Understanding Tensorflow 2 source code

D. Gueorguiev 4/27/20

Contents

[Preliminaries 2](#_Toc44883108)

[Tensorflow globals and macros 3](#_Toc44883109)

[TF\_PREDICT\_TRUE and TF\_PREDICT\_FALSE 3](#_Toc44883110)

[TF\_DISALLOW\_COPY\_AND\_ASSIGN 3](#_Toc44883111)

[TF\_ARRAYSIZE 3](#_Toc44883112)

[TF\_FALLTHROUGH\_INTENDED 4](#_Toc44883113)

[Macros utilizing compiler attributes 4](#_Toc44883114)

[Tensorflow Logging Internals 6](#_Toc44883115)

[Class LogMessage 6](#_Toc44883116)

[Minimum Log and Vlog level from the environment 11](#_Toc44883117)

[The LOG macro 12](#_Toc44883118)

[The VLOG macro 13](#_Toc44883119)

[Helper classes requiring synchronization – LogEveryNState, LogFirstNState, LogEveryPow2State, LogEveryNSecState 14](#_Toc44883120)

[The helper macro LOGGING\_INTERNAL\_STATEFUL\_CONDITION and the LOG\_EVERY\_### macros 16](#_Toc44883121)

[The CHECK\_X macros and related internals 19](#_Toc44883122)

[Tensorflow Internal structures, containers and interfaces 26](#_Toc44883123)

[Internal Data Structures and Synchronization Primitives 26](#_Toc44883124)

[Internal helper structs FlatMap::Bucket and FlatSet::Bucket 26](#_Toc44883125)

[Class gtl::internal::FlatRep - the internal representation for FlatMap and FlatSet 28](#_Toc44883126)

[Class gtl::FlatMap 42](#_Toc44883127)

[Class gtl::FlatSet 50](#_Toc44883128)

[Class CompactPointerSet<T> 56](#_Toc44883129)

[The Abseil Hashing Framework 60](#_Toc44883130)

[Class ArraySlice 88](#_Toc44883131)

[Class InlinedVector 110](#_Toc44883132)

[Class IntType 149](#_Toc44883133)

[Class iterator\_range 156](#_Toc44883134)

[map\_util and map\_traits headers 157](#_Toc44883135)

[C++ External and Internal API 163](#_Toc44883136)

[TF\_CAPI\_EXPORT directive 164](#_Toc44883137)

[TF\_VERSION string 164](#_Toc44883138)

[TF\_Buffer struct and functionality for manipulating it 165](#_Toc44883139)

[Global functions for manipulation of TF\_SessionOptions 166](#_Toc44883140)

[New graph construction API (under construction) 168](#_Toc44883141)

[Classes Graph and GraphDef 170](#_Toc44883142)

[Protobuf interfaces and formats 179](#_Toc44883143)

[Core/Protobuf/Config.proto 179](#_Toc44883144)

[Exploring the C++ code examples 180](#_Toc44883145)

[Exploring //tensorflow/core/example 180](#_Toc44883146)

[Proto file for ***Feature*** 180](#_Toc44883147)

[Proto file for ***Example***: 182](#_Toc44883148)

[Building the tensorflow libraries and examples using Bazel 183](#_Toc44883149)

[Building tensorflow libraries 183](#_Toc44883150)

[Building tensorflow core components 184](#_Toc44883151)

[Building Example Code 184](#_Toc44883152)

[Build Folder structure 184](#_Toc44883153)

[Build first C++ Tensorflow app 186](#_Toc44883154)

[Third Party and External Packages 186](#_Toc44883155)

[**BoringSSL:** https://boringssl.googlesource.com/boringssl/ 186](#_Toc44883156)

[**FarmHash:** https://github.com/google/farmhash 186](#_Toc44883157)

[*Introducing FarmHash* 186](#_Toc44883158)

[**HighwayHash**: https://github.com/google/highwayhash 187](#_Toc44883159)

[Appendix A: Bazel tutorial for Tensorflow Builds 187](#_Toc44883160)

[Appendix B: Hashing in Tensorflow 187](#_Toc44883161)

[Bibliography 190](#_Toc44883162)

# 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](https://github.com/dimitarpg13/tensorflow/blob/master/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 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 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](https://en.cppreference.com/w/cpp/numeric/fenv/feexceptflag). Refer to 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))](https://en.cppreference.com/w/cpp/language/attributes/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))](https://gcc.gnu.org/onlinedocs/gcc/Inline.html) 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))](https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Function-Attributes.html) 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))](https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Function-Attributes.html) 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))](https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Function-Attributes.html). 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))](https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Function-Attributes.html) 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](https://github.com/dimitarpg13/tensorflow/blob/master/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 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 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 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 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 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 Function ParseInteger – it uses [std::istringstream](http://www.cplusplus.com/reference/sstream/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 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 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](https://llvm.org/docs/LibFuzzer.html). For automated Fuzz testing see the following wiki page – [Fuzzing](https://en.wikipedia.org/wiki/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 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 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](https://github.com/abseil/abseil-cpp) 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 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 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 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 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 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());

}

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 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 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)

# 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 FlatMap::Bucket and 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](https://github.com/dimitarpg13/tensorflow/blob/master/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](https://en.wikipedia.org/wiki/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 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](https://c9x.me/x86/html/file_module_x86_id_252.html) 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](https://github.com/dimitarpg13/tensorflow/blob/master/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;

}

};

### 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.

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 classic example of type-erased Callback struct shown below. In it it is used a helper WrappingCallback 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);

}

Let us see how the type-erasure is implemented in HashState-

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;

}

. . .

}

The contents of [absl/hash/hash.h](https://github.com/abseil/abseil-cpp/blob/20200225.2/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](https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern) (***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>{};

The contents of [absl/hash/internal/hash.h](https://github.com/abseil/abseil-cpp/blob/20200225.2/absl/hash/internal/hash.h) :

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

### 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](https://github.com/dimitarpg13/abseil-cpp/blob/lts_2020_02_25/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](https://en.cppreference.com/w/cpp/language/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](https://stackoverflow.com/questions/48014146/abseil-c-template-argument/48014251#48014251?newreg=8b4a6ea3be5e4a05af4f1a4509a8c1a7) 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](https://github.com/abseil/abseil-cpp/blob/20200225.2/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](https://github.com/abseil/abseil-cpp/blob/20200225.2/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](https://github.com/dimitarpg13/tensorflow/blob/master/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](https://gcc.gnu.org/wiki/Visibility). 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](#UlrichDrepperSharedLibrary) about writing shared libraries by Ulrich Drepper can be found [here](https://akkadia.org/drepper/dsohowto.pdf).

### 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](https://semver.org/spec/v2.0.0.html)) 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](https://github.com/dimitarpg13/tensorflow/blob/master/tensorflow/core/public/session_options.h#L28-L61)

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](https://github.com/dimitarpg13/tensorflow/blob/master/tensorflow/core/graph/graph.h#L473-L761) 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](https://github.com/dimitarpg13/tensorflow/blob/master/tensorflow/core/protobuf/config.proto#L18-L194)).

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](https://github.com/dimitarpg13/tensorflow/blob/master/tensorflow/core/protobuf/config.proto#L100-L188)).

1. Optimizer tuning parameters (see message [OptimizerOptions](https://github.com/dimitarpg13/tensorflow/blob/master/tensorflow/core/protobuf/config.proto#L197-L242)). 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.
2. Various graph options (see message [GraphOptions](https://github.com/dimitarpg13/tensorflow/blob/master/tensorflow/core/protobuf/config.proto#L244-L289)). 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
3. 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 :



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

![A screenshot of a cell phone

Description automatically generated]()

The location of eigen source: ROOT\_FOLDER/bazel-tensorflow/external/eigen\_archive

![A picture containing monitor

Description automatically generated]()

The location of grpc source: ROOT\_FOLDER/bazel-tensorflow/external/com\_github\_grpc\_grpc

![A screen shot of a monitor

Description automatically generated]()

The location of absl source: ROOT\_FOLDER/bazel-tensorflow/external/com\_google\_absl



The location of zlib source: ROOT\_FOLDER/bazel-tensorflow/external/zlib

![A close up of a screen

Description automatically generated]()

# 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.

1. Build

# Third Party and External Packages

## **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*](http://code.google.com/p/farmhash/)*family of hash functions for strings.  FarmHash is a successor to*[*CityHash*](http://code.google.com/p/cityhash/)*, and includes many of the same tricks and techniques, several of them taken from Austin Appleby’s*[*MurmurHash*](https://code.google.com/p/smhasher/)*.  
  
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

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>

# Bibliography

1. *How to write shared libraries*, Ulrich Drepper, [drepper@gmail.com](mailto:drepper@gmail.com), Dec 10, 2011
2. *Semantic Versioning v2.0.0* <https://semver.org/spec/v2.0.0.html>