoad64(_p))

#define ABSL_INTERNAL_UNALIGNED_STORE16(_p, _val) \
  (absl::base_internal::UnalignedStore16(_p, _val))
#define ABSL_INTERNAL_UNALIGNED_STORE32(_p, _val) \
  (absl::base_internal::UnalignedStore32(_p, _val))
#define ABSL_INTERNAL_UNALIGNED_STORE64(_p, _val) \
  (absl::base_internal::UnalignedStore64(_p, _val))
```

absl::base_internal::UnalignedLoadxx and absl::base_internal::UnalignedStorexx use memcpy:

```
inline uint16_t UnalignedLoad16(const void *p) {
  uint16_t t;
  memcpy(&t, p, sizeof t);
  return t;
}

inline uint32_t UnalignedLoad32(const void *p) {
  uint32_t t;
  memcpy(&t, p, sizeof t);
  return t;
}
```

```
inline uint64_t UnalignedLoad64(const void *p) {
  uint64_t t;
  memcpy(&t, p, sizeof t);
  return t;
}

inline void UnalignedStore16(void *p, uint16_t v) { memcpy(p, &v, sizeof v); }

inline void UnalignedStore32(void *p, uint32_t v) { memcpy(p, &v, sizeof v); }

inline void UnalignedStore64(void *p, uint64_t v) { memcpy(p, &v, sizeof v); }
```

Code of city.cc:

```
namespace absl {
ABSL_NAMESPACE_BEGIN
namespace hash_internal {
```

Swap the byte order for big endian architectures. We discussed absl::gbswap_32 and absl::gbswap_64 in the previous paragraph.

```
#ifdef ABSL_IS_BIG_ENDIAN
#define uint32_in_expected_order(x) (absl::gbswap_32(x))
#define uint64_in_expected_order(x) (absl::gbswap_64(x))
#else
#define uint32_in_expected_order(x) (x)
#define uint64_in_expected_order(x) (x)
#endif


static uint64_t Fetch64(const char *p) {
  return uint64_in_expected_order(ABSL_INTERNAL_UNALIGNED_LOAD64(p));
}

static uint32_t Fetch32(const char *p) {
  return uint32_in_expected_order(ABSL_INTERNAL_UNALIGNED_LOAD32(p));
}

// Some primes between 2^63 and 2^64 for various uses.
static const uint64_t k0 = 0xc3a5c85c97cb3127ULL;
static const uint64_t k1 = 0xb492b66fbe98f273ULL;
static const uint64_t k2 = 0x9ae16a3b2f90404fULL;

// Magic numbers for 32-bit hashing.  Copied from Murmur3.
static const uint32_t c1 = 0xcc9e2d51;
static const uint32_t c2 = 0x1b873593;

// A 32-bit to 32-bit integer hash copied from Murmur3.
static uint32_t fmix(uint32_t h) {
  h ^= h >> 16;
  h *= 0x85ebca6b;
  h ^= h >> 13;
  h *= 0xc2b2ae35;
  h ^= h >> 16;
  return h;
}

static uint32_t Rotate32(uint32_t val, int shift) {
  // Avoid shifting by 32: doing so yields an undefined result.
  return shift == 0 ? val : ((val >> shift) | (val << (32 - shift)));
}

#undef PERMUTE3
#define PERMUTE3(a, b, c) \
  do {                    \
    std::swap(a, b);      \
    std::swap(a, c);      \
  } while (0)

static uint32_t Mur(uint32_t a, uint32_t h) {
  // Helper from Murmur3 for combining two 32-bit values.
  a *= c1;
  a = Rotate32(a, 17);
  a *= c2;
  h ^= a;
  h = Rotate32(h, 19);
  return h * 5 + 0xe6546b64;
}

static uint32_t Hash32Len13to24(const char *s, size_t len) {
  uint32_t a = Fetch32(s - 4 + (len >> 1));
  uint32_t b = Fetch32(s + 4);
  uint32_t c = Fetch32(s + len - 8);
  uint32_t d = Fetch32(s + (len >> 1));
  uint32_t e = Fetch32(s);
  uint32_t f = Fetch32(s + len - 4);
  uint32_t h = len;

  return fmix(Mur(f, Mur(e, Mur(d, Mur(c, Mur(b, Mur(a, h)))))));
}

static uint32_t Hash32Len0to4(const char *s, size_t len) {
  uint32_t b = 0;
  uint32_t c = 9;
  for (size_t i = 0; i < len; i++) {
    signed char v = s[i];
    b = b * c1 + v;
    c ^= b;
  }
  return fmix(Mur(b, Mur(len, c)));
}

static uint32_t Hash32Len5to12(const char *s, size_t len) {
  uint32_t a = len, b = len * 5, c = 9, d = b;
  a += Fetch32(s);
  b += Fetch32(s + len - 4);
  c += Fetch32(s + ((len >> 1) & 4));
  return fmix(Mur(c, Mur(b, Mur(a, d))));
}

uint32_t CityHash32(const char *s, size_t len) {
  if (len <= 24) {
    return len <= 12
               ? (len <= 4 ? Hash32Len0to4(s, len) : Hash32Len5to12(s, len))
               : Hash32Len13to24(s, len);
  }

  // len > 24
  uint32_t h = len, g = c1 * len, f = g;

  uint32_t a0 = Rotate32(Fetch32(s + len - 4) * c1, 17) * c2;
  uint32_t a1 = Rotate32(Fetch32(s + len - 8) * c1, 17) * c2;
  uint32_t a2 = Rotate32(Fetch32(s + len - 16) * c1, 17) * c2;
  uint32_t a3 = Rotate32(Fetch32(s + len - 12) * c1, 17) * c2;
  uint32_t a4 = Rotate32(Fetch32(s + len - 20) * c1, 17) * c2;
  h ^= a0;
  h = Rotate32(h, 19);
  h = h * 5 + 0xe6546b64;
  h ^= a2;
  h = Rotate32(h, 19);
  h = h * 5 + 0xe6546b64;
  g ^= a1;
  g = Rotate32(g, 19);
  g = g * 5 + 0xe6546b64;
  g ^= a3;
  g = Rotate32(g, 19);
  g = g * 5 + 0xe6546b64;
  f += a4;
  f = Rotate32(f, 19);
  f = f * 5 + 0xe6546b64;
  size_t iters = (len - 1) / 20;
  do {
    uint32_t b0 = Rotate32(Fetch32(s) * c1, 17) * c2;
    uint32_t b1 = Fetch32(s + 4);
    uint32_t b2 = Rotate32(Fetch32(s + 8) * c1, 17) * c2;
    uint32_t b3 = Rotate32(Fetch32(s + 12) * c1, 17) * c2;
    uint32_t b4 = Fetch32(s + 16);
    h ^= b0;
    h = Rotate32(h, 18);
    h = h * 5 + 0xe6546b64;
    f += b1;
    f = Rotate32(f, 19);
    f = f * c1;
    g += b2;
    g = Rotate32(g, 18);
    g = g * 5 + 0xe6546b64;
    h ^= b3 + b1;
    h = Rotate32(h, 19);
    h = h * 5 + 0xe6546b64;
    g ^= b4;
    g = absl::gbswap_32(g) * 5;
    h += b4 * 5;
    h = absl::gbswap_32(h);
    f += b0;
    PERMUTE3(f, h, g);
    s += 20;
  } while (--iters != 0);
  g = Rotate32(g, 11) * c1;
  g = Rotate32(g, 17) * c1;
  f = Rotate32(f, 11) * c1;
  f = Rotate32(f, 17) * c1;
  h = Rotate32(h + g, 19);
  h = h * 5 + 0xe6546b64;
  h = Rotate32(h, 17) * c1;
  h = Rotate32(h + f, 19);
  h = h * 5 + 0xe6546b64;
  h = Rotate32(h, 17) * c1;
  return h;
}

// Bitwise right rotate.  Normally this will compile to a single
// instruction, especially if the shift is a manifest constant.
static uint64_t Rotate(uint64_t val, int shift) {
  // Avoid shifting by 64: doing so yields an undefined result.
  return shift == 0 ? val : ((val >> shift) | (val << (64 - shift)));
}

static uint64_t ShiftMix(uint64_t val) { return val ^ (val >> 47); }

static uint64_t HashLen16(uint64_t u, uint64_t v) {
  return Hash128to64(uint128(u, v));
}
```

```
static uint64_t HashLen16(uint64_t u, uint64_t v, uint64_t mul) {
  // Murmur-inspired hashing.
  uint64_t a = (u ^ v) * mul;
  a ^= (a >> 47);
  uint64_t b = (v ^ a) * mul;
  b ^= (b >> 47);
  b *= mul;
  return b;
}

static uint64_t HashLen0to16(const char *s, size_t len) {
  if (len >= 8) {
    uint64_t mul = k2 + len * 2;
    uint64_t a = Fetch64(s) + k2;
    uint64_t b = Fetch64(s + len - 8);
    uint64_t c = Rotate(b, 37) * mul + a;
    uint64_t d = (Rotate(a, 25) + b) * mul;
    return HashLen16(c, d, mul);
  }
  if (len >= 4) {
    uint64_t mul = k2 + len * 2;
    uint64_t a = Fetch32(s);
    return HashLen16(len + (a << 3), Fetch32(s + len - 4), mul);
  }
  if (len > 0) {
    uint8_t a = s[0];
    uint8_t b = s[len >> 1];
    uint8_t c = s[len - 1];
    uint32_t y = static_cast<uint32_t>(a) + (static_cast<uint32_t>(b) << 8);
    uint32_t z = len + (static_cast<uint32_t>(c) << 2);
    return ShiftMix(y * k2 ^ z * k0) * k2;
  }
  return k2;
}

// This probably works well for 16-byte strings as well, but it may be overkill
// in that case.
static uint64_t HashLen17to32(const char *s, size_t len) {
  uint64_t mul = k2 + len * 2;
  uint64_t a = Fetch64(s) * k1;
  uint64_t b = Fetch64(s + 8);
  uint64_t c = Fetch64(s + len - 8) * mul;
  uint64_t d = Fetch64(s + len - 16) * k2;
  return HashLen16(Rotate(a + b, 43) + Rotate(c, 30) + d,
                   a + Rotate(b + k2, 18) + c, mul);
}

// Return a 16-byte hash for 48 bytes.  Quick and dirty.
// Callers do best to use "random-looking" values for a and b.
static std::pair<uint64_t, uint64_t> WeakHashLen32WithSeeds(uint64_t w, uint64_t x,
                                          uint64_t y, uint64_t z,
                                          uint64_t a, uint64_t b) {
  a += w;
  b = Rotate(b + a + z, 21);
  uint64_t c = a;
  a += x;
  a += y;
  b += Rotate(a, 44);
  return std::make_pair(a + z, b + c);
}

// Return a 16-byte hash for s[0] ... s[31], a, and b.  Quick and dirty.
static std::pair<uint64_t, uint64_t> WeakHashLen32WithSeeds(const char *s, uint64_t a,
                                          uint64_t b) {
  return WeakHashLen32WithSeeds(Fetch64(s), Fetch64(s + 8), Fetch64(s + 16),
                        Fetch64(s + 24), a, b);
}

// Return an 8-byte hash for 33 to 64 bytes.
static uint64_t HashLen33to64(const char *s, size_t len) {
  uint64_t mul = k2 + len * 2;
  uint64_t a = Fetch64(s) * k2;
  uint64_t b = Fetch64(s + 8);
  uint64_t c = Fetch64(s + len - 24);
  uint64_t d = Fetch64(s + len - 32);
  uint64_t e = Fetch64(s + 16) * k2;
  uint64_t f = Fetch64(s + 24) * 9;
  uint64_t g = Fetch64(s + len - 8);
  uint64_t h = Fetch64(s + len - 16) * mul;
  uint64_t u = Rotate(a + g, 43) + (Rotate(b, 30) + c) * 9;
  uint64_t v = ((a + g) ^ d) + f + 1;
  uint64_t w = absl::gbswap_64((u + v) * mul) + h;
  uint64_t x = Rotate(e + f, 42) + c;
  uint64_t y = (absl::gbswap_64((v + w) * mul) + g) * mul;
  uint64_t z = e + f + c;
  a = absl::gbswap_64((x + z) * mul + y) + b;
  b = ShiftMix((z + a) * mul + d + h) * mul;
  return b + x;
}

uint64_t CityHash64(const char *s, size_t len) {
  if (len <= 32) {
    if (len <= 16) {
      return HashLen0to16(s, len);
    } else {
      return HashLen17to32(s, len);
    }
  } else if (len <= 64) {
    return HashLen33to64(s, len);
  }

  // For strings over 64 bytes we hash the end first, and then as we
  // loop we keep 56 bytes of state: v, w, x, y, and z.
  uint64_t x = Fetch64(s + len - 40);
  uint64_t y = Fetch64(s + len - 16) + Fetch64(s + len - 56);
  uint64_t z = HashLen16(Fetch64(s + len - 48) + len, Fetch64(s + len - 24));
  std::pair<uint64_t, uint64_t> v = WeakHashLen32WithSeeds(s + len - 64, len, z);
  std::pair<uint64_t, uint64_t> w = WeakHashLen32WithSeeds(s + len - 32, y + k1, x);
  x = x * k1 + Fetch64(s);

  // Decrease len to the nearest multiple of 64, and operate on 64-byte chunks.
  len = (len - 1) & ~static_cast<size_t>(63);
  do {
    x = Rotate(x + y + v.first + Fetch64(s + 8), 37) * k1;
    y = Rotate(y + v.second + Fetch64(s + 48), 42) * k1;
    x ^= w.second;
    y += v.first + Fetch64(s + 40);
    z = Rotate(z + w.first, 33) * k1;
    v = WeakHashLen32WithSeeds(s, v.second * k1, x + w.first);
    w = WeakHashLen32WithSeeds(s + 32, z + w.second, y + Fetch64(s + 16));
    std::swap(z, x);
    s += 64;
    len -= 64;
  } while (len != 0);
  return HashLen16(HashLen16(v.first, w.first) + ShiftMix(y) * k1 + z,
                   HashLen16(v.second, w.second) + x);
}

uint64_t CityHash64WithSeed(const char *s, size_t len, uint64_t seed) {
  return CityHash64WithSeeds(s, len, k2, seed);
}

uint64_t CityHash64WithSeeds(const char *s, size_t len, uint64_t seed0,
                             uint64_t seed1) {
  return HashLen16(CityHash64(s, len) - seed0, seed1);
}

}  // namespace hash_internal
ABSL_NAMESPACE_END
}  // namespace absl
```

Take a look into the Google's CityHash repo:
https://github.com/google/cityhash

Understanding Murmur3 Hasher

// TODO: finish this section

Austin Appleby's Murmur3 github repo: https://github.com/aappleby/smhasher/wiki/MurmurHash3
MurmurHash3 is a successor of MurmurHash2. It comes in 3 variants – a 32-bit version that targets low latency for hash table use, and two 128-bit versions for generating unique identifiers for large blocks of data, one each for x86 and x64 platforms.

Understanding FarmHash

// TODO: finish this section

Bibliography

1. *How to write shared libraries*, Ulrich Drepper, drepper@gmail.com, Dec 10, 2011
2. *Semantic Versioning v2.0.0* https://semver.org/spec/v2.0.0.html