Developer Guide

Welcome to the developer documentation for protocol buffers – a language-neutral, platform-neutral, extensible way of serializing structured data for use in communications protocols, data storage, and more.

This documentation is aimed at Java, C++, or Python developers who want to use protocol buffers in their applications. This overview introduces protocol buffers and tells you what you need to do to get started – you can then go on to follow the tutorials or delve deeper into protocol buffer encoding. API reference documentation is also provided for all three languages, as well as language and style guides for writing `.proto` files.

What are protocol buffers?

Protocol buffers are a flexible, efficient, automated mechanism for serializing structured data – think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages. You can even update your data structure without breaking deployed programs that are compiled against the "old" format.

How do they work?

You specify how you want the information you're serializing to be structured by defining protocol buffer message types in `.proto` files. Each protocol buffer message is a small logical record of information, containing a series of name-value pairs. Here's a very basic example of a `.proto` file that defines a message containing information about a person:

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}
```

As you can see, the message format is simple – each message type has one or more uniquely numbered fields, and each field has a name and a value type, where value types can be numbers (integer or floating-point), booleans, strings, raw bytes, or even (as in the example above) other protocol buffer message types, allowing you to structure your data hierarchically. You can specify optional fields, required fields, and repeated fields. You can find more information about writing `.proto` files in the Protocol Buffer Language Guide.

Once you've defined your messages, you run the protocol buffer compiler for your application's language on your `.proto` file to generate data access classes. These provide simple accessors for each field (like `query()` and `set_query()`) as well as methods to serialize/parse the whole structure to/from raw bytes – so, for instance, if your chosen language is C++, running the compiler on the above example will generate a class called `Person`. You can then use this class in your application to populate, serialize, and retrieve `Person` protocol buffer messages. You might then write some code like this:

```
Person person;
person.set_name("John Doe");
person.set_id(1234);
person.set_email("jdoe@example.com");
fstream output("myfile", ios::out | ios::binary);
person.SerializeToOstream(&output);
```

Then, later on, you could read your message back in:

```
fstream input("myfile", ios::in | ios::binary);
Person person;
person.ParseFromIstream(&input);
cout << "Name: " << person.name() << endl;
cout << "E-mail: " << person.email() << endl;
```

You can add new fields to your message formats without breaking backwards-compatibility; old binaries simply ignore the new field when parsing. So if you have a communications protocol that uses protocol buffers as its data format, you can extend your protocol without having to worry about breaking existing code.

You'll find a complete reference for using generated protocol buffer code in the API Reference section, and you can find out more about how protocol buffer messages are encoded in Protocol Buffer Encoding.

Why not just use XML?

Protocol buffers have many advantages over XML for serializing structured data. Protocol buffers:

- are simpler
- are 3 to 10 times smaller
- are 20 to 100 times faster
- are less ambiguous
- generate data access classes that are easier to use programmatically

For example, let's say you want to model a `person` with a `name` and an `email`. In XML, you need to do:

```
  <person>
    <name>John Doe</name>
    <email>jdoe@example.com</email>
  </person>
```

while the corresponding protocol buffer message (in protocol buffer text format) is:

```
# Textual representation of a protocol buffer.
# This is *not* the binary format used on the wire.
person {
  name: "John Doe"
  email: "jdoe@example.com"
}
```

When this message is encoded to the protocol buffer binary format (the text format above is just a convenient human-readable representation for debugging and editing), it would probably be 28 bytes long and take around 100-200 nanoseconds to parse. The XML version is at least 69 bytes if you remove whitespace, and would take around 5,000-10,000 nanoseconds to parse.

Also, manipulating a protocol buffer is much easier:

```
 cout << "Name: " << person.name() << endl;
  cout << "E-mail: " << person.email() << endl;
```

Whereas with XML you would have to do something like:

```
cout << "Name: "
     << person.getElementsByTagName("name")->item(0)->innerText()
     << endl;
cout << "E-mail: "
     << person.getElementsByTagName("email")->item(0)->innerText()
     << endl;
```

However, protocol buffers are not always a better solution than XML – for instance, protocol buffers would not be a good way to model a text-based document with markup (e.g. HTML), since you cannot easily interleave structure with text. In addition, XML is human-readable and human-editable; protocol buffers, at least in their native format, are not. XML is also – to some extent – self-describing. A protocol buffer is only meaningful if you have the message definition (the `.proto` file).

Sounds like the solution for me! How do I get started?

Download the package – this contains the complete source code for the Java, Python, and C++ protocol buffer compilers, as well as the classes you need for I/O and testing. To build and install your compiler, follow the instructions in the README.

Once you're all set, try following the tutorial for your chosen language – this will step you through creating a simple application that uses protocol buffers.

A bit of history

Protocol buffers were initially developed at Google to deal with an index server request/response protocol. Prior to protocol buffers, there was a format for requests and responses that used hand marshalling/unmarshalling of requests and responses, and that supported a number of versions of the protocol. This resulted in some very ugly code, like:

```
if (version == 3) {
   ...
 } else if (version > 4) {
   if (version == 5) {
     ...
   }
   ...
 }
```

Explicitly formatted protocols also complicated the rollout of new protocol versions, because developers had to make sure that all servers between the originator of the request and the actual server handling the request understood the new protocol before they could flip a switch to start using the new protocol.

Protocol buffers were designed to solve many of these problems:

- New fields could be easily introduced, and intermediate servers that didn't need to inspect the data could simply parse it and pass through the data without needing to know about all the fields.
- Formats were more self-describing, and could be dealt with from a variety of languages (C++, Java, etc.)

However, users still needed to hand-write their own parsing code.

As the system evolved, it acquired a number of other features and uses:

- Automatically-generated serialization and deserialization code avoided the need for hand parsing.
- In addition to being used for short-lived RPC (Remote Procedure Call) requests, people started to use protocol buffers as a handy self-describing format for storing data persistently (for example, in Bigtable).
- Server RPC interfaces started to be declared as part of protocol files, with the protocol compiler generating stub classes that users could override with actual implementations of the server's interface.

Protocol buffers are now Google's *lingua franca* for data – at time of writing, there are 48,162 different message types defined in the Google code tree across 12,183 `.proto` files. They're used both in RPC systems and for persistent storage of data in a variety of storage systems.

Language Guide

This guide describes how to use the protocol buffer language to structure your protocol buffer data, including `proto` file syntax and how to generate data access classes from your `.proto` files.

This is a reference guide – for a step by step example that uses many of the features described in this document, see the tutorial for your chosen language.

Defining A Message Type

First let's look at a very simple example. Let's say you want to define a search request message format, where each search request has a query string, the particular page of results you are interested in, and a number of results per page. Here's the `.proto` file you use to define the message type.

```
message SearchRequest {
  required string query = 1;
  optional int32 page_number = 2;
  optional int32 result_per_page = 3;
}
```

The `SearchRequest` message definition specifies three fields (name/value pairs), one for each piece of data that you want to include in this type of message. Each field has a name and a type.

Specifying Field Types

In the above example, all the fields are scalar types: two integers (`page_number` and `result_per_page`) and a string (`query`). However, you can also specify composite types for your fields, including enumerations and other message types.

Assigning Tags

As you can see, each field in the message definition has a **unique numbered tag**. These tags are used to identify your fields in the message binary format, and should not be changed once your message type is in use. Note that tags with values in the range 1 through 15 take one byte to encode. Tags in the range 16 through 2047 take two bytes. So you should reserve the tags 1 through 15 for very frequently occurring message elements. Remember to leave some room for frequently occurring elements that might be added in the future.

The smallest tag number you can specify is 1, and the largest is 2^{29} - 1, or 536,870,911. You also cannot use the numbers 19000 though 19999 (`FieldDescriptor::kFirstReservedNumber` through `FieldDescriptor::kLastReservedNumber`), as they are reserved for the Protocol Buffers implementation -the protocol buffer compiler will complain if you use one of these reserved numbers in your `.proto`.

Specifying Field Rules

You specify that message fields are one of the following:

- `required`: a well-formed message must have exactly one of this field.
- `optional`: a well-formed message can have zero or one of this field (but not more than one).
- `repeated`: this field can be repeated any number of times (including zero) in a well-formed message. The order of the repeated values will be preserved.

Required Is Forever You should be very careful about marking fields as `required`. If at some point you wish to stop writing or sending a required field, it will be problematic to change the field to an optional field – old readers will consider messages without this field to be incomplete and may reject or drop them unintentionally. You should consider writing application-specific custom validation routines for your buffers instead. Some engineers at Google have come to the conclusion that using `required` does more harm than good; they prefer to use only `optional` and `repeated`. However, this view is not universal.

Adding More Message Types

Multiple message types can be defined in a single `.proto` file. This is useful if you are defining multiple related messages – so, for example, if you wanted to define the reply message format that corresponds to your `SearchResponse` message type, you could add it to the same `.proto`:

```
message SearchRequest {
  required string query = 1;
  optional int32 page_number = 2;
  optional int32 result_per_page = 3;
}

message SearchResponse {
 ...
}
```

Adding Comments

To add comments to your `.proto` files, use C/C++-style `//` syntax.

```
message SearchRequest {
  required string query = 1;
  optional int32 page_number = 2;// Which page number do we want?
  optional int32 result_per_page = 3;// Number of results to return per page.
}
```

What's Generated From Your `.proto`?

When you run the protocol buffer compiler on a `.proto`, the compiler generates the code in your chosen language you'll need to work with the message types you've described in the file, including getting and setting field values, serializing your messages to an output stream, and parsing your messages from an input stream.

For **C++**, the compiler generates a `.h` and `.cc` file from each `.proto`, with a class for each message type described in your file.

For **Java**, the compiler generates a `.java` file with a class for each message type, as well as a special `Builder` classes for creating message class instances.

Python is a little different – the Python compiler generates a module with a static descriptor of each message type in your `.proto`, which is then used with *ametaclass* to create the necessary Python data access class at runtime.

You can find out more about using the APIs for each language by following the tutorial for your chosen language. For even more API details, see the relevant API reference.

Scalar Value Types

A scalar message field can have one of the following types– the table shows the type specified in the `.proto` file, and the corresponding type in the automatically generated class:

.proto Type	Notes	C++ Type	Java Type
double		double	double
float		float	float
int32	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.	int32	int
int64	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.	int64	long
uint32	Uses variable-length encoding.	uint32	int
uint64	Uses variable-length encoding.	uint64	long
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.	int32	int
sint64	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than	int64	long

	regular int64s.		
fixed32	Always four bytes. More efficient than uint32 if values are often greater than 2^{28}.	uint32	int
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than 2^{56}.	uint64	long
sfixed32	Always four bytes.	int32	int
sfixed64	Always eight bytes.	int64	long
bool		bool	boolean
string	A string must always contain UTF-8 encoded or 7-bit ASCII text.	string	String
bytes	May contain any arbitrary sequence of bytes.	string	ByteString

You can find out more about how these types are encoded when you serialize your message in Protocol Buffer Encoding.

Optional Fields And Default Values

As mentioned above, elements in a message description can be labeled `optional`. A well-formed message may or may not contain an optional element. When a message is parsed, if it does not contain an optional element, the corresponding field in the parsed object is set to the default value for that field. The default value can be specified as part of the message description. For example, let's say you want to provide a default value of 10 for a `SearchRequest`'s `result_per_page` value.

```
optional int32 result_per_page = 3 [default = 10];
```

If the default value is not specified for an optional element, a type-specific default value is used instead: for strings, the default value is the empty string. For bools, the default value is false. For numeric types, the default value is zero.

Enumerations

When you're defining a message type, you might want one of its fields to only have one of a pre-defined list of values. For example, let's say you want to add a `corpus` field for each `SearchRequest`, where the corpus can be `UNIVERSAL`, `WEB`, `IMAGES`, `LOCAL`, `NEWS`, `PRODUCTS` or `VIDEO`. You can do this very simply by adding an `enum` to your message definition - a field with an `enum` type can only have one of a specified set of constants as its value (if you try to provide a different value, the parser will treat it like an unknown field). In the following example we've added an `enum` called `Corpus` with all the possible values, and a field of type `Corpus`:

```
message SearchRequest {
```

```
  required string query = 1;
  optional int32 page_number = 2;
  optional int32 result_per_page = 3 [default = 10];
  enum Corpus {
    UNIVERSAL = 0;
    WEB = 1;
    IMAGES = 2;
    LOCAL = 3;
    NEWS = 4;
    PRODUCTS = 5;
    VIDEO = 6;
  }
  optional Corpus corpus = 4 [default = UNIVERSAL];
}
```

Enumerator constants must be in the range [0, 2147483647]. You can define `enum`s within a message definition, as in the above example, or outside – these `enum`s can be reused in any message definition in your `.proto` file. You can also use an `enum` type declared in one message as the type of a field in a different message, using the syntax *`MessageType.EnumType`*.

When you run the protocol buffer compiler on a `.proto` that uses an `enum`, the generated code will have a corresponding `enum` for Java or C++, or a special `EnumDescriptor` class for Python that's used to create a set of symbolic constants with integer values in the runtime-generated class.

For more information about how to work with message `enum`s in your applications, see the generated code guide for your chosen language.

Using Other Message Types

You can use other message types as field types. For example, let's say you wanted to include `Result` messages in each `SearchResponse` message – to do this, you can define a `Result` message type in the same `.proto` and then specify a field of type `Result` in `SearchResponse`:

```
message SearchResponse {
  repeated Result result = 1;
}

message Result {
  required string url = 1;
  optional string title = 2;
  repeated string snippets = 3;
}
```

Importing Definitions

In the above example, the `Result` message type is defined in the same file as `SearchResponse` – what if the message type you want to use as a field type is already defined in another `.proto` file?

You can use definitions from other `.proto` files by *importing* them. To import another `.proto`'s definitions, you add an import statement to the top of your file:

```
import "myproject/other_protos.proto";
```

The protocol compiler searches for imported files in a set of directories specified on the protocol compiler command line using the `-I`/`--import_path` flag. If no flag was given, it looks in the directory in which the compiler was invoked.

Nested Types

You can define and use message types inside other message types, as in the following example – here the `Result` message is defined inside the `SearchResponse` message:

```
message SearchResponse {
  message Result {
    required string url = 1;
    optional string title = 2;
    repeated string snippets = 3;
  }
  repeated Result result = 1;
}
```

If you want to reuse this message type outside its parent message type, you refer to it as *`Parent.Type`*:

```
message SomeOtherMessage {
  optional SearchResponse.Result result = 1;
}
```

You can nest messages as deeply as you like:

```
message Outer {                  // Level 0
  message MiddleAA {  // Level 1
    message Inner {   // Level 2
      required int64 ival = 1;
      optional bool  booly = 2;
    }
  }
  message MiddleBB = {  // Level 1
    message Inner = {   // Level 2
      required int32 ival = 1;
      optional bool  booly = 2;
    }
  }
}
```

Groups

Note that this feature is deprecated and should not be used when creating new message types – use nested message types instead.

Groups are another way to nest information in your message definitions. For example, another way to specify a `SearchResponse` containing a number of `Results` is as follows:

```
message SearchResponse {
  repeated group Result = 1 {
    required string url = 2;
    optional string title = 3;
    repeated string snippets = 4;
  }
}
```

A group simply combines a nested message type and a field into a single declaration. In your code, you can treat this message just as if it had a `Result` type field called `result` (the latter name is converted to lower-case so that it does not conflict with the former). Therefore, this example is exactly equivalent to the `SearchResponse` above, except that the message has a different wire format.

Updating A Message Type

If an existing message type no longer meets all your needs – for example, you'd like the message format to have an extra field – but you'd still like to use code created with the old format, don't worry! It's very simple to update message types without breaking any of your existing code. Just remember the following rules:

- Don't change the numeric tags for any existing fields.

- Any new fields that you add should be `optional` or `repeated`. This means that any messages serialized by code using your "old" message format can be parsed by your new generated code, as they won't be missing any `required` elements. You should set up sensible default values for these elements so that new code can properly interact with messages generated by old code. Similarly, messages created by your new code can be parsed by your old code: old binaries simply ignore the new field when parsing. However, the unknown fields are not discarded, and if the message is later serialized, the unknown fields are serialized along with it – so if the message is passed on to new code, the new fields are still available. Note that preservation of unknown fields is currently not available for Python.
- Non-required fields can be removed, as long as the tag number is not used again in your updated message type (it may be better to rename the field instead, perhaps adding the prefix "OBSOLETE_", so that future users of your `.proto` can't accidentally reuse the number).
- A non-required field can be converted to an extension and vice versa, as long as the type and number stay the same.
- `int32`, `uint32`, `int64`, `uint64`, and `bool` are all compatible – this means you can change a field from one of these types to another without breaking forwards- or backwards-compatibility. If a number is parsed from the wire which doesn't fit in the corresponding type, you will get the same effect as if you had cast the number to that type in C++ (e.g. if a 64-bit number is read as an int32, it will be truncated to 32 bits).
- `sint32` and `sint64` are compatible with each other but are *not* compatible with the other integer types.
- `string` and `bytes` are compatible as long as the bytes are valid UTF-8.
- Embedded messages are compatible with `bytes` if the bytes contain an encoded version of the message.
- `fixed32` is compatible with `sfixed32`, and `fixed64` with `sfixed64`.

Extensions

Extensions let you declare that a range of field numbers in a message are available for third-party extensions. Other people can then declare new fields for your message type with those numeric tags in their own `.proto` files without having to edit the original file. Let's look at an example:

```
message Foo {
  // ...
  extensions 100 to 199;
}
```

This says that the range of field numbers [100, 199] in `Foo` is reserved for extensions. Other users can now add new fields to `Foo` in their own `.proto` files that import your `.proto`, using tags within your specified range – for example:

```
extend Foo {
  optional int32 bar = 126;
}
```

This says that `Foo` now has an optional `int32` field called `bar`.

When your user's `Foo` messages are encoded, the wire format is exactly the same as if the user defined the new field inside `Foo`. However, the way you access extension fields in your application code is slightly different to accessing regular fields – your generated data access code has special accessors for working with extensions. So, for example, here's how you set the value of `bar` in C++:

```
Foo foo;
foo.SetExtension(bar, 15);
```

Similarly, the `Foo` class defines templated accessors `HasExtension()`, `ClearExtension()`, `GetExtension()`, `MutableExtension()`, and `AddExtension()`. All have semantics matching the corresponding generated accessors for a normal field. For more information about working with extensions, see the generated code reference for your chosen language.

Note that extensions can be of any field type, including message types.

Nested Extensions

You can declare extensions in the scope of another type:

```
message Baz {
  extend Foo {
    optional int32 bar = 126;
  }
  ...
}
```

In this case, the C++ code to access this extension is:

```
Foo foo;
foo.SetExtension(Baz::bar, 15);
```

In other words, the only effect is that `bar` is defined within the scope of `Baz`.

This is a common source of confusion: Declaring an `extend` block nested inside a message type ***does not*** imply any relationship between the outer type and the extended type. In particular, the above example ***does not*** mean that `Baz` is any sort of subclass of `Foo`. All it means is that the symbol `bar` is declared inside the scope of `Baz`; it's simply a static member.

A common pattern is to define extensions inside the scope of the extension's field type – for example, here's an extension to `Foo` of type `Baz`, where the extension is defined as part of `Baz`:

```
message Baz {
  extend Foo {
    optional Baz foo_ext = 127;
  }
  ...
}
```

However, there is no requirement that an extension with a message type be defined inside that type. You can also do this:

```
message Baz {
  ...
}

// This can even be in a different file.
extend Foo {
  optional Baz foo_baz_ext = 127;
}
```

In fact, this syntax may be preferred to avoid confusion. As mentioned above, the nested syntax is often mistaken for subclassing by users who are not already familiar with extensions.

Choosing Extension Numbers

It's very important to make sure that two users don't add extensions to the same message type using the same numeric tag – data corruption can result if an extension is accidentally interpreted as the wrong type. You may want to consider defining an extension numbering convention for your project to prevent this happening.

If your numbering convention might involve extensions having very large numbers as tags, you can specify that your extension range goes up to the maximum possible field number using the `max` keyword:

```
message Foo {
  extensions 1000 to max;
}
```

`max` is $2^{29} - 1$, or 536,870,911.

As when choosing tag numbers in general, your numbering convention also needs to avoid field numbers 19000 though 19999 (`FieldDescriptor::kFirstReservedNumber` through

FieldDescriptor::kLastReservedNumber), as they are reserved for the Protocol Buffers implementation. You can define an extension range that includes this range, but the protocol compiler will not allow you to define actual extensions with these numbers.

Packages

You can add an optional package specifier to a .proto file to prevent name clashes between protocol message types.

```
package foo.bar;
message Open { ... }
```

You can then use the package specifier when defining fields of your message type:

```
message Foo {
  ...
  required foo.bar.Open open = 1;
  ...
}
```

The way a package specifier affects the generated code depends on your chosen language:

- In **C++** the generated classes are wrapped inside a C++ namespace. For example, Open would be in the namespace foo::bar.
- In **Java**, the package is used as the Java package, unless you explicitly provide a option java_package in your .proto file.
- In **Python**, the package directive is ignored, since Python modules are organized according to their location in the file system.

Defining Services

If you want to use your message types with an RPC (Remote Procedure Call) system, you can define an RPC service interface in a .proto file and the protocol buffer compiler will generate service interface code and stubs in your chosen language. So, for example, if you want to define an RPC service with a method that takes your SearchRequest and returns a SearchResponse, you can define it in your .proto file as follows:

```
service SearchService {
  rpc Search (SearchRequest) returns (SearchResponse);
}
```

The protocol compiler will then generate an abstract interface called SearchService and a corresponding "stub" implementation. The stub forwards all calls to an RpcChannel, which in turn is an abstract interface that you must define yourself in terms of your own RPC system. For example, you might implement an RpcChannel which serializes the message and sends it to a server via HTTP. In other words, the generated stub provides a type-safe interface for making protocol-buffer-based RPC calls, without locking you into any particular RPC implementation. So, in C++, you might end up with code like this:

```
using google::protobuf;
protobuf::RpcChannel* channel;
protobuf::RpcController* controller;
SearchService* service;
SearchRequest request;
SearchResponse response;
void DoSearch() {
  // You provide classes MyRpcChannel and MyRpcController, which implement
  // the abstract interfaces protobuf::RpcChannel and protobuf::RpcController.
  channel = new MyRpcChannel("somehost.example.com:1234");
  controller = new MyRpcController;
  // The protocol compiler generates the SearchService class based on the
  // definition given above.
  service = new SearchService::Stub(channel);
```

```
  // Set up the request.
  request.set_query("protocol buffers");
  // Execute the RPC.
  service->Search(controller, request, response, protobuf::NewCallback(&Done));
}
void Done() {
  delete service;
  delete channel;
  delete controller;
}
```

All service classes also implement the `Service` interface, which provides a way to call specific methods without knowing the method name or its input and output types at compile time. On the server side, this can be used to implement an RPC server with which you could register services.

```
using google::protobuf;
class ExampleSearchService : public SearchService {
 public:
  void Search(protobuf::RpcController* controller,
              const SearchRequest* request,
              SearchResponse* response,
              protobuf::Closure* done) {
    if (request->query() == "google") {
      response->add_result()->set_url("http://www.google.com");
    } else if (request->query() == "protocol buffers") {
      response->add_result()->set_url("http://protobuf.googlecode.com");
    }
    done->Run();
  }
};
int main() {
  // You provide class MyRpcServer.  It does not have to implement any
  // particular interface; this is just an example.
  MyRpcServer server;
  protobuf::Service* service = new ExampleSearchService;
  server.ExportOnPort(1234, service);
  server.Run();
  delete service;
  return 0;
}
```

There are a number of ongoing third-party projects to develop RPC implementations for Protocol Buffers. For a list of links to projects we know about, see the RPC Implementations wiki page.

Options

Individual declarations in a `.proto` file can be annotated with a number of *options*. Options do not change the overall meaning of a declaration, but may affect the way it is handled in a particular context. The complete list of available options is defined in `google/protobuf/descriptor.proto`.

Some options are file-level options, meaning they should be written at the top-level scope, not inside any message, enum, or service definition. Some options are message-level options, meaning they should be written inside message definitions. Options can also be written on fields, enum types, enum values, service types, and service methods; however, no useful options currently exist for any of these.

Here are a few of the most commonly used options:

- `java_package` (file option): The package you want to use for your generated Java classes. If no explicit `java_package` option is given in the `.proto` file, then by default the proto package (specified using the "package" keyword in the `.proto` file) will be used. However, proto packages generally do not make good Java packages since proto packages are not expected to start with reverse domain names. If not generating Java code, this option has no effect.

```
option java_package = "com.example.foo";
```


- `java_outer_classname` (file option): The class name for the outermost Java class (and hence the file name) you want to generate. If no explicit `java_outer_classname` is specified in the `.proto` file, the class name will be constructed by converting the `.proto` file name to camel-case (so `foo_bar.proto` becomes `FooBar.java`). If not generating Java code, this option has no effect.

```
option java_outer_classname = "Ponycopter";
```

- `optimize_for` (file option): Can be set to `CODE_SIZE` (the default) or `SPEED`. Setting this to `SPEED` tells the C++ and Java code generators to generate significantly more code in order to make operations like parsing and serializing significantly faster (typically an order of magnitude or two). We recommend only enabling this option if profiling indicates that lots of time is being spent in the Protocol Buffer library.

```
option optimize_for = SPEED;
```

- `message_set_wire_format` (message option): If set to `true`, the message uses a different binary format intended to be compatible with an old format used inside Google called `MessageSet`. Users outside Google will probably never need to use this option. The message must be declared exactly as follows:

```
message Foo {
  option message_set_wire_format = true;
  extensions 4 to max;
}
```

Generating Your Classes

To generate the Java, Python, or C++ code you need to work with the message types defined in a `.proto` file, you need to run the protocol buffer compiler `protoc` on the `.proto`. If you haven't installed the compiler, download the package and follow the instructions in the README.

The Protocol Compiler is invoked as follows:

```
protoc --proto_path=IMPORT_PATH --cpp_out=DST_DIR --java_out=DST_DIR --
python_out=DST_DIR path/to/file.proto
```

- `IMPORT_PATH` specifies a directory in which to look for `.proto` files when resolving `import` directives. If omitted, the current directory is used. Multiple import directories can be specified by passing the `--proto_path` option multiple times; they will be searched in order. `-I=IMPORT_PATH` can be used as a short form of `--proto_path`.
- You can provide one or more *output directives*:
  - `--cpp_out` generates C++ code in `DST_DIR`. See the C++ generated code reference for more.
  - `--java_out` generates Java code in `DST_DIR`. See the Java generated code reference for more.
  - `--python_out` generates Python code in `DST_DIR`. See the Python generated code reference for more.
- You must provide one or more `.proto` files as input. Multiple `.proto` files can be specified at once. Although the files are named relative to the current directory, each file must reside in one of the `IMPORT_PATH`s so that the compiler can determine its canonical name.

Style Guide

This document provides a style guide for `.proto` files. By following these conventions, you'll make your protocol buffer message definitions and their corresponding classes consistent and easy to read.

Message And Field Names

Use CamelCase (with an initial capital) for message names– for example, `SongServerRequest`. Use underscore_separated_names for field names – for example, `song_name`.

```
message SongServerRequest {
  required string song_name = 1;
}
```

Using this naming convention for field names gives you accessors like the following:

```
C++:
  const string& song_name() { ... }
  void set_song_name(const string& x) { ... }
Java:
  public String getSongName() { ... }
  public Builder setSongName(String v) { ... }
```

Enums

Use CamelCase (with an initial capital) for enum type names and CAPITALS_WITH_UNDERSCORES for value names:

```
enum Foo {
  FIRST_VALUE = 1;
  SECOND_VALUE = 2;
}
```

Each enum value should end with a semicolon, not a comma.

Services

If your `.proto` defines an RPC service, you should use CamelCase (with aninitial capital) for both the service name anc any RPC method names:

```
service FooService {
  rpc GetSomething(FooRequest) returns (FooResponse);
}
```

Encoding

This document describes the binary wire format for protocol buffer messages. You don't need to understand this to use protocol buffers in your applications, but it can be very useful to know how different protocol buffer formats affect the size of your encoded messages.

A Simple Message

Let's say you have the following very simple message definition:

```
message Test1 {
  required int32 a = 1;
}
```

In an application, you create a `Test1` message and set `a` to 150. You then serialize the message to an output stream. If you were able to examine the encoded message, you'd see three bytes:

```
08 96 01
```

So far, so small and numeric – but what does it mean? Read on...

Base 128 Varints

To understand your simple protocol buffer encoding, you first need to understand *varints*. Varints are a method of serializing integers using one or more bytes. Smaller numbers take a smaller number of bytes.

Each byte in a varint, except the last byte, has the *most significant bit* (msb) set – this indicates that there are further bytes to come. The lower 7 bits of each byte are used to store the two's complement representation of the number in groups of 7 bits, **least significant group first**

So, for example, here is the number 1 – it's a single byte, so the msb is not set:

```
0000 0001
```

And here is 300 – this is a bit more complicated:

```
1010 1100 0000 0010
```

How do you figure out that this is 300? First you drop the msb from each byte, as this is just there to tell us whether we've reached the end of the number (as you can see, it's set in the first byte as there is more than one byte in the varint):

```
1010 1100 0000 0010
→ 010 1100  000 0010
```

You reverse the two groups of 7 bits because, as you remember, varints store numbers with the least significant group first. Then you concatenate them to get your final value:

```
000 0010  010 1100
→  000 0010 ++ 010 1100
→  100101100
→  256 + 32 + 8 + 4 = 300
```

Message Structure

As you know, a protocol buffer message is a series of key-value pairs. The binary version of a message just uses the field's number as the key – the name and declared type for each field can only be determined on the decoding end by referencing the message type's definition (i.e. the `.proto` file).

When a message is encoded, the keys and values are concatenated into a byte stream. When the message is being decoded, the parser needs to be able to skip fields that it doesn't recognize. This way, new fields can be added to a message without breaking old programs that do not know about them. To this end, the "key" for each pair in a wire-format message is actually two values– the field number from your `.proto` file, plus a *wire type* that provides just enough information to find the length of the following value.

The available wire types are as follows:

Type	Meaning	Used For
0	Varint	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	64-bit	fixed64, sfixed64, double
2	Length-delimited	string, bytes, embedded messages
3	Start group	groups (deprecated)
4	End group	groups (deprecated)
5	32-bit	fixed32, sfixed32, float

Each key in the streamed message is a varint with the value `(field_number << 3) | wire_type` – in other words, the last three bits of the number store the wire type.

Now let's look at our simple example again. You now know that the first number in the stream is always a varint key, and here it's 08, or (dropping the msb):

```
000 1000
```

You take the last three bits to get the wire type (0) and then right-shift by three to get the field number (1). So you now know that the tag is 1 and the following value is a varint. Using your varint-decoding knowledge from the previous section, you can see that the next two bytes store the value 150.

```
96 01 = 1001 0110  0000 0001
       → 000 0001  ++  001 0110 (drop the msb and reverse the groups of 7 bits)
       → 10010110
       → 2 + 4 + 16 + 128 = 150
```

More Value Types

Signed Integers

As you saw in the previous section, all the protocol buffer types associated with wire type 0 are encoded as varints. However, there is an important difference between the signed int types (`sint32` and `sint64`) and the "standard" int types (`int32` and `int64`) when it comes to encoding negative numbers. If you use `int32` or `int64` as the type for a negative number, the resulting varint is *always ten bytes long* – it is, effectively, treated like a very large unsigned integer. If you use one of the signed types, the resulting varint uses ZigZag encoding, which is much more efficient.

ZigZag encoding maps signed integers to unsigned integers so that numbers with a small *absolute value* (for instance, -1) have a small varint encoded value too. It does this in a way that "zig-zags" back and forth through the positive and negative integers, so that -1 is encoded as 1, 1 is encoded as 2, -2 is encoded as 3, and so on, as you can see in the following table:

Signed Original	Encoded As
0	0
-1	1
1	2
-2	3
2147483647	4294967294
-2147483648	4294967295

In other words, each value `n` is encoded using

```
(n << 1) ^ (n >> 31)
```

for `sint32`s, or

```
(n << 1) ^ (n >> 63)
```

for the 64-bit version.

Note that the second shift – the `(n >> 31)` part – is an arithmetic shift. So, in other words, the result of the shift is either a number that is all zero bits (if `n` is positive) or all one bits (if `n` is negative).

When the `sint32` or `sint64` is parsed, its value is decoded back to the original, signed version.

Non-varint Numbers

Non-varint numeric types are simple – `double` and `fixed64` have wire type 1, which tells the parser to expect a fixed 64-bit lump of data; similarly `float` and `fixed32` have wire type 5, which tells it to expect 32 bits. In both cases the values are stored in little-endian byte order.

Strings

A wire type of 2 (length-delimited) means that the value is a varint encoded length followed by the specified number of bytes of data.

```
message Test2 {
  required string b = 2;
}
```

Setting the value of b to "testing" gives you:

```
12 07 74 65 73 74 69 6e 67
```

The red bytes are the UTF8 of "testing". The key here is 0x12 → tag = 2, type = 2. The length varint in the value is 7 and lo and behold, we find seven bytes following it – our string.

Embedded Messages

Here's a message definition with an embedded message of our example type, Test1:

```
message Test3 {
  required Test1 c = 3;
}
```

And here's the encoded version:

```
 1a 03 08 96 01
```

As you can see, the last three bytes are exactly the same as our first example (`08 96 01`), and they're preceded by the number 3 – embedded messages are treated in exactly the same way as strings (wire type = 2).

Optional And Repeated Elements

If your message definition has `repeated` elements, the encoded message has zero or more key-value pairs with the same tag number. These repeated values do not have to appear consecutively; they may be interleaved with other fields. The order of the elements with respect to each other is preserved when parsing, though the ordering with respect to other fields is lost.

If any of your elements are `optional`, the encoded message may or may not have a key-value pair with that tag number.

Normally, an encoded message would never have more than one instance of an `optional` or `required` field. However, parsers are expected to handle the case in which they do. For numeric types and strings, if the same value appears multiple times, the parser accepts the *last* value it sees. For embedded message fields, the parser merges multiple instances of the same field, as if with the `Message::MergeFrom` method – that is, all singular scalar fields in the latter instance replace those in the former, singular embedded messages are merged, and repeated fields are concatenated. The effect of these rules is that parsing the concatenation of two encoded messages produces exactly the same result as if you had parsed the two messages separately and merged the resulting objects. That is, this:

```
MyMessage message;
message.ParseFromString(str1 + str2);
```

is equivalent to this:

```
MyMessage message, message2;
message.ParseFromString(str1);
message2.ParseFromString(str2);
message.MergeFrom(message2);
```

This property is occasionally useful, as it allows you to merge two messages even if you do not know their types.

Field Order

While you can use field numbers in any order in a `.proto`, when a message is serialized its known fields should be written sequentially by field number, as in the provided C++, Java, and Python serialization code. This allows parsing code to use optimizations that rely on field numbers being in sequence. However, protocol buffer parsers must be able to parse fields in any order, as not all messages are created by simply serializing an object – for instance, it's sometimes useful to merge two messages by simply concatenating them.

If a message has unknown fields, the current Java and C++ implementations write them in arbitrary order after the sequentially-ordered known fields. The current Python implementation does not track unknown fields.

Tutorials

Each tutorial in this section shows you how to implement a simple application using protocol buffers in your favourite language, introducing you to the language's protocol buffer API as well as showing you the basics of creating and using .proto files. The complete sample code for each application is also provided.

The tutorials don't assume that you know anything about protocol buffers, but do assume that you are comfortable writing code in your chosen language, including using file I/O.

- C++ Tutorial
- Java Tutorial
- Python Tutorial

Protocol Buffer Basics: C++

This tutorial provides a basic C++ programmer's introduction to working with protocol buffers. By walking through creating a simple example application, it shows you how to

- Define message formats in a `.proto` file.
- Use the protocol buffer compiler.
- Use the C++ protocol buffer API to write and read messages.

This isn't a comprehensive guide to using protocol buffers in C++. For more detailed reference information, see the Protocol Buffer Language Guide, the C++ API Reference, the C++ Generated Code Guide, and the Encoding Reference.

Why Use Protocol Buffers?

The example we're going to use is a very simple "address book" application that can read and write people's contact details to and from a file. Each person in the address book has a name, an ID, an email address, and a contact phone number.

How do you serialize and retrieve structured data like this? There are a few ways to solve this problem:

- The raw in-memory data structures can be sent/saved in binary form. Over time, this is a fragile approach, as the receiving/reading code must be compiled with exactly the same memory layout, endianness, etc. Also, as files accumulate data in the raw format and copies of software that are wired for that format are spread around, it's very hard to extend the format.
- You can invent an ad-hoc way to encode the data items into a single string – such as encoding 4 ints as "12:3:-23:67". This is a simple and flexible approach, although it does require writing one-off encoding and parsing code, and the parsing imposes a small run-time cost. This works best for encoding very simple data.
- Serialize the data to XML. This approach can be very attractive since XML is (sort of) human readable and there are binding libraries for lots of languages. This can be a good choice if you want to share data with other applications/projects. However, XML is notoriously space intensive, and encoding/decoding it can impose a huge performance penalty on applications. Also, navigating an XML DOM tree is considerably more complicated than navigating simple fields in a class normally would be.

Protocol buffers are the flexible, efficient, automated solution to solve exactly this problem. With protocol buffers, you write a `.proto` description of the data structure you wish to store. From that, the protocol buffer compiler creates a class that implements automatic encoding and parsing of the protocol buffer data with an efficient binary format. The generated class provides getters and setters for the fields that make up a protocol buffer and takes care of the details of reading and writing the protocol buffer as a unit. Importantly, the protocol buffer format supports the idea of extending the format over time in such a way that the code can still read data encoded with the old format.

Where to Find the Example Code

The example code is included in the source code package, under the "examples" directory. Download it here.

Defining Your Protocol Format

To create your address book application, you'll need to start with a `.proto` file. The definitions in a `.proto` file are simple: you add a *message* for each data structure you want to serialize, then specify a name and a type for each field in the message. Here is the `.proto` file that defines your messages, `addressbook.proto`.

```
package tutorial;

message Person {
  required string name = 1;
```

```
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}

message AddressBook {
  repeated Person person = 1;
}
```

As you can see, the syntax is similar to C++ or Java. Let's go through each part of the file and see what it does.

The `.proto` file starts with a package declaration, which helps to prevent naming conflicts between different projects. In C++, your generated classes will be placed in a namespace matching the package name.

Next, you have your message definitions. A message is just an aggregate containing a set of typed fields. Many standard simple data types are available as field types, including `bool`, `int32`, `float`, `double`, and `string`. You can also add further structure to your messages by using other message types as field types – in the above example the `Person` message contains `PhoneNumber` messages, while the `AddressBook` message contains `Person` messages. You can even define message types nested inside other messages – as you can see, the `PhoneNumber` type is defined inside `Person`. You can also define `enum` types if you want one of your fields to have one of a predefined list of values – here you want to specify that a phone number can be one of `MOBILE`, `HOME`, or `WORK`.

The " = 1", " = 2" markers on each element identify the unique "tag" that field uses in the binary encoding. Tag numbers 1-15 require one less byte to encode than higher numbers, so as an optimization you can decide to use those tags for the commonly used or repeated elements, leaving tags 16 and higher for less-commonly used optional elements. Each element in a repeated field requires re-encoding the tag number, so repeated fields are particularly good candidates for this optimization.

Each field must be annotated with one of the following modifiers:

- `required`: a value for the field must be provided, otherwise the message will be considered "uninitialized". If `libprotobuf` is compiled in debug mode, serializing an uninitialized message will cause an assertion failure. In optimized builds, the check is skipped and the message will be written anyway. However, parsing an uninitialized message will always fail (by returning `false` from the parse method). Other than this, a required field behaves exactly like an optional field.
- `optional`: the field may or may not be set. If an optional field value isn't set, a default value is used. For simple types, you can specify your own default value, as we've done for the phone number `type` in the example. Otherwise, a system default is used: zero for numeric types, the empty string for strings, false for bools. For embedded messages, the default value is always the "default instance" or "prototype" of the message, which has none of its fields set. Calling the accessor to get the value of an optional (or required) field which has not been explicitly set always returns that field's default value.
- `repeated`: the field may be repeated any number of times (including zero). The order of the repeated values will b preserved in the protocol buffer. Think of repeated fields as dynamically sized arrays.

Required Is Forever You should be very careful about marking fields as `required`. If at some point you wish to stop writing or sending a required field, it will be problematic to change the field to an optional field – old readers will consider messages without this field to be incomplete and may reject or drop them unintentionally. You should consider writing application-specific custom validation routines for your buffers instead. Some engineers at Google have come to the conclusion that using `required` does more harm than good; they prefer to use only `optional` and `repeated`. However, this view is not universal.

You'll find a complete guide to writing `.proto` files – including all the possible field types – in the Protocol Buffer Language Guide. Don't go looking for facilities similar to class inheritance, though – protocol buffers don't do that.

Compiling Your Protocol Buffers

Now that you have a `.proto`, the next thing you need to do is generate the classes you'll need to read and write `AddressBook` (and hence `Person` and `PhoneNumber`) messages. To do this, you need to run the protocol buffer compiler `protoc` on your `.proto`:

1. If you haven't installed the compiler, [download the package]() and follow the instructions in the README.
2. Now run the compiler, specifying the source directory (where your application's source code lives – the current directory is used if you don't provide a value), the destination directory (where you want the generated code to go; often the same as `$SRC_DIR`), and the path to your `.proto` relative to the source directory. In this case, you...:

   ```
   protoc -I=$SRC_DIR --cpp_out=$DST_DIR addressbook.proto
   ```

   Because you want C++ classes, you use the `--cpp_out` option – similar options are provided for other supported languages.

This generates the following files in your specified destination directory:

- `addressbook.pb.h`, the header which declares your generated classes.
- `addressbook.pb.cc`, which contains the implementation of your classes.

The Protocol Buffer API

Let's look at some of the generated code and see what classes and functions the compiler has created for you. If you look in `tutorial.pb.h`, you can see that you have a class for each message you specified in `tutorial.proto`. Looking closer at the `Person` class, you can see that the complier has generated accessors for each field. For example, for the `name`, `id`, `email`, and `phone` fields, you have these methods:

```
 // name
  inline bool has_name() const;
  inline void clear_name();
  inline const ::std::string& name() const;
  inline void set_name(const ::std::string& value);
  inline void set_name(const char* value);
  inline ::std::string* mutable_name();
  // id
  inline bool has_id() const;
  inline void clear_id();
  inline int32_t id() const;
  inline void set_id(int32_t value);
  // email
  inline bool has_email() const;
  inline void clear_email();
  inline const ::std::string& email() const;
  inline void set_email(const ::std::string& value);
  inline void set_email(const char* value);
  inline ::std::string* mutable_email();
  // phone
  inline int phone_size() const;
  inline void clear_phone();
  inline const ::google::protobuf::RepeatedPtrField< ::tutorial::Person_PhoneNumber
>& phone() const;
  inline ::google::protobuf::RepeatedPtrField< ::tutorial::Person_PhoneNumber >*
mutable_phone();
  inline const ::tutorial::Person_PhoneNumber& phone(int index) const;
  inline ::tutorial::Person_PhoneNumber* mutable_phone(int index);
  inline ::tutorial::Person_PhoneNumber* add_phone();
```

As you can see, the getters have exactly the name as the field in lowercase, and the setter methods begin with `set_`. There are also `has_` methods for each singular (required or optional) field which return true if that field has been set. Finally, each field has a `clear_` method that un-sets the field back to its empty state.

While the numeric `id` field just has the basic accessor set described above, the `name` and `email` fields have a couple of extra methods because they're strings – a `mutable_` getter that lets you get a direct pointer to the string, and an extra setter. Note that you can call `mutable_email()` even if `email` is not already set; it will be initialized to an empty string automatically. If you had a singular message field in this example, it would also have a `mutable_` method but not a `set_` method.

Repeated fields also have some special methods – if you look at the methods for the repeated `phone` field, you'll see that you can

- check the repeated field's `_size` (in other words, how many phone numbers are associated with this `Person`).
- get a specified phone number using its index.
- update an existing phone number at the specified index.
- add another phone number to the message which you can then edit (repeated scalar types have an `add_` that just lets you pass in the new value).

For more information on exactly what members the protocol compiler generates for any particular field definition, see the C++ generated code reference.

Enums and Nested Classes

The generated code includes a `PhoneType` enum that corresponds to your `.proto` enum. You can refer to this type as `Person::PhoneType` and its values as `Person::MOBILE`, `Person::HOME`, and `Person::WORK` (the implementation details are a little more complicated, but you don't need to understand them to use the enum).

The compiler has also generated a nested class for you called `Person::PhoneNumber`. If you look at the code, you can see that the "real" class is actually called `Person_PhoneNumber`, but a typedef defined inside `Person` allows you to treat it as if it were a nested class. The only case where this makes a difference is if you want to forward-declare the class in another file – you cannot forward-declare nested types in C++, but you can forward-declare `Person_PhoneNumber`.

Standard Message Methods

Each message class also contains a number of other methods that let you check or manipulate the entire message, including:

- `bool IsInitialized() const;`: checks if all the required fields have been set.
- `string DebugString() const;`: returns a human-readable representation of the message, particularly useful for debugging.
- `void CopyFrom(const Person& from);`: overwrites the message with the given message's values.
- `void Clear();`: clears all the elements back to the empty state.

These and the I/O methods described in the following section implement the `Message` interface shared by all C++ protocol buffer classes. For more info, see the complete API documentation for `Message`.

Parsing and Serialization

Finally, each protocol buffer class has methods for writing and reading messages of your chosen type using the protocol buffer binary format. These include:

- `bool SerializeToString(string* output) const;`: serializes the message and stores the bytes in the given string. Note that the bytes are binary, not text; we only use the `string` class as a convenient container.
- `bool ParseFromString(const string& data);`: parses a message from the given string.
- `bool SerializeToOstream(ostream* output) const;`: writes the message to the given C++ `ostream`.
- `bool ParseFromIStream(istream* input);`: parses a message from the given C++ `istream`.

These are just a couple of the options provided for parsing and serialization. Again, see the `Message` API reference for a complete list.

Protocol Buffers and O-O Design Protocol buffer classes are basically dumb data holders (like structs in C++); they don't make good first class citizens in an object model. If you want to add richer behaviour to a generated class, the best way to do this is to wrap the generated protocol buffer class in an application-specific class. Wrapping protocol buffers is also a good idea if you don't have control over the design of the `.proto` file (if, say, you're reusing one from another project). In that case, you can use the wrapper class to craft an interface better suited to the unique environment of your application: hiding some data and methods, exposing convenience functions, etc. **You should never add behaviour to the generated classes by inheriting from them**. This will break internal mechanisms and is not good object-oriented practice anyway.

Writing A Message

Now let's try using your protocol buffer classes. The first thing you want your address book application to be able to do is write personal details to your address book file. To do this, you need to create and populate instances of your protocol buffer classes and then write them to an output stream.

Here is a program which reads an `AddressBook` from a file, adds one new `Person` to it based on user input, and writes the new `AddressBook` back out to the file again. The parts whichdirectly call or reference code generated by the protocol compiler are highlighted.

```
#include <iostream>
#include <fstream>
#include <string>
#include "addressbook.pb.h"
using namespace std;
// This function fills in a Person message based on user input.
void PromptForAddress(tutorial::Person* person) {
  cout << "Enter person ID number: " ;
  int id;
  cin >> id;
  person->set_id(id);
  cin.ignore(256, '\n');
  cout << "Enter name: " ;
  getline(cin, *person->mutable_name());
  cout << "Enter email address (blank for none): " ;
  string email;
  getline(cin, email);
  if (!email.empty()) {
    person->set_email(email);
  }
  while (true) {
    cout << "Enter a phone number (or leave blank to finish): " ;
    string number;
    getline(cin, number);
    if (number.empty()) {
      break;
    }
    tutorial::Person::PhoneNumber* phone_number = person->add_phone();
    phone_number->set_number(number);
    cout << "Is this a mobile, home, or work phone? " ;
    string type;
    getline(cin, type);
    if (type == "mobile") {
      phone_number->set_type(tutorial::Person::MOBILE);
    } else if (type == "home") {
      phone_number->set_type(tutorial::Person::HOME);
    } else if (type == "work") {
      phone_number->set_type(tutorial::Person::WORK);
    } else {
      cout << "Unknown phone type.  Using default." << endl;
    }
  }
}
// Main function:  Reads the entire address book from a file,
//   adds one person based on user input, then writes it back out to the same
//   file.
int main(int argc, char* argv[]) {
  // Verify that the version of the library that we linked against is
  // compatible with the version of the headers we compiled against.
  GOOGLE_PROTOBUF_VERIFY_VERSION ;
  if (argc != 2) {
    cerr << "Usage:  " << argv[0] << " ADDRESS_BOOK_FILE" << endl;
    return -1;
  }
  tutorial::AddressBook address_book ;
```

```
  {
    // Read the existing address book.
    fstream input(argv[1], ios::in | ios::binary);
    if (!input) {
      cout << argv[1] << ": File not found.  Creating a new file." << endl;
    } else if (!address_book.ParseFromIstream(&input)) {
      cerr << "Failed to parse address book." << endl;
      return -1;
    }
  }
  // Add an address.
  PromptForAddress(address_book.add_person());
  {
    // Write the new address book back to disk.
    fstream output(argv[1], ios::out | ios::trunc | ios::binary);
    if (!address_book.SerializeToOstream(&output)) {
      cerr << "Failed to write address book." << endl;
      return -1;
    }
  }
  return 0;
}
```

Notice the `GOOGLE_PROTOBUF_VERIFY_VERSION` macro. It is good practice – though not strictly necessary– to execute this macro before using the C++ Protocol Buffer library. It verifies that you have not accidentally linked against a version of the library which is incompatible with the version of the headers you compiled with. If a version mismatch is detected, the program will abort. Note that every `.pb.cc` file automatically invokes this macro on startup.

Reading A Message

Of course, an address book wouldn't be much use if you couldn't get any information out of it! This example reads the file created by the above example and prints all the information in it.

```
#include <iostream>
#include <fstream>
#include <string>
#include "addressbook.pb.h"
using namespace std;
// Iterates though all people in the AddressBook and prints info about them.
void ListPeople(const tutorial::AddressBook& address_book) {
  for (int i = 0; i < address_book.person_size(); i++) {
    const tutorial::Person& person = address_book.person(i);
    cout << "Person ID: " << person.id() << endl;
    cout << "  Name: " << person.name() << endl;
    if (person.has_email()) {
      cout << "  E-mail address: " << person.email() << endl;
    }
    for (int j = 0; j < person.phone_size(); j++) {
      const tutorial::Person::PhoneNumber& phone_number = person.phone(j);
      switch (phone_number.type()) {
        case tutorial::Person::MOBILE:
          cout << "  Mobile phone #: ";
          break;
        case tutorial::Person::HOME:
          cout << "  Home phone #: ";
          break;
        case tutorial::Person::WORK:
          cout << "  Work phone #: ";
          break;
      }
      cout << phone_number.number() << endl;
    }
  }
}
```

```
// Main function:  Reads the entire address book from a file and prints all
//   the information inside.
int main(int argc, char* argv[]) {
  // Verify that the version of the library that we linked against is
  // compatible with the version of the headers we compiled against.
  GOOGLE_PROTOBUF_VERIFY_VERSION ;
  if (argc != 2) {
    cerr << "Usage:  " << argv[0] << " ADDRESS_BOOK_FILE" << endl;
    return -1;
  }
  tutorial::AddressBook address_book ;
  {
    // Read the existing address book.
    fstream input(argv[1], ios::in | ios::binary);
    if (!address_book .ParseFromIstream (&input)) {
      cerr << "Failed to parse address book."  << endl;
      return -1;
    }
  }
  ListPeople(address_book );
  return 0;
}
```

Extending a Protocol Buffer

Sooner or later after you release the code that uses your protocol buffer, you will undoubtedly want to "improve" the protocol buffer's definition. If you want your new buffers to be backwards-compatible, and your old buffers to be forward-compatible – and you almost certainly do want this– then there are some rules you need to follow. In the new version of the protocol buffer:

- you *must not* change the tag numbers of any existing fields.
- you *must not* add or delete any required fields.
- you *may* delete optional or repeated fields.
- you *may* add new optional or repeated fields but you must use fresh tag numbers (i.e. tag numbers that were never used in this protocol buffer, not even by deleted fields).

(There are some exceptions to these rules, but they are rarely used.)

If you follow these rules, old code will happily read new messages and simply ignore any new fields. To the old code, optional fields that were deleted will simply have their default value, and deleted repeated fields will be empty. New code will also transparently read old messages. However, keep in mind that new optional fields will not be present in old messages, so you will need to either check explicitly whether they're set with `has_`, or provide a reasonable default value in your `.proto` file with `[default = value]` after the tag number. If the default value is notspecified for an optional element, a type-specific default value is used instead: for strings, the default value is the empty string. For booleans, the default value is false. For numeric types, the default value is zero. Note also that if you added a new repeated field, your new code will not be able to tell whether it was left empty (by new code) or never set at all (by old code) since there is no `has_` flag for it.

Getting More Speed

By default, the protocol buffer compiler tries to generate smaller files by using reflection to implement most functionality (e.g. parsing and serialization). However, the compiler can also generate code optimizedexplicitly for your message types often providing an order of magnitude performance boost, but also doubling the size of the code. If profiling shows that your application is spending a lot of time in the protocol buffer library, you should try changing the optimization mode. Simply add the following line to your `.proto` file:

```
option optimize_for = SPEED;
```

Re-run the protocol compiler, and it will generate extremely fast parsing, serialization, and other code.

If that's still not enough for you, another good thing to try in C++ is to reuse your message objects. Messages try to keep around any memory they allocate for reuse, even when they are cleared. Thus, if you are handling many messages with the same type and similar structure in succession, it is a good idea to reuse the same message object each time to take load off the memory allocator. However, be careful about doing this when parsing untrusted data! A malicious user could send your app a sequence of messages which have very different structures (for example, each could set a different set of unknown fields). Even if each message is small, it could cause your object to allocate some memory that will not be reused for future messages. Over time, these can add up to a lot of memory. To be safe, you should allocate a fresh object each time you parse a message you do not trust.

Advanced Usage

Protocol buffers have uses that go beyond simple accessors and serialization. Be sure to explore the C++ API reference to see what else you can do with them.

One key feature provided by protocol message classes is *reflection*. You can iterate over the fields of a message and manipulate their values without writing your code against any specific message type. One very useful way to use reflection is for converting protocol messages to and from other encodings, such as XML or JSON. A more advanced use of reflection might be to find differences between two messages of the same type, or to develop a sort of "regular expressions for protocol messages" in which you can write expressions that match certain message contents. If you use your imagination, it's possible to apply Protocol Buffers to a much wider range of problems than you might initially expect!

Reflection is provided by the `Message::Reflection` interface.

API Reference

This section contains reference documentation for working with protocol buffer classes in C++, Java, and Python. The documentation for each language includes:

- A reference guide to the code generated by the protocol buffer compiler from your `.proto` files.
- Generated API documentation for the provided source code.

Note that there are APIs for several more languages in the pipeline – for details, see the other languages wiki page.

C++ Reference

- C++ Generated Code Guide
- C++ API

Java Reference

- Java Generated Code Guide
- Java API (Javadoc)

Python Reference

- Python Generated Code Guide
- Python API (Epydoc)

C++ Generated Code

This page describes exactly what C++ code the protocol buffer compiler generates for any given protocol definition. You should read the language guide before reading this document.

Compiler Invocation

The protocol buffer compiler produces C++ output when invoked with the `--cpp_out=` command-line flag. The parameter to the `--cpp_out=` option is the directory where you want the compiler to write your C++ output. The compiler creates a header file and an implementation file for each `.proto` file input. The names of the output files are computed by taking the name of the `.proto` file and making two changes:

- The extension (`.proto`) is replaced with either `.pb.h` or `.pb.cc` for the header or implementation file, respectively.
- The proto path (specified with the `--proto_path=` or `-I` command-line flag) is replaced with the output path (specified with the `--cpp_out=` flag).

So, for example, let's say you invoke the compiler as follows:

```
protoc --proto_path=src --cpp_out=build/gen src/foo.proto src/bar/baz.proto
```

The compiler will read the files `src/foo.proto` and `src/bar/baz.proto` and produce four output files: `build/gen/foo.pb.h`, `build/gen/foo.pb.cc`, `build/gen/bar/baz.pb.h`, `build/gen/bar/baz.pb.cc`. The compiler will automatically create the directory `build/gen/bar` if necessary, but it will *not* create `build` or `build/gen`; they must already exist.

Packages

If a `.proto` file contains a `package` declaration, the entire contents of the file will be placed in a corresponding C++ namespace. For example, given the `package` declaration:

```
package foo.bar;
```

All declarations in the file will reside in the `foo::bar` namespace.

Messages

Given a simple message declaration:

```
message Foo {}
```

The protocol buffer compiler generates a class called `Foo`, which publicly derives from google::protobuf::Message. The class is a concrete class; no pure-virtual methods are left unimplemented. Methods that are virtual in Message but not pure-virtual may or may not be overridden by Foo, depending on the optimization mode. By default, Foo only implements the minimum set of methods necessary to function. However, if the .proto file contains the line:

```
option optimize_for = SPEED;
```

then Foo will override all virtual methods with fast, generated implementations. This can significantly increase the size of the generated code, so you should only use this option if profiling indicates that it is needed.

You should *not* create your own Foo subclasses. If you subclass this class and override a virtual method, the override may be ignored, as many generated method calls are de-virtualized to improve performance.

In addition to the methods defined by the Message interface, the Foo class defines the following methods:

- Foo(): Default constructor.
- ~Foo(): Default destructor.
- Foo(const Foo& other): Copy constructor.
- Foo& operator=(const Foo& other): Assignment operator.
- const UnknownFieldSet& unknown_fields() const: Returns the set of unknown fields encountered while parsing this message.
- UnknownFieldSet* mutable_unknown_fields(): Returns a mutable pointer to the set of unknown fields encountered while parsing this message.

The class also defines the following static methods:

- static const Descriptor& descriptor(): Returns the type's descriptor. This contains information about the type, including what fields it has and what their types are. This can be used with reflection to inspect fields programmatically.
- static const Foo& default_instance(): Returns a const singleton instance of Foo which is identical to a newly-constructed instance of Foo (so all singular fields are unset and all repeated fields are empty). Note that the default instance of a message can be used as a factory by calling its New() method.

A message can be declared inside another message. For example: message Foo { message Bar { } }

In this case, the compiler generates two classes: Foo and Foo_Bar. In addition, the compiler generates a typedef inside Foo as follows:

```
typedef Foo_Bar Bar;
```

This means that you can use the nested type's class as if it was the nested class Foo::Bar. However, note that C++ does not allow nested types to be forward-declared. If you want to forward-declare Bar in another file and use that declaration, you must identify it as Foo_Bar.

Fields

In addition to the methods described in the previous section, the protocol buffer compiler generates a set of accessor methods for each field defined within the message in the .proto file.

Singular Numeric Fields

For either of these field definitions:

```
optional int32 foo = 1;
required int32 foo = 1;
```

The compiler will generate the following accessor methods:

- bool has_foo() const: Returns true if the field is set.
- int32 foo() const: Returns the current value of the field. If the field is not set, returns the default value.
- void set_foo(int32 value): Sets the value of the field. After calling this, has_foo() will return true and foo() will return value.

- `void clear_foo()`: Clears the value of the field. After calling this, `has_foo()` will return `false` and `foo()` will return the default value.

For other numeric field types (including `bool`), `int32` is replaced with the corresponding C++ type according to the scalar value types table.

Singular String Fields

For any of these field definitions:

```
optional string foo = 1;
required string foo = 1;
optional bytes foo = 1;
required bytes foo = 1;
```

The compiler will generate the following accessor methods:

- `bool has_foo() const`: Returns `true` if the field is set.
- `const string& foo() const`: Returns the current value of the field. If the field is not set, returns the default value.
- `void set_foo(const string& value)`: Sets the value of the field. After calling this, `has_foo()` will return `true` and `foo()` will return a copy of `value`.
- `void set_foo(const char* value)`: Sets the value of the field using a C-style null-terminated string. After calling this, `has_foo()` will return `true` and `foo()` will return a copy of `value`.
- `string* mutable_foo()`: Returns a mutable pointer to the `string` object that stores the field's value. If the field was not set prior to the call, then the returned string will be empty (*not* the default value). After calling this, `has_foo()` will return `true` and `foo()` will return whatever value is written into the given string. The pointer is invalidated by a call to `Clear()` or `clear_foo()`.
- `void clear_foo()`: Clears the value of the field. After calling this, `has_foo()` will return `false` and `foo()` will return the default value.

Singular Enum Fields

Given the enum type:

```
enum Bar {
  BAR_VALUE = 1;
}
```

For either of these field definitions:

```
optional Bar foo = 1;
required Bar foo = 1;
```

The compiler will generate the following accessor methods:

- `bool has_foo() const`: Returns `true` if the field is set.
- `Bar foo() const`: Returns the current value of the field. If the field is not set, returns the default value.
- `void set_foo(Bar value)`: Sets the value of the field. After calling this, `has_foo()` will return `true` and `foo()` will return `value`. In debug mode (i.e. NDEBUG is not defined), if `value` does not match any of the values defined for `Bar`, this method will abort the process.
- `void clear_foo()`: Clears the value of the field. After calling this, `has_foo()` will return `false` and `foo()` will return the default value.

Singular Embedded Message Fields

Given the message type:

```
message Bar {}
```

For either of these field definitions:

```
optional Bar foo = 1;
required Bar foo = 1;
```

The compiler will generate the following accessor methods:

- `bool has_foo() const`: Returns `true` if the field is set.
- `const Bar& foo() const`: Returns the current value of the field. If the field is not set, returns a `Bar` with none of its fields set (possibly `Bar::default_instance()`).
- `Bar* mutable_foo()`: Returns a mutable pointer to the `Bar` object that stores the field's value. If the field was not set prior to the call, then the returned `Bar` will have none of its fields set (i.e. it will be identical to a newly-allocated `Bar`). After calling this, `has_foo()` will return `true` and `foo()` will return a reference to the same instance of `Bar`. The pointer is invalidated by a call to `Clear()` or `clear_foo()`.
- `void clear_foo()`: Clears the value of the field. After calling this, `has_foo()` will return `false` and `foo()` will return the default value.

Repeated Numeric Fields

For this field definition:

```
repeated int32 foo = 1;
```

The compiler will generate the following accessor methods:

- `int foo_size() const`: Returns the number of elements currently in the field.
- `int32 foo(int index) const`: Returns the element at the given zero-based index.
- `void set_foo(int index, int32 value)`: Sets the value of the element at the given zero-based index.
- `void add_foo(int32 value)`: Appends a new element to the field with the given value.
- `void clear_foo()`: Removes all elements from the field. After calling this, `foo_size()` will return zero.
- `const RepeatedField<int32>& foo() const`: Returns the underlying `RepeatedField` that stores the field's elements. This container class provides STL-like iterators and other methods.
- `RepeatedField<int32>*` mutable_foo(): Returns a mutable pointer to the underlying `RepeatedField` that stores the field's elements. This container class provides STL-like iterators and other methods.

For other numeric field types (including `bool`), `int32` is replaced with the corresponding C++ type according to the scalar value types table.

Repeated String Fields

For either of these field definitions:

```
repeated string foo = 1;
repeated bytes foo = 1;
```

The compiler will generate the following accessor methods:

- `int foo_size() const`: Returns the number of elements currently in the field.
- `const string& foo(int index) const`: Returns the element at the given zero-based index.
- `void set_foo(int index, const string& value)`: Sets the value of the element at the given zero-based index.
- `void set_foo(int index, const char* value)`: Sets the value of the element at the given zero-based index using a C-style null-terminated string.
- `string* mutable_foo(int index)`: Returns a mutable pointer to the `string` object that stores the value of the element at the given zero-based index. The pointer is invalidated by a call to `Clear()` or `clear_foo()`, or by manipulating the underlying `RepeatedPtrField` in a way that would remove this element.
- `void add_foo(const string& value)`: Appends a new element to the field with the given value.
- `void add_foo(const char* value)`: Appends a new element to the field using a C-style null-terminated string.

- `string* add_foo()`: Adds a new empty string element and returns a pointer to it. The pointer is invalidated by a call to `Clear()` or `clear_foo()`, or by manipulating the underlying `RepeatedPtrField` in a way that would remove this element.
- `void clear_foo()`: Removes all elements from the field. After calling this, `foo_size()` will return zero.
- `const RepeatedPtrField<string>& foo() const`: Returns the underlying `RepeatedPtrField` that stores the field's elements. This container class provides STL-like iterators and other methods.
- `RepeatedPtrField<string>*` mutable_foo(): Returns a mutable pointer to the underlying `RepeatedPtrField` that stores the field's elements. This container class provides STL-like iterators and other methods.

Repeated Enum Fields

Given the enum type:

```
enum Bar {
  BAR_VALUE = 1;
}
```

For this field definition:

```
repeated Bar foo = 1;
```

The compiler will generate the following accessor methods:

- `int foo_size() const`: Returns the number of elements currently in the field.
- `Bar foo(int index) const`: Returns the element at the given zero-based index.
- `void set_foo(int index, Bar value)`: Sets the value of the element at the given zero-based index. In debug mode (i.e. NDEBUG is not defined), if `value` does not match any of the values defined for `Bar`, this method will abort the process.
- `void add_foo(Bar value)`: Appends a new element to the field with the given value. In debug mode (i.e. NDEBUG is not defined), if `value` does not match any of the values defined for `Bar`, this method will abort the process.
- `void clear_foo()`: Removes all elements from the field. After calling this, `foo_size()` will return zero.
- `const RepeatedField<Bar>& foo() const`: Returns the underlying `RepeatedField` that stores the field's elements. This container class provides STL-like iterators and other methods.
- `RepeatedField<Bar>*` mutable_foo(): Returns a mutable pointer to the underlying `RepeatedField` that stores the field's elements. This container class provides STL-like iterators and other methods.

Repeated Embedded Message Fields

Given the message type:

```
message Bar {}
```

For this field definitions:

```
repeated Bar foo = 1;
```

The compiler will generate the following accessor methods:

- `int foo_size() const`: Returns the number of elements currently in the field.
- `const Bar& foo(int index) const`: Returns the element at the given zero-based index.
- `Bar* mutable_foo(int index)`: Returns a mutable pointer to the `Bar` object that stores the value of the element at the given zero-based index. The pointer is invalidated by a call to `Clear()` or `clear_foo()`, or by manipulating the underlying `RepeatedPtrField` in a way that would remove this element.
- `Bar* add_foo()`: Adds a new element and returns a pointer to it. The returned `Bar` will have none of its fields set (i.e. it will be identical to a newly-allocated `Bar`). The pointer is invalidated by a call to `Clear()` or `clear_foo()`, or by manipulating the underlying `RepeatedPtrField` in a way that would remove this element.

- `void clear_foo()`: Removes all elements from the field. After calling this, `foo_size()` will return zero.
- `const RepeatedPtrField<Bar>& foo() const`: Returns the underlying `RepeatedPtrField` that stores the field's elements. This container class provides STL-like iterators and other methods.
- `RepeatedPtrField<Bar>*` mutable_foo(): Returns a mutable pointer to the underlying `RepeatedPtrField` that stores the field's elements. This container class provides STL-like iterators and other methods.

Enumerations

Given an enum definition like:

```
enum Foo {
  VALUE_A = 1;
  VALUE_B = 5;
  VALUE_C = 1234;
}
```

The protocol buffer compiler will generate a C++ enum type called `Foo` with the same set of values. In addition, the compiler will generate the following functions:

- `const EnumDescriptor* Foo_descriptor()`: Returns the type's descriptor, which contains information about what values this enum type defines.
- `bool Foo_IsValid(int value)`: Returns `true` if the given numeric value matches one of `Foo`'s defined values. In the above example, it would return `true` if the input were 1, 5, or 1234.

Be careful when casting integers to enums. If an integer is cast to an enum value, the integer *must* be one of the valid values for than enum, or the results may be undefined. If in doubt, use the generated `Foo_IsValid()` function to test if the cast is valid. Setting an enum-typed field of a protocol message to an invalid value may cause an assertion failure. If an invalid enum value is read when parsing a message, it will be treated as an unknown field.

You can define an enum inside a message type. In this case, the protocol buffer compiler generates code that makes it appear that the enum type itself was declared nested inside the message's class. The `Foo_descriptor()` and `Foo_IsValid()` functions are declared as static methods. In reality,the enum type itself and its values are declared at the global scope with mangled names, and are imported into the class's scope with a typedef and a series of constant definitions. This is done only to get around problems with declaration ordering. Do not depend on the mangled top-level names; pretend the enum really is nested in the message class.

Extensions

Given a message with an extension range:

```
message Foo {
  extensions 100 to 199;
}
```

The protocol buffer compiler will generate some additional methods for `Foo`: `HasExtension()`, `ExtensionSize()`, `ClearExtension()`, `GetExtension()`, `SetExtension()`, `MutableExtension()`, and `AddExtension()`. Each of these methods takes, as its first parameter, an extension identifier (described below), which identifies an extension field. The remaining parameters and the return value are exactly the same as those for the corresponding accessor methods that would be generated for a normal (non-extension) field of the same type as the extension identifier. (`GetExtension()` corresponds to the accessors with no special prefix.)

Given an extension definition:

```
extend Foo {
  optional int32 bar = 123;
}
```

The protocol buffer compiler generates an "extension identifier" called `bar`, which you can use with `Foo`'s extension accessors to access this extension, like so:

```
Foo foo;
assert(!foo.HasExtension(bar));
foo.SetExtension(bar, 1);
assert(foo.HasExtension(bar));
assert(foo.GetExttension(bar) == 1);
foo.ClearExtension(bar);
assert(!foo.HasExtension(bar));
```

(The exact implementation of extension identifiers is complicated and involves magical use of templates – however, you don't need to worry about how extension identifiers work to use them.)

Extensions can be declared nested inside of another type. For example, a common pattern is to do something like this:

```
message Baz {
  extend Foo {
    optional Baz foo_ext = 124;
  }
}
```

In this case, the extension identifier `foo_ext` is declared nested inside `Baz`. It can be used as follows:

```
Foo foo;
Baz* baz = foo.MutableExtension(Baz::foo_ext);
FillInMyBaz(baz);
```

Services

Interface

Given a service definition:

```
service Foo {
  rpc Bar(FooRequest) returns(FooResponse);
}
```

The protocol buffer compiler will generate a class `Foo` to represent this service. `Foo` will have a virtual method for each method defined in the service definition. In this case, the method `Bar` is defined as:

```
virtual void Bar(RpcController* controller, const FooRequest* request,
                 FooResponse * response, Closure* done);
```

The parameters are equivalent to the parameters of `Service::CallMethod()`, except that the `method` argument is implied and `request` and `response` specify their exact type.

These generated methods are virtual, but not pure-virtual. The default implementations simply call `controller->SetFailed()` with an error message indicating that the method is unimplemented, then invoke the `done` callback. When implementing your own service, you must subclass this generated service and implement its methods as appropriate.

`Foo` subclasses the `Service` interface. The protocol buffer compiler automatically generates implementations of the methods of `Service` as follows:

- `GetDescriptor`: Returns the service's `ServiceDescriptor`.
- `CallMethod`: Determines which method is being called based on the provided method descriptor and calls it directly, down-casting the request and response messages objects to the correct types.
- `GetRequestPrototype` and `GetResponsePrototype`: Returns the default instance of the request or response of the correct type for the given method.

The following static method is also generated:

- `static ServiceDescriptor descriptor()`: Returns the type's descriptor, which contains information about what methods this service has and what their input and output types are.

Stub

The protocol buffer compiler also generates a "stub" implementation of every service interface, which is used by clients wishing to send requests to servers implementing the service. For the `Foo` service (above), the stub implementation `Foo_Stub` will be defined. As with nested message types, a typedef is used so that `Foo_Stub` can also be referred to as `Foo::Stub`.

`Foo_Stub` is a subclass of `Foo` which also implements the following methods:

- `Foo_Stub(RpcChannel* channel)`: Constructs a new stub which sends requests on the given channel.
- `Foo_Stub(RpcChannel* channel, ChannelOwnership ownership)`: Constructs a new stub which sends requests on the given channel and possibly owns that channel. If `ownership` is `Service::STUB_OWNS_CHANNEL` then when the stub object is deleted it will delete the channel as well.
- `RpcChannel* channel()`: Returns this stub's channel, as passed to the constructor.

The stub additionally implements each of the service's methods as a wrapper around the channel. Calling one of the methods simply calls `channel->CallMethod()`.

The Protocol Buffer library does not include an RPC implementation. However, it includes all of the tools you need to hook up a generated service class to any arbitrary RPC implementation of your choice. You need only provide implementations of `RpcChannel` and `RpcController`. See the documentation for `service.h` for more information.

C++ API

Packages
google::protobuf *Core components of the Protocol Buffers runtime library.*
google::protobuf::io *Auxiliary classes used for I/O.*
google::protobuf::compiler *Implementation of the Protocol Buffer compiler.*

google::protobuf

Core components of the Protocol Buffers runtime library.

The files in this package represent the core of the Protocol Buffer system. All of them are part of the libprotobuf library.

A note on thread-safety:

Thread-safety in the Protocol Buffer library follows a simple rule: unless explicitly noted otherwise, it is always safe to use an object from multiple threads simultaneously as long as the object is declared const in all threads (or, it is only used in ways that would be allowed if it were declared const). However, if an object is accessed in one thread in a way that would not be allowed if it were const, then it is not safe to access that object in any other thread simultaneously.

Put simply, read-only access to an object can happen in multiple threads simultaneously, but write access can only happen in a single thread at a time.

The implementation does contain some "const" methods which actually modify the object behind the scenes -- e.g., to cache results -- but in these cases mutex locking is used to make the access thread-safe.

Files
google/protobuf/descriptor.h *This file contains classes which describe a type of protocol message.*
google/protobuf/descriptor.pb.h *Protocol buffer representations of descriptors.*
google/protobuf/descriptor_database.h *Interface for manipulating databases of descriptors.*
google/protobuf/dynamic_message.h *Defines an implementation of Message which can emulate types which are not known at compile-time.*
google/protobuf/message.h *This file contains the abstract interface for all protocol messages.*
google/protobuf/repeated_field.h *RepeatedField and RepeatedPtrField are used by generated protocol message classes to manipulate repeated fields.*
google/protobuf/service.h *This module declares the abstract interfaces underlying proto2 RPC services.*

google/protobuf/text_format.h *Utilities for printing and parsing protocol messages in a human-readable, text-based format.*
google/protobuf/unknown_field_set.h *Contains classes used to keep track of unrecognized fields seen while parsing a protocol message.*
google/protobuf/stubs/common.h *Contains basic types and utilities used by the rest of the library.*

google::protobuf::io

Auxiliary classes used for I/O.

The Protocol Buffer library uses the classes in this package to deal with I/O and encoding/decoding raw bytes. Most users will not need to deal with this package. However, users who want to adapt the system to work with their own I/O abstractions -- e.g., to allow Protocol Buffers to be read from a different kind of input stream without the need for a temporary buffer -- should take a closer look.

Files
google/protobuf/io/coded_stream.h *This file contains the CodedInputStream and CodedOutputStream classes, which wrap a ZeroCopyInputStream or ZeroCopyOutputStream, respectively, and allow you to read or write individual pieces of data in various formats.*
google/protobuf/io/printer.h *Utility class for writing text to a ZeroCopyOutputStream.*
google/protobuf/io/tokenizer.h *Class for parsing tokenized text from a ZeroCopyInputStream.*
google/protobuf/io/zero_copy_stream.h *This file contains the ZeroCopyInputStream and ZeroCopyOutputStream interfaces, which represent abstract I/O streams to and from which protocol buffers can be read and written.*
google/protobuf/io/zero_copy_stream_impl.h *This file contains common implementations of the interfaces defined in zero_copy_stream.h.*

google::protobuf::compiler

Implementation of the Protocol Buffer compiler.

This package contains code for parsing .proto files and generating code based on them. There are two reasons you might be interested in this package:

- You want to parse .proto files at runtime. In this case, you should look at importer.h. Since this functionality is widely useful, it is included in the libprotobuf base library; you do not have to link against libprotoc.
- You want to write a custom protocol compiler which generates different kinds of code, e.g. code in a different language which is not supported by the official compiler. For this purpose, command_line_interface.h provides you with a complete compiler front-end, so all you need to do is write a custom implementation of CodeGenerator and a trivial main() function. You can even make your compiler support the official languages in addition to your own. Since this functionality is only useful to those writing custom compilers, it is in a separate library called "libprotoc" which you will have to link against.

Files
google/protobuf/compiler/code_generator.h *Defines the abstract interface implemented by each of the language-specific code generators.*
google/protobuf/compiler/command_line_interface.h *Implements the Protocol Compiler front-end such that it may be reused by custom compilers written to support other languages.*

google/protobuf/compiler/importer.h *This file is the public interface to the .proto file parser.*
google/protobuf/compiler/parser.h *Implements parsing of .proto files to FileDescriptorProtos.*
google/protobuf/compiler/cpp/cpp_generator.h *Generates C++ code for a given .proto file.*
google/protobuf/compiler/java/java_generator.h *Generates Java code for a given .proto file.*
google/protobuf/compiler/python/python_generator.h *Generates Python code for a given .proto file.*

descriptor.h

```
#include <google/protobuf/descriptor.h>
namespace google::protobuf
```

This file contains classes which describe a type of protocol message.

You can use a message's descriptor to learn at runtime what fields it contains and what the types of those fields are. The Message interface also allows you to dynamically access and modify individual fields by passing the FieldDescriptor of the field you are interested in.

Most users will not care about descriptors, because they will write code specific to certain protocol types and will simply use the classes generated by the protocol compiler directly. Advanced users who want to operate on arbitrary types (not known at compile time) may want to read descriptors in order to learn about the contents of a message. A very small number of users will want to construct their own Descriptors, either because they are implementing Message manually or because they are writing something like the protocol compiler.

For an example of how you might use descriptors, see the code example at the top of message.h.

Classes in this file

`Descriptor`
Describes a type of protocol message, or a particular group within a message.

`Descriptor::ExtensionRange`
A range of field numbers which are designated for third-party extensions.

`FieldDescriptor`
Describes a single field of a message.

`EnumDescriptor`
Describes an enum type defined in a .proto file.

`EnumValueDescriptor`
Describes an individual enum constant of a particular type.

`ServiceDescriptor`
Describes an RPC service.

`MethodDescriptor`
Describes an individual service method.

`FileDescriptor`
Describes a whole .proto file.

`DescriptorPool`
Used to construct descriptors.

`DescriptorPool::ErrorCollector`
When converting a FileDescriptorProto to a FileDescriptor, various errors might be detected in the input.

class Descriptor

```
#include <google/protobuf/descriptor.h>
namespace google::protobuf
```

Describes a type of protocol message, or a particular group within a message.

To obtain the Descriptor for a given message object, call Message::GetDescriptor(). Generated message classes also have a static method called descriptor() which returns the type's descriptor. Use DescriptorPool to construct your own descriptors.

Members

const string &	**name**() const *The name of the message type, not including its scope.*
const string &	**full_name**() const *The fully-qualified name of the message type, scope delimited by periods. more...*
int	**index**() const *Index of this descriptor within the file or containing type's message type array.*
const FileDescriptor *	**file**() const *The .proto file in which this message type was defined. Never NULL.*
const Descriptor *	**containing_type**() const *If this Descriptor describes a nested type, this returns the type in which it is nested. more...*
const MessageOptions &	**options**() const *Get options for this message type. more...*
void	**CopyTo**(DescriptorProto * proto) const *Write the contents of this Descriptor into the given DescriptorProto. more...*
string	**DebugString**() const *Write the contents of this decriptor in a human-readable form. more...*

Field stuff

int	**field_count**() const *The number of fields in this message type.*
const FieldDescriptor *	**field**(int index) const *Gets a field by index, where 0 <= index < field_count(). more...*
const FieldDescriptor *	**FindFieldByNumber**(int number) const *Looks up a field by declared tag number. more...*
const FieldDescriptor *	**FindFieldByName**(const string & name) const *Looks up a field by name. Returns NULL if no such field exists.*

Nested type stuff

int	**nested_type_count**() const *The number of nested types in this message type.*
const Descriptor *	**nested_type**(int index) const *Gets a nested type by index, where 0 <= index < nested_type_count(). more...*
const Descriptor *	**FindNestedTypeByName**(const string & name) const *Looks up a nested type by name. more...*

Enum stuff

int	**enum_type_count**() const

	The number of enum types in this message type.
const EnumDescriptor *	**enum_type**(int index) const Gets an enum type by index, where 0 <= index < enum_type_count(). more...
const EnumDescriptor *	**FindEnumTypeByName**(const string & name) const Looks up an enum type by name. Returns NULL if no such enum type exists.
const EnumValueDescriptor *	**FindEnumValueByName**(const string & name) const Looks up an enum value by name, among all enum types in this message. more...
Extensions	
int	**extension_range_count**() const The number of extension ranges in this message type.
const ExtensionRange *	**extension_range**(int index) const Gets an extension range by index, where 0 <= index < extension_range_count(). more...
bool	**IsExtensionNumber**(int number) const Returns true if the number is in one of the extension ranges.
int	**extension_count**() const The number of extensions -- extending *other* messages -- that were defined nested within this message type's scope.
const FieldDescriptor *	**extension**(int index) const Get an extension by index, where 0 <= index < extension_count(). more...
const FieldDescriptor *	**FindExtensionByName**(const string & name) const Looks up a named extension (which extends some *other* message type) defined within this message type's scope.

---

const string & Descriptor::full_name() const

The fully-qualified name of the message type, scope delimited by periods.

For example, message type "Foo" which is declared in package "bar" has full name "bar.Foo". If a type "Baz" is nested within Foo, Baz's full_name is "bar.Foo.Baz". To get only the part that comes after the last '.', use name().

---

const Descriptor * Descriptor::containing_type() const

If this Descriptor describes a nested type, this returns the type in which it is nested.

Otherwise, returns NULL.

---

const MessageOptions & Descriptor::options() const

Get options for this message type.

These are specified in the .proto file by placing lines like "option foo = 1234;" in the message definition. The exact set of known options is defined by MessageOptions in google/protobuf/descriptor.proto.

```
void Descriptor::CopyTo(
        DescriptorProto * proto) const
```

Write the contents of this Descriptor into the given DescriptorProto.

The target DescriptorProto must be clear before calling this; if it isn't, the result may be garbage.

```
string Descriptor::DebugString() const
```

Write the contents of this decriptor in a human-readable form.

Output will be suitable for re-parsing.

```
const FieldDescriptor *
    Descriptor::field(
        int index) const
```

Gets a field by index, where 0 <= index < field_count().

These are returned in the order they were defined in the .proto file.

```
const FieldDescriptor *
    Descriptor::FindFieldByNumber(
        int number) const
```

Looks up a field by declared tag number.

Returns NULL if no such field exists.

```
const Descriptor *
    Descriptor::nested_type(
        int index) const
```

Gets a nested type by index, where 0 <= index < nested_type_count().

These are returned in the order they were defined in the .proto file.

```
const Descriptor *
    Descriptor::FindNestedTypeByName(
        const string & name) const
```

Looks up a nested type by name.

Returns NULL if no such nested type exists.

```
const EnumDescriptor *
    Descriptor::enum_type(
        int index) const
```

Gets an enum type by index, where 0 <= index < enum_type_count().

These are returned in the order they were defined in the .proto file.

```
const EnumValueDescriptor *
    Descriptor::FindEnumValueByName(
        const string & name) const
```

Looks up an enum value by name, among all enum types in this message.

Returns NULL if no such value exists.

```
const ExtensionRange *
    Descriptor::extension_range(
        int index) const
```

Gets an extension range by index, where 0 <= index < extension_range_count().

These are returned in the order they were defined in the .proto file.

```
const FieldDescriptor *
    Descriptor::extension(
        int index) const
```

Get an extension by index, where 0 <= index < extension_count().

These are returned in the order they were defined in the .proto file.

struct Descriptor::ExtensionRange

```
#include <google/protobuf/descriptor.h>
namespace google::protobuf
```

A range of field numbers which are designated for third-party extensions.

Members	
`int`	**`start`** *inclusive*
`int`	**`end`** *exclusive*

class FieldDescriptor

```
#include <google/protobuf/descriptor.h>
namespace google::protobuf
```

Describes a single field of a message.

To get the descriptor for a given field, first get the Descriptor for the message in which it is defined, then call

Descriptor::FindFieldByName() To get a FieldDescriptor for an extension, do one of the following:

- Get the Descriptor or FileDescriptor for its containing scope, then call Descriptor::FindExtensionByName() or FileDescriptor::FindExtensionByName()
- Given a DescriptorPool, call DescriptorPool::FindExtensionByNumber()
- Given a Reflection for a message object, call Reflection::FindKnownExtensionByName() or Reflection::FindKnownExtensionByNumber() Use DescriptorPool to construct your own descriptors.

Members

enum	**Type** *Identifies a field type. more...*
enum	**CppType** *Specifies the C++ data type used to represent the field. more...*
enum	**Label** *Identifies whether the field is optional, required, or repeated. more...*
const string &	**name**() const *Name of this field within the message.*
const string &	**full_name**() const *Fully-qualified name of the field.*
const FileDescriptor *	**file**() const *File in which this field was defined.*
bool	**is_extension**() const *Is this an extension field?*
int	**number**() const *Declared tag number.*
Type	**type**() const *Declared type of this field.*
CppType	**cpp_type**() const *C++ type of this field.*
Label	**label**() const *optional/required/repeated*
bool	**is_required**() const *shorthand for label() == LABEL_REQUIRED*
bool	**is_optional**() const *shorthand for label() == LABEL_OPTIONAL*
bool	**is_repeated**() const *shorthand for label() == LABEL_REPEATED*
int	**index**() const *Index of this field within the message's field array, or the file or extension scope's extensions array.*
bool	**has_default_value**() const *Does this field have an explicitly-declared default value?*
int32	**default_value_int32**() const *Get the field default value if cpp_type() == CPPTYPE_INT32. more...*
int64	**default_value_int64**() const *Get the field default value if cpp_type() == CPPTYPE_INT64. more...*

uint32	**default_value_uint32**() const *Get the field default value if cpp_type() == CPPTYPE_UINT32. more...*
uint64	**default_value_uint64**() const *Get the field default value if cpp_type() == CPPTYPE_UINT64. more...*
float	**default_value_float**() const *Get the field default value if cpp_type() == CPPTYPE_FLOAT. more...*
double	**default_value_double**() const *Get the field default value if cpp_type() == CPPTYPE_DOUBLE. more...*
bool	**default_value_bool**() const *Get the field default value if cpp_type() == CPPTYPE_BOOL. more...*
const EnumValueDescriptor *	**default_value_enum**() const *Get the field default value if cpp_type() == CPPTYPE_ENUM. more...*
const string &	**default_value_string**() const *Get the field default value if cpp_type() == CPPTYPE_STRING. more...*
const Descriptor *	**containing_type**() const *The Descriptor for the message of which this is a field. more...*
const Descriptor *	**extension_scope**() const *An extension may be declared within the scope of another message. more...*
const Descriptor *	**message_type**() const *If type is TYPE_MESSAGE or TYPE_GROUP, returns a descriptor for the message or the group type. more...*
const EnumDescriptor *	**enum_type**() const *If type is TYPE_ENUM, returns a descriptor for the enum. more...*
const FieldDescriptor *	**experimental_map_key**() const *EXPERIMENTAL; DO NOT USE. more...*
const FieldOptions &	**options**() const *Get the FieldOptions for this field. more...*
void	**CopyTo**(FieldDescriptorProto * proto) const *See Descriptor::CopyTo().*
string	**DebugString**() const *See Descriptor::DebugString().*
const int	**kMaxNumber** = (1 << 29) - 1 *Valid field numbers are positive integers up to kMaxNumber.*
const int	**kFirstReservedNumber** = 19000 *First field number reserved for the protocol buffer library implementation. more...*
const int	**kLastReservedNumber** = 19999 *Last field number reserved for the protocol buffer library implementation. more...*

```
enum FieldDescriptor::Type {
  TYPE_DOUBLE = 1,
  TYPE_FLOAT = 2,
```

```
  TYPE_INT64 = 3,
  TYPE_UINT64 = 4,
  TYPE_INT32 = 5,
  TYPE_FIXED64 = 6,
  TYPE_FIXED32 = 7,
  TYPE_BOOL = 8,
  TYPE_STRING = 9,
  TYPE_GROUP = 10,
  TYPE_MESSAGE = 11,
  TYPE_BYTES = 12,
  TYPE_UINT32 = 13,
  TYPE_ENUM = 14,
  TYPE_SFIXED32 = 15,
  TYPE_SFIXED64 = 16,
  TYPE_SINT32 = 17,
  TYPE_SINT64 = 18,
  MAX_TYPE = 18
}
```

Identifies a field type.

0 is reserved for errors. The order is weird for historical reasons. Types 12 and up are new in proto2.

TYPE_DOUBLE	double, exactly eight bytes on the wire.
TYPE_FLOAT	float, exactly four bytes on the wire.
TYPE_INT64	int64, varint on the wire. Negative numbers take 10 bytes. Use TYPE_SINT64 if negative values are likely.
TYPE_UINT64	uint64, varint on the wire.
TYPE_INT32	int32, varint on the wire. Negative numbers take 10 bytes. Use TYPE_SINT32 if negative values are likely.
TYPE_FIXED64	uint64, exactly eight bytes on the wire.
TYPE_FIXED32	uint32, exactly four bytes on the wire.
TYPE_BOOL	bool, varint on the wire.
TYPE_STRING	UTF-8 text.
TYPE_GROUP	Tag-delimited message. Deprecated.
TYPE_MESSAGE	Length-delimited message.
TYPE_BYTES	Arbitrary byte array.
TYPE_UINT32	uint32, varint on the wire
TYPE_ENUM	Enum, varint on the wire.
TYPE_SFIXED32	int32, exactly four bytes on the wire
TYPE_SFIXED64	int64, exactly eight bytes on the wire
TYPE_SINT32	int32, ZigZag-encoded varint on the wire
TYPE_SINT64	int64, ZigZag-encoded varint on the wire
MAX_TYPE	Constant useful for defining lookup tables indexed by Type.

```
enum FieldDescriptor::CppType {
  CPPTYPE_INT32 = 1,
```

```
  CPPTYPE_INT64 = 2,
  CPPTYPE_UINT32 = 3,
  CPPTYPE_UINT64 = 4,
  CPPTYPE_DOUBLE = 5,
  CPPTYPE_FLOAT = 6,
  CPPTYPE_BOOL = 7,
  CPPTYPE_ENUM = 8,
  CPPTYPE_STRING = 9,
  CPPTYPE_MESSAGE = 10,
  MAX_CPPTYPE = 10
}
```

Specifies the C++ data type used to represent the field.

There is a fixed mapping from Type to CppType where each Type maps to exactly one CppType. 0 is reserved for errors.

CPPTYPE_INT32	TYPE_INT32, TYPE_SINT32, TYPE_SFIXED32.
CPPTYPE_INT64	TYPE_INT64, TYPE_SINT64, TYPE_SFIXED64.
CPPTYPE_UINT32	TYPE_UINT32, TYPE_FIXED32.
CPPTYPE_UINT64	TYPE_UINT64, TYPE_FIXED64.
CPPTYPE_DOUBLE	TYPE_DOUBLE.
CPPTYPE_FLOAT	TYPE_FLOAT.
CPPTYPE_BOOL	TYPE_BOOL.
CPPTYPE_ENUM	TYPE_ENUM.
CPPTYPE_STRING	TYPE_STRING, TYPE_BYTES.
CPPTYPE_MESSAGE	TYPE_MESSAGE, TYPE_GROUP.
MAX_CPPTYPE	Constant useful for defining lookup tables indexed by CppType.

```
enum FieldDescriptor::Label {
  LABEL_OPTIONAL = 1,
  LABEL_REQUIRED = 2,
  LABEL_REPEATED = 3,
  MAX_LABEL = 3
}
```

Identifies whether the field is optional, required, or repeated.

0 is reserved for errors.

LABEL_OPTIONAL	optional
LABEL_REQUIRED	required
LABEL_REPEATED	repeated
MAX_LABEL	Constant useful for defining lookup tables indexed by Label.

---

int32 FieldDescriptor::default_value_int32() const

Get the field default value if cpp_type() == CPPTYPE_INT32.

If no explicit default was defined, the default is 0.

int64 FieldDescriptor::default_value_int64() const

Get the field default value if cpp_type() == CPPTYPE_INT64.

If no explicit default was defined, the default is 0.

uint32 FieldDescriptor::default_value_uint32() const

Get the field default value if cpp_type() == CPPTYPE_UINT32.

If no explicit default was defined, the default is 0.

uint64 FieldDescriptor::default_value_uint64() const

Get the field default value if cpp_type() == CPPTYPE_UINT64.

If no explicit default was defined, the default is 0.

float FieldDescriptor::default_value_float() const

Get the field default value if cpp_type() == CPPTYPE_FLOAT.

If no explicit default was defined, the default is 0.0.

double FieldDescriptor::default_value_double() const

Get the field default value if cpp_type() == CPPTYPE_DOUBLE.

If no explicit default was defined, the default is 0.0.

bool FieldDescriptor::default_value_bool() const

Get the field default value if cpp_type() == CPPTYPE_BOOL.

If no explicit default was defined, the default is false.

const EnumValueDescriptor * FieldDescriptor::default_value_enum() const

Get the field default value if cpp_type() == CPPTYPE_ENUM.

If no explicit default was defined, the default is the first value defined in the enum type (all enum types are required to have at least one value). This never returns NULL.

const string & FieldDescriptor::default_value_string() const

Get the field default value if cpp_type() == CPPTYPE_STRING.

If no explicit default was defined, the default is the empty string.

---

const Descriptor * FieldDescriptor::containing_type() const

The Descriptor for the message of which this is a field.

For extensions, this is the extended type. Never NULL.

---

const Descriptor * FieldDescriptor::extension_scope() const

An extension may be declared within the scope of another message.

If this field is an extension (is_extension() is true), then extension_scope() returns that message, or NULL if the extension was declared at global scope. If this is not an extension, extension_scope() is undefined (may assert-fail).

---

const Descriptor * FieldDescriptor::message_type() const

If type is TYPE_MESSAGE or TYPE_GROUP, returns a descriptor for the message or the group type.

Otherwise, undefined.

---

const EnumDescriptor * FieldDescriptor::enum_type() const

If type is TYPE_ENUM, returns a descriptor for the enum.

Otherwise, undefined.

---

const FieldDescriptor * FieldDescriptor::experimental_map_key() const

EXPERIMENTAL; DO NOT USE.

If this field is a map field, experimental_map_key() is the field that is the key for this map. experimental_map_key()->containing_type() is the same as message_type().

---

const FieldOptions & FieldDescriptor::options() const

Get the FieldOptions for this field.

This includes things listed in square brackets after the field definition. E.g., the field:

```
optional string text = 1 [ctype=CORD];
```

has the "ctype" option set. FieldOptions is actually a protocol message, which makes it easier to extend.

---

const int FieldDescriptor::kFirstReservedNumber = 1900(

First field number reserved for the protocol buffer library implementation.

Users may not declare fields that use reserved numbers.

---

const int FieldDescriptor::kLastReservedNumber = 1999!

Last field number reserved for the protocol buffer library implementation.

Users may not declare fields that use reserved numbers.

class EnumDescriptor

```
#include <google/protobuf/descriptor.h>
namespace google::protobuf
```

Describes an enum type defined in a .proto file.

To get the EnumDescriptor for a generated enum type, call TypeName_descriptor(). Use DescriptorPool to construct your own descriptors.

Members	
const string &	**name**() const *The name of this enum type in the containing scope.*
const string &	**full_name**() const *The fully-qualified name of the enum type, scope delimited by periods.*
int	**index**() const *Index of this enum within the file or containing message's enum array.*
const FileDescriptor *	**file**() const *The .proto file in which this enum type was defined. Never NULL.*
int	**value_count**() const *The number of values for this EnumDescriptor. more...*
const EnumValueDescriptor *	**value**(int index) const *Gets a value by index, where 0 <= index < value_count(). more...*
const EnumValueDescriptor *	**FindValueByName**(const string & name) const *Looks up a value by name. Returns NULL if no such value exists.*
const EnumValueDescriptor *	**FindValueByNumber**(int number) const *Looks up a value by number. more...*
const Descriptor *	**containing_type**() const *If this enum type is nested in a message type, this is that message type. more...*
const EnumOptions &	**options**() const

	Get options for this enum type. more...
void	**CopyTo**(EnumDescriptorProto * proto) const *See Descriptor::CopyTo().*
string	**DebugString**() const *See Descriptor::DebugString().*

```
int EnumDescriptor::value_count() const
```

The number of values for this EnumDescriptor.

Guaranteed to be greater than zero.

```
const EnumValueDescriptor *
    EnumDescriptor::value(
        int index) const
```

Gets a value by index, where 0 <= index < value_count().

These are returned in the order they were defined in the .proto file.

```
const EnumValueDescriptor *
    EnumDescriptor::FindValueByNumber(
        int number) const
```

Looks up a value by number.

Returns NULL if no such value exists. If multiple values have this number, the first one defined is returned.

```
const Descriptor *
    EnumDescriptor::containing_type() const
```

If this enum type is nested in a message type, this is that message type.

Otherwise, NULL.

```
const EnumOptions &
    EnumDescriptor::options() const
```

Get options for this enum type.

These are specified in the .proto file by placing lines like "option foo = 1234;" in the enum definition. The exact set of known options is defined by EnumOptions in google/protobuf/descriptor.proto.

class EnumValueDescriptor

```
#include <google/protobuf/descriptor.h>
namespace google::protobuf
```

Describes an individual enum constant of a particular type.

To get the EnumValueDescriptor for a given enum value, first get the EnumDescriptor for its type, then use EnumDescriptor::FindValueByName() or EnumDescriptor::FindValueByNumber(). Use DescriptorPool to construct your own descriptors.

Members

`const string &`	**`name`**`() const` *Name of this enum constant.*
`int`	**`index`**`() const` *Index within the enums's Descriptor.*
`int`	**`number`**`() const` *Numeric value of this enum constant.*
`const string &`	**`full_name`**`() const` *The full_name of an enum value is a sibling symbol of the enum type. more...*
`const EnumDescriptor *`	**`type`**`() const` *The type of this value. Never NULL.*
`const EnumValueOptions &`	**`options`**`() const` *Get options for this enum value. more...*
`void`	**`CopyTo`**`(EnumValueDescriptorProto * proto) const` *See Descriptor::CopyTo().*
`string`	**`DebugString`**`() const` *See Descriptor::DebugString().*

`const string & EnumValueDescriptor::full_name() const`

The full_name of an enum value is a sibling symbol of the enum type.

e.g. the full name of FieldDescriptorProto::TYPE_INT32 is actually "google.protobuf.FieldDescriptorProto.TYPE_INT32", NOT "google.protobuf.FieldDescriptorProto.Type.TYPE_INT32". This is to conform with C++ scoping rules for enums.

`const EnumValueOptions & EnumValueDescriptor::options() const`

Get options for this enum value.

These are specified in the .proto file by adding text like "[foo = 1234]" after an enum value definition. The exact set of known options is defined by EnumValueOptions in google/protobuf/descriptor.proto.

class ServiceDescriptor

```
#include <google/protobuf/descriptor.h>
namespace google::protobuf
```

Describes an RPC service.

To get the ServiceDescriptor for a service, call Service::GetDescriptor(). Generated service classes also have a static method called descriptor() which returns the type's ServiceDescriptor. Use DescriptorPool to construct your own descriptors.

Members

`const string &`	`name() const` *The name of the service, not including its containing scope.*
`const string &`	`full_name() const` *The fully-qualified name of the service, scope delimited by periods.*
`int`	`index() const` *Index of this service within the file's services array.*
`const FileDescriptor *`	`file() const` *The .proto file in which this service was defined. Never NULL.*
`const ServiceOptions &`	`options() const` *Get options for this service type. more...*
`int`	`method_count() const` *The number of methods this service defines.*
`const MethodDescriptor *`	`method(int index) const` *Gets a MethodDescriptor by index, where 0 <= index < method_count(). more...*
`const MethodDescriptor *`	`FindMethodByName(const string & name) const` *Look up a MethodDescriptor by name.*
`void`	`CopyTo(ServiceDescriptorProto * proto) const` *See Descriptor::CopyTo().*
`string`	`DebugString() const` *See Descriptor::DebugString().*

```
const ServiceOptions &
    ServiceDescriptor::options() const
```

Get options for this service type.

These are specified in the .proto file by placing lines like "option foo = 1234;" in the service definition. The exact set of known options is defined by ServiceOptions in google/protobuf/descriptor.proto.

```
const MethodDescriptor *
    ServiceDescriptor::method(
        int index) const
```

Gets a MethodDescriptor by index, where 0 <= index < method_count().

These are returned in the order they were defined in the .proto file.

class MethodDescriptor

```
#include <google/protobuf/descriptor.h>
namespace google::protobuf
```

Describes an individual service method.

To obtain a MethodDescriptor given a service, first get its ServiceDescriptor, then call ServiceDescriptor::FindMethodByName() Use DescriptorPool to construct your own descriptors.

Members	
const string &	**name**() const *Name of this method, not including containing scope.*
const string &	**full_name**() const *The fully-qualified name of the method, scope delimited by periods.*
int	**index**() const *Index within the service's Descriptor.*
const ServiceDescriptor *	**service**() const *Gets the service to which this method belongs. Never NULL.*
const Descriptor *	**input_type**() const *Gets the type of protocol message which this method accepts as input.*
const Descriptor *	**output_type**() const *Gets the type of protocol message which this message produces as output.*
const MethodOptions &	**options**() const *Get options for this method. more...*
void	**CopyTo**(MethodDescriptorProto * proto) const *See Descriptor::CopyTo().*
string	**DebugString**() const *See Descriptor::DebugString().*

```
const MethodOptions &
    MethodDescriptor::options() const
```

Get options for this method.

These are specified in the .proto file by placing lines like "option foo = 1234;" in curly-braces after a method declaration. The exact set of known options is defined by MethodOptions in google/protobuf/descriptor.proto.

class FileDescriptor

```
#include <google/protobuf/descriptor.h>
namespace google::protobuf
```

Describes a whole .proto file.

To get the FileDescriptor for a compiled-in file, get the descriptor for something defined in that file and call descriptor->file (). Use DescriptorPool to construct your own descriptors.

Members	
const string &	**name**() const *The filename, relative to the source tree. more...*
const string &	**package**() const *The package, e.g. "google.protobuf.compiler".*
const DescriptorPool *	**pool**() const *The DescriptorPool in which this FileDescriptor and all its contents were allocated. more...*

int	**dependency_count**() const *The number of files imported by this one.*
const [FileDescriptor](#) *	**dependency**(int index) const *Gets an imported file by index, where 0 <= index < [dependency_count()](#). [more...](#)*
int	**message_type_count**() const *Number of top-level message types defined in this file. [more...](#)*
const [Descriptor](#) *	**message_type**(int index) const *Gets a top-level message type, where 0 <= index < [message_type_count()](#). [more...](#)*
int	**enum_type_count**() const *Number of top-level enum types defined in this file. [more...](#)*
const [EnumDescriptor](#) *	**enum_type**(int index) const *Gets a top-level enum type, where 0 <= index < [enum_type_count()](#). [more...](#)*
int	**service_count**() const *Number of services defined in this file.*
const [ServiceDescriptor](#) *	**service**(int index) const *Gets a service, where 0 <= index < [service_count()](#). [more...](#)*
int	**extension_count**() const *Number of extensions defined at file scope. [more...](#)*
const [FieldDescriptor](#) *	**extension**(int index) const *Gets an extension's descriptor, where 0 <= index < [extension_count()](#). [more...](#)*
const [FileOptions](#) &	**options**() const *Get options for this file. [more...](#)*
const [Descriptor](#) *	**FindMessageTypeByName**(const string & name) const *Find a top-level message type by name. Returns NULL if not found.*
const [EnumDescriptor](#) *	**FindEnumTypeByName**(const string & name) const *Find a top-level enum type by name. Returns NULL if not found.*
const [EnumValueDescriptor](#) *	**FindEnumValueByName**(const string & name) const *Find an enum value defined in any top-level enum by name. [more...](#)*
const [ServiceDescriptor](#) *	**FindServiceByName**(const string & name) const *Find a service definition by name. Returns NULL if not found.*
const [FieldDescriptor](#) *	**FindExtensionByName**(const string & name) const *Find a top-level extension definition by name. Returns NULL if not found.*
void	**CopyTo**([FileDescriptorProto](#) * proto) const *See [Descriptor::CopyTo()](#).*
string	**DebugString**() const *See [Descriptor::DebugString()](#).*

---

const string & FileDescriptor::name() const

The filename, relative to the source tree.

e.g. "google/protobuf/descriptor.proto"

```
const DescriptorPool *
    FileDescriptor::pool() const
```

The DescriptorPool in which this FileDescriptor and all its contents were allocated.

Never NULL.

```
const FileDescriptor *
    FileDescriptor::dependency(
        int index) const
```

Gets an imported file by index, where 0 <= index < dependency_count().

These are returned in the order they were defined in the .proto file.

```
int FileDescriptor::message_type_count() const
```

Number of top-level message types defined in this file.

(This does not include nested types.)

```
const Descriptor *
    FileDescriptor::message_type(
        int index) const
```

Gets a top-level message type, where 0 <= index < message_type_count().

These are returned in the order they were defined in the .proto file.

```
int FileDescriptor::enum_type_count() const
```

Number of top-level enum types defined in this file.

(This does not include nested types.)

```
const EnumDescriptor *
    FileDescriptor::enum_type(
        int index) const
```

Gets a top-level enum type, where 0 <= index < enum_type_count().

These are returned in the order they were defined in the .proto file.

```
const ServiceDescriptor *
    FileDescriptor::service(
```

```
        int index) const
```

Gets a service, where 0 <= index < service_count().

These are returned in the order they were defined in the .proto file.

```
int FileDescriptor::extension_count() const
```

Number of extensions defined at file scope.

(This does not include extensions nested within message types.)

```
const FieldDescriptor *
    FileDescriptor::extension(
        int index) const
```

Gets an extension's descriptor, where 0 <= index < extension_count()

These are returned in the order they were defined in the .proto file.

```
const FileOptions &
    FileDescriptor::options() const
```

Get options for this file.

These are specified in the .proto file by placing lines like "option foo = 1234;" at the top level, outside of any other definitions. The exact set of known options is defined by FileOptions in google/protobuf/descriptor.proto.

```
const EnumValueDescriptor *
    FileDescriptor::FindEnumValueByName(
        const string & name) const
```

Find an enum value defined in any top-level enum by name.

Returns NULL if not found.

class DescriptorPool

```
#include <google/protobuf/descriptor.h>
namespace google::protobuf
```

Used to construct descriptors.

Normally you won't want to build your own descriptors. Message classes constructed by the protocol compiler will provide them for you. However, if you are implementing Message on your own, or if you are writing a program which can operate on totally arbitrary types and needs to load them from some sort of database, you might need to.

Since Descriptors are composed of a whole lot of cross-linked bits of data that would be a pain to put together manually, the DescriptorPool class is provided to make the process easier. It can take a FileDescriptorProto (defined in descriptor.proto), validate it, and convert it to a set of nicely cross-linked Descriptors.

DescriptorPool also helps with memory management. Descriptors are composed of many objects containing static data and pointers to each other. In all likelihood, when it comes time to delete this data, you'll want to delete it all at once. In fact, it is not uncommon to have a whole pool of descriptors all cross-linked with each other which you wish to delete all at once. This class represents such a pool, and handles the memory management for you.

You can also search for descriptors within a DescriptorPool by name, and extensions by number.

Members

	DescriptorPool() *Create a normal, empty DescriptorPool.*
explicit	**DescriptorPool**(DescriptorDatabase * fallback_database, ErrorCollector * error_collector = NULL)
	~DescriptorPool()
const FileDescriptor *	**FindFileByName**(const string & name) const *Find a FileDescriptor in the pool by file name. more...*
const FileDescriptor *	**FindFileContainingSymbol**(const string & symbol_name) const *Find the FileDescriptor in the pool which defines the given symbol. more...*
static const DescriptorPool *	**generated_pool**() *Get a pointer to the generated pool. more...*

Looking up descriptors

These find descriptors by fully-qualified name. These will find both top-level descriptors and nested descriptors. They return NULL if not found.

const Descriptor *	**FindMessageTypeByName**(const string & name) const
const FieldDescriptor *	**FindFieldByName**(const string & name) const
const FieldDescriptor *	**FindExtensionByName**(const string & name) const
const EnumDescriptor *	**FindEnumTypeByName**(const string & name) const
const EnumValueDescriptor *	**FindEnumValueByName**(const string & name) const
const ServiceDescriptor *	**FindServiceByName**(const string & name) const
const MethodDescriptor *	**FindMethodByName**(const string & name) const
const FieldDescriptor *	**FindExtensionByNumber**(const Descriptor * extendee, int number) const *Finds an extension of the given type by number. more...*

Building descriptors

const FileDescriptor *	**BuildFile**(const FileDescriptorProto & proto) *Convert the FileDescriptorProto to real descriptors and place them in this DescriptorPool. more...*
const FileDescriptor *	**BuildFileCollectingErrors**(const FileDescriptorProto & proto, ErrorCollector * error_collector) *Same as BuildFile() except errors are sent to the given ErrorCollector.*

Internal stuff

These methods MUST NOT be called from outside the proto2 library. These methods may contain hidden pitfalls and may be removed in a future library version.

explicit	**DescriptorPool**(const DescriptorPool * underlay) *DEPRECATED: Use of underlays can lead to many subtle gotchas. more...*

const FileDescriptor *	**InternalBuildGeneratedFile**(const void * data, int size) *Called by generated classes at init time. more...*
void	**InternalDontEnforceDependencies**() *For internal use only: Changes the behavior of BuildFile() such that it allows the file to make reference to message types declared in other files which it did not officially declare as dependencies.*
void	**internal_set_underlay**(const DescriptorPool * underlay) *For internal use only.*
static DescriptorPool *	**internal_generated_pool**() *For internal use only: Gets a non-const pointer to the generated pool. more...*

```
const FileDescriptor *
    DescriptorPool::FindFileByName(
        const string & name) const
```

Find a FileDescriptor in the pool by file name.

Returns NULL if not found.

```
const FileDescriptor *
    DescriptorPool::FindFileContainingSymbol(
        const string & symbol_name) const
```

Find the FileDescriptor in the pool which defines the given symbol.

If any of the Find*ByName() methods below would succeed, then this is equivalent to calling that method and calling the result's file() method. Otherwise this returns NULL.

```
static const DescriptorPool *
    DescriptorPool::generated_pool()
```

Get a pointer to the generated pool.

Generated protocol message classes which are compiled into the binary will allocate their descriptors in this pool. Do not add your own descriptors to this pool.

```
const FieldDescriptor *
    DescriptorPool::FindExtensionByNumber(
        const Descriptor * extendee,
        int number) const
```

Finds an extension of the given type by number.

The extendee must be a member of this DescriptorPool or one of its underlays.

```
const FileDescriptor *
```

```
    DescriptorPool::BuildFile(
        const FileDescriptorProto & proto)
```

Convert the FileDescriptorProto to real descriptors and place them in this DescriptorPool.

All dependencies of the file must already be in the pool. Returns the resulting FileDescriptor, or NULL if there were problems with the input (e.g. the message was invalid, or dependencies were missing). Details about the errors are written to GOOGLE_LOG(ERROR).

```
explicit DescriptorPool::DescriptorPool(
        const DescriptorPool * underlay)
```

DEPRECATED: Use of underlays can lead to many subtle gotchas.

Instead, try to formulate what you want to do in terms of DescriptorDatabases. This constructor will be removed soon.

Create a DescriptorPool which is overlaid on top of some other pool. If you search for a descriptor in the overlay and it is not found, the underlay will be searched as a backup. If the underlay has its own underlay, that will be searched next, and so on. This also means that files built in the overlay will be cross-linked with the underlay's descriptors if necessary. The underlay remains property of the caller; it must remain valid for the lifetime of the newly-constructed pool.

Example: Say you want to parse a .proto file at runtime in order to use its type with a DynamicMessage. Say this .proto file has dependencies, but you know that all the dependencies will be things that are already compiled into the binary. For ease of use, you'd like to load the types right out of generated_pool() rather than have to parse redundant copies of all these .protos and runtime. But, you don't want to add the parsed types directly into generated_pool(): this is not allowed, and would be bad design anyway. So, instead, you could use generated_pool() as an underlay for a new DescriptorPool in which you add only the new file.

```
const FileDescriptor *
    DescriptorPool::InternalBuildGeneratedFile(
        const void * data,
        int size)
```

Called by generated classes at init time.

Do NOT call this in your own code!

```
static DescriptorPool * DescriptorPool::internal_generated_pool()
```

For internal use only: Gets a non-const pointer to the generated pool.

This is called at static-initialization time only, so thread-safety is not a concern. If both an underlay and a fallback database are present, the fallback database takes precedence.

class DescriptorPool::ErrorCollector

```
#include <google/protobuf/descriptor.h>
namespace google::protobuf
```

When converting a FileDescriptorProto to a FileDescriptor, various errors might be detected in the input.

The caller may handle these programmatically by implementing an ErrorCollector.

Members	
enum	**ErrorLocation** *These constants specify what exact part of the construct is broken. more...*
	ErrorCollector()

`virtual`	`~ErrorCollector()`
`virtual void`	`AddError(const string & filename, const string & element_name, const Message * descriptor, ErrorLocation location, const string & message) = 0` *Reports an error in the FileDescriptorProto.*


```
enum ErrorCollector::ErrorLocation {
  NAME,
  NUMBER,
  TYPE,
  EXTENDEE,
  DEFAULT_VALUE,
  INPUT_TYPE,
  OUTPUT_TYPE,
  OTHER
}
```

These constants specify what exact part of the construct is broken.

This is useful e.g. for mapping the error back to an exact location in a .proto file.

NAME	the symbol name, or the package name for files
NUMBER	field or extension range number
TYPE	field type
EXTENDEE	field extendee
DEFAULT_VALUE	field default value
INPUT_TYPE	method input type
OUTPUT_TYPE	method output type
OTHER	some other problem

descriptor.pb.h

```
#include <google/protobuf/descriptor.pb.h>
namespace google::protobuf
```

Protocol buffer representations of descriptors.

This file defines a set of protocol message classes which represent the same information represented by the classes defined in descriptor.h. You can convert a FileDescriptorProto to a FileDescriptor using the DescriptorPool class. Thus, the classes in this file allow protocol type definitions to be communicated efficiently between processes.

The protocol compiler currently doesn't support auto-generated documentation, hence this page contains no descriptions. This file was generated by the protocol compiler from `descriptor.proto`, whose contents are as follows:

```
// Protocol Buffers - Google's data interchange format
// Copyright 2008 Google Inc.
// http://code.google.com/p/protobuf/
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

// Author: kenton@google.com (Kenton Varda)
//  Based on original Protocol Buffers design by
//  Sanjay Ghemawat, Jeff Dean, and others.
//
// The messages in this file describe the definitions found in .proto files.
// A valid .proto file can be translated directly to a FileDescriptorProto
// without any other information (e.g. without reading its imports).



package google.protobuf;
option java_package = "com.google.protobuf";
option java_outer_classname = "DescriptorProtos";

// descriptor.proto must be optimized for speed because reflection-based
// algorithms don't work during bootstrapping.
option optimize_for = SPEED;

// The protocol compiler can output a FileDescriptorSet containing the .proto
// files it parses.
message FileDescriptorSet {
  repeated FileDescriptorProto file = 1;
}

// Describes a complete .proto file.
message FileDescriptorProto {
  optional string name = 1;       // file name, relative to root of source tree
  optional string package = 2;    // e.g. "foo", "foo.bar", etc.

  // Names of files imported by this file.
  repeated string dependency = 3;

  // All top-level definitions in this file.
  repeated DescriptorProto message_type = 4;
  repeated EnumDescriptorProto enum_type = 5;
```

```
  repeated ServiceDescriptorProto service = 6;
  repeated FieldDescriptorProto extension = 7;

  optional FileOptions options = 8;
}

// Describes a message type.
message DescriptorProto {
  optional string name = 1;

  repeated FieldDescriptorProto field = 2;
  repeated FieldDescriptorProto extension = 6;

  repeated DescriptorProto nested_type = 3;
  repeated EnumDescriptorProto enum_type = 4;

  message ExtensionRange {
    optional int32 start = 1;
    optional int32 end = 2;
  }
  repeated ExtensionRange extension_range = 5;

  optional MessageOptions options = 7;
}

// Describes a field within a message.
message FieldDescriptorProto {
  enum Type {
    // 0 is reserved for errors.
    // Order is weird for historical reasons.
    TYPE_DOUBLE         = 1;
    TYPE_FLOAT          = 2;
    TYPE_INT64          = 3;   // Not ZigZag encoded.  Negative numbers
                               // take 10 bytes.  Use TYPE_SINT64 if negative
                               // values are likely.
    TYPE_UINT64         = 4;
    TYPE_INT32          = 5;   // Not ZigZag encoded.  Negative numbers
                               // take 10 bytes.  Use TYPE_SINT32 if negative
                               // values are likely.
    TYPE_FIXED64        = 6;
    TYPE_FIXED32        = 7;
    TYPE_BOOL           = 8;
    TYPE_STRING         = 9;
    TYPE_GROUP          = 10;  // Tag-delimited aggregate.
    TYPE_MESSAGE        = 11;  // Length-delimited aggregate.

    // New in version 2.
    TYPE_BYTES          = 12;
    TYPE_UINT32         = 13;
    TYPE_ENUM           = 14;
    TYPE_SFIXED32       = 15;
    TYPE_SFIXED64       = 16;
    TYPE_SINT32         = 17;  // Uses ZigZag encoding.
    TYPE_SINT64         = 18;  // Uses ZigZag encoding.
  };

  enum Label {
    // 0 is reserved for errors
    LABEL_OPTIONAL      = 1;
    LABEL_REQUIRED      = 2;
    LABEL_REPEATED      = 3;
    // TODO(sanjay): Should we add LABEL_MAP?
  };

  optional string name = 1;
  optional int32 number = 3;
  optional Label label = 4;

  // If type_name is set, this need not be set.  If both this and type_name
  // are set, this must be either TYPE_ENUM or TYPE_MESSAGE.
  optional Type type = 5;

  // For message and enum types, this is the name of the type.  If the name
  // starts with a '.', it is fully-qualified.  Otherwise, C++-like scoping
```

```
  // rules are used to find the type (i.e. first the nested types within this
  // message are searched, then within the parent, on up to the root
  // namespace).
  optional string type_name = 6;

  // For extensions, this is the name of the type being extended.  It is
  // resolved in the same manner as type_name.
  optional string extendee = 2;

  // For numeric types, contains the original text representation of the value.
  // For booleans, "true" or "false".
  // For strings, contains the default text contents (not escaped in any way).
  // For bytes, contains the C escaped value.  All bytes >= 128 are escaped.
  // TODO(kenton):  Base-64 encode?
  optional string default_value = 7;

  optional FieldOptions options = 8;
}

// Describes an enum type.
message EnumDescriptorProto {
  optional string name = 1;

  repeated EnumValueDescriptorProto value = 2;

  optional EnumOptions options = 3;
}

// Describes a value within an enum.
message EnumValueDescriptorProto {
  optional string name = 1;
  optional int32 number = 2;

  optional EnumValueOptions options = 3;
}

// Describes a service.
message ServiceDescriptorProto {
  optional string name = 1;
  repeated MethodDescriptorProto method = 2;

  optional ServiceOptions options = 3;
}

// Describes a method of a service.
message MethodDescriptorProto {
  optional string name = 1;

  // Input and output type names.  These are resolved in the same way as
  // FieldDescriptorProto.type_name, but must refer to a message type.
  optional string input_type = 2;
  optional string output_type = 3;

  optional MethodOptions options = 4;
}

// ===================================================================
// Options

// Each of the definitions above may have "options" attached.  These are
// just annotations which may cause code to be generated slightly differently
// or may contain hints for code that manipulates protocol messages.

// TODO(kenton):  Allow extensions to options.

message FileOptions {

  // Sets the Java package where classes generated from this .proto will be
  // placed.  By default, the proto package is used, but this is often
  // inappropriate because proto packages do not normally start with backwards
  // domain names.
  optional string java_package = 1;
```

```
  // If set, all the classes from the .proto file are wrapped in a single
  // outer class with the given name.  This applies to both Proto1
  // (equivalent to the old "--one_java_file" option) and Proto2 (where
  // a .proto always translates to a single class, but you may want to
  // explicitly choose the class name).
  optional string java_outer_classname = 8;

  // If set true, then the Java code generator will generate a separate .java
  // file for each top-level message, enum, and service defined in the .proto
  // file.  Thus, these types will *not* be nested inside the outer class
  // named by java_outer_classname.  However, the outer class will still be
  // generated to contain the file's getDescriptor() method as well as any
  // top-level extensions defined in the file.
  optional bool java_multiple_files = 10 [default=false];

  // Generated classes can be optimized for speed or code size.
  enum OptimizeMode {
    SPEED = 1;      // Generate complete code for parsing, serialization, etc.
    CODE_SIZE = 2;  // Use ReflectionOps to implement these methods.
  }
  optional OptimizeMode optimize_for = 9 [default=CODE_SIZE];
}

message MessageOptions {
  // Set true to use the old proto1 MessageSet wire format for extensions.
  // This is provided for backwards-compatibility with the MessageSet wire
  // format.  You should not use this for any other reason:  It's less
  // efficient, has fewer features, and is more complicated.
  //
  // The message must be defined exactly as follows:
  //   message Foo {
  //     option message_set_wire_format = true;
  //     extensions 4 to max;
  //   }
  // Note that the message cannot have any defined fields; MessageSets only
  // have extensions.
  //
  // All extensions of your type must be singular messages; e.g. they cannot
  // be int32s, enums, or repeated messages.
  //
  // Because this is an option, the above two restrictions are not enforced by
  // the protocol compiler.
  optional bool message_set_wire_format = 1 [default=false];
}

message FieldOptions {
  // The ctype option instructs the C++ code generator to use a different
  // representation of the field than it normally would.  See the specific
  // options below.  This option is not yet implemented in the open source
  // release -- sorry, we'll try to include it in a future version!
  optional CType ctype = 1;
  enum CType {
    CORD = 1;

    STRING_PIECE = 2;
  }

  // EXPERIMENTAL.  DO NOT USE.
  // For "map" fields, the name of the field in the enclosed type that
  // is the key for this map.  For example, suppose we have:
  //   message Item {
  //     required string name = 1;
  //     required string value = 2;
  //   }
  //   message Config {
  //     repeated Item items = 1 [experimental_map_key="name"];
  //   }
  // In this situation, the map key for Item will be set to "name".
  // TODO: Fully-implement this, then remove the "experimental_" prefix.
  optional string experimental_map_key = 9;
}

message EnumOptions {
}
```

```
message EnumValueOptions {
}

message ServiceOptions {

  // Note:  Field numbers 1 through 32 are reserved for Google's internal RPC
  //   framework.  We apologize for hoarding these numbers to ourselves, but
  //   we were already using them long before we decided to release Protocol
  //   Buffers.
}

message MethodOptions {

  // Note:  Field numbers 1 through 32 are reserved for Google's internal RPC
  //   framework.  We apologize for hoarding these numbers to ourselves, but
  //   we were already using them long before we decided to release Protocol
  //   Buffers.
}
```

Classes in this file

`FileDescriptorSet`

See the docs for descriptor.pb.h for more information about this class.

`FileDescriptorProto`

See the docs for descriptor.pb.h for more information about this class.

`DescriptorProto_ExtensionRange`

See the docs for descriptor.pb.h for more information about this class.

`DescriptorProto`

See the docs for descriptor.pb.h for more information about this class.

`FieldDescriptorProto`

See the docs for descriptor.pb.h for more information about this class.

`EnumDescriptorProto`

See the docs for descriptor.pb.h for more information about this class.

`EnumValueDescriptorProto`

See the docs for descriptor.pb.h for more information about this class.

`ServiceDescriptorProto`

See the docs for descriptor.pb.h for more information about this class.

`MethodDescriptorProto`

See the docs for descriptor.pb.h for more information about this class.

`FileOptions`

See the docs for descriptor.pb.h for more information about this class.

`MessageOptions`

See the docs for descriptor.pb.h for more information about this class.

`FieldOptions`

See the docs for descriptor.pb.h for more information about this class.

`EnumOptions`

See the docs for descriptor.pb.h for more information about this class.

`EnumValueOptions`

See the docs for descriptor.pb.h for more information about this class.

`ServiceOptions`

See the docs for descriptor.pb.h for more information about this class.

`MethodOptions`

See the docs for descriptor.pb.h for more information about this class.

File Members

These definitions are not part of any class.

enum	**FieldDescriptorProto_Type** *more...*
enum	**FieldDescriptorProto_Label** *more...*
enum	**FileOptions_OptimizeMode** *more...*
enum	**FieldOptions_CType** *more...*
void	**protobuf_BuildDesc_google_2fprotobuf_2fdescriptor_2eproto** () *Internal implementation detail -- do not call this.*
const EnumDescriptor *	**FieldDescriptorProto_Type_descriptor** ()
bool	**FieldDescriptorProto_Type_IsValid** (int value)
const EnumDescriptor *	**FieldDescriptorProto_Label_descriptor** ()
bool	**FieldDescriptorProto_Label_IsValid** (int value)
const EnumDescriptor *	**FileOptions_OptimizeMode_descriptor** ()
bool	**FileOptions_OptimizeMode_IsValid** (int value)
const EnumDescriptor *	**FieldOptions_CType_descriptor** ()
bool	**FieldOptions_CType_IsValid** (int value)
const FieldDescriptorProto_Type	**FieldDescriptorProto_Type_Type_MIN** = FieldDescriptorProto_Type_TYPE_DOUBLE
const FieldDescriptorProto_Type	**FieldDescriptorProto_Type_Type_MAX** = FieldDescriptorProto_Type_TYPE_SINT64
const FieldDescriptorProto_Label	**FieldDescriptorProto_Label_Label_MIN** = FieldDescriptorProto_Label_LABEL_OPTIONAL
const FieldDescriptorProto_Label	**FieldDescriptorProto_Label_Label_MAX** = FieldDescriptorProto_Label_LABEL_REPEATED
const FileOptions_OptimizeMode	**FileOptions_OptimizeMode_OptimizeMode_MIN** = FileOptions_OptimizeMode_SPEED
const FileOptions_OptimizeMode	**FileOptions_OptimizeMode_OptimizeMode_MAX** = FileOptions_OptimizeMode_CODE_SIZE
const FieldOptions_CType	**FieldOptions_CType_CType_MIN** = FieldOptions_CType_CORD
const FieldOptions_CType	**FieldOptions_CType_CType_MAX** = FieldOptions_CType_STRING_PIECE

```
enum protobuf::FieldDescriptorProto_Type {
  FieldDescriptorProto_Type_TYPE_DOUBLE = 1,
  FieldDescriptorProto_Type_TYPE_FLOAT = 2,
  FieldDescriptorProto_Type_TYPE_INT64 = 3,
  FieldDescriptorProto_Type_TYPE_UINT64 = 4,
  FieldDescriptorProto_Type_TYPE_INT32 = 5,
  FieldDescriptorProto_Type_TYPE_FIXED64 = 6,
  FieldDescriptorProto_Type_TYPE_FIXED32 = 7,
  FieldDescriptorProto_Type_TYPE_BOOL = 8,
  FieldDescriptorProto_Type_TYPE_STRING = 9,
  FieldDescriptorProto_Type_TYPE_GROUP = 10,
  FieldDescriptorProto_Type_TYPE_MESSAGE = 11,
  FieldDescriptorProto_Type_TYPE_BYTES = 12,
```

```
  FieldDescriptorProto_Type_TYPE_UINT32 = 13,
  FieldDescriptorProto_Type_TYPE_ENUM = 14,
  FieldDescriptorProto_Type_TYPE_SFIXED32 = 15,
  FieldDescriptorProto_Type_TYPE_SFIXED64 = 16,
  FieldDescriptorProto_Type_TYPE_SINT32 = 17,
  FieldDescriptorProto_Type_TYPE_SINT64 = 18
}
```

FieldDescriptorProto_Type_TYPE_DOUBLE	
FieldDescriptorProto_Type_TYPE_FLOAT	
FieldDescriptorProto_Type_TYPE_INT64	
FieldDescriptorProto_Type_TYPE_UINT64	
FieldDescriptorProto_Type_TYPE_INT32	
FieldDescriptorProto_Type_TYPE_FIXED64	
FieldDescriptorProto_Type_TYPE_FIXED32	
FieldDescriptorProto_Type_TYPE_BOOL	
FieldDescriptorProto_Type_TYPE_STRING	
FieldDescriptorProto_Type_TYPE_GROUP	
FieldDescriptorProto_Type_TYPE_MESSAGE	
FieldDescriptorProto_Type_TYPE_BYTES	
FieldDescriptorProto_Type_TYPE_UINT32	
FieldDescriptorProto_Type_TYPE_ENUM	
FieldDescriptorProto_Type_TYPE_SFIXED32	
FieldDescriptorProto_Type_TYPE_SFIXED64	
FieldDescriptorProto_Type_TYPE_SINT32	
FieldDescriptorProto_Type_TYPE_SINT64	

```
enum protobuf::FieldDescriptorProto_Label {
  FieldDescriptorProto_Label_LABEL_OPTIONAL = 1,
  FieldDescriptorProto_Label_LABEL_REQUIRED = 2,
  FieldDescriptorProto_Label_LABEL_REPEATED = 3
}
```

FieldDescriptorProto_Label_LABEL_OPTIONAL	
FieldDescriptorProto_Label_LABEL_REQUIRED	
FieldDescriptorProto_Label_LABEL_REPEATED	

```
enum protobuf::FileOptions_OptimizeMode {
  FileOptions_OptimizeMode_SPEED = 1,
  FileOptions_OptimizeMode_CODE_SIZE = 2
}
```

FileOptions_OptimizeMode_SPEED	
FileOptions_OptimizeMode_CODE_SIZE	

```
enum protobuf::FieldOptions_CType {
  FieldOptions_CType_CORD = 1,
  FieldOptions_CType_STRING_PIECE = 2
}
```

FieldOptions_CType_CORD	
FieldOptions_CType_STRING_PIECE	

class FileDescriptorSet: public Message

```
#include <google/protobuf/descriptor.pb.h>
namespace google::protobuf
```

See the docs for descriptor.pb.h for more information about this class.

Members	
	FileDescriptorSet()
virtual	**~FileDescriptorSet**()
	FileDescriptorSet(const FileDescriptorSet & from)
FileDescriptorSet &	**operator=**(const FileDescriptorSet & from)
const UnknownFieldSet &	**unknown_fields**() const
UnknownFieldSet *	**mutable_unknown_fields**()
static const FileDescriptorSet &	**default_instance**()
static const Descriptor *	**descriptor**()
implements Message	
virtual FileDescriptorSet *	**New**() const *Construct a new instance of the same type. more...*
virtual void	**CopyFrom**(const Message & from) *Make this message into a copy of the given message. more...*
virtual void	**MergeFrom**(const Message & from) *Merge the fields from the given message into this message. more...*
void	**CopyFrom**(const FileDescriptorSet & from)
void	**MergeFrom**(const FileDescriptorSet & from)
virtual void	**Clear**() *Clear all fields of the message and set them to their default values. more...*
virtual bool	**IsInitialized**() const *Quickly check if all required fields have values set.*
virtual int	**ByteSize**() const *Computes the serialized size of the message. more...*
virtual bool	**MergePartialFromCodedStream**(io::CodedInputStream * input) *Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input. more...*
virtual bool	**SerializeWithCachedSizes**(io::CodedOutputStream *

	`output) const` *Serializes the message without recomputing the size. more...*
`virtual int`	**`GetCachedSize`**`() const` *Returns the result of the last call to ByteSize(). more...*
`virtual const Descriptor *`	**`GetDescriptor`**`() const` *Get a Descriptor for this message's type. more...*
`virtual const Reflection *`	**`GetReflection`**`() const` *Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time). more...*
accessors	
`int`	**`file_size`**`() const` *repeated .google.protobuf.FileDescriptorProto file = 1;*
`void`	**`clear_file`**`()`
`const RepeatedPtrField< FileDescriptorProto > &`	**`file`**`() const`
`RepeatedPtrField< FileDescriptorProto > *`	**`mutable_file`**`()`
`const FileDescriptorProto &`	**`file`**`(int index) const`
`FileDescriptorProto *`	**`mutable_file`**`(int index)`
`FileDescriptorProto *`	**`add_file`**`()`

`virtual FileDescriptorSet * FileDescriptorSet::New() const`

Construct a new instance of the same type.

Ownership is passed to the caller.

`virtual void FileDescriptorSet::CopyFrom( const Message & from)`

Make this message into a copy of the given message.

The given message must have the same descriptor, but need not necessarily be the same class. By default this is just implemented as "Clear(); MergeFrom(from);".

`virtual void FileDescriptorSet::MergeFrom( const Message & from)`

Merge the fields from the given message into this message.

Singular fields will be overwritten, except for embedded messages which will be merged. Repeated fields will be concatenated. The given message must be of the same type as this message (i.e. the exact same class).

`virtual void FileDescriptorSet::Clear()`

Clear all fields of the message and set them to their default values.

Clear() avoids freeing memory, assuming that any memory allocated to hold parts of the message will be needed again to hold the next message. If you actually want to free the memory used by a Message, you must delete it.

`virtual int FileDescriptorSet::ByteSize() const`

Computes the serialized size of the message.

This recursively calls ByteSize() on all embedded messages. If a subclass does not override this, it MUST override SetCachedSize().

`virtual bool FileDescriptorSet::MergePartialFromCodedStream( io::CodedInputStream * input)`

Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input.

MergeFromCodedStream() is just implemented as MergePartialFromCodedStream() followed by IsInitialized().

`virtual bool FileDescriptorSet::SerializeWithCachedSizes( io::CodedOutputStream * output) const`

Serializes the message without recomputing the size.

The message must not have changed since the last call to ByteSize(); if it has, the results are undefined.

`virtual int FileDescriptorSet::GetCachedSize() const`

Returns the result of the last call to ByteSize().

An embedded message's size is needed both to serialize it (because embedded messages are length-delimited) and to compute the outer message's size. Caching the size avoids computing it multiple times.

ByteSize() does not automatically use the cached size when available because this would require invalidating it every time the message was modified, which would be too hard and expensive. (E.g. if a deeply-nested sub-message is changed, all of its parents' cached sizes would need to be invalidated, which is too much work for an otherwise inlined setter method.)

`virtual const Descriptor * FileDescriptorSet::GetDescriptor() const`

Get a Descriptor for this message's type.

This describes what fields the message contains, the types of those fields, etc.

`virtual const Reflection * FileDescriptorSet::GetReflection() const`

Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time).

This object remains property of the Message.

---

class FileDescriptorProto: public Message

```
#include <google/protobuf/descriptor.pb.h >
namespace google::protobuf
```

See the docs for descriptor.pb.h for more information about this class.

Members

	FileDescriptorProto ()
virtual	**~FileDescriptorProto** ()
	FileDescriptorProto (const FileDescriptorProto & from)
FileDescriptorProto &	**operator=** (const FileDescriptorProto & from)
const UnknownFieldSet &	**unknown_fields** () const
UnknownFieldSet *	**mutable_unknown_fields** ()
static const FileDescriptorProto &	**default_instance** ()
static const Descriptor *	**descriptor** ()

implements Message

virtual FileDescriptorProto *	**New**() const *Construct a new instance of the same type. more...*
virtual void	**CopyFrom**(const Message & from) *Make this message into a copy of the given message. more...*
virtual void	**MergeFrom**(const Message & from) *Merge the fields from the given message into this message. more...*
void	**CopyFrom**(const FileDescriptorProto & from)
void	**MergeFrom**(const FileDescriptorProto & from)
virtual void	**Clear**() *Clear all fields of the message and set them to their default values. more...*
virtual bool	**IsInitialized**() const *Quickly check if all required fields have values set.*
virtual int	**ByteSize**() const *Computes the serialized size of the message. more...*
virtual bool	**MergePartialFromCodedStream** (io::CodedInputStream * input) *Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input. more...*
virtual bool	**SerializeWithCachedSizes** (io::CodedOutputStream * output) const *Serializes the message without recomputing the size. more...*
virtual int	**GetCachedSize**() const *Returns the result of the last call to ByteSize(). more...*
virtual const Descriptor *	**GetDescriptor**() const *Get a Descriptor for this message's type. more...*
virtual const Reflection *	**GetReflection**() const *Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time). more...*

accessors

bool	**has_name**() const *optional string name = 1;*
void	**clear_name**()
const ::std::string &	**name**() const
void	**set_name**(const ::std::string & value)
void	**set_name**(const char * value)
inline::std::string *	**mutable_name**()
bool	**has_package**() const *optional string package = 2;*
void	**clear_package**()
const ::std::string &	**package**() const
void	**set_package**(const ::std::string & value)
void	**set_package**(const char * value)
inline::std::string *	**mutable_package**()
int	**dependency_size**() const *repeated string dependency = 3;*
void	**clear_dependency**()
const RepeatedPtrField< ::std::string > &	**dependency**() const
RepeatedPtrField< ::std::string > *	**mutable_dependency**()
const ::std::string &	**dependency**(int index) const
inline::std::string *	**mutable_dependency**(int index)
void	**set_dependency**(int index, const ::std::string & value)
void	**set_dependency**(int index, const char * value)
inline::std::string *	**add_dependency**()
void	**add_dependency**(const ::std::string & value)
void	**add_dependency**(const char * value)
int	**message_type_size**() const *repeated .google.protobuf.DescriptorProto message_type = 4;*
void	**clear_message_type**()
const RepeatedPtrField< DescriptorProto > &	**message_type**() const
RepeatedPtrField< DescriptorProto > *	**mutable_message_type**()
const DescriptorProto &	**message_type**(int index) const
DescriptorProto *	**mutable_message_type**(int index)
DescriptorProto *	**add_message_type**()

int	**enum_type_size**() const *repeated .google.protobuf.EnumDescriptorProto enum_type = 5;*
void	**clear_enum_type**()
const RepeatedPtrField< EnumDescriptorProto > &	**enum_type**() const
RepeatedPtrField< EnumDescriptorProto > *	**mutable_enum_type**()
const EnumDescriptorProto &	**enum_type**(int index) const
EnumDescriptorProto *	**mutable_enum_type**(int index)
EnumDescriptorProto *	**add_enum_type**()
int	**service_size**() const *repeated .google.protobuf.ServiceDescriptorProto service = 6;*
void	**clear_service**()
const RepeatedPtrField< ServiceDescriptorProto > &	**service**() const
RepeatedPtrField< ServiceDescriptorProto > *	**mutable_service**()
const ServiceDescriptorProto &	**service**(int index) const
ServiceDescriptorProto *	**mutable_service**(int index)
ServiceDescriptorProto *	**add_service**()
int	**extension_size**() const *repeated .google.protobuf.FieldDescriptorProto extension = 7;*
void	**clear_extension**()
const RepeatedPtrField< FieldDescriptorProto > &	**extension**() const
RepeatedPtrField< FieldDescriptorProto > *	**mutable_extension**()
const FieldDescriptorProto &	**extension**(int index) const
FieldDescriptorProto *	**mutable_extension**(int index)
FieldDescriptorProto *	**add_extension**()
bool	**has_options**() const *optional .google.protobuf.FileOptions options = 8;*
void	**clear_options**()
const FileOptions &	**options**() const
FileOptions *	**mutable_options**()

virtual FileDescriptorProto * FileDescriptorProto::New() const

Construct a new instance of the same type.

Ownership is passed to the caller.

`virtual void FileDescriptorProto::CopyFrom( const Message & from)`

Make this message into a copy of the given message.

The given message must have the same descriptor, but need not necessarily be the same class. By default this is just implemented as "Clear(); MergeFrom(from);".

`virtual void FileDescriptorProto::MergeFrom( const Message & from)`

Merge the fields from the given message into this message.

Singular fields will be overwritten, except for embedded messages which will be merged. Repeated fields will be concatenated. The given message must be of the same type as this message (i.e. the exact same class).

`virtual void FileDescriptorProto::Clear()`

Clear all fields of the message and set them to their default values.

Clear() avoids freeing memory, assuming that any memory allocated to hold parts of the message will be needed again to hold the next message. If you actually want to free the memory used by a Message, you must delete it.

`virtual int FileDescriptorProto::ByteSize() const`

Computes the serialized size of the message.

This recursively calls ByteSize() on all embedded messages. If a subclass does not override this, it MUST override SetCachedSize().

`virtual bool FileDescriptorProto::MergePartialFromCodedStream( io::CodedInputStream * input)`

Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input.

MergeFromCodedStream() is just implemented as MergePartialFromCodedStream() followed by IsInitialized().

`virtual bool FileDescriptorProto::SerializeWithCachedSizes( io::CodedOutputStream * output) const`

Serializes the message without recomputing the size.

The message must not have changed since the last call to ByteSize(); if it has, the results are undefined.

`virtual int FileDescriptorProto::GetCachedSize() const`

Returns the result of the last call to ByteSize().

An embedded message's size is needed both to serialize it (because embedded messages are length-delimited) and to compute the outer message's size. Caching the size avoids computing it multiple times.

ByteSize() does not automatically use the cached size when available because this would require invalidating it every time the message was modified, which would be too hard and expensive. (E.g. if a deeply-nested sub-message is changed, all of its parents' cached sizes would need to be invalidated, which is too much work for an otherwise inlined setter method.)

virtual const Descriptor *
FileDescriptorProto::GetDescriptor() const

Get a Descriptor for this message's type.

This describes what fields the message contains, the types of those fields, etc.

---

virtual const Reflection *
FileDescriptorProto::GetReflection() const

Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time).

This object remains property of the Message.

class DescriptorProto_ExtensionRange: public Message

```
#include <google/protobuf/descriptor.pb.h >
namespace google::protobuf
```

See the docs for descriptor.pb.h for more information about this class.

Members

	DescriptorProto_ExtensionRange ()
virtual	**~DescriptorProto_ExtensionRange** ()
	DescriptorProto_ExtensionRange (const DescriptorProto_ExtensionRange & from)
DescriptorProto_ExtensionRange &	**operator=** (const DescriptorProto_ExtensionRange & from)
const UnknownFieldSet &	**unknown_fields** () const
UnknownFieldSet *	**mutable_unknown_fields** ()
static const DescriptorProto_ExtensionRange &	**default_instance** ()
static const Descriptor *	**descriptor** ()

implements Message

virtual DescriptorProto_ExtensionRange *	**New**() const *Construct a new instance of the same type. more...*
virtual void	**CopyFrom** (const Message & from) *Make this message into a copy of the given message. more...*
virtual void	**MergeFrom** (const Message & from) *Merge the fields from the given message into this message. more...*
void	**CopyFrom** (const DescriptorProto_ExtensionRange & from)
void	**MergeFrom** (const DescriptorProto_ExtensionRange & from)
virtual void	**Clear** () *Clear all fields of the message and set them to their default values. more...*

virtual bool	**IsInitialized**() const *Quickly check if all required fields have values set.*
virtual int	**ByteSize**() const *Computes the serialized size of the message. more...*
virtual bool	**MergePartialFromCodedStream**(io::CodedInputStream * input) *Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input. more...*
virtual bool	**SerializeWithCachedSizes**(io::CodedOutputStream * output) const *Serializes the message without recomputing the size. more...*
virtual int	**GetCachedSize**() const *Returns the result of the last call to ByteSize(). more...*
virtual const Descriptor *	**GetDescriptor**() const *Get a Descriptor for this message's type. more...*
virtual const Reflection *	**GetReflection**() const *Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time). more...*
accessors	
bool	**has_start**() const *optional int32 start = 1;*
void	**clear_start**()
int32	**start**() const
void	**set_start**(int32 value)
bool	**has_end**() const *optional int32 end = 2;*
void	**clear_end**()
int32	**end**() const
void	**set_end**(int32 value)

---

virtual DescriptorProto_ExtensionRange * DescriptorProto_ExtensionRange::New() const

Construct a new instance of the same type.

Ownership is passed to the caller.

---

virtual void DescriptorProto_ExtensionRange::CopyFrom(const Message & from)

Make this message into a copy of the given message.

The given message must have the same descriptor, but need not necessarily be the same class. By default this is just implemented as "Clear(); MergeFrom(from);".

---

virtual void DescriptorProto_ExtensionRange::MergeFrom(

```
        const Message & from)
```

Merge the fields from the given message into this message.

Singular fields will be overwritten, except for embedded messages which will be merged. Repeated fields will be concatenated. The given message must be of the same type as this message (i.e. the exact same class).

---

virtual void DescriptorProto_ExtensionRange::Clear()

Clear all fields of the message and set them to their default values.

Clear() avoids freeing memory, assuming that any memory allocated to hold parts of the message will be needed again to hold the next message. If you actually want to free the memory used by a Message, you must delete it.

---

virtual int DescriptorProto_ExtensionRange::ByteSize() const

Computes the serialized size of the message.

This recursively calls ByteSize() on all embedded messages. If a subclass does not override this, it MUST override SetCachedSize().

---

virtual bool DescriptorProto_ExtensionRange::MergePartialFromCodedStream(io::CodedInputStream * input)

Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input.

MergeFromCodedStream() is just implemented as MergePartialFromCodedStream() followed by IsInitialized().

---

virtual bool DescriptorProto_ExtensionRange::SerializeWithCachedSizes(io::CodedOutputStream * output) const

Serializes the message without recomputing the size.

The message must not have changed since the last call to ByteSize(); if it has, the results are undefined.

---

virtual int DescriptorProto_ExtensionRange::GetCachedSize() const

Returns the result of the last call to ByteSize().

An embedded message's size is needed both to serialize it (because embedded messages are length-delimited) and to compute the outer message's size. Caching the size avoids computing it multiple times.

ByteSize() does not automatically use the cached size when available because this would require invalidating it every time the message was modified, which would be too hard and expensive. (E.g. if a deeply-nested sub-message is changed, all of its parents' cached sizes would need to be invalidated, which is too much work for an otherwise inlined setter method.)

---

virtual const Descriptor * DescriptorProto_ExtensionRange::GetDescriptor() const

Get a Descriptor for this message's type.

This describes what fields the message contains, the types of those fields, etc.

---

virtual const Reflection *

DescriptorProto_ExtensionRange::GetReflection() const

Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time).

This object remains property of the Message.

class DescriptorProto: public Message

```
#include <google/protobuf/descriptor.pb.h >
namespace google::protobuf
```

See the docs for descriptor.pb.h for more information about this class.

Members

	DescriptorProto ()
virtual	**~DescriptorProto** ()
	DescriptorProto (const DescriptorProto & from)
DescriptorProto &	**operator=** (const DescriptorProto & from)
const UnknownFieldSet &	**unknown_fields** () const
UnknownFieldSet *	**mutable_unknown_fields** ()
static const DescriptorProto &	**default_instance** ()
static const Descriptor *	**descriptor** ()

nested types

typedef	DescriptorProto_ExtensionRange **ExtensionRange**

implements Message

virtual DescriptorProto *	**New**() const *Construct a new instance of the same type. more...*
virtual void	**CopyFrom** (const Message & from) *Make this message into a copy of the given message. more...*
virtual void	**MergeFrom** (const Message & from) *Merge the fields from the given message into this message. more...*
void	**CopyFrom** (const DescriptorProto & from)
void	**MergeFrom** (const DescriptorProto & from)
virtual void	**Clear** () *Clear all fields of the message and set them to their default values. more...*
virtual bool	**IsInitialized** () const *Quickly check if all required fields have values set.*
virtual int	**ByteSize** () const *Computes the serialized size of the message. more...*
virtual bool	**MergePartialFromCodedStream** (io::CodedInputStream * input) *Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input. more...*
virtual bool	**SerializeWithCachedSizes** (io::CodedOutputStream * output) const *Serializes the message without recomputing the size. more...*

virtual int	**GetCachedSize**() const *Returns the result of the last call to ByteSize(). more...*
virtual const Descriptor *	**GetDescriptor**() const *Get a Descriptor for this message's type. more...*
virtual const Reflection *	**GetReflection**() const *Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time). more...*
accessors	
bool	**has_name**() const *optional string name = 1;*
void	**clear_name**()
const ::std::string &	**name**() const
void	**set_name**(const ::std::string & value)
void	**set_name**(const char * value)
inline::std::string *	**mutable_name**()
int	**field_size**() const *repeated .google.protobuf.FieldDescriptorProto field = 2;*
void	**clear_field**()
const RepeatedPtrField< FieldDescriptorProto > &	**field**() const
RepeatedPtrField< FieldDescriptorProto > *	**mutable_field**()
const FieldDescriptorProto &	**field**(int index) const
FieldDescriptorProto *	**mutable_field**(int index)
FieldDescriptorProto *	**add_field**()
int	**extension_size**() const *repeated .google.protobuf.FieldDescriptorProto extension = 6;*
void	**clear_extension**()
const RepeatedPtrField< FieldDescriptorProto > &	**extension**() const
RepeatedPtrField< FieldDescriptorProto > *	**mutable_extension**()
const FieldDescriptorProto &	**extension**(int index) const
FieldDescriptorProto *	**mutable_extension**(int index)
FieldDescriptorProto *	**add_extension**()
int	**nested_type_size**() const *repeated .google.protobuf.DescriptorProto nested_type = 3;*
void	**clear_nested_type**()
const RepeatedPtrField< DescriptorProto > &	**nested_type**() const
RepeatedPtrField< DescriptorProto > *	**mutable_nested_type**()
const DescriptorProto &	**nested_type**(int index) const

DescriptorProto *	**mutable_nested_type** (int index)
DescriptorProto *	**add_nested_type** ()
int	**enum_type_size** () const *repeated .google.protobuf.EnumDescriptorProto enum_type = 4;*
void	**clear_enum_type** ()
const RepeatedPtrField< EnumDescriptorProto > &	**enum_type** () const
RepeatedPtrField< EnumDescriptorProto > *	**mutable_enum_type** ()
const EnumDescriptorProto &	**enum_type** (int index) const
EnumDescriptorProto *	**mutable_enum_type** (int index)
EnumDescriptorProto *	**add_enum_type** ()
int	**extension_range_size** () const *repeated .google.protobuf.DescriptorProto.ExtensionRange extension_range = 5;*
void	**clear_extension_range** ()
const RepeatedPtrField< DescriptorProto_ExtensionRange > &	**extension_range** () const
RepeatedPtrField< DescriptorProto_ExtensionRange > *	**mutable_extension_range** ()
const DescriptorProto_ExtensionRange &	**extension_range** (int index) const
DescriptorProto_ExtensionRange *	**mutable_extension_range** (int index)
DescriptorProto_ExtensionRange *	**add_extension_range** ()
bool	**has_options** () const *optional .google.protobuf.MessageOptions options = 7;*
void	**clear_options** ()
const MessageOptions &	**options** () const
MessageOptions *	**mutable_options** ()

virtual DescriptorProto * DescriptorProto::New() const

Construct a new instance of the same type.

Ownership is passed to the caller.

virtual void DescriptorProto::CopyFrom(const Message & from)

Make this message into a copy of the given message.

The given message must have the same descriptor, but need not necessarily be the same class. By default this is just implemented as "Clear(); MergeFrom(from);".

```
virtual void DescriptorProto::MergeFrom(
        const Message & from)
```

Merge the fields from the given message into this message.

Singular fields will be overwritten, except for embedded messages which will be merged. Repeated fields will be concatenated. The given message must be of the same type as this message (i.e. the exact same class).

```
virtual void DescriptorProto::Clear()
```

Clear all fields of the message and set them to their default values.

Clear() avoids freeing memory, assuming that any memory allocated to hold parts of the message will be needed again to hold the next message. If you actually want to free the memory used by a Message, you must delete it.

```
virtual int DescriptorProto::ByteSize() const
```

Computes the serialized size of the message.

This recursively calls ByteSize() on all embedded messages. If a subclass does not override this, it MUST override SetCachedSize().

```
virtual bool DescriptorProto::MergePartialFromCodedStream(
        io::CodedInputStream * input)
```

Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input.

MergeFromCodedStream() is just implemented as MergePartialFromCodedStream() followed by IsInitialized().

```
virtual bool DescriptorProto::SerializeWithCachedSizes(
        io::CodedOutputStream * output) const
```

Serializes the message without recomputing the size.

The message must not have changed since the last call to ByteSize(); if it has, the results are undefined.

```
virtual int DescriptorProto::GetCachedSize() const
```

Returns the result of the last call to ByteSize().

An embedded message's size is needed both to serialize it (because embedded messages are length-delimited) and to compute the outer message's size. Caching the size avoids computing it multiple times.

ByteSize() does not automatically use the cached size when available because this would require invalidating it every time the message was modified, which would be too hard and expensive. (E.g. if a deeply-nested sub-message is changed, all of its parents' cached sizes would need to be invalidated, which is too much work for an otherwise inlined setter method.)

```
virtual const Descriptor *
    DescriptorProto::GetDescriptor() const
```

Get a Descriptor for this message's type.

This describes what fields the message contains, the types of those fields, etc.

```
virtual const Reflection *
    DescriptorProto::GetReflection() const
```

Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time).

This object remains property of the Message.

class FieldDescriptorProto: public Message

```
#include <google/protobuf/descriptor.pb.h>
namespace google::protobuf
```

See the docs for descriptor.pb.h for more information about this class.

Members	
	FieldDescriptorProto()
virtual	**~FieldDescriptorProto**()
	FieldDescriptorProto(const FieldDescriptorProto & from)
FieldDescriptorProto &	**operator=**(const FieldDescriptorProto & from)
const UnknownFieldSet &	**unknown_fields**() const
UnknownFieldSet *	**mutable_unknown_fields**()
static const FieldDescriptorProto &	**default_instance**()
static const Descriptor *	**descriptor**()
nested types	
typedef	FieldDescriptorProto_Type **Type**
typedef	FieldDescriptorProto_Label **Label**
static const EnumDescriptor *	**Type_descriptor**()
static bool	**Type_IsValid**(int value)
static const EnumDescriptor *	**Label_descriptor**()
static bool	**Label_IsValid**(int value)
const Type	**TYPE_DOUBLE** = FieldDescriptorProto_Type_TYPE_DOUBLE
const Type	**TYPE_FLOAT** = FieldDescriptorProto_Type_TYPE_FLOAT
const Type	**TYPE_INT64** = FieldDescriptorProto_Type_TYPE_INT64
const Type	**TYPE_UINT64** = FieldDescriptorProto_Type_TYPE_UINT64
const Type	**TYPE_INT32** = FieldDescriptorProto_Type_TYPE_INT32
const Type	**TYPE_FIXED64** = FieldDescriptorProto_Type_TYPE_FIXED64
const Type	**TYPE_FIXED32** = FieldDescriptorProto_Type_TYPE_FIXED32

const Type	**TYPE_BOOL** = FieldDescriptorProto_Type_TYPE_BOOL
const Type	**TYPE_STRING** = FieldDescriptorProto_Type_TYPE_STRING
const Type	**TYPE_GROUP** = FieldDescriptorProto_Type_TYPE_GROUP
const Type	**TYPE_MESSAGE** = FieldDescriptorProto_Type_TYPE_MESSAGE
const Type	**TYPE_BYTES** = FieldDescriptorProto_Type_TYPE_BYTES
const Type	**TYPE_UINT32** = FieldDescriptorProto_Type_TYPE_UINT32
const Type	**TYPE_ENUM** = FieldDescriptorProto_Type_TYPE_ENUM
const Type	**TYPE_SFIXED32** = FieldDescriptorProto_Type_TYPE_SFIXED32
const Type	**TYPE_SFIXED64** = FieldDescriptorProto_Type_TYPE_SFIXED64
const Type	**TYPE_SINT32** = FieldDescriptorProto_Type_TYPE_SINT32
const Type	**TYPE_SINT64** = FieldDescriptorProto_Type_TYPE_SINT64
const Type	**Type_MIN** = FieldDescriptorProto_Type_Type_MIN
const Type	**Type_MAX** = FieldDescriptorProto_Type_Type_MAX
const Label	**LABEL_OPTIONAL** = FieldDescriptorProto_Label_LABEL_OPTIONAL
const Label	**LABEL_REQUIRED** = FieldDescriptorProto_Label_LABEL_REQUIRED
const Label	**LABEL_REPEATED** = FieldDescriptorProto_Label_LABEL_REPEATED
const Label	**Label_MIN** = FieldDescriptorProto_Label_Label_MIN
const Label	**Label_MAX** = FieldDescriptorProto_Label_Label_MAX

implements Message

virtual FieldDescriptorProto *	**New**() const *Construct a new instance of the same type. more...*
virtual void	**CopyFrom**(const Message & from) *Make this message into a copy of the given message. more...*
virtual void	**MergeFrom**(const Message & from) *Merge the fields from the given message into this message. more...*
void	**CopyFrom**(const FieldDescriptorProto & from)
void	**MergeFrom**(const FieldDescriptorProto & from)
virtual void	**Clear**() *Clear all fields of the message and set them to their default values. more...*
virtual bool	**IsInitialized**() const *Quickly check if all required fields have values set.*
virtual int	**ByteSize**() const *Computes the serialized size of the message. more...*

virtual bool	**MergePartialFromCodedStream** (io::CodedInputStream * input) *Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input. more...*
virtual bool	**SerializeWithCachedSizes** (io::CodedOutputStream * output) const *Serializes the message without recomputing the size. more...*
virtual int	**GetCachedSize** () const *Returns the result of the last call to ByteSize(). more...*
virtual const Descriptor *	**GetDescriptor** () const *Get a Descriptor for this message's type. more...*
virtual const Reflection *	**GetReflection** () const *Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time). more...*

accessors

bool	**has_name** () const *optional string name = 1;*
void	**clear_name** ()
const ::std::string &	**name** () const
void	**set_name** (const ::std::string & value)
void	**set_name** (const char * value)
inline::std::string *	**mutable_name** ()
bool	**has_number** () const *optional int32 number = 3;*
void	**clear_number** ()
int32	**number** () const
void	**set_number** (int32 value)
bool	**has_label** () const *optional .google.protobuf.FieldDescriptorProto.Label label = 4;*
void	**clear_label** ()
FieldDescriptorProto_Label	**label** () const
void	**set_label** (FieldDescriptorProto_Label value)
bool	**has_type** () const *optional .google.protobuf.FieldDescriptorProto.Type type = 5;*
void	**clear_type** ()
FieldDescriptorProto_Type	**type** () const
void	**set_type** (FieldDescriptorProto_Type value)
bool	**has_type_name** () const *optional string type_name = 6;*
void	**clear_type_name** ()
const ::std::string &	**type_name** () const
void	**set_type_name** (const ::std::string & value)
void	**set_type_name** (const char * value)
inline::std::string *	**mutable_type_name** ()

bool	**has_extendee**() const *optional string extendee = 2;*
void	**clear_extendee**()
const ::std::string &	**extendee**() const
void	**set_extendee**(const ::std::string & value)
void	**set_extendee**(const char * value)
inline::std::string *	**mutable_extendee**()
bool	**has_default_value**() const *optional string default_value = 7;*
void	**clear_default_value**()
const ::std::string &	**default_value**() const
void	**set_default_value**(const ::std::string & value)
void	**set_default_value**(const char * value)
inline::std::string *	**mutable_default_value**()
bool	**has_options**() const *optional .google.protobuf.FieldOptions options = 8;*
void	**clear_options**()
const FieldOptions &	**options**() const
FieldOptions *	**mutable_options**()

virtual FieldDescriptorProto * FieldDescriptorProto::New() const

Construct a new instance of the same type.

Ownership is passed to the caller.

virtual void FieldDescriptorProto::CopyFrom(const Message & from)

Make this message into a copy of the given message.

The given message must have the same descriptor, but need not necessarily be the same class. By default this is just implemented as "Clear(); MergeFrom(from);".

virtual void FieldDescriptorProto::MergeFrom(const Message & from)

Merge the fields from the given message into this message.

Singular fields will be overwritten, except for embedded messages which will be merged. Repeated fields will be concatenated. The given message must be of the same type as this message (i.e. the exact same class).

virtual void FieldDescriptorProto::Clear()

Clear all fields of the message and set them to their default values.

Clear() avoids freeing memory, assuming that any memory allocated to hold parts of the message will be needed again to hold the next message. If you actually want to free the memory used by a Message, you must delete it.

---

virtual int FieldDescriptorProto::ByteSize() const

Computes the serialized size of the message.

This recursively calls ByteSize() on all embedded messages. If a subclass does not override this, it MUST override SetCachedSize().

---

virtual bool FieldDescriptorProto::MergePartialFromCodedStream(io::CodedInputStream * input)

Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input.

MergeFromCodedStream() is just implemented as MergePartialFromCodedStream() followed by IsInitialized().

---

virtual bool FieldDescriptorProto::SerializeWithCachedSizes(io::CodedOutputStream * output) const

Serializes the message without recomputing the size.

The message must not have changed since the last call to ByteSize(); if it has, the results are undefined.

---

virtual int FieldDescriptorProto::GetCachedSize() const

Returns the result of the last call to ByteSize().

An embedded message's size is needed both to serialize it (because embedded messages are length-delimited) and to compute the outer message's size. Caching the size avoids computing it multiple times.

ByteSize() does not automatically use the cached size when available because this would require invalidating it every time the message was modified, which would be too hard and expensive. (E.g. if a deeply-nested sub-message is changed, all of its parents' cached sizes would need to be invalidated, which is too much work for an otherwise inlined setter method.)

---

virtual const Descriptor * FieldDescriptorProto::GetDescriptor() const

Get a Descriptor for this message's type.

This describes what fields the message contains, the types of those fields, etc.

---

virtual const Reflection * FieldDescriptorProto::GetReflection() const

Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time).

This object remains property of the Message.

---

class EnumDescriptorProto: public Message

```
#include <google/protobuf/descriptor.pb.h >
namespace google::protobuf
```

See the docs for descriptor.pb.h for more information about this class.

Members

	EnumDescriptorProto ()
virtual	**~EnumDescriptorProto** ()
	EnumDescriptorProto (const EnumDescriptorProto & from)
EnumDescriptorProto &	**operator=**(const EnumDescriptorProto & from)
const UnknownFieldSet &	**unknown_fields** () const
UnknownFieldSet *	**mutable_unknown_fields** ()
static const EnumDescriptorProto &	**default_instance** ()
static const Descriptor *	**descriptor** ()

implements Message

virtual EnumDescriptorProto *	**New**() const *Construct a new instance of the same type. more...*
virtual void	**CopyFrom**(const Message & from) *Make this message into a copy of the given message. more...*
virtual void	**MergeFrom**(const Message & from) *Merge the fields from the given message into this message. more...*
void	**CopyFrom**(const EnumDescriptorProto & from)
void	**MergeFrom**(const EnumDescriptorProto & from)
virtual void	**Clear**() *Clear all fields of the message and set them to their default values. more...*
virtual bool	**IsInitialized**() const *Quickly check if all required fields have values set.*
virtual int	**ByteSize**() const *Computes the serialized size of the message. more...*
virtual bool	**MergePartialFromCodedStream** (io::CodedInputStream * input) *Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input. more...*
virtual bool	**SerializeWithCachedSizes** (io::CodedOutputStream * output) const *Serializes the message without recomputing the size. more...*
virtual int	**GetCachedSize** () const *Returns the result of the last call to ByteSize(). more...*
virtual const Descriptor *	**GetDescriptor** () const *Get a Descriptor for this message's type. more...*
virtual const Reflection *	**GetReflection** () const *Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time). more...*

accessors

bool	**has_name** () const *optional string name = 1;*
void	**clear_name** ()

const ::std::string &	**name**() const
void	**set_name**(const ::std::string & value)
void	**set_name**(const char * value)
inline::std::string *	**mutable_name**()
int	**value_size**() const *repeated .google.protobuf.EnumValueDescriptorProto value = 2;*
void	**clear_value**()
const RepeatedPtrField< EnumValueDescriptorProto > &	**value**() const
RepeatedPtrField< EnumValueDescriptorProto > *	**mutable_value**()
const EnumValueDescriptorProto &	**value**(int index) const
EnumValueDescriptorProto *	**mutable_value**(int index)
EnumValueDescriptorProto *	**add_value**()
bool	**has_options**() const *optional .google.protobuf.EnumOptions options = 3;*
void	**clear_options**()
const EnumOptions &	**options**() const
EnumOptions *	**mutable_options**()

---

virtual EnumDescriptorProto * EnumDescriptorProto::New() const

Construct a new instance of the same type.

Ownership is passed to the caller.

---

virtual void EnumDescriptorProto::CopyFrom(const Message & from)

Make this message into a copy of the given message.

The given message must have the same descriptor, but need not necessarily be the same class. By default this is just implemented as "Clear(); MergeFrom(from);".

---

virtual void EnumDescriptorProto::MergeFrom(const Message & from)

Merge the fields from the given message into this message.

Singular fields will be overwritten, except for embedded messages which will be merged. Repeated fields will be concatenated. The given message must be of the same type as this message (i.e. the exact same class).

---

virtual void EnumDescriptorProto::Clear()

Clear all fields of the message and set them to their default values.

Clear() avoids freeing memory, assuming that any memory allocated to hold parts of the message will be needed again to hold the next message. If you actually want to free the memory used by a Message, you must delete it.

---

virtual int EnumDescriptorProto::ByteSize() const

Computes the serialized size of the message.

This recursively calls ByteSize() on all embedded messages. If a subclass does not override this, it MUST override SetCachedSize().

---

virtual bool EnumDescriptorProto::MergePartialFromCodedStream(io::CodedInputStream * input)

Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input.

MergeFromCodedStream() is just implemented as MergePartialFromCodedStream() followed by IsInitialized().

---

virtual bool EnumDescriptorProto::SerializeWithCachedSizes(io::CodedOutputStream * output) const

Serializes the message without recomputing the size.

The message must not have changed since the last call to ByteSize(); if it has, the results are undefined.

---

virtual int EnumDescriptorProto::GetCachedSize() const

Returns the result of the last call to ByteSize().

An embedded message's size is needed both to serialize it (because embedded messages are length-delimited) and to compute the outer message's size. Caching the size avoids computing it multiple times.

ByteSize() does not automatically use the cached size when available because this would require invalidating it every time the message was modified, which would be too hard and expensive. (E.g. if a deeply-nested sub-message is changed, all of its parents' cached sizes would need to be invalidated, which is too much work for an otherwise inlined setter method.)

---

virtual const Descriptor * EnumDescriptorProto::GetDescriptor() const

Get a Descriptor for this message's type.

This describes what fields the message contains, the types of those fields, etc.

---

virtual const Reflection * EnumDescriptorProto::GetReflection() const

Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time).

This object remains property of the Message.

---

class EnumValueDescriptorProto: public Message

```
#include <google/protobuf/descriptor.pb.h >
namespace google::protobuf
```

See the docs for descriptor.pb.h for more information about this class.

Members

	`EnumValueDescriptorProto ()`
`virtual`	`~EnumValueDescriptorProto ()`
	`EnumValueDescriptorProto (const EnumValueDescriptorProto & from)`
`EnumValueDescriptorProto &`	`operator=(const EnumValueDescriptorProto & from)`
`const UnknownFieldSet &`	`unknown_fields () const`
`UnknownFieldSet *`	`mutable_unknown_fields ()`
`static const EnumValueDescriptorProto &`	`default_instance ()`
`static const Descriptor *`	`descriptor ()`

implements Message

`virtual EnumValueDescriptorProto *`	`New() const` *Construct a new instance of the same type. more...*
`virtual void`	`CopyFrom(const Message & from)` *Make this message into a copy of the given message. more...*
`virtual void`	`MergeFrom(const Message & from)` *Merge the fields from the given message into this message. more...*
`void`	`CopyFrom(const EnumValueDescriptorProto & from)`
`void`	`MergeFrom(const EnumValueDescriptorProto & from)`
`virtual void`	`Clear()` *Clear all fields of the message and set them to their default values. more...*
`virtual bool`	`IsInitialized() const` *Quickly check if all required fields have values set.*
`virtual int`	`ByteSize() const` *Computes the serialized size of the message. more...*
`virtual bool`	`MergePartialFromCodedStream (io::CodedInputStream * input)` *Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input. more...*
`virtual bool`	`SerializeWithCachedSizes (io::CodedOutputStream * output) const` *Serializes the message without recomputing the size. more...*
`virtual int`	`GetCachedSize() const` *Returns the result of the last call to ByteSize(). more...*
`virtual const Descriptor *`	`GetDescriptor() const` *Get a Descriptor for this message's type. more...*
`virtual const Reflection *`	`GetReflection() const` *Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time). more...*

accessors

`bool`	`has_name() const` *optional string name = 1;*

void	**clear_name**()
const ::std::string &	**name**() const
void	**set_name**(const ::std::string & value)
void	**set_name**(const char * value)
inline::std::string *	**mutable_name**()
bool	**has_number**() const *optional int32 number = 2;*
void	**clear_number**()
int32	**number**() const
void	**set_number**(int32 value)
bool	**has_options**() const *optional .google.protobuf.EnumValueOptions options = 3;*
void	**clear_options**()
const EnumValueOptions &	**options**() const
EnumValueOptions *	**mutable_options**()

---

virtual EnumValueDescriptorProto * EnumValueDescriptorProto::New() const

Construct a new instance of the same type.

Ownership is passed to the caller.

---

virtual void EnumValueDescriptorProto::CopyFrom(const Message & from)

Make this message into a copy of the given message.

The given message must have the same descriptor, but need not necessarily be the same class. By default this is just implemented as "Clear(); MergeFrom(from);".

---

virtual void EnumValueDescriptorProto::MergeFrom(const Message & from)

Merge the fields from the given message into this message.

Singular fields will be overwritten, except for embedded messages which will be merged. Repeated fields will be concatenated. The given message must be of the same type as this message (i.e. the exact same class).

---

virtual void EnumValueDescriptorProto::Clear()

Clear all fields of the message and set them to their default values.

Clear() avoids freeing memory, assuming that any memory allocated to hold parts of the message will be needed again to hold the next message. If you actually want to free the memory used by a Message, you must delete it.

---

virtual int EnumValueDescriptorProto::ByteSize() const

Computes the serialized size of the message.

This recursively calls ByteSize() on all embedded messages. If a subclass does not override this, it MUST override SetCachedSize().

```
virtual bool EnumValueDescriptorProto::MergePartialFromCodedStream(
        io::CodedInputStream * input)
```

Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input.

MergeFromCodedStream() is just implemented as MergePartialFromCodedStream() followed by IsInitialized().

```
virtual bool EnumValueDescriptorProto::SerializeWithCachedSizes(
        io::CodedOutputStream * output) const
```

Serializes the message without recomputing the size.

The message must not have changed since the last call to ByteSize(); if it has, the results are undefined.

```
virtual int EnumValueDescriptorProto::GetCachedSize() const
```

Returns the result of the last call to ByteSize().

An embedded message's size is needed both to serialize it (because embedded messages are length-delimited) and to compute the outer message's size. Caching the size avoids computing it multiple times.

ByteSize() does not automatically use the cached size when available because this would require invalidating it every time the message was modified, which would be too hard and expensive. (E.g. if a deeply-nested sub-message is changed, all of its parents' cached sizes would need to be invalidated, which is too much work for an otherwise inlined setter method.)

```
virtual const Descriptor *
    EnumValueDescriptorProto::GetDescriptor() const
```

Get a Descriptor for this message's type.

This describes what fields the message contains, the types of those fields, etc.

```
virtual const Reflection *
    EnumValueDescriptorProto::GetReflection() const
```

Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time).

This object remains property of the Message.

class ServiceDescriptorProto: public Message

```
#include <google/protobuf/descriptor.pb.h>
namespace google::protobuf
```

See the docs for descriptor.pb.h for more information about this class.

Members	
	ServiceDescriptorProto ()
virtual	**~ServiceDescriptorProto** ()

	ServiceDescriptorProto (const ServiceDescriptorProto & from)
ServiceDescriptorProto &	**operator=**(const ServiceDescriptorProto & from)
const UnknownFieldSet &	**unknown_fields** () const
UnknownFieldSet *	**mutable_unknown_fields** ()
static const ServiceDescriptorProto &	**default_instance** ()
static const Descriptor *	**descriptor** ()

implements Message

virtual ServiceDescriptorProto *	**New**() const *Construct a new instance of the same type. more...*
virtual void	**CopyFrom**(const Message & from) *Make this message into a copy of the given message. more...*
virtual void	**MergeFrom**(const Message & from) *Merge the fields from the given message into this message. more...*
void	**CopyFrom**(const ServiceDescriptorProto & from)
void	**MergeFrom**(const ServiceDescriptorProto & from)
virtual void	**Clear**() *Clear all fields of the message and set them to their default values. more...*
virtual bool	**IsInitialized**() const *Quickly check if all required fields have values set.*
virtual int	**ByteSize**() const *Computes the serialized size of the message. more...*
virtual bool	**MergePartialFromCodedStream** (io::CodedInputStream * input) *Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input. more...*
virtual bool	**SerializeWithCachedSizes** (io::CodedOutputStream * output) const *Serializes the message without recomputing the size. more...*
virtual int	**GetCachedSize**() const *Returns the result of the last call to ByteSize(). more...*
virtual const Descriptor *	**GetDescriptor**() const *Get a Descriptor for this message's type. more...*
virtual const Reflection *	**GetReflection**() const *Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time). more...*

accessors

bool	**has_name**() const *optional string name = 1;*
void	**clear_name**()
const ::std::string &	**name**() const
void	**set_name**(const ::std::string & value)
void	**set_name**(const char * value)

inline::std::string *	**mutable_name**()
int	**method_size**() const *repeated .google.protobuf.MethodDescriptorProto method = 2;*
void	**clear_method**()
const RepeatedPtrField< MethodDescriptorProto > &	**method**() const
RepeatedPtrField< MethodDescriptorProto > *	**mutable_method**()
const MethodDescriptorProto &	**method**(int index) const
MethodDescriptorProto *	**mutable_method**(int index)
MethodDescriptorProto *	**add_method**()
bool	**has_options**() const *optional .google.protobuf.ServiceOptions options = 3;*
void	**clear_options**()
const ServiceOptions &	**options**() const
ServiceOptions *	**mutable_options**()

---

virtual ServiceDescriptorProto * ServiceDescriptorProto::New() const

Construct a new instance of the same type.

Ownership is passed to the caller.

---

virtual void ServiceDescriptorProto::CopyFrom(const Message & from)

Make this message into a copy of the given message.

The given message must have the same descriptor, but need not necessarily be the same class. By default this is just implemented as "Clear(); MergeFrom(from);".

---

virtual void ServiceDescriptorProto::MergeFrom(const Message & from)

Merge the fields from the given message into this message.

Singular fields will be overwritten, except for embedded messages which will be merged. Repeated fields will be concatenated. The given message must be of the same type as this message (i.e. the exact same class).

---

virtual void ServiceDescriptorProto::Clear()

Clear all fields of the message and set them to their default values.

Clear() avoids freeing memory, assuming that any memory allocated to hold parts of the message will be needed again to hold the next message. If you actually want to free the memory used by a Message, you must delete it.

```
virtual int ServiceDescriptorProto::ByteSize() const
```

Computes the serialized size of the message.

This recursively calls ByteSize() on all embedded messages. If a subclass does not override this, it MUST override SetCachedSize().

```
virtual bool ServiceDescriptorProto::MergePartialFromCodedStream(
        io::CodedInputStream * input)
```

Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input.

MergeFromCodedStream() is just implemented as MergePartialFromCodedStream() followed by IsInitialized().

```
virtual bool ServiceDescriptorProto::SerializeWithCachedSizes(
        io::CodedOutputStream * output) const
```

Serializes the message without recomputing the size.

The message must not have changed since the last call to ByteSize(); if it has, the results are undefined.

```
virtual int ServiceDescriptorProto::GetCachedSize() const
```

Returns the result of the last call to ByteSize().

An embedded message's size is needed both to serialize it (because embedded messages are length-delimited) and to compute the outer message's size. Caching the size avoids computing it multiple times.

ByteSize() does not automatically use the cached size when available because this would require invalidating it every time the message was modified, which would be too hard and expensive. (E.g. if a deeply-nested sub-message is changed, all of its parents' cached sizes would need to be invalidated, which is too much work for an otherwise inlined setter method.)

```
virtual const Descriptor *
    ServiceDescriptorProto::GetDescriptor() const
```

Get a Descriptor for this message's type.

This describes what fields the message contains, the types of those fields, etc.

```
virtual const Reflection *
    ServiceDescriptorProto::GetReflection() const
```

Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time).

This object remains property of the Message.

class MethodDescriptorProto: public Message

```
#include <google/protobuf/descriptor.pb.h>
namespace google::protobuf
```

See the docs for descriptor.pb.h for more information about this class.

Members
`MethodDescriptorProto()`

virtual	**~MethodDescriptorProto** ()
	MethodDescriptorProto (const MethodDescriptorProto & from)
MethodDescriptorProto &	**operator=** (const MethodDescriptorProto & from)
const UnknownFieldSet &	**unknown_fields** () const
UnknownFieldSet *	**mutable_unknown_fields** ()
static const MethodDescriptorProto &	**default_instance** ()
static const Descriptor *	**descriptor** ()

implements Message

virtual MethodDescriptorProto *	**New**() const *Construct a new instance of the same type. more...*
virtual void	**CopyFrom** (const Message & from) *Make this message into a copy of the given message. more...*
virtual void	**MergeFrom** (const Message & from) *Merge the fields from the given message into this message. more...*
void	**CopyFrom** (const MethodDescriptorProto & from)
void	**MergeFrom** (const MethodDescriptorProto & from)
virtual void	**Clear** () *Clear all fields of the message and set them to their default values. more...*
virtual bool	**IsInitialized** () const *Quickly check if all required fields have values set.*
virtual int	**ByteSize** () const *Computes the serialized size of the message. more...*
virtual bool	**MergePartialFromCodedStream** (io::CodedInputStream * input) *Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input. more...*
virtual bool	**SerializeWithCachedSizes** (io::CodedOutputStream * output) const *Serializes the message without recomputing the size. more...*
virtual int	**GetCachedSize** () const *Returns the result of the last call to ByteSize(). more...*
virtual const Descriptor *	**GetDescriptor** () const *Get a Descriptor for this message's type. more...*
virtual const Reflection *	**GetReflection** () const *Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time). more...*

accessors

bool	**has_name** () const *optional string name = 1;*
void	**clear_name** ()
const ::std::string &	**name** () const
void	**set_name** (const ::std::string & value)

void	**set_name**(const char * value)
inline::std::string *	**mutable_name**()
bool	**has_input_type**() const *optional string input_type = 2;*
void	**clear_input_type**()
const ::std::string &	**input_type**() const
void	**set_input_type**(const ::std::string & value)
void	**set_input_type**(const char * value)
inline::std::string *	**mutable_input_type**()
bool	**has_output_type**() const *optional string output_type = 3;*
void	**clear_output_type**()
const ::std::string &	**output_type**() const
void	**set_output_type**(const ::std::string & value)
void	**set_output_type**(const char * value)
inline::std::string *	**mutable_output_type**()
bool	**has_options**() const *optional .google.protobuf.MethodOptions options = 4;*
void	**clear_options**()
const MethodOptions &	**options**() const
MethodOptions *	**mutable_options**()

---

virtual MethodDescriptorProto * MethodDescriptorProto::New() const

Construct a new instance of the same type.

Ownership is passed to the caller.

---

virtual void MethodDescriptorProto::CopyFrom(const Message & from)

Make this message into a copy of the given message.

The given message must have the same descriptor, but need not necessarily be the same class. By default this is just implemented as "Clear(); MergeFrom(from);".

---

virtual void MethodDescriptorProto::MergeFrom(const Message & from)

Merge the fields from the given message into this message.

Singular fields will be overwritten, except for embedded messages which will be merged. Repeated fields will be concatenated. The given message must be of the same type as this message (i.e. the exact same class).

---

virtual void MethodDescriptorProto::Clear()

Clear all fields of the message and set them to their default values.

Clear() avoids freeing memory, assuming that any memory allocated to hold parts of the message will be needed again to hold the next message. If you actually want to free the memory used by aMessage, you must delete it.

---

virtual int MethodDescriptorProto::ByteSize() const

Computes the serialized size of the message.

This recursively calls ByteSize() on all embedded messages. If a subclass does not override this, it MUST override SetCachedSize().

---

virtual bool MethodDescriptorProto::MergePartialFromCodedStream(io::CodedInputStream * input)

Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input.

MergeFromCodedStream() is just implemented as MergePartialFromCodedStream() followed by IsInitialized().

---

virtual bool MethodDescriptorProto::SerializeWithCachedSizes(io::CodedOutputStream * output) const

Serializes the message without recomputing the size.

The message must not have changed since the last call toByteSize(); if it has, the results are undefined.

---

virtual int MethodDescriptorProto::GetCachedSize() const

Returns the result of the last call to ByteSize().

An embedded message's size is needed both to serialize it (because embedded messages are length-delimited) and to compute the outer message's size. Caching the size avoids computing it multiple times.

ByteSize() does not automatically use the cached size when available because this would require invalidating it every time the message was modified, which would be too hard and expensive. (E.g. if a deeply-nested sub-message is changed, all of its parents' cached sizes would need to be invalidated, which is too much work for an otherwise inlined setter method.)

---

virtual const Descriptor * MethodDescriptorProto::GetDescriptor() const

Get a Descriptor for this message's type.

This describes what fields the message contains, the types of those fields, etc.

---

virtual const Reflection * MethodDescriptorProto::GetReflection() const

Get the Reflection interface for this Message, which can be used to read and modify the fields of theMessage dynamically (in other words, without knowing the message type at compile time).

This object remains property of the Message.

class FileOptions: public Message

```
#include <google/protobuf/descriptor.pb.h>
namespace google::protobuf
```

See the docs for descriptor.pb.h for more information about this class.

Members

	FileOptions ()
virtual	**~FileOptions** ()
	FileOptions (const FileOptions & from)
FileOptions &	**operator=** (const FileOptions & from)
const UnknownFieldSet &	**unknown_fields** () const
UnknownFieldSet *	**mutable_unknown_fields** ()
static const FileOptions &	**default_instance** ()
static const Descriptor *	**descriptor** ()

nested types

typedef	FileOptions_OptimizeMode **OptimizeMode**
static const EnumDescriptor *	**OptimizeMode_descriptor** ()
static bool	**OptimizeMode_IsValid** (int value)
const OptimizeMode	**SPEED** = FileOptions_OptimizeMode_SPEED
const OptimizeMode	**CODE_SIZE** = FileOptions_OptimizeMode_CODE_SIZE
const OptimizeMode	**OptimizeMode_MIN** = FileOptions_OptimizeMode_OptimizeMode_MIN
const OptimizeMode	**OptimizeMode_MAX** = FileOptions_OptimizeMode_OptimizeMode_MAX

implements Message

virtual FileOptions *	**New**() const *Construct a new instance of the same type. more...*
virtual void	**CopyFrom**(const Message & from) *Make this message into a copy of the given message. more...*
virtual void	**MergeFrom**(const Message & from) *Merge the fields from the given message into this message. more...*
void	**CopyFrom**(const FileOptions & from)
void	**MergeFrom**(const FileOptions & from)
virtual void	**Clear**() *Clear all fields of the message and set them to their default values. more...*
virtual bool	**IsInitialized**() const *Quickly check if all required fields have values set.*
virtual int	**ByteSize**() const *Computes the serialized size of the message. more...*
virtual bool	**MergePartialFromCodedStream** (io::CodedInputStream * input)

	Like [MergeFromCodedStream()](), but succeeds even if required fields are missing in the input. [more...]()
virtual bool	**SerializeWithCachedSizes** ([io::CodedOutputStream]() * output) const *Serializes the message without recomputing the size. [more...]()*
virtual int	**GetCachedSize** () const *Returns the result of the last call to [ByteSize](). [more...]()*
virtual const [Descriptor]() *	**GetDescriptor** () const *Get a [Descriptor]() for this message's type. [more...]()*
virtual const [Reflection]() *	**GetReflection** () const *Get the [Reflection]() interface for this [Message](), which can be used to read and modify the fields of the [Message]() dynamically (in other words, without knowing the message type at compile time). [more...]()*
accessors	
bool	**has_java_package** () const *optional string java_package = 1;*
void	**clear_java_package** ()
const ::std::string &	**java_package** () const
void	**set_java_package** (const ::std::string & value)
void	**set_java_package** (const char * value)
inline::std::string *	**mutable_java_package** ()
bool	**has_java_outer_classname** () const *optional string java_outer_classname = 8;*
void	**clear_java_outer_classname** ()
const ::std::string &	**java_outer_classname** () const
void	**set_java_outer_classname** (const ::std::string & value)
void	**set_java_outer_classname** (const char * value)
inline::std::string *	**mutable_java_outer_classname** ()
bool	**has_java_multiple_files** () const *optional bool java_multiple_files = 10 [default = false];*
void	**clear_java_multiple_files** ()
bool	**java_multiple_files** () const
void	**set_java_multiple_files** (bool value)
bool	**has_optimize_for** () const *optional [.google.protobuf.FileOptions.OptimizeMode]() optimize_for = 9 [default = CODE_SIZE];*
void	**clear_optimize_for** ()
[FileOptions_OptimizeMode]()	**optimize_for** () const
void	**set_optimize_for** ([FileOptions_OptimizeMode]() value)

---

virtual [FileOptions]() * FileOptions::New() const

Construct a new instance of the same type.

Ownership is passed to the caller.

`virtual void FileOptions::CopyFrom(const Message & from)`

Make this message into a copy of the given message.

The given message must have the same descriptor, but need not necessarily be the same class. By default this is just implemented as "Clear(); MergeFrom(from);".

`virtual void FileOptions::MergeFrom(const Message & from)`

Merge the fields from the given message into this message.

Singular fields will be overwritten, except for embedded messages which will be merged. Repeated fields will be concatenated. The given message must be of the same type as this message (i.e. the exact same class).

`virtual void FileOptions::Clear()`

Clear all fields of the message and set them to their default values.

Clear() avoids freeing memory, assuming that any memory allocated to hold parts of the message will be needed again to hold the next message. If you actually want to free the memory used by a Message, you must delete it.

`virtual int FileOptions::ByteSize() const`

Computes the serialized size of the message.

This recursively calls ByteSize() on all embedded messages. If a subclass does not override this, it MUST override SetCachedSize().

`virtual bool FileOptions::MergePartialFromCodedStream(io::CodedInputStream * input)`

Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input.

MergeFromCodedStream() is just implemented as MergePartialFromCodedStream() followed by IsInitialized().

`virtual bool FileOptions::SerializeWithCachedSizes(io::CodedOutputStream * output) const`

Serializes the message without recomputing the size.

The message must not have changed since the last call to ByteSize(); if it has, the results are undefined.

`virtual int FileOptions::GetCachedSize() const`

Returns the result of the last call to ByteSize().

An embedded message's size is needed both to serialize it (because embedded messages are length-delimited) and to compute the outer message's size. Caching the size avoids computing it multiple times.

ByteSize() does not automatically use the cached size when available because this would require invalidating it every time the message was modified, which would be too hard and expensive. (E.g. if a deeply-nested sub-message is changed, all of its parents' cached sizes would need to be invalidated, which is too much work for an otherwise inlined setter method.)

---

virtual const [Descriptor](#) *
FileOptions::GetDescriptor() const

Get a Descriptor for this message's type.

This describes what fields the message contains, the types of those fields, etc.

---

virtual const Reflection *
FileOptions::GetReflection() const

Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time).

This object remains property of the Message.

class MessageOptions: public Message

```
#include <google/protobuf/descriptor.pb.h>
namespace google::protobuf
```

See the docs for descriptor.pb.h for more information about this class.

Members	
	MessageOptions()
virtual	**~MessageOptions**()
	MessageOptions(const MessageOptions & from)
MessageOptions &	**operator=**(const MessageOptions & from)
const UnknownFieldSet &	**unknown_fields**() const
UnknownFieldSet *	**mutable_unknown_fields**()
static const MessageOptions &	**default_instance**()
static const Descriptor *	**descriptor**()
implements Message	
virtual MessageOptions *	**New**() const *Construct a new instance of the same type. more...*
virtual void	**CopyFrom**(const Message & from) *Make this message into a copy of the given message. more...*
virtual void	**MergeFrom**(const Message & from) *Merge the fields from the given message into this message. more...*
void	**CopyFrom**(const MessageOptions & from)
void	**MergeFrom**(const MessageOptions & from)
virtual void	**Clear**() *Clear all fields of the message and set them to their default values. more...*
virtual bool	**IsInitialized**() const *Quickly check if all required fields have values set.*
virtual int	**ByteSize**() const

	Computes the serialized size of the message. *more...*
virtual bool	**MergePartialFromCodedStream** (io::CodedInputStream * input) *Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input.* *more...*
virtual bool	**SerializeWithCachedSizes** (io::CodedOutputStream * output) const *Serializes the message without recomputing the size.* *more...*
virtual int	**GetCachedSize** () const *Returns the result of the last call to ByteSize().* *more...*
virtual const Descriptor *	**GetDescriptor** () const *Get a Descriptor for this message's type.* *more...*
virtual const Reflection *	**GetReflection** () const *Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time).* *more...*
accessors	
bool	**has_message_set_wire_format** () const *optional bool message_set_wire_format = 1 [default = false];*
void	**clear_message_set_wire_format** ()
bool	**message_set_wire_format** () const
void	**set_message_set_wire_format** (bool value)

virtual MessageOptions * MessageOptions::New() const

Construct a new instance of the same type.

Ownership is passed to the caller.

virtual void MessageOptions::CopyFrom(
const Message & from)

Make this message into a copy of the given message.

The given message must have the same descriptor, but need not necessarily be the same class. By default this is just implemented as "Clear(); MergeFrom(from);".

virtual void MessageOptions::MergeFrom(
const Message & from)

Merge the fields from the given message into this message.

Singular fields will be overwritten, except for embedded messages which will be merged. Repeated fields will be concatenated. The given message must be of the same type as this message (i.e. the exact same class).

virtual void MessageOptions::Clear()

Clear all fields of the message and set them to their default values.

Clear() avoids freeing memory, assuming that any memory allocated to hold parts of the message will be needed again to hold the next message. If you actually want to free the memory used by a Message, you must delete it.

virtual int MessageOptions::ByteSize() const

Computes the serialized size of the message.

This recursively calls ByteSize() on all embedded messages. If a subclass does not override this, it MUST override SetCachedSize().

---

virtual bool MessageOptions::MergePartialFromCodedStream(io::CodedInputStream * input)

Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input.

MergeFromCodedStream() is just implemented as MergePartialFromCodedStream() followed by IsInitialized().

---

virtual bool MessageOptions::SerializeWithCachedSizes(io::CodedOutputStream * output) const

Serializes the message without recomputing the size.

The message must not have changed since the last call to ByteSize(); if it has, the results are undefined.

---

virtual int MessageOptions::GetCachedSize() const

Returns the result of the last call to ByteSize().

An embedded message's size is needed both to serialize it (because embedded messages are length-delimited) and to compute the outer message's size. Caching the size avoids computing it multiple times.

ByteSize() does not automatically use the cached size when available because this would require invalidating it every time the message was modified, which would be too hard and expensive. (E.g. if a deeply-nested sub-message is changed, all of its parents' cached sizes would need to be invalidated, which is too much work for an otherwise inlined setter method.)

---

virtual const Descriptor * MessageOptions::GetDescriptor() const

Get a Descriptor for this message's type.

This describes what fields the message contains, the types of those fields, etc.

---

virtual const Reflection * MessageOptions::GetReflection() const

Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time).

This object remains property of the Message.

class FieldOptions: public Message

```
#include <google/protobuf/descriptor.pb.h >
namespace google::protobuf
```

See the docs for descriptor.pb.h for more information about this class.

Members

	FieldOptions()
virtual	**~FieldOptions**()
	FieldOptions(const FieldOptions & from)
FieldOptions &	**operator=**(const FieldOptions & from)
const UnknownFieldSet &	**unknown_fields**() const
UnknownFieldSet *	**mutable_unknown_fields**()
static const FieldOptions &	**default_instance**()
static const Descriptor *	**descriptor**()

nested types

typedef	FieldOptions_CType **CType**
static const EnumDescriptor *	**CType_descriptor**()
static bool	**CType_IsValid**(int value)
const CType	**CORD** = FieldOptions_CType_CORD
const CType	**STRING_PIECE** = FieldOptions_CType_STRING_PIECE
const CType	**CType_MIN** = FieldOptions_CType_CType_MIN
const CType	**CType_MAX** = FieldOptions_CType_CType_MAX

implements Message

virtual FieldOptions *	**New**() const *Construct a new instance of the same type. more...*
virtual void	**CopyFrom**(const Message & from) *Make this message into a copy of the given message. more...*
virtual void	**MergeFrom**(const Message & from) *Merge the fields from the given message into this message. more...*
void	**CopyFrom**(const FieldOptions & from)
void	**MergeFrom**(const FieldOptions & from)
virtual void	**Clear**() *Clear all fields of the message and set them to their default values. more...*
virtual bool	**IsInitialized**() const *Quickly check if all required fields have values set.*
virtual int	**ByteSize**() const *Computes the serialized size of the message. more...*
virtual bool	**MergePartialFromCodedStream**(io::CodedInputStream * input) *Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input. more...*
virtual bool	**SerializeWithCachedSizes**(io::CodedOutputStream * output) const *Serializes the message without recomputing the size. more...*
virtual int	**GetCachedSize**() const *Returns the result of the last call to ByteSize(). more...*

virtual const Descriptor *	**GetDescriptor**() const *Get a Descriptor for this message's type. more...*
virtual const Reflection *	**GetReflection**() const *Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time). more...*
accessors	
bool	**has_ctype**() const *optional .google.protobuf.FieldOptions.CType ctype = 1;*
void	**clear_ctype**()
FieldOptions_CType	**ctype**() const
void	**set_ctype**(FieldOptions_CType value)
bool	**has_experimental_map_key**() const *optional string experimental_map_key = 9;*
void	**clear_experimental_map_key**()
const ::std::string &	**experimental_map_key**() const
void	**set_experimental_map_key**(const ::std::string & value)
void	**set_experimental_map_key**(const char * value)
inline::std::string *	**mutable_experimental_map_key**()

```
virtual FieldOptions * FieldOptions::New() const
```

Construct a new instance of the same type.

Ownership is passed to the caller.

```
virtual void FieldOptions::CopyFrom(
        const Message & from)
```

Make this message into a copy of the given message.

The given message must have the same descriptor, but need not necessarily be the same class. By default this is just implemented as "Clear(); MergeFrom(from);".

```
virtual void FieldOptions::MergeFrom(
        const Message & from)
```

Merge the fields from the given message into this message.

Singular fields will be overwritten, except for embedded messages which will be merged. Repeated fields will be concatenated. The given message must be of the same type as this message (i.e. the exact same class).

```
virtual void FieldOptions::Clear()
```

Clear all fields of the message and set them to their default values.

Clear() avoids freeing memory, assuming that any memory allocated to hold parts of the message will be needed again to hold the next message. If you actually want to free the memory used by aMessage, you must delete it.

```
virtual int FieldOptions::ByteSize() const
```

Computes the serialized size of the message.

This recursively calls ByteSize() on all embedded messages. If a subclass does not override this, it MUST override SetCachedSize().

```
virtual bool FieldOptions::MergePartialFromCodedStream(
        io::CodedInputStream * input)
```

Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input.

MergeFromCodedStream() is just implemented as MergePartialFromCodedStream() followed by IsInitialized().

```
virtual bool FieldOptions::SerializeWithCachedSizes(
        io::CodedOutputStream * output) const
```

Serializes the message without recomputing the size.

The message must not have changed since the last call to ByteSize(); if it has, the results are undefined.

```
virtual int FieldOptions::GetCachedSize() const
```

Returns the result of the last call to ByteSize().

An embedded message's size is needed both to serialize it (because embedded messages are length-delimited) and to compute the outer message's size. Caching the size avoids computing it multiple times.

ByteSize() does not automatically use the cached size when available because this would require invalidating it every time the message was modified, which would be too hard and expensive. (E.g. if a deeply-nested sub-message is changed, all of its parents' cached sizes would need to be invalidated, which is too much work for an otherwise inlined setter method.)

```
virtual const Descriptor *
    FieldOptions::GetDescriptor() const
```

Get a Descriptor for this message's type.

This describes what fields the message contains, the types of those fields, etc.

```
virtual const Reflection *
    FieldOptions::GetReflection() const
```

Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time).

This object remains property of the Message.

class EnumOptions: public Message

```
#include <google/protobuf/descriptor.pb.h >
namespace google::protobuf
```

See the docs for descriptor.pb.h for more information about this class.

Members

	EnumOptions()
virtual	**~EnumOptions**()
	EnumOptions(const EnumOptions & from)
EnumOptions &	**operator=**(const EnumOptions & from)
const UnknownFieldSet &	**unknown_fields**() const
UnknownFieldSet *	**mutable_unknown_fields**()
static const EnumOptions &	**default_instance**()
static const Descriptor *	**descriptor**()
implements Message	
virtual EnumOptions *	**New**() const *Construct a new instance of the same type. more...*
virtual void	**CopyFrom**(const Message & from) *Make this message into a copy of the given message. more...*
virtual void	**MergeFrom**(const Message & from) *Merge the fields from the given message into this message. more...*
void	**CopyFrom**(const EnumOptions & from)
void	**MergeFrom**(const EnumOptions & from)
virtual void	**Clear**() *Clear all fields of the message and set them to their default values. more...*
virtual bool	**IsInitialized**() const *Quickly check if all required fields have values set.*
virtual int	**ByteSize**() const *Computes the serialized size of the message. more...*
virtual bool	**MergePartialFromCodedStream**(io::CodedInputStream * input) *Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input. more...*
virtual bool	**SerializeWithCachedSizes**(io::CodedOutputStream * output) const *Serializes the message without recomputing the size. more...*
virtual int	**GetCachedSize**() const *Returns the result of the last call to ByteSize(). more...*
virtual const Descriptor *	**GetDescriptor**() const *Get a Descriptor for this message's type. more...*
virtual const Reflection *	**GetReflection**() const *Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time). more...*

---

virtual EnumOptions * EnumOptions::New() const

Construct a new instance of the same type.

Ownership is passed to the caller.

`virtual void EnumOptions::CopyFrom( const Message & from)`

Make this message into a copy of the given message.

The given message must have the same descriptor, but need not necessarily be the same class. By default this is just implemented as "Clear(); MergeFrom(from);".

`virtual void EnumOptions::MergeFrom( const Message & from)`

Merge the fields from the given message into this message.

Singular fields will be overwritten, except for embedded messages which will be merged. Repeated fields will be concatenated. The given message must be of the same type as this message (i.e. the exact same class).

`virtual void EnumOptions::Clear()`

Clear all fields of the message and set them to their default values.

Clear() avoids freeing memory, assuming that any memory allocated to hold parts of the message will be needed again to hold the next message. If you actually want to free the memory used by a Message, you must delete it.

`virtual int EnumOptions::ByteSize() const`

Computes the serialized size of the message.

This recursively calls ByteSize() on all embedded messages. If a subclass does not override this, it MUST override SetCachedSize().

`virtual bool EnumOptions::MergePartialFromCodedStream( io::CodedInputStream * input)`

Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input.

MergeFromCodedStream() is just implemented as MergePartialFromCodedStream() followed by IsInitialized().

`virtual bool EnumOptions::SerializeWithCachedSizes( io::CodedOutputStream * output) const`

Serializes the message without recomputing the size.

The message must not have changed since the last call to ByteSize(); if it has, the results are undefined.

`virtual int EnumOptions::GetCachedSize() const`

Returns the result of the last call to ByteSize().

An embedded message's size is needed both to serialize it (because embedded messages are length-delimited) and to compute the outer message's size. Caching the size avoids computing it multiple times.

ByteSize() does not automatically use the cached size when available because this would require invalidating it every time the message was modified, which would be too hard and expensive. (E.g. if a deeply-nested sub-message is changed, all of its parents' cached sizes would need to be invalidated, which is too much work for an otherwise inlined setter method.)

---

virtual const Descriptor *
EnumOptions::GetDescriptor() const

Get a Descriptor for this message's type.

This describes what fields the message contains, the types of those fields, etc.

---

virtual const Reflection *
EnumOptions::GetReflection() const

Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time).

This object remains property of the Message.

class EnumValueOptions: public Message

```
#include <google/protobuf/descriptor.pb.h>
namespace google::protobuf
```

See the docs for descriptor.pb.h for more information about this class.

Members	
	EnumValueOptions()
virtual	**~EnumValueOptions**()
	EnumValueOptions(const EnumValueOptions & from)
EnumValueOptions &	**operator=**(const EnumValueOptions & from)
const UnknownFieldSet &	**unknown_fields**() const
UnknownFieldSet *	**mutable_unknown_fields**()
static const EnumValueOptions &	**default_instance**()
static const Descriptor *	**descriptor**()
implements Message	
virtual EnumValueOptions *	**New**() const *Construct a new instance of the same type. more...*
virtual void	**CopyFrom**(const Message & from) *Make this message into a copy of the given message. more...*
virtual void	**MergeFrom**(const Message & from) *Merge the fields from the given message into this message. more...*
void	**CopyFrom**(const EnumValueOptions & from)
void	**MergeFrom**(const EnumValueOptions & from)
virtual void	**Clear**() *Clear all fields of the message and set them to their default values. more...*
virtual bool	**IsInitialized**() const *Quickly check if all required fields have values set.*

virtual int	**ByteSize**() const *Computes the serialized size of the message. more...*
virtual bool	**MergePartialFromCodedStream** (io::CodedInputStream * input) *Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input. more...*
virtual bool	**SerializeWithCachedSizes** (io::CodedOutputStream * output) const *Serializes the message without recomputing the size. more...*
virtual int	**GetCachedSize** () const *Returns the result of the last call to ByteSize(). more...*
virtual const Descriptor *	**GetDescriptor** () const *Get a Descriptor for this message's type. more...*
virtual const Reflection *	**GetReflection** () const *Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time). more...*

---

virtual EnumValueOptions * EnumValueOptions::New() const

Construct a new instance of the same type.

Ownership is passed to the caller.

---

virtual void EnumValueOptions::CopyFrom(const Message & from)

Make this message into a copy of the given message.

The given message must have the same descriptor, but need not necessarily be the same class. By default this is just implemented as "Clear(); MergeFrom(from);".

---

virtual void EnumValueOptions::MergeFrom(const Message & from)

Merge the fields from the given message into this message.

Singular fields will be overwritten, except for embedded messages which will be merged. Repeated fields will be concatenated. The given message must be of the same type as this message (i.e. the exact same class).

---

virtual void EnumValueOptions::Clear()

Clear all fields of the message and set them to their default values.

Clear() avoids freeing memory, assuming that any memory allocated to hold parts of the message will be needed again to hold the next message. If you actually want to free the memory used by a Message, you must delete it.

---

virtual int EnumValueOptions::ByteSize() const

Computes the serialized size of the message.

This recursively calls ByteSize() on all embedded messages. If a subclass does not override this, it MUST override

SetCachedSize().

---

virtual bool EnumValueOptions::MergePartialFromCodedStream(
io::CodedInputStream * input)

Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input.

MergeFromCodedStream() is just implemented as MergePartialFromCodedStream() followed by IsInitialized().

---

virtual bool EnumValueOptions::SerializeWithCachedSizes(
io::CodedOutputStream * output) const

Serializes the message without recomputing the size.

The message must not have changed since the last call to ByteSize(); if it has, the results are undefined.

---

virtual int EnumValueOptions::GetCachedSize() const

Returns the result of the last call to ByteSize().

An embedded message's size is needed both to serialize it (because embedded messages are length-delimited) and to compute the outer message's size. Caching the size avoids computing it multiple times.

ByteSize() does not automatically use the cached size when available because this would require invalidating it every time the message was modified, which would be too hard and expensive. (E.g. if a deeply-nested sub-message is changed, all of its parents' cached sizes would need to be invalidated, which is too much work for an otherwise inlined setter method.)

---

virtual const Descriptor *
EnumValueOptions::GetDescriptor() const

Get a Descriptor for this message's type.

This describes what fields the message contains, the types of those fields, etc.

---

virtual const Reflection *
EnumValueOptions::GetReflection() const

Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time).

This object remains property of the Message.

class ServiceOptions: public Message

```
#include <google/protobuf/descriptor.pb.h>
namespace google::protobuf
```

See the docs for descriptor.pb.h for more information about this class.

Members	
	ServiceOptions ()
virtual	**~ServiceOptions** ()
	ServiceOptions (const ServiceOptions & from)
ServiceOptions &	**operator=** (const ServiceOptions & from)

const UnknownFieldSet &	**unknown_fields** () const
UnknownFieldSet *	**mutable_unknown_fields** ()
static const ServiceOptions &	**default_instance** ()
static const Descriptor *	**descriptor** ()
implements Message	
virtual ServiceOptions *	**New**() const *Construct a new instance of the same type. more...*
virtual void	**CopyFrom**(const Message & from) *Make this message into a copy of the given message. more...*
virtual void	**MergeFrom**(const Message & from) *Merge the fields from the given message into this message. more...*
void	**CopyFrom**(const ServiceOptions & from)
void	**MergeFrom**(const ServiceOptions & from)
virtual void	**Clear**() *Clear all fields of the message and set them to their default values. more...*
virtual bool	**IsInitialized**() const *Quickly check if all required fields have values set.*
virtual int	**ByteSize**() const *Computes the serialized size of the message. more...*
virtual bool	**MergePartialFromCodedStream** (io::CodedInputStream * input) *Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input. more...*
virtual bool	**SerializeWithCachedSizes** (io::CodedOutputStream * output) const *Serializes the message without recomputing the size. more...*
virtual int	**GetCachedSize** () const *Returns the result of the last call to ByteSize(). more...*
virtual const Descriptor *	**GetDescriptor** () const *Get a Descriptor for this message's type. more...*
virtual const Reflection *	**GetReflection** () const *Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time). more...*

---

virtual ServiceOptions * ServiceOptions::New() const

Construct a new instance of the same type.

Ownership is passed to the caller.

---

virtual void ServiceOptions::CopyFrom(
const Message & from)

Make this message into a copy of the given message.

The given message must have the same descriptor, but need not necessarily be the same class. By default this is just implemented as "Clear(); MergeFrom(from);".

---

virtual void ServiceOptions::MergeFrom(const Message & from)

Merge the fields from the given message into this message.

Singular fields will be overwritten, except for embedded messages which will be merged. Repeated fields will be concatenated. The given message must be of the same type as this message (i.e. the exact same class).

---

virtual void ServiceOptions::Clear()

Clear all fields of the message and set them to their default values.

Clear() avoids freeing memory, assuming that any memory allocated to hold parts of the message will be needed again to hold the next message. If you actually want to free the memory used by aMessage, you must delete it.

---

virtual int ServiceOptions::ByteSize() const

Computes the serialized size of the message.

This recursively calls ByteSize() on all embedded messages. If a subclass does not override this, it MUST override SetCachedSize().

---

virtual bool ServiceOptions::MergePartialFromCodedStream(io::CodedInputStream * input)

Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input.

MergeFromCodedStream() is just implemented as MergePartialFromCodedStream() followed by IsInitialized().

---

virtual bool ServiceOptions::SerializeWithCachedSizes(io::CodedOutputStream * output) const

Serializes the message without recomputing the size.

The message must not have changed since the last call toByteSize(); if it has, the results are undefined.

---

virtual int ServiceOptions::GetCachedSize() const

Returns the result of the last call to ByteSize().

An embedded message's size is needed both to serialize it (because embedded messages are length-delimited) and to compute the outer message's size. Caching the size avoids computing it multiple times.

ByteSize() does not automatically use the cached size when available because this would require invalidating it every time the message was modified, which would be too hard and expensive. (E.g. if a deeply-nested sub-message is changed, all of its parents' cached sizes would need to be invalidated, which is too much work for an otherwise inlined setter method.)

---

virtual const Descriptor * ServiceOptions::GetDescriptor() const

Get a Descriptor for this message's type.

This describes what fields the message contains, the types of those fields, etc.

```
virtual const Reflection *
    ServiceOptions::GetReflection() const
```

Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time).

This object remains property of the Message.

class MethodOptions: public Message

```
#include <google/protobuf/descriptor.pb.h>
namespace google::protobuf
```

See the docs for descriptor.pb.h for more information about this class.

Members

	MethodOptions()
virtual	**~MethodOptions**()
	MethodOptions(const MethodOptions & from)
MethodOptions &	**operator=**(const MethodOptions & from)
const UnknownFieldSet &	**unknown_fields**() const
UnknownFieldSet *	**mutable_unknown_fields**()
static const MethodOptions &	**default_instance**()
static const Descriptor *	**descriptor**()

implements Message

virtual MethodOptions *	**New**() const *Construct a new instance of the same type. more...*
virtual void	**CopyFrom**(const Message & from) *Make this message into a copy of the given message. more...*
virtual void	**MergeFrom**(const Message & from) *Merge the fields from the given message into this message. more...*
void	**CopyFrom**(const MethodOptions & from)
void	**MergeFrom**(const MethodOptions & from)
virtual void	**Clear**() *Clear all fields of the message and set them to their default values. more...*
virtual bool	**IsInitialized**() const *Quickly check if all required fields have values set.*
virtual int	**ByteSize**() const *Computes the serialized size of the message. more...*
virtual bool	**MergePartialFromCodedStream**(io::CodedInputStream * input) *Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input. more...*

virtual bool	**SerializeWithCachedSizes** (io::CodedOutputStream * output) const *Serializes the message without recomputing the size. more...*
virtual int	**GetCachedSize** () const *Returns the result of the last call to ByteSize(). more...*
virtual const Descriptor *	**GetDescriptor** () const *Get a Descriptor for this message's type. more...*
virtual const Reflection *	**GetReflection** () const *Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time). more...*

---

virtual MethodOptions * MethodOptions::New() const

Construct a new instance of the same type.

Ownership is passed to the caller.

---

virtual void MethodOptions::CopyFrom(const Message & from)

Make this message into a copy of the given message.

The given message must have the same descriptor, but need not necessarily be the same class. By default this is just implemented as "Clear(); MergeFrom(from);".

---

virtual void MethodOptions::MergeFrom(const Message & from)

Merge the fields from the given message into this message.

Singular fields will be overwritten, except for embedded messages which will be merged. Repeated fields will be concatenated. The given message must be of the same type as this message (i.e. the exact same class).

---

virtual void MethodOptions::Clear()

Clear all fields of the message and set them to their default values.

Clear() avoids freeing memory, assuming that any memory allocated to hold parts of the message will be needed again to hold the next message. If you actually want to free the memory used by a Message, you must delete it.

---

virtual int MethodOptions::ByteSize() const

Computes the serialized size of the message.

This recursively calls ByteSize() on all embedded messages. If a subclass does not override this, it MUST override SetCachedSize().

---

virtual bool MethodOptions::MergePartialFromCodedStream(io::CodedInputStream * input)

Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input.

MergeFromCodedStream() is just implemented as MergePartialFromCodedStream() followed by IsInitialized().

virtual bool MethodOptions::SerializeWithCachedSizes(io::CodedOutputStream * output) const

Serializes the message without recomputing the size.

The message must not have changed since the last call to ByteSize(); if it has, the results are undefined.

virtual int MethodOptions::GetCachedSize() const

Returns the result of the last call to ByteSize().

An embedded message's size is needed both to serialize it (because embedded messages are length-delimited) and to compute the outer message's size. Caching the size avoids computing it multiple times.

ByteSize() does not automatically use the cached size when available because this would require invalidating it every time the message was modified, which would be too hard and expensive. (E.g. if a deeply-nested sub-message is changed, all of its parents' cached sizes would need to be invalidated, which is too much work for an otherwise inlined setter method.)

virtual const Descriptor * MethodOptions::GetDescriptor() const

Get a Descriptor for this message's type.

This describes what fields the message contains, the types of those fields, etc.

virtual const Reflection * MethodOptions::GetReflection() const

Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time).

This object remains property of the Message.

descriptor_database.h

`#include <google/protobuf/descriptor_database.h>`
`namespace google::protobuf`

Interface for manipulating databases of descriptors.

Classes in this file
`DescriptorDatabase` *Abstract interface for a database of descriptors.*
`SimpleDescriptorDatabase` *A DescriptorDatabase into which you can insert files manually.*
`DescriptorPoolDatabase` *A DescriptorDatabase that fetches files from a given pool.*
`MergedDescriptorDatabase` *A DescriptorDatabase that wraps two or more others.*

class DescriptorDatabase

`#include <google/protobuf/descriptor_database.h>`
`namespace google::protobuf`

Abstract interface for a database of descriptors.

This is useful if you want to create a DescriptorPool which loads descriptors on-demand from some sort of large database. If the database is large, it may be inefficient to enumerate every .proto file inside it calling DescriptorPool::BuildFile() for each one. Instead, a DescriptorPool can be created which wraps a DescriptorDatabase and only builds particular descriptors when they are needed.

Known subclasses:

- `SourceTreeDescriptorDatabase`
- `DescriptorPoolDatabase`
- `MergedDescriptorDatabase`
- `SimpleDescriptorDatabase`

Members	
	`DescriptorDatabase()`
`virtual`	`~DescriptorDatabase()`
`virtual bool`	`FindFileByName(const string & filename, FileDescriptorProto * output) = 0` *Find a file by file name. more...*
`virtual bool`	`FindFileContainingSymbol(const string & symbol_name, FileDescriptorProto * output) = 0` *Find the file that declares the given fully-qualified symbol name. more...*
`virtual bool`	`FindFileContainingExtension(const string & containing_type, int field_number, FileDescriptorProto * output) = 0`

	Find the file which defines an extension extending the given message type with the given field number. more...

```
virtual bool DescriptorDatabase::FindFileByName(
        const string & filename,
        FileDescriptorProto * output) = 0
```

Find a file by file name.

Fills in in *output and returns true if found. Otherwise, returns false, leaving the contents of *output undefined.

```
virtual bool DescriptorDatabase::FindFileContainingSymbol
        const string & symbol_name,
        FileDescriptorProto * output) = 0
```

Find the file that declares the given fully-qualified symbol name.

If found, fills in *output and returns true, otherwise returns false and leaves *output undefined.

```
virtual bool DescriptorDatabase::FindFileContainingExtension
        const string & containing_type,
        int field_number,
        FileDescriptorProto * output) = 0
```

Find the file which defines an extension extending the given message type with the given field number.

If found, fills in *output and returns true, otherwise returns false and leaves *output undefined. containing_type must be a fully-qualified type name.

class SimpleDescriptorDatabase: public DescriptorDatabase

```
#include <google/protobuf/descriptor_database.h>
namespace google::protobuf
```

A DescriptorDatabase into which you can insert files manually.

FindFileContainingSymbol() is fully-implemented. When you add a file, its symbols will be indexed for this purpose.

FindFileContainingExtension() is mostly-implemented. It works if and only if the original FieldDescriptorProto defining the extension has a fully-qualified type name in its "extendee" field (i.e. starts with a '.'). If the extendee is a relative name, SimpleDescriptorDatabase will not attempt to resolve the type, so it will not know what type the extension is extending. Therefore, calling FindFileContainingExtension() with the extension's containing type will never actually find that extension. Note that this is an unlikely problem, as all FileDescriptorProtos created by the protocol compiler (as well as ones created by calling FileDescriptor::CopyTo()) will always use fully-qualified names for all types. You only need to worry if you are constructing FileDescriptorProtos yourself, or are calling compiler::Parser directly.

Members	
	SimpleDescriptorDatabase()
	~SimpleDescriptorDatabase()
void	**Add**(const FileDescriptorProto & file) *Adds the FileDescriptorProto to the database, making a copy. more...*
void	**AddAndOwn**(const FileDescriptorProto * file)

	Adds the FileDescriptorProto to the database and takes ownership of it.
implements DescriptorDatabase	
`virtual bool`	`FindFileByName(const string & filename, FileDescriptorProto * output)` *Find a file by file name. more...*
`virtual bool`	`FindFileContainingSymbol(const string & symbol_name, FileDescriptorProto * output)` *Find the file that declares the given fully-qualified symbol name. more...*
`virtual bool`	`FindFileContainingExtension(const string & containing_type, int field_number, FileDescriptorProto * output)` *Find the file which defines an extension extending the given message type with the given field number. more...*

```
void SimpleDescriptorDatabase::Add(
        const FileDescriptorProto & file)
```

Adds the FileDescriptorPrototo the database, making a copy.

The object can be deleted afterAdd() returns.

```
virtual bool SimpleDescriptorDatabase::FindFileByName(
        const string & filename,
        FileDescriptorProto * output)
```

Find a file by file name.

Fills in in *output and returns true if found. Otherwise, returns false, leaving the contents of *output undefined.

```
virtual bool SimpleDescriptorDatabase::FindFileContainingSymbol
        const string & symbol_name,
        FileDescriptorProto * output)
```

Find the file that declares the given fully-qualified symbol name.

If found, fills in *output and returns true, otherwise returns false and leaves *output undefined.

```
virtual bool SimpleDescriptorDatabase::FindFileContainingExtension
        const string & containing_type,
        int field_number,
        FileDescriptorProto * output)
```

Find the file which defines an extension extending the given message type with the given field number.

If found, fills in *output and returns true, otherwise returns false and leaves *output undefined. containing_type must be a fully-qualified type name.

class DescriptorPoolDatabase: public DescriptorDatabase

```
#include <google/protobuf/descriptor_database.h>
```

`namespace google::protobuf`

A DescriptorDatabase that fetches files from a given pool.

Members	
	DescriptorPoolDatabase(const DescriptorPool & pool)
	~DescriptorPoolDatabase()
implements DescriptorDatabase	
virtual bool	**FindFileByName**(const string & filename, FileDescriptorProto * output) *Find a file by file name. more...*
virtual bool	**FindFileContainingSymbol**(const string & symbol_name, FileDescriptorProto * output) *Find the file that declares the given fully-qualified symbol name. more...*
virtual bool	**FindFileContainingExtension**(const string & containing_type, int field_number, FileDescriptorProto * output) *Find the file which defines an extension extending the given message type with the given field number. more...*

```
virtual bool DescriptorPoolDatabase::FindFileByName(
        const string & filename,
        FileDescriptorProto * output)
```

Find a file by file name.

Fills in in *output and returns true if found. Otherwise, returns false, leaving the contents of *output undefined.

```
virtual bool DescriptorPoolDatabase::FindFileContainingSymbol(
        const string & symbol_name,
        FileDescriptorProto * output)
```

Find the file that declares the given fully-qualified symbol name.

If found, fills in *output and returns true, otherwise returns false and leaves *output undefined.

```
virtual bool DescriptorPoolDatabase::FindFileContainingExtension(
        const string & containing_type,
        int field_number,
        FileDescriptorProto * output)
```

Find the file which defines an extension extending the given message type with the given field number.

If found, fills in *output and returns true, otherwise returns false and leaves *output undefined. containing_type must be a fully-qualified type name.

class MergedDescriptorDatabase: public DescriptorDatabase

```
#include <google/protobuf/descriptor_database.h>
namespace google::protobuf
```

A DescriptorDatabase that wraps two or more others.

It first searches the first database and, if that fails, tries the second, and so on.

Members	
	MergedDescriptorDatabase(DescriptorDatabase * source1, DescriptorDatabase * source2) *Merge just two databases. The sources remain property of the caller.*
	MergedDescriptorDatabase(const vector< DescriptorDatabase * > & sources) *Merge more than two databases. more...*
	~MergedDescriptorDatabase( )
implements DescriptorDatabase	
virtual bool	**FindFileByName**(const string & filename, FileDescriptorProto * output) *Find a file by file name. more...*
virtual bool	**FindFileContainingSymbol**(const string & symbol_name, FileDescriptorProto * output) *Find the file that declares the given fully-qualified symbol name. more...*
virtual bool	**FindFileContainingExtension**(const string & containing_type, int field_number, FileDescriptorProto * output) *Find the file which defines an extension extending the given message type with the given field number. more...*

```
MergedDescriptorDatabase::MergedDescriptorDatabase(
        const vector< DescriptorDatabase * > & sources)
```

Merge more than two databases.

The sources remain property of the caller. The vector may be deleted after the constructor returns but the DescriptorDatabases need to stick around.

```
virtual bool MergedDescriptorDatabase::FindFileByName(
        const string & filename,
        FileDescriptorProto * output)
```

Find a file by file name.

Fills in in *output and returns true if found. Otherwise, returns false, leaving the contents of *output undefined.

```
virtual bool MergedDescriptorDatabase::FindFileContainingSymbol(
        const string & symbol_name,
        FileDescriptorProto * output)
```

Find the file that declares the given fully-qualified symbol name.

If found, fills in *output and returns true, otherwise returns false and leaves *output undefined.

```
virtual bool MergedDescriptorDatabase::FindFileContainingExtension
```

```
const string & containing_type,
int field_number,
FileDescriptorProto * output)
```

Find the file which defines an extension extending the given message type with the given field number.

If found, fills in *output and returns true, otherwise returns false and leaves *output undefined. containing_type must be a fully-qualified type name.

dynamic_message.h

```
#include <google/protobuf/dynamic_message.h>
namespace google::protobuf
```

Defines an implementation of Message which can emulate types which are not known at compile-time.

Classes in this file

DynamicMessageFactory *Constructs implementations of Message which can emulate types which are not known at compile-time.*

class DynamicMessageFactory: public MessageFactory

```
#include <google/protobuf/dynamic_message.h>
namespace google::protobuf
```

Constructs implementations of Message which can emulate types which are not known at compile-time.

Sometimes you want to be able to manipulate protocol types that you don't know about at compile time. It would be nice to be able to construct a Message object which implements the message type given by any arbitrary Descriptor. DynamicMessage provides this.

As it turns out, a DynamicMessage needs to construct extra information about its type in order to operate. Most of this information can be shared between all DynamicMessages of the same type. But, caching this information in some sort of global map would be a bad idea, since the cached information for a particular descriptor could outlive the descriptor itself. To avoid this problem, DynamicMessageFactory encapsulates this "cache". All DynamicMessages of the same type created from the same factory will share the same support data. Any Descriptors used with a particular factory must outlive the factory.

Members

	DynamicMessageFactory() *Construct a DynamicMessageFactory that will search for extensions in the DescriptorPool in which the exendee is defined.*
	DynamicMessageFactory(const DescriptorPool * pool) *Construct a DynamicMessageFactory that will search for extensions in the given DescriptorPool.*
	~DynamicMessageFactory()
implements MessageFactory	
virtual const Message *	**GetPrototype**(const Descriptor * type) *Given a Descriptor, constructs the default (prototype) Message of that type. more...*

```
virtual const Message * DynamicMessageFactory::GetPrototype(
        const Descriptor * type)
```

Given a Descriptor, constructs the default (prototype) Message of that type.

You can then call that message's New() method to construct a mutable message of that type.

Calling this method twice with the same Descriptor returns the same object. The returned object remains property of the factory and will be destroyed when the factory is destroyed. Also, any objects created by calling the prototype's New() method share some data with the prototype, so these must be destoyed before the DynamicMessageFactory is destroyed.

The given descriptor must outlive the returned message, and hence must outlive the DynamicMessageFactory.

Note that while GetPrototype() is idempotent, it is not const. This implies that it is not thread-safe to call GetPrototype() on the same DynamicMessageFactory in two different threads simultaneously. However, the returned objects are just as thread-safe as any other Message.

message.h

```
#include <google/protobuf/message.h>
namespace google::protobuf
```

This file contains the abstract interface for all protocol messages.

Although it's possible to implement this interface manually, most users will use the protocol compiler to generate implementations.

Example usage:

Say you have a message defined as:

```
message Foo {
  optional string text = 1;
  repeated int32 numbers = 2;
}
```

Then, if you used the protocol compiler to generate a class from the above definition, you could use it like so:

```
string data;  // Will store a serialized version of the message.

{
  // Create a message and serialize it.
  Foo foo;
  foo.set_text("Hello World!");
  foo.add_numbers(1);
  foo.add_numbers(5);
  foo.add_numbers(42);

  foo.SerializeToString(&data);
}

{
  // Parse the serialized message and check that it contains the
  // correct data.
  Foo foo;
  foo.ParseFromString(data);

  assert(foo.text() == "Hello World!");
  assert(foo.numbers_size() == 3);
  assert(foo.numbers(0) == 1);
  assert(foo.numbers(1) == 5);
  assert(foo.numbers(2) == 42);
}

{
  // Same as the last block, but do it dynamically via the Message
  // reflection interface.
  Message* foo = new Foo;
  Descriptor* descriptor = foo->GetDescriptor();

  // Get the descriptors for the fields we're interested in and verify
  // their types.
  FieldDescriptor* text_field = descriptor->FindFieldByName("text");
  assert(text_field != NULL);
```

```
  assert(text_field->type() == FieldDescriptor::TYPE_STRING );
  assert(text_field->label() == FieldDescriptor::TYPE_OPTIONAL);
  FieldDescriptor* numbers_field = descriptor->FindFieldByName("numbers");
  assert(numbers_field != NULL);
  assert(numbers_field->type() == FieldDescriptor::TYPE_INT32 );
  assert(numbers_field->label() == FieldDescriptor::TYPE_REPEATED);

  // Parse the message.
  foo->ParseFromString(data);

  // Use the reflection interface to examine the contents.
  Reflection* reflection = foo->GetReflection();
  assert(reflection->GetString(foo, text_field) == "Hello World!");
  assert(reflection->CountField(foo, numbers_field) == 3);
  assert(reflection->GetInt32(foo, numbers_field, 0) == 1);
  assert(reflection->GetInt32(foo, numbers_field, 1) == 5);
  assert(reflection->GetInt32(foo, numbers_field, 2) == 42);

  delete foo;
}
```

Classes in this file

`Message`	*Abstract interface for protocol messages.*
`Reflection`	*This interface contains methods that can be used to dynamically access and modify the fields of a protocol message.*
`MessageFactory`	*Abstract interface for a factory for message objects.*

class Message

```
#include <google/protobuf/message.h>
namespace google::protobuf
```

Abstract interface for protocol messages.

The methods of this class that are virtual but not pure-virtual have default implementations based on reflectionMessage classes which are optimized for speed will want to override these with faster implementations, but classes optimized for code size may be happy with keeping them. See the optimize_for option in descriptor.proto.

Known subclasses:

- `DescriptorProto`
- `DescriptorProto_ExtensionRange`
- `EnumDescriptorProto`
- `EnumOptions`
- `EnumValueDescriptorProto`
- `EnumValueOptions`
- `FieldDescriptorProto`
- `FieldOptions`
- `FileDescriptorProto`
- `FileDescriptorSet`
- `FileOptions`
- `MessageOptions`
- `MethodDescriptorProto`
- `MethodOptions`
- `ServiceDescriptorProto`

- ServiceOptions

Members	
	Message()
virtual	**~Message**()

Introspection	
typedef	google::protobuf::Reflection **Reflection** *Typedef for backwards-compatibility.*
virtual const Descriptor *	**GetDescriptor**() const = 0 *Get a Descriptor for this message's type. more...*
virtual const Reflection *	**GetReflection**() const = 0 *Get the Reflection interface for this Message, which can be used to read and modify the fields of the Message dynamically (in other words, without knowing the message type at compile time). more...*

Basic Operations	
virtual Message *	**New**() const = 0 *Construct a new instance of the same type. more...*
virtual void	**CopyFrom**(const Message & from) *Make this message into a copy of the given message. more...*
virtual void	**MergeFrom**(const Message & from) *Merge the fields from the given message into this message. more...*
virtual void	**Clear**() *Clear all fields of the message and set them to their default values. more...*
virtual bool	**IsInitialized**() const *Quickly check if all required fields have values set.*
void	**CheckInitialized**() const *Verifies that IsInitialized() returns true. more...*
void	**FindInitializationErrors**(vector< string > * errors) const *Slowly build a list of all required fields that are not set. more...*
string	**InitializationErrorString**() const *Like FindInitializationErrors, but joins all the strings, delimited by commas, and returns them.*
virtual void	**DiscardUnknownFields**() *Clears all unknown fields from this message and all embedded messages. more...*

Debugging	
string	**DebugString**() const *Generates a human readable form of this message, useful for debugging and other purposes.*
string	**ShortDebugString**() const *Like DebugString(), but with less whitespace.*
void	**PrintDebugString**() const *Convenience function useful in GDB. Prints DebugString() to stdout.*

Parsing	

Methods for parsing in protocol buffer format. Most of these are just simple wrappers around MergeFromCodedStream().

bool	**ParseFromCodedStream**(io::CodedInputStream * input) *Fill the message with a protocol buffer parsed from the given input stream. more...*
bool	**ParsePartialFromCodedStream**(io::CodedInputStream * input) *Like ParseFromCodedStream(), but accepts messages that are missing required fields.*
bool	**ParseFromZeroCopyStream**(io::ZeroCopyInputStream * input) *Read a protocol buffer from the given zero-copy input stream. more...*
bool	**ParsePartialFromZeroCopyStream**(io::ZeroCopyInputStream * input) *Like ParseFromZeroCopyStream(), but accepts messages that are missing required fields.*
bool	**ParseFromString**(const string & data) *Parse a protocol buffer contained in a string.*
bool	**ParsePartialFromString**(const string & data) *Like ParseFromString(), but accepts messages that are missing required fields.*
bool	**ParseFromArray**(const void * data, int size) *Parse a protocol buffer contained in an array of bytes.*
bool	**ParsePartialFromArray**(const void * data, int size) *Like ParseFromArray(), but accepts messages that are missing required fields.*
bool	**ParseFromFileDescriptor**(int file_descriptor) *Parse a protocol buffer from a file descriptor. more...*
bool	**ParsePartialFromFileDescriptor**(int file_descriptor) *Like ParseFromFileDescriptor(), but accepts messages that are missing required fields.*
bool	**ParseFromIstream**(istream * input) *Parse a protocol buffer from a C++ istream. more...*
bool	**ParsePartialFromIstream**(istream * input) *Like ParseFromIstream(), but accepts messages that are missing required fields.*
bool	**MergeFromCodedStream**(io::CodedInputStream * input) *Reads a protocol buffer from the stream and merges it into this Message. more...*
virtual bool	**MergePartialFromCodedStream**(io::CodedInputStream * input) *Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input. more...*

Serialization

Methods for serializing in protocol buffer format. Most of these are just simple wrappers around ByteSize() and SerializeWithCachedSizes().

bool	**SerializeToCodedStream**(io::CodedOutputStream * output) const *Write a protocol buffer of this message to the given output. more...*
bool	**SerializePartialToCodedStream**(io::CodedOutputStream * output) const *Like SerializeToCodedStream(), but allows missing required fields.*
bool	**SerializeToZeroCopyStream**(io::ZeroCopyOutputStream * output) const *Write the message to the given zero-copy output stream. more...*

bool	**SerializePartialToZeroCopyStream**(io::ZeroCopyOutputStream * output) const *Like SerializeToZeroCopyStream(), but allows missing required fields.*
bool	**SerializeToString**(string * output) const *Serialize the message and store it in the given string. more...*
bool	**SerializePartialToString**(string * output) const *Like SerializeToString(), but allows missing required fields.*
bool	**SerializeToArray**(void * data, int size) const *Serialize the message and store it in the given byte array. more...*
bool	**SerializePartialToArray**(void * data, int size) const *Like SerializeToArray(), but allows missing required fields.*
bool	**SerializeToFileDescriptor**(int file_descriptor) const *Serialize the message and write it to the given file descriptor. more...*
bool	**SerializePartialToFileDescriptor**(int file_descriptor) const *Like SerializeToFileDescriptor(), but allows missing required fields.*
bool	**SerializeToOstream**(ostream * output) const *Serialize the message and write it to the given C++ ostream. more...*
bool	**SerializePartialToOstream**(ostream * output) const *Like SerializeToOstream(), but allows missing required fields.*
bool	**AppendToString**(string * output) const *Like SerializeToString(), but appends to the data to the string's existing contents. more...*
bool	**AppendPartialToString**(string * output) const *Like AppendToString(), but allows missing required fields.*
virtual int	**ByteSize**() const *Computes the serialized size of the message. more...*
virtual bool	**SerializeWithCachedSizes**(io::CodedOutputStream * output) const *Serializes the message without recomputing the size. more...*
virtual int	**GetCachedSize**() const = 0 *Returns the result of the last call to ByteSize(). more...*

---

virtual const Descriptor * Message::GetDescriptor() const = 0

Get a Descriptor for this message's type.

This describes what fields the message contains, the types of those fields, etc.

---

virtual const Reflection * Message::GetReflection() const = 0

Get the Reflection interface for this Message, which can be used to read and modify the fields of theMessage dynamically (in other words, without knowing the message type at compile time).

This object remains property of the Message.

virtual Message * Message::New() const = 0

Construct a new instance of the same type.

Ownership is passed to the caller.

---

virtual void Message::CopyFrom(
const Message & from)

Make this message into a copy of the given message.

The given message must have the same descriptor, but need not necessarily be the same class. By default this is just implemented as "Clear(); MergeFrom(from);".

---

virtual void Message::MergeFrom(
const Message & from)

Merge the fields from the given message into this message.

Singular fields will be overwritten, except for embedded messages which will be merged. Repeated fields will be concatenated. The given message must be of the same type as this message (i.e. the exact same class).

---

virtual void Message::Clear()

Clear all fields of the message and set them to their default values.

Clear() avoids freeing memory, assuming that any memory allocated to hold parts of the message will be needed again to hold the next message. If you actually want to free the memory used by a Message, you must delete it.

---

void Message::CheckInitialized() const

Verifies that IsInitialized() returns true.

GOOGLE_CHECK-fails otherwise, with a nice error message.

---

void Message::FindInitializationErrors(
vector< string > * errors) const

Slowly build a list of all required fields that are not set.

This is much, much slower than IsInitialized() as it is implemented purely via reflection. Generally, you should not call this unless you have already determined that an error exists by calling IsInitialized().

---

virtual void Message::DiscardUnknownFields()

Clears all unknown fields from this message and all embedded messages.

Normally, if unknown tag numbers are encountered when parsing a message, the tag and value are stored in the message's UnknownFieldSet and then written back out when the message is serialized. This allows servers which simply route messages to other servers to pass through messages that have new field definitions which they don't yet know about. However, this behavior can have security implications. To avoid it, call this method after parsing.

See Reflection::GetUnknownFields() for more on unknown fields.

```
bool Message::ParseFromCodedStream(
        io::CodedInputStream * input)
```

Fill the message with a protocol buffer parsed from the given input stream.

Returns false on a read error or if the input is in the wrong format.

```
bool Message::ParseFromZeroCopyStream(
        io::ZeroCopyInputStream * input)
```

Read a protocol buffer from the given zero-copy input stream.

If successful, the entire input will be consumed.

```
bool Message::ParseFromFileDescriptor(
        int file_descriptor)
```

Parse a protocol buffer from a file descriptor.

If successful, the entire input will be consumed.

```
bool Message::ParseFromIstream(
        istream * input)
```

Parse a protocol buffer from a C++ istream.

If successful, the entire input will be consumed.

```
bool Message::MergeFromCodedStream(
        io::CodedInputStream * input)
```

Reads a protocol buffer from the stream and merges it into this Message.

Singular fields read from the input overwrite what is already in the Message and repeated fields are appended to those already present.

It is the responsibility of the caller to call input->LastTagWas() (for groups) or input->ConsumedEntireMessage() (for non-groups) after this returns to verify that the message's end was delimited correctly.

ParsefromCodedStream() is implemented as Clear() followed by MergeFromCodedStream().

```
virtual bool Message::MergePartialFromCodedStream(
        io::CodedInputStream * input)
```

Like MergeFromCodedStream(), but succeeds even if required fields are missing in the input.

MergeFromCodedStream() is just implemented as MergePartialFromCodedStream() followed by IsInitialized()

```
bool Message::SerializeToCodedStream(
        io::CodedOutputStream * output) const
```

Write a protocol buffer of this message to the given output.

Returns false on a write error. If the message is missing required fields, this may GOOGLE_CHECK-fail.

```
bool Message::SerializeToZeroCopyStream(
        io::ZeroCopyOutputStream * output) const
```

Write the message to the given zero-copy output stream.

All required fields must be set.

```
bool Message::SerializeToString(
        string * output) const
```

Serialize the message and store it in the given string.

All required fields must be set.

```
bool Message::SerializeToArray(
        void * data,
        int size) const
```

Serialize the message and store it in the given byte array.

All required fields must be set.

```
bool Message::SerializeToFileDescriptor(
        int file_descriptor) const
```

Serialize the message and write it to the given file descriptor.

All required fields must be set.

```
bool Message::SerializeToOstream(
        ostream * output) const
```

Serialize the message and write it to the given C++ ostream.

All required fields must be set.

```
bool Message::AppendToString(
        string * output) const
```

Like SerializeToString(), but appends to the data to the string's existing contents.

All required fields must be set.

```
virtual int Message::ByteSize() const
```

Computes the serialized size of the message.

This recursively callsByteSize() on all embedded messages. If a subclass does not override this, it MUST override SetCachedSize().

```
virtual bool Message::SerializeWithCachedSizes(
        io::CodedOutputStream * output) const
```

Serializes the message without recomputing the size.

The message must not have changed since the last call toByteSize(); if it has, the results are undefined.

```
virtual int Message::GetCachedSize() const = 0
```

Returns the result of the last call toByteSize().

An embedded message's size is needed both to serialize it (because embedded messages are length-delimited) and to compute the outer message's size. Caching the size avoids computing it multiple times.

ByteSize() does not automatically use the cached size when available because this would require invalidating it every time the message was modified, which would be too hard and expensive. (E.g. if a deeply-nested sub-message is changed, all of its parents' cached sizes would need to be invalidated, which is too much work for an otherwise inlined setter method.)

class Reflection

```
#include <google/protobuf/message.h>
namespace google::protobuf
```

This interface contains methods that can be used to dynamically access and modify the fields of a protocol message.

Their semantics are similar to the accessors the protocol compiler generates.

To get the Reflection for a given Message, call Message::GetReflection().

This interface is separate from Message only for efficiency reasons; the vast majority of implementations ofMessage will share the same implementation of Reflection (GeneratedMessageReflection, defined in generated_message.h), and all Messages of a particular class should share the sameReflection object (though you should not rely on the latter fact).

There are several ways that these methods can be used incorrectly. For example, any of the following conditions will lead to undefined results (probably assertion failures):

- The FieldDescriptor is not a field of this message type.
- The method called is not appropriate for the field's type. For each field type in FieldDescriptor::TYPE_*, there is only one Get*() method, one Set*() method, and one Add*() method that is valid for that type. It should be obvious which (except maybe for TYPE_BYTES, which are represented using strings in C++).
- A Get*() or Set*() method for singular fields is called on a repeated field.
- GetRepeated*(), SetRepeated*(), or Add*() is called on a non-repeated field.

- The Message object passed to any method is not of the right type for this Reflection object (i.e. message.GetReflection() != reflection).

You might wonder why there is not any abstract representation for a field of arbitrary type. E.g., why isn't there just a "GetField()" method that returns "const Field&", where "Field" is some class with accessors like "GetInt32Value()". The problem is that someone would have to deal with allocating these Field objects. For generated message classes, having to allocate space for an additional object to wrap every field would at least double the message's memory footprint, probably worse. Allocating the objects on-demand, on the other hand, would be expensive and prone to memory leaks. So, instead we ended up with this flat interface.

TODO(kenton): Create a utility class which callers can use to read and write fields from a Reflection without paying attention to the type.

Members

	`Reflection()` *TODO(kenton): Remove parameter.*
`virtual`	`~Reflection()`
`virtual const UnknownFieldSet &`	`GetUnknownFields(const Message & message) const = 0` *Get the UnknownFieldSet for the message. more...*
`virtual UnknownFieldSet *`	`MutableUnknownFields(Message * message) const = 0` *Get a mutable pointer to the UnknownFieldSet for the message. more...*
`virtual bool`	`HasField(const Message & message, const FieldDescriptor * field) const = 0` *Check if the given non-repeated field is set.*
`virtual int`	`FieldSize(const Message & message, const FieldDescriptor * field) const = 0` *Get the number of elements of a repeated field.*
`virtual void`	`ClearField(Message * message, const FieldDescriptor * field) const = 0` *Clear the value of a field, so that HasField() returns false or FieldSize() returns zero.*
`virtual void`	`ListFields(const Message & message, vector< const FieldDescriptor * > * output) const = 0` *List all fields of the message which are currently set. more...*

Singular field getters

These get the value of a non-repeated field. They return the default value for fields that aren't set.

`virtual int32`	`GetInt32(const Message & message, const FieldDescriptor * field) const = 0`
`virtual int64`	`GetInt64(const Message & message, const FieldDescriptor * field) const = 0`
`virtual uint32`	`GetUInt32(const Message & message, const FieldDescriptor * field) const = 0`
`virtual uint64`	`GetUInt64(const Message & message, const FieldDescriptor * field) const = 0`
`virtual float`	`GetFloat(const Message & message, const FieldDescriptor * field) const = 0`
`virtual double`	`GetDouble(const Message & message, const FieldDescriptor * field) const = 0`
`virtual bool`	`GetBool(const Message & message, const FieldDescriptor * field) const = 0`

virtual string	**GetString**(const Message & message, const FieldDescriptor * field) const = 0
virtual const EnumValueDescriptor *	**GetEnum**(const Message & message, const FieldDescriptor * field) const = 0
virtual const Message &	**GetMessage**(const Message & message, const FieldDescriptor * field) const = 0
virtual const string &	**GetStringReference**(const Message & message, const FieldDescriptor * field, string * scratch) const = 0 *Get a string value without copying, if possible.* more...

Singular field mutators

These mutate the value of a non-repeated field.

virtual void	**SetInt32**(Message * message, const FieldDescriptor * field, int32 value) const = 0
virtual void	**SetInt64**(Message * message, const FieldDescriptor * field, int64 value) const = 0
virtual void	**SetUInt32**(Message * message, const FieldDescriptor * field, uint32 value) const = 0
virtual void	**SetUInt64**(Message * message, const FieldDescriptor * field, uint64 value) const = 0
virtual void	**SetFloat**(Message * message, const FieldDescriptor * field, float value) const = 0
virtual void	**SetDouble**(Message * message, const FieldDescriptor * field, double value) const = 0
virtual void	**SetBool**(Message * message, const FieldDescriptor * field, bool value) const = 0
virtual void	**SetString**(Message * message, const FieldDescriptor * field, const string & value) const = 0
virtual void	**SetEnum**(Message * message, const FieldDescriptor * field, const EnumValueDescriptor * value) const = 0
virtual Message *	**MutableMessage**(Message * message, const FieldDescriptor * field) const = 0 *Get a mutable pointer to a field with a message type.*

Repeated field getters

These get the value of one element of a repeated field.

virtual int32	**GetRepeatedInt32**(const Message & message, const FieldDescriptor * field, int index) const = 0
virtual int64	**GetRepeatedInt64**(const Message & message, const FieldDescriptor * field, int index) const = 0
virtual uint32	**GetRepeatedUInt32**(const Message & message, const FieldDescriptor * field, int index) const = 0
virtual uint64	**GetRepeatedUInt64**(const Message & message, const FieldDescriptor * field, int index) const = 0
virtual float	**GetRepeatedFloat**(const Message & message, const FieldDescriptor * field, int index) const = 0

virtual double	**GetRepeatedDouble**(const Message & message, const FieldDescriptor * field, int index) const = 0
virtual bool	**GetRepeatedBool**(const Message & message, const FieldDescriptor * field, int index) const = 0
virtual string	**GetRepeatedString**(const Message & message, const FieldDescriptor * field, int index) const = 0
virtual const EnumValueDescriptor *	**GetRepeatedEnum**(const Message & message, const FieldDescriptor * field, int index) const = 0
virtual const Message &	**GetRepeatedMessage**(const Message & message, const FieldDescriptor * field, int index) const = 0
virtual const string &	**GetRepeatedStringReference**(const Message & message, const FieldDescriptor * field, int index, string * scratch) const = 0 *See GetStringReference(), above.*

Repeated field mutators

These mutate the value of one element of a repeated field.

virtual void	**SetRepeatedInt32**(Message * message, const FieldDescriptor * field, int index, int32 value) const = 0
virtual void	**SetRepeatedInt64**(Message * message, const FieldDescriptor * field, int index, int64 value) const = 0
virtual void	**SetRepeatedUInt32**(Message * message, const FieldDescriptor * field, int index, uint32 value) const = 0
virtual void	**SetRepeatedUInt64**(Message * message, const FieldDescriptor * field, int index, uint64 value) const = 0
virtual void	**SetRepeatedFloat**(Message * message, const FieldDescriptor * field, int index, float value) const = 0
virtual void	**SetRepeatedDouble**(Message * message, const FieldDescriptor * field, int index, double value) const = 0
virtual void	**SetRepeatedBool**(Message * message, const FieldDescriptor * field, int index, bool value) const = 0
virtual void	**SetRepeatedString**(Message * message, const FieldDescriptor * field, int index, const string & value) const = 0
virtual void	**SetRepeatedEnum**(Message * message, const FieldDescriptor * field, int index, const EnumValueDescriptor * value) const = 0
virtual Message *	**MutableRepeatedMessage**(Message * message, const FieldDescriptor * field, int index) const = 0 *Get a mutable pointer to an element of a repeated field with a message type.*

Repeated field adders

These add an element to a repeated field.

virtual void	**AddInt32**(Message * message, const FieldDescriptor * field, int32 value) const = 0
virtual void	**AddInt64**(Message * message, const FieldDescriptor * field, int64 value) const = 0
virtual void	**AddUInt32**(Message * message, const FieldDescriptor * field, uint32 value) const = 0
virtual void	**AddUInt64**(Message * message, const FieldDescriptor * field, uint64 value) const = 0
virtual void	**AddFloat**(Message * message, const FieldDescriptor * field, float value) const = 0
virtual void	**AddDouble**(Message * message, const FieldDescriptor * field, double value) const = 0
virtual void	**AddBool**(Message * message, const FieldDescriptor * field, bool value) const = 0
virtual void	**AddString**(Message * message, const FieldDescriptor * field, const string & value) const = 0
virtual void	**AddEnum**(Message * message, const FieldDescriptor * field, const EnumValueDescriptor * value) const = 0
virtual Message *	**AddMessage**(Message * message, const FieldDescriptor * field) const = 0
Extensions	
virtual const FieldDescriptor *	**FindKnownExtensionByName**(const string & name) const = 0 *Try to find an extension of this message type by fully-qualified field name. more...*
virtual const FieldDescriptor *	**FindKnownExtensionByNumber**(int number) const = 0 *Try to find an extension of this message type by field number. more...*

```
virtual const UnknownFieldSet &
    Reflection::GetUnknownFields(
        const Message & message) const = 0
```

Get the UnknownFieldSet for the message.

This contains fields which were seen when the Message was parsed but were not recognized according to the Message's definition.

```
virtual UnknownFieldSet *
    Reflection::MutableUnknownFields(
        Message * message) const = 0
```

Get a mutable pointer to the UnknownFieldSet for the message.

This contains fields which were seen when the Message was parsed but were not recognized according to the Message's definition.

```
virtual void Reflection::ListFields(
        const Message & message,
        vector< const FieldDescriptor * > * output) const = 0
```

List all fields of the message which are currently set.

This includes extensions. Singular fields will only be listed if HasField(field) would return true and repeated fields will only be listed if FieldSize(field) would return non-zero. Fields (both normal fields and extension fields) will be listed ordered by field number.

```
virtual const string & Reflection::GetStringReference(
        const Message & message,
        const FieldDescriptor * field,
        string * scratch) const = 0
```

Get a string value without copying, if possible.

GetString() necessarily returns a copy of the string. This can be inefficient when the string is already stored in a string object in the underlying message. GetStringReference() will return a reference to the underlying string in this case. Otherwise, it will copy the string into scratch and return that.

Note: It is perfectly reasonable and useful to write code like:

```
str = reflection->GetStringReference(field, &str);
```

This line would ensure that only one copy of the string is made regardless of the field's underlying representation. When initializing a newly-constructed string, though, it's just as fast and more readable to use code like:

```
string str = reflection->GetString(field);
```

```
virtual const FieldDescriptor *
    Reflection::FindKnownExtensionByName(
        const string & name) const = 0
```

Try to find an extension of this message type by fully-qualified field name.

Returns NULL if no extension is known for this name or number.

```
virtual const FieldDescriptor *
    Reflection::FindKnownExtensionByNumber(
        int number) const = 0
```

Try to find an extension of this message type by field number.

Returns NULL if no extension is known for this name or number.

class MessageFactory

```
#include <google/protobuf/message.h>
namespace google::protobuf
```

Abstract interface for a factory for message objects.

Known subclasses:

- DynamicMessageFactory

Members

	MessageFactory()
virtual	**~MessageFactory**()
virtual const Message *	**GetPrototype**(const Descriptor * type) = 0 *Given a Descriptor, gets or constructs the default (prototype) Message of that type. more...*
static MessageFactory *	**generated_factory**() *Gets a MessageFactory which supports all generated, compiled-in messages. more...*
static void	**InternalRegisterGeneratedMessage**(const Descriptor * descriptor, const Message * prototype) *For internal use only: Registers a message type at static initialization time, to be placed in generated_factory().*

---

virtual const Message * MessageFactory::GetPrototype(const Descriptor * type) = 0

Given a Descriptor, gets or constructs the default (prototype) Message of that type.

You can then call that message's New() method to construct a mutable message of that type.

Calling this method twice with the same Descriptor returns the same object. The returned object remains property of the factory. Also, any objects created by calling the prototype's New() method share some data with the prototype, so these must be destoyed before the MessageFactory is destroyed.

The given descriptor must outlive the returned message, and hence must outlive the MessageFactory.

Some implementations do not support all types. GetPrototype() will return NULL if the descriptor passed in is not supported.

This method may or may not be thread-safe depending on the implementation. Each implementation should document its own degree thread-safety.

---

static MessageFactory * MessageFactory::generated_factory()

Gets a MessageFactory which supports all generated, compiled-in messages.

In other words, for any compiled-in type FooMessage, the following is true:

```
MessageFactory::generated_factory ()->GetPrototype(
  FooMessage::descriptor()) == FooMessage::default_instance()
```

This factory supports all types which are found in DescriptorPool::generated_pool(). If given a descriptor from any other pool, GetPrototype() will return NULL. (You can also check if a descriptor is for a generated message by checking if descriptor->file()->pool() == DescriptorPool::generated_pool().)

This factory is 100% thread-safe; calling GetPrototype() does not modify any shared data.

This factory is a singleton. The caller must not delete the object.

repeated_field.h

```
#include <google/protobuf/repeated_field.h>
namespace google::protobuf
```

RepeatedField and RepeatedPtrField are used by generated protocol message classes to manipulate repeated fields.

These classes are very similar to STL's vector, but include a number of optimizations found to be useful specifically in the case of Protocol Buffers. RepeatedPtrField is particularly different from STL vector as it manages ownership of the pointers that it contains.

Typically, clients should not need to access RepeatedField objects directly, but should instead use the accessor functions generated automatically by the protocol compiler.

Classes in this file
RepeatedField *RepeatedField is used to represent repeated fields of a primitive type (in other words, everything except strings and nested Messages).*
RepeatedPtrField *RepeatedPtrField is like RepeatedField, but used for repeated strings or Messages.*

template class RepeatedField

```
#include <google/protobuf/repeated_field.h>
namespace google::protobuf

template <typename Element>
```

RepeatedField is used to represent repeated fields of a primitive type (in other words, everything except strings and nested Messages).

Most users will not ever use a RepeatedField directly; they will use the get-by-index, set-by-index, and add accessors that are generated for all repeated fields.

Members	
typedef	Element * **iterator** *STL-like iterator support.*
typedef	const Element * **const_iterator**
	RepeatedField()
	~RepeatedField()
int	**size**() const
Element	**Get**(int index) const
Element *	**Mutable**(int index)
void	**Set**(int index, Element value)
void	**Add**(Element value)

void	**RemoveLast**() *Remove the last element in the array. more...*
void	**Clear**()
void	**MergeFrom**(const RepeatedField & other)
void	**Reserve**(int new_size) *Reserve space to expand the field to at least the given size. more...*
Element *	**mutable_data**() *Gets the underlying array. more...*
const Element *	**data**() const
void	**Swap**(RepeatedField * other) *Swap entire contents with "other". more...*
iterator	**begin**()
const_iterator	**begin**() const
iterator	**end**()
const_iterator	**end**() const

---

void RepeatedField::RemoveLast()

Remove the last element in the array.

We don't provide a way to remove any element other than the last because it invites inefficient use, such as O(n^2) filtering loops that should have been O(n). If you want to remove an element other than the last, the best way to do it is to re-arrange the elements so that the one you want removed is at the end, then call RemoveLast().

---

void RepeatedField::Reserve(
int new_size)

Reserve space to expand the field to at least the given size.

If the array is grown, it will always be at least doubled in size.

Element * RepeatedField::mutable_data()

Gets the underlying array.

This pointer is possibly invalidated by any add or remove operation.

---

void RepeatedField::Swap(
RepeatedField * other)

Swap entire contents with "other".

We may not be using initial_space_ but it's not worth checking. Just copy it anyway.

template class RepeatedPtrField

```
#include <google/protobuf/repeated_field.h>
namespace google::protobuf

template <typename Element>
```

RepeatedPtrFieldis like RepeatedField but used for repeated strings or Messages.

Members

typedef	internal::RepeatedPtrIterator< Element ** > **iterator** *STL-like iterator support.*
typedef	internal::RepeatedPtrIterator< const Element *const * > **const_iterator**
	RepeatedPtrField()
explicit	**RepeatedPtrField**(const Message * prototype) *This constructor is only defined for RepeatedPtrField<Message>. more...*
	~RepeatedPtrField()
const Message *	**prototype**() const *Returns the prototype if one was passed to the constructor.*
int	**size**() const
const Element &	**Get**(int index) const
Element *	**Mutable**(int index)
Element *	**Add**()
void	**RemoveLast**() *Remove the last element in the array.*
void	**Clear**()
void	**MergeFrom**(const RepeatedPtrField & other)
void	**Reserve**(int new_size) *Reserve space to expand the field to at least the given size. more...*
Element **	**mutable_data**() *Gets the underlying array. more...*
const Element *const *	**data**() const
void	**Swap**(RepeatedPtrField * other) *Swap entire contents with "other". more...*
iterator	**begin**()
const_iterator	**begin**() const
iterator	**end**()
const_iterator	**end**() const
	RepeatedPtrField(const Message * prototype)
const Message *	**prototype**() const

Message *	**NewElement**()

Advanced memory management

When hardcore memory management becomes necessary -- as it often does here at Google -- the following methods may be useful.

void	**AddAllocated**(Element * value) *Add an already-allocated object, passing ownership to the RepeatedPtrField. more...*
Element *	**ReleaseLast**() *Remove the last element and return it, passing ownership to the caller. more...*
int	**ClearedCount**() *Get the number of cleared objects that are currently being kept around for reuse.*
void	**AddCleared**(Element * value) *Add an element to the pool of cleared objects, passing ownership to the RepeatedPtrField. more...*
Element *	**ReleaseCleared**() *Remove a single element from the cleared pool and return it, passing ownership to the caller. more...*

```
explicit RepeatedPtrField::RepeatedPtrField(
        const Message * prototype)
```

This constructor is only defined for RepeatedPtrField<Message>.

When a RepeatedPtrFieldis created using this constructor, prototype->New() will be called to allocate new elements, rather than just using the "new" operator. This is useful for the implementation of DynamicMessage, but is not used by normal generated messages.

```
void RepeatedPtrField::Reserve(
        int new_size)
```

Reserve space to expand the field to at least the given size.

This only resizes the pointer array; it doesn't allocate any objects. If the array is grown, it will always be at least doubled in size.

```
Element ** RepeatedPtrField::mutable_data()
```

Gets the underlying array.

This pointer is possibly invalidated by any add or remove operation.

```
void RepeatedPtrField::Swap(
        RepeatedPtrField * other)
```

Swap entire contents with "other".

We may not be using initial_space_ but it's not worth checking. Just copy it anyway.

```
void RepeatedPtrField::AddAllocated(
        Element * value)
```

Add an already-allocated object, passing ownership to the RepeatedPtrField

We don't care about the order of cleared elements, so if there's one in the way, just move it to the back of the array.

```
Element * RepeatedPtrField::ReleaseLast()
```

Remove the last element and return it, passing ownership to the caller.

Requires: size() > 0

```
void RepeatedPtrField::AddCleared(
        Element * value)
```

Add an element to the pool of cleared objects, passing ownership to the RepeatedPtrField

The element must be cleared prior to calling this method.

```
Element * RepeatedPtrField::ReleaseCleared()
```

Remove a single element from the cleared pool and return it, passing ownership to the caller.

The element is guaranteed to be cleared. Requires ClearedCount() > 0

service.h

`#include <google/protobuf/service.h>`
`namespace google::protobuf`

This module declares the abstract interfaces underlying proto2 RPC services.

These are intented to be independent of any particular RPC implementation, so that proto2 services can be used on top o a variety of implementations.

When you use the protocol compiler to compile a service definition, it generates two classes: An abstract interface for the service (with methods matching the service definition) and a "stub" implementation. A stub is just a type-safe wrapper around an RpcChannel which emulates a local implementation of the service.

For example, the service definition:

```
service MyService {
  rpc Foo(MyRequest) returns(MyResponse);
}
```

will generate abstract interface "MyService" and class "MyService::Stub". You could implement a MyService as follows:

```
class MyServiceImpl : public MyService {
 public:
  MyServiceImpl() {}
  ~MyServiceImpl() {}

  // implements MyService ---------------------------------------

  void Foo(google::protobuf::RpcController* controller,
           const MyRequest* request,
           MyResponse* response,
           Closure* done) {
    // ... read request and fill in response ...
    done->Run();
  }
};
```

You would then register an instance of MyServiceImpl with your RPC server implementation. (How to do that depends on the implementation.)

To call a remote MyServiceImpl, first you need an RpcChannel connected to it. How to construct a channel depends, again, on your RPC implementation. Here we use a hypothentical "MyRpcChannel" as an example:

```
MyRpcChannel channel("rpc:hostname:1234/myservice");
MyRpcController controller;
MyServiceImpl::Stub stub(&channel);
FooRequest request;
FooRespnose response;

// ... fill in request ...

stub.Foo(&controller, request, &response, NewCallback(HandleResponse));
```

On Thread-Safety:

Different RPC implementations may make different guarantees about what threads they may run callbacks on, and what threads the application is allowed to use to call the RPC system. Portable software should be ready for callbacks to be called on any thread, but should not try to call the RPC system from any thread except for the ones on which it received the callbacks. Realistically, though, simple software will probably want to use a single-threaded RPC system while high-end software will want to use multiple threads. RPC implementations should provide multiple choices.

Classes in this file
Service *Abstract base interface for protocol-buffer-based RPC services.*
RpcController *An RpcController mediates a single method call.*
RpcChannel *Abstract interface for an RPC channel.*

class Service

```
#include <google/protobuf/service.h>
namespace google::protobuf
```

Abstract base interface for protocol-buffer-based RPC services.

Services themselves are abstract interfaces (implemented either by servers or as stubs), but they subclass this base interface. The methods of this interface can be used to call the methods of the Service without knowing its exact type at compile time (analogous to Reflection).

Members	
enum	**ChannelOwnership** *When constructing a stub, you may pass STUB_OWNS_CHANNEL as the second parameter to the constructor to tell it to delete its RpcChannel when destroyed. more...*
	Service()
virtual	**~Service**()
virtual const ServiceDescriptor *	**GetDescriptor**() = 0 *Get the ServiceDescriptor describing this service and its methods.*
virtual void	**CallMethod**(const MethodDescriptor * method, RpcController * controller, const Message * request, Message * response, Closure * done) = 0 *Call a method of the service specified by MethodDescriptor. more...*
virtual const Message &	**GetRequestPrototype**(const MethodDescriptor * method) const = 0 *CallMethod() requires that the request and response passed in are of a particular subclass of Message. more...*
virtual const Message &	**GetResponsePrototype**(const MethodDescriptor * method) const = 0

```
enum Service::ChannelOwnership {
  STUB_OWNS_CHANNEL,
  STUB_DOESNT_OWN_CHANNEL
}
```

When constructing a stub, you may pass STUB_OWNS_CHANNEL as the second parameter to the constructor to tell it to delete its RpcChannel when destroyed.

STUB_OWNS_CHANNEL	
STUB_DOESNT_OWN_CHANNEL	

```
virtual void Service::CallMethod(
        const MethodDescriptor * method,
        RpcController * controller,
        const Message * request,
        Message * response,
        Closure * done) = 0
```

Call a method of the service specified by MethodDescriptor.

This is normally implemented as a simple switch() that calls the standard definitions of the service's methods.

Preconditions:

- method->service() == GetDescriptor()
- request and response are of the exact same classes as the objects returned by GetRequestPrototype(method) and GetResponsePrototype(method).
- After the call has started, the request must not be modified and the response must not be accessed at all until "done" is called.
- "controller" is of the correct type for the RPC implementation being used by this Service. For stubs, the "correct type" depends on the RpcChannel which the stub is using. Server-side Service implementations are expected to accept whatever type of RpcController the server-side RPC implementation uses.

Postconditions:

- "done" will be called when the method is complete. This may be before CallMethod() returns or it may be at some point in the future.
- If the RPC succeeded, "response" contains the response returned by the server.
- If the RPC failed, "response"'s contents are undefined. The RpcController can be queried to determine if an error occurred and possibly to get more information about the error.

```
virtual const Message & Service::GetRequestPrototype(
        const MethodDescriptor * method) const = 0
```

CallMethod() requires that the request and response passed in are of a particular subclass of Message.

GetRequestPrototype() and GetResponsePrototype() get the default instances of these required types. You can then call Message::New() on these instances to construct mutable objects which you can then pass to CallMethod().

Example:

```
const MethodDescriptor* method =
  service->GetDescriptor()->FindMethodByName("Foo");
Message* request  = stub->GetRequestPrototype (method)->New();
Message* response = stub->GetResponsePrototype(method)->New();
request->ParseFromString(input);
service->CallMethod(method, *request, response, callback);
```

class RpcController

```
#include <google/protobuf/service.h>
namespace google::protobuf
```

An RpcController mediates a single method call.

The primary purpose of the controller is to provide a way to manipulate settings specific to the RPC implementation and t find out about RPC-level errors.

The methods provided by the RpcController interface are intended to be a "least common denominator" set of features which we expect all implementations to support. Specific implementations may provide more advanced features (e.g. deadline propagation).

Members	
	RpcController()
virtual	**~RpcController**()
Client-side methods *These calls may be made from the client side only. Their results are undefined on the server side (may crash).*	
virtual void	**Reset**() = 0 *Resets the RpcController to its initial state so that it may be reused in a new call. more...*
virtual bool	**Failed**() const = 0 *After a call has finished, returns true if the call failed. more...*
virtual string	**ErrorText**() const = 0 *If Failed() is true, returns a human-readable description of the error.*
virtual void	**StartCancel**() = 0 *Advises the RPC system that the caller desires that the RPC call be canceled. more...*
Server-side methods *These calls may be made from the server side only. Their results are undefined on the client side (may crash).*	
virtual void	**SetFailed**(const string & reason) = 0 *Causes Failed() to return true on the client side. more...*
virtual bool	**IsCanceled**() const = 0 *If true, indicates that the client canceled the RPC, so the server may as well give up on replying to it. more...*
virtual void	**NotifyOnCancel**(Closure * callback) = 0 *Asks that the given callback be called when the RPC is canceled. more...*

virtual void RpcController::Reset() = 0

Resets the RpcController to its initial state so that it may be reused in a new call.

Must not be called while an RPC is in progress.

virtual bool RpcController::Failed() const = 0

After a call has finished, returns true if the call failed.

The possible reasons for failure depend on the RPC implementation. Failed() must not be called before a call has finished. If Failed() returns true, the contents of the response message are undefined.

virtual void RpcController::StartCancel() = 0

Advises the RPC system that the caller desires that the RPC call be canceled.

The RPC system may cancel it immediately, may wait awhile and then cancel it, or may not even cancel the call at all. If the call is canceled, the "done" callback will still be called and the RpcController will indicate that the call failed at that time.

`virtual void RpcController::SetFailed(const string & reason) = 0`

Causes Failed() to return true on the client side.

"reason" will be incorporated into the message returned by ErrorText(). If you find you need to return machine-readable information about failures, you should incorporate it into your response protocol buffer and should NOT call SetFailed().

`virtual bool RpcController::IsCanceled() const = 0`

If true, indicates that the client canceled the RPC, so the server may as well give up on replying to it.

The server should still call the final "done" callback.

`virtual void RpcController::NotifyOnCancel(Closure * callback) = 0`

Asks that the given callback be called when the RPC is canceled.

The callback will always be called exactly once. If the RPC completes without being canceled, the callback will be called after completion. If the RPC has already been canceled when NotifyOnCancel() is called, the callback will be called immediately.

NotifyOnCancel() must be called no more than once per request.

class RpcChannel

`#include <google/protobuf/service.h>`
`namespace google::protobuf`

Abstract interface for an RPC channel.

An RpcChannel represents a communication line to a Service which can be used to call that Service's methods. The Service may be running on another machine. Normally, you should not call an RpcChannel directly, but instead construct a stub Service wrapping it. Example:

```
RpcChannel* channel = new MyRpcChannel("remotehost.example.com:1234");
MyService* service = new MyService::Stub(channel);
service->MyMethod(request, &response, callback);
```

Members	
	RpcChannel()
virtual	**~RpcChannel**()
virtual void	**CallMethod**(const MethodDescriptor * method, RpcController * controller, const Message * request, Message * response, Closure * done) = 0 *Call the given method of the remote service. more...*

```
virtual void RpcChannel::CallMethod(
        const MethodDescriptor * method,
        RpcController * controller,
        const Message * request,
        Message * response,
        Closure * done) = 0
```

Call the given method of the remote service.

The signature of this procedure looks the same as Service::CallMethod() but the requirements are less strict in one important way: the request and response objects need not be of any specific class as long as their descriptors are method->input_type() and method->output_type().

text_format.h

```
#include <google/protobuf/text_format.h>
namespace google::protobuf
```

Utilities for printing and parsing protocol messages in a human-readable, text-based format.

Classes in this file

`TextFormat` *This class implements protocol buffer text format.*
`TextFormat::Parser` *For more control over parsing, use this class.*

class TextFormat

```
#include <google/protobuf/text_format.h>
namespace google::protobuf
```

This class implements protocol buffer text format.

Printing and parsing protocol messages in text format is useful for debugging and human editing of messages.

This class is really a namespace that contains only static methods.

Members

`static bool`	**`Print`**`(const Message & message, io::ZeroCopyOutputStream * output)` *Outputs a textual representation of the given message to the given output stream.*
`static bool`	**`PrintUnknownFields`**`(const UnknownFieldSet & unknown_fields, io::ZeroCopyOutputStream * output)` *Print the fields in an UnknownFieldSet. more...*
`static bool`	**`PrintToString`**`(const Message & message, string * output)` *Like Print(), but outputs directly to a string.*
`static bool`	**`PrintUnknownFieldsToString`**`(const UnknownFieldSet & unknown_fields, string * output)` *Like PrintUnknownFields(), but outputs directly to a string.*
`static void`	**`PrintFieldValueToString`**`(const Message & message, const FieldDescriptor * field, int index, string * output)` *Outputs a textual representation of the value of the field supplied on the message supplied. more...*
`static bool`	**`Parse`**`(io::ZeroCopyInputStream * input, Message * output)` *Parses a text-format protocol message from the given input stream to the given message object. more...*
`static bool`	**`ParseFromString`**`(const string & input, Message * output)` *Like Parse(), but reads directly from a string.*
`static`	**`Merge`**`(io::ZeroCopyInputStream * input, Message * output)`

bool	*Like Parse(), but the data is merged into the given message, as if using Message::MergeFrom().*
static bool	**MergeFromString**(const string & input, Message * output) *Like Merge(), but reads directly from a string.*

```
static bool TextFormat::PrintUnknownFields(
        const UnknownFieldSet & unknown_fields,
        io::ZeroCopyOutputStream * output)
```

Print the fields in an UnknownFieldSet

They are printed by tag number only. Embedded messages are heuristically identified by attempting to parse them.

```
static void TextFormat::PrintFieldValueToString(
        const Message & message,
        const FieldDescriptor * field,
        int index,
        string * output)
```

Outputs a textual representation of the value of the field supplied on the message supplied.

For non-repeated fields, an index of -1 must be supplied. Note that this method will print the default value for a field if it is not set.

```
static bool TextFormat::Parse(
        io::ZeroCopyInputStream * input,
        Message * output)
```

Parses a text-format protocol message from the given input stream to the given message object.

This function parses the format written by Print().

class TextFormat::Parser

```
#include <google/protobuf/text_format.h>
namespace google::protobuf
```

For more control over parsing, use this class.

Members	
	Parser()
	~Parser()
bool	**Parse**(io::ZeroCopyInputStream * input, Message * output) *Like TextFormat::Parse().*
bool	**ParseFromString**(const string & input, Message * output) *Like TextFormat::ParseFromString().*
bool	**Merge**(io::ZeroCopyInputStream * input, Message * output) *Like TextFormat::Merge().*
bool	**MergeFromString**(const string & input, Message * output)

	Like TextFormat::MergeFromString().
void	**RecordErrorsTo**(io::ErrorCollector * error_collector) *Set where to report parse errors. more...*
void	**AllowPartialMessage**(bool allow) *Normally parsing fails if, after parsing, output->IsInitialized() returns false. more...*

---

void Parser::RecordErrorsTo(io::ErrorCollector * error_collector)

Set where to report parse errors.

If NULL (the default), errors will be printed to stderr.

---

void Parser::AllowPartialMessage(bool allow)

Normally parsing fails if, after parsing, output->IsInitialized() returns false.

Call AllowPartialMessage(true) to skip this check.

unknown_field_set.h

```
#include <google/protobuf/unknown_field_set.h>
namespace google::protobuf
```

Contains classes used to keep track of unrecognized fields seen while parsing a protocol message.

Classes in this file

UnknownFieldSet *An UnknownFieldSet contains fields that were encountered while parsing a message but were not defined by its type.*
UnknownField *Represents one field in an UnknownFieldSet.*

class UnknownFieldSet

```
#include <google/protobuf/unknown_field_set.h>
namespace google::protobuf
```

An UnknownFieldSet contains fields that were encountered while parsing a message but were not defined by its type.

Keeping track of these can be useful, especially in that they may be written if the message is serialized again without being cleared in between. This means that software which simply receives messages and forwards them to other servers does not need to be updated every time a new field is added to the message definition.

To get the UnknownFieldSet attached to any message, call Reflection::GetUnknownFields().

This class is necessarily tied to the protocol buffer wire format, unlike the Reflection interface which is independent of any serialization scheme.

Members

	UnknownFieldSet()
	~UnknownFieldSet()
void	**Clear**() *Remove all fields.*
bool	**empty**() const *Is this set empty?*
void	**MergeFrom**(const UnknownFieldSet & other) *Merge the contents of some other UnknownFieldSet with this one.*
int	**field_count**() const *Returns the number of fields present in the UnknownFieldSet.*
const UnknownField &	**field**(int index) const *Get a field in the set, where 0 <= index < field_count(). more...*
UnknownField *	**mutable_field**(int index) *Get a mutable pointer to a field in the set, where 0 <= index < field_count(). more...*

const UnknownField *	**FindFieldByNumber**(int number) const *Find a field by field number. Returns NULL if not found.*
UnknownField *	**AddField**(int number) *Add a field by field number. more...*

Parsing helpers

These work exactly like the similarly-named methods of Message.

bool	**MergeFromCodedStream**(io::CodedInputStream * input)
bool	**ParseFromCodedStream**(io::CodedInputStream * input)
bool	**ParseFromZeroCopyStream**(io::ZeroCopyInputStream * input)
bool	**ParseFromArray**(const void * data, int size)
bool	**ParseFromString**(const string & data)

---

const UnknownField & UnknownFieldSet::field(int index) const

Get a field in the set, where 0 <= index < field_count().

The fields appear in arbitrary order.

UnknownField * UnknownFieldSet::mutable_field(int index)

Get a mutable pointer to a field in the set, where 0 <= index < field_count().

The fields appear in arbitrary order.

UnknownField * UnknownFieldSet::AddField(int number)

Add a field by field number.

If the field number already exists, returns the existing UnknownField.

class UnknownField

```
#include <google/protobuf/unknown_field_set.h>
namespace google::protobuf
```

Represents one field in an UnknownFieldSet.

UnknownFiled's accessors are similar to those that would be produced by the protocol compiler for the fields:

```
repeated uint64 varint;
repeated fixed32 fixed32;
repeated fixed64 fixed64;
repeated bytes length_delimited;
```

```
repeated UnknownFieldSet group;
```

(OK, so the last one isn't actually a valid field type but you get the idea.)

Members

	~UnknownField()
void	**Clear**() *Clears all fields.*
void	**MergeFrom**(const UnknownField & other) *Merge the contents of some other UnknownField with this one. more...*
int	**number**() const *The field's tag number, as seen on the wire.*
int	**index**() const *The index of this UnknownField within the UknownFieldSet (e.g. more...*
int	**varint_size**() const
int	**fixed32_size**() const
int	**fixed64_size**() const
int	**length_delimited_size**() const
int	**group_size**() const
uint64	**varint**(int index) const
uint32	**fixed32**(int index) const
uint64	**fixed64**(int index) const
const string &	**length_delimited**(int index) const
const UnknownFieldSet &	**group**(int index) const
void	**set_varint**(int index, uint64 value)
void	**set_fixed32**(int index, uint32 value)
void	**set_fixed64**(int index, uint64 value)
void	**set_length_delimited**(int index, const string & value)
string *	**mutable_length_delimited**(int index)
UnknownFieldSet *	**mutable_group**(int index)
void	**add_varint**(uint64 value)
void	**add_fixed32**(uint32 value)
void	**add_fixed64**(uint64 value)
void	**add_length_delimited**(const string & value)
string *	**add_length_delimited**()
UnknownFieldSet *	**add_group**()
void	**clear_varint**()

void	**clear_fixed32**()
void	**clear_fixed64**()
void	**clear_length_delimited**()
void	**clear_group**()
const RepeatedField< uint64 > &	**varint**() const
const RepeatedField< uint32 > &	**fixed32**() const
const RepeatedField< uint64 > &	**fixed64**() const
const RepeatedPtrField< string > &	**length_delimited**() const
const RepeatedPtrField< UnknownFieldSet > &	**group**() const
RepeatedField< uint64 > *	**mutable_varint**()
RepeatedField< uint32 > *	**mutable_fixed32**()
RepeatedField< uint64 > *	**mutable_fixed64**()
RepeatedPtrField< string > *	**mutable_length_delimited**()
RepeatedPtrField< UnknownFieldSet > *	**mutable_group**()

```
void UnknownField::MergeFrom(
        const UnknownField & other)
```

Merge the contents of some other UnknownField with this one.

For each wire type, the values are simply concatenated.

```
int UnknownField::index() const
```

The index of this UnknownField within the UknownFieldSet (e.g.

set.field(field.index()) == field).

common.h

```
#include <google/protobuf/stubs/common.h>
namespace google::protobuf
```

Contains basic types and utilities used by the rest of the library.

Classes in this file

`LogSilencer` *Create a LogSilencer if you want to temporarily suppress all log messages.*
`Closure` *Abstract interface for a callback.*

File Members

These definitions are not part of any class.

`typedef`	`unsigned int` **`uint`**
`typedef`	`int8_t` **`int8`**
`typedef`	`int16_t` **`int16`**
`typedef`	`int32_t` **`int32`**
`typedef`	`int64_t` **`int64`**
`typedef`	`uint8_t` **`uint8`**
`typedef`	`uint16_t` **`uint16`**
`typedef`	`uint32_t` **`uint32`**
`typedef`	`uint64_t` **`uint64`**
`typedef`	`void` **`LogHandler`**`(LogLevel level, const char *filename, int line, const string &message)`
`enum`	**`LogLevel`** *more...*
`LogHandler *`	**`SetLogHandler`**`(LogHandler * new_func)` *The protobuf library sometimes writes warning and error messages to stderr. more...*
`Closure *`	**`NewCallback`**`(void(*)() function)` *See Closure.*
`Closure *`	**`NewPermanentCallback`**`(void(*)() function)` *See Closure.*
`template Closure *`	**`NewCallback`**`(Class * object, void(Class::*)() method)` *See Closure.*
`template Closure *`	**`NewPermanentCallback`**`(Class * object, void(Class::*)() method)` *See Closure.*

template Closure *	**NewCallback**(void(*)(Arg1) function, Arg1 arg1) *See Closure.*
template Closure *	**NewPermanentCallback**(void(*)(Arg1) function, Arg1 arg1) *See Closure.*
template Closure *	**NewCallback**(Class * object, void(Class::*)(Arg1) method, Arg1 arg1) *See Closure.*
template Closure *	**NewPermanentCallback**(Class * object, void(Class::*)(Arg1) method, Arg1 arg1) *See Closure.*
template Closure *	**NewCallback**(void(*)(Arg1, Arg2) function, Arg1 arg1, Arg2 arg2) *See Closure.*
template Closure *	**NewPermanentCallback**(void(*)(Arg1, Arg2) function, Arg1 arg1, Arg2 arg2) *See Closure.*
template Closure *	**NewCallback**(Class * object, void(Class::*)(Arg1, Arg2) method, Arg1 arg1, Arg2 arg2) *See Closure.*
template Closure *	**NewPermanentCallback**(Class * object, void(Class::*)(Arg1, Arg2) method, Arg1 arg1, Arg2 arg2) *See Closure.*
void	**DoNothing**() *A function which does nothing. more...*
const int32	**kint32max** = 0x7FFFFFFF
const int32	**kint32min** = -kint32max - 1
const int64	**kint64max** = 0x7FFFFFFFFFFFFFFFLL
const int64	**kint64min** = -kint64max - 1
const uint32	**kuint32max** = 0xFFFFFFFFu
const uint64	**kuint64max** = 0xFFFFFFFFFFFFFFFFULL

```
enum protobuf::LogLevel {
  LOGLEVEL_INFO,
  LOGLEVEL_WARNING,
  LOGLEVEL_ERROR,
  LOGLEVEL_FATAL,
  LOGLEVEL_DFATAL = LOGLEVEL_FATAL
}
```

LOGLEVEL_INFO	Informational. This is never actually used by libprotobuf.
LOGLEVEL_WARNING	Warns about issues that, although not technically a problem now, could cause problems in the future. For example, a // warning will be printed when parsing a message that is near the message size limit.

LOGLEVEL_ERROR	An error occurred which should never happen during normal use.
LOGLEVEL_FATAL	An error occurred from which the library cannot recover. This usually indicates a programming error in the code which calls the library, especially when compiled in debug mode.
LOGLEVEL_DFATAL	

LogHandler * protobuf::SetLogHandler(
LogHandler * new_func)

The protobuf library sometimes writes warning and error messages to stderr.

These messages are primarily useful for developers, but may also help end users figure out a problem. If you would prefer that these messages be sent somewhere other than stderr, call SetLogHandler() to set your own handler. This returns the old handler. Set the handler to NULL to ignore log messages (but see also LogSilencer, below).

Obviously, SetLogHandler is not thread-safe. You should only call it at initialization time, and probably not from library code. If you simply want to suppress log messages temporarily (e.g. because you have some code that tends to trigger them frequently and you know the warnings are not important to you), use the LogSilencer class below.

---

void protobuf::DoNothing()

A function which does nothing.

Useful for creating no-op callbacks, e.g.:

```
Closure* nothing = NewCallback(&DoNothing);
```

class LogSilencer

```
#include <google/protobuf/stubs/common.h>
namespace google::protobuf
```

Create a LogSilencer if you want to temporarily suppress all log messages.

As long as any LogSilencer objects exist, non-fatal log messages will be discarded (the current LogHandler will *not* be called). Constructing a LogSilencer is thread-safe. You may accidentally suppress log messages occurring in another thread, but since messages are generally for debugging purposes only, this isn't a big deal. If you want to intercept log messages, use SetLogHandler().

Members
LogSilencer()
~LogSilencer()

class Closure

```
#include <google/protobuf/stubs/common.h>
namespace google::protobuf
```

Abstract interface for a callback.

When calling an RPC, you must provide a Closure to call when the procedure completes. See the Service interface in service.h.

To automatically construct a Closure which calls a particular function or method with a particular set of parameters, use the NewCallback() function. Example:

```
void FooDone(const FooResponse* response) {
  ...
}

void CallFoo() {
  ...
  // When done, call FooDone() and pass it a pointer to the response.
  Closure* callback = NewCallback(&FooDone, response);
  // Make the call.
  service->Foo(controller, request, response, callback);
}
```

Example that calls a method:

```
class Handler {
 public:
  ...

  void FooDone(const FooResponse* response) {
    ...
  }

  void CallFoo() {
    ...
    // When done, call FooDone() and pass it a pointer to the response.
    Closure* callback = NewCallback(this, &Handler::FooDone, response);
    // Make the call.
    service->Foo(controller, request, response, callback);
  }
};
```

Currently NewCallback() supports binding zero, one, or two arguments.

Callbacks created with NewCallback() automatically delete themselves when executed. They should be used when a callback is to be called exactly once (usually the case with RPC callbacks). If a callback may be called a different number of times (including zero), create it with NewPermanentCallback() instead. You are then responsible for deleting the callback (using the "delete" keyword as normal).

Note that NewCallback() is a bit touchy regarding argument types. Generally, the values you provide for the parameter bindings must exactly match the types accepted by the callback function. For example:

```
void Foo(string s);
NewCallback(&Foo, "foo");          // WON'T WORK:  const char* != string
NewCallback(&Foo, string("foo"));  // WORKS
```

Also note that the arguments cannot be references:

```
void Foo(const string& s);
string my_str;
NewCallback(&Foo, my_str);  // WON'T WORK:  Can't use referecnes.
```

However, correctly-typed pointers will work just fine.

Members	
	Closure()
virtual	**~Closure**()

```
virtual void Run() = 0
```

coded_stream.h

```
#include <google/protobuf/io/coded_stream.h>
namespace google::protobuf::io
```

This file contains the CodedInputStream and CodedOutputStream classes, which wrap a ZeroCopyInputStream or ZeroCopyOutputStream, respectively, and allow you to read or write individual pieces of data in various formats.

In particular, these implement the varint encoding for integers, a simple variable-length encoding in which smaller numbers take fewer bytes.

Typically these classes will only be used internally by the protocol buffer library in order to encode and decode protocol buffers. Clients of the library only need to know about this class if they wish to write custom message parsing or serialization procedures.

CodedOutputStream example:

```
// Write some data to "myfile".  First we write a 4-byte "magic number"
// to identify the file type, then write a length-delimited string.  The
// string is composed of a varint giving the length followed by the raw
// bytes.
int fd = open("myfile", O_WRONLY);
ZeroCopyOutputStream* raw_output = new FileOutputStream(fd);
CodedOutputStream* coded_output = new CodedOutputStream(raw_output);

int magic_number = 1234;
char text[] = "Hello world!";
coded_output->WriteLittleEndian32(magic_number);
coded_output->WriteVarint32(strlen(text));
coded_output->WriteRaw(text, strlen(text));

delete coded_output;
delete raw_output;
close(fd);
```

CodedInputStream example:

```
// Read a file created by the above code.
int fd = open("myfile", O_RDONLY);
ZeroCopyInputStream* raw_input = new FileInputStream(fd);
CodedInputStream coded_input = new CodedInputStream(raw_input);

coded_input->ReadLittleEndian32(&magic_number);
if (magic_number != 1234) {
  cerr << "File not in expected format." << endl;
  return;
}

uint32 size;
coded_input->ReadVarint32(&size);

char* text = new char[size + 1];
coded_input->ReadRaw(buffer, size);
text[size] = '\0';

delete coded_input;
delete raw_input;
close(fd);
```

```
cout << "Text is: " << text << endl;
delete [] text;
```

For those who are interested, varint encoding is defined as follows:

The encoding operates on unsigned integers of up to 64 bits in length. Each byte of the encoded value has the format:

- bits 0-6: Seven bits of the number being encoded.
- bit 7: Zero if this is the last byte in the encoding (in which case all remaining bits of the number are zero) or 1 if more bytes follow. The first byte contains the least-significant 7 bits of the number, the second byte (if present) contains the next-least-significant 7 bits, and so on. So, the binary number 1011000101011 would be encoded in two bytes as "10101011 00101100".

In theory, varint could be used to encode integers of any length. However, for practicality we set a limit at 64 bits. The maximum encoded length of a number is thus 10 bytes.

Classes in this file
`CodedInputStream` *Class which reads and decodes binary data which is composed of varint- encoded integers and fixed-width pieces.*
`CodedOutputStream` *Class which encodes and writes binary data which is composed of varint- encoded integers and fixed-width pieces.*

class CodedInputStream

```
#include <google/protobuf/io/coded_stream.h>
namespace google::protobuf::io
```

Class which reads and decodes binary data which is composed of varint- encoded integers and fixed-width pieces.

Wraps a ZeroCopyInputStream Most users will not need to deal with CodedInputStream

Most methods of CodedInputStream that return a bool return false if an underlying I/O error occurs or if the data is malformed. Once such a failure occurs, the CodedInputStream is broken and is no longer useful.

Members	
`explicit`	**`CodedInputStream`**`(ZeroCopyInputStream * input)` *Create a CodedInputStream that reads from the given ZeroCopyInputStream.*
	`~CodedInputStream``()` *Destroy the CodedInputStream and position the underlying ZeroCopyInputStream at the first unread byte. more...*
`bool`	**`Skip`**`(int count)` *Skips a number of bytes. more...*
`bool`	**`ReadRaw`**`(void * buffer, int size)` *Read raw bytes, copying them into the given buffer.*
`bool`	**`ReadString`**`(string * buffer, int size)` *Like ReadRaw, but reads into a string. more...*
`bool`	**`ReadLittleEndian32`**`(uint32 * value)` *Read a 32-bit little-endian integer.*
`bool`	**`ReadLittleEndian64`**`(uint64 * value)` *Read a 64-bit little-endian integer.*
`bool`	**`ReadVarint32`**`(uint32 * value)` *Read an unsigned integer with Varint encoding, truncating to 32 bits. more...*

bool	**ReadVarint64**(uint64 * value) *Read an unsigned integer with Varint encoding.*
uint32	**ReadTag**() *Read a tag. more...*
bool	**ExpectTag**(uint32 expected) *Usually returns true if calling ReadVarint32() now would produce the given value. more...*
bool	**ExpectAtEnd**() *Usually returns true if no more bytes can be read. more...*
bool	**LastTagWas**(uint32 expected) *If the last call to ReadTag() returned the given value, returns true. more...*
bool	**ConsumedEntireMessage**() *When parsing message (but NOT a group), this method must be called immediately after MergeFromCodedStream() returns (if it returns true) to further verify that the message ended in a legitimate way. more...*

Limits

Limits are used when parsing length-delimited embedded messages. After the message's length is read, PushLimit() is used to prevent the CodedInputStream from reading beyond that length. Once the embedded message has been parsed, PopLimit() is called to undo the limit.

typedef	int **Limit** *Opaque type used with PushLimit() and PopLimit(). more...*
Limit	**PushLimit**(int byte_limit) *Places a limit on the number of bytes that the stream may read, starting from the current position. more...*
void	**PopLimit**(Limit limit) *Pops the last limit pushed by PushLimit(). more...*
int	**BytesUntilLimit**() *Returns the number of bytes left until the nearest limit on the stack is hit, or -1 if no limits are in place.*

Total Bytes Limit

To prevent malicious users from sending excessively large messages and causing integer overflows or memory exhaustion, CodedInputStream imposes a hard limit on the total number of bytes it will read.

void	**SetTotalBytesLimit**(int total_bytes_limit, int warning_threshold) *Sets the maximum number of bytes that this CodedInputStream will read before refusing to continue. more...*

Recursion Limit

To prevent corrupt or malicious messages from causing stack overflows, we must keep track of the depth of recursion when parsing embedded messages and groups. CodedInputStream keeps track of this because it is the only object that is passed down the stack during parsing.

void	**SetRecursionLimit**(int limit) *Sets the maximum recursion depth. The default is 64.*
bool	**IncrementRecursionDepth**() *Increments the current recursion depth. more...*
void	**DecrementRecursionDepth**() *Decrements the recursion depth.*

CodedInputStream::~CodedInputStream()

Destroy the CodedInputStream and position the underlying ZeroCopyInputStream at the first unread byte.

If an error occurred while reading (causing a method to return false), then the exact position of the input stream may be anywhere between the last value that was read successfully and the stream's byte limit.

---

bool CodedInputStream::Skip(

```
        int count)
```

Skips a number of bytes.

Returns false if an underlying read error occurs.

---

bool CodedInputStream::ReadString(

```
        string * buffer,
        int size)
```

Like ReadRaw, but reads into a string.

Implementation Note: ReadString() grows the string gradually as it reads in the data, rather than allocating the entire requested size upfront. This prevents denial-of-service attacks in which a client could claim that a string is going to be MAX_INT bytes long in order to crash the server because it can't allocate this much space at once.

---

bool CodedInputStream::ReadVarint32(

```
        uint32 * value)
```

Read an unsigned integer with Varint encoding, truncating to 32 bits.

Reading a 32-bit value is equivalent to reading a 64-bit one and casting it to uint32, but may be more efficient.

---

uint32 CodedInputStream::ReadTag()

Read a tag.

This calls ReadVarint32() and returns the result, or returns zero (which is not a valid tag) if ReadVarint32() fails. Also, it updates the last tag value, which can be checked with LastTagWas(). Always inline because this is only called in once place per parse loop but it is called for every iteration of said loop, so it should be fast. GCC doesn't want to inline this by default.

---

bool CodedInputStream::ExpectTag(

```
        uint32 expected)
```

Usually returns true if calling ReadVarint32() now would produce the given value.

Will always return false if ReadVarint32() would not return the given value. If ExpectTag() returns true, it also advances past the varint. For best performance, use a compile-time constant as the parameter. Always inline because this collapses to a small number of instructions when given a constant parameter, but GCC doesn't want to inline by default.

---

bool CodedInputStream::ExpectAtEnd()

Usually returns true if no more bytes can be read.

Always returns false if more bytes can be read. If ExpectAtEnd() returns true, a subsequent call to LastTagWas() will act as if ReadTag() had been called and returned zero, and ConsumedEntireMessage() will return true.

---

bool CodedInputStream::LastTagWas(
uint32 expected)

If the last call to ReadTag() returned the given value, returns true.

Otherwise, returns false;

This is needed because parsers for some types of embedded messages (with field type TYPE_GROUP) don't actually know that they've reached the end of a message until they see an ENDGROUP tag, which was actually part of the enclosing message. The enclosing message would like to check that tag to make sure it had the right number, so it calls LastTagWas() on return from the embedded parser to check.

---

bool CodedInputStream::ConsumedEntireMessage()

When parsing message (but NOT a group), this method must be called immediately after MergeFromCodedStream() returns (if it returns true) to further verify that the message ended in a legitimate way.

For example, this verifies that parsing did not end on an end-group tag. It also checks for some cases where, due to optimizations, MergeFromCodedStream() can incorrectly return true.

---

typedef CodedInputStream::Limit

Opaque type used with PushLimit() and PopLimit().

Do not modify values of this type yourself. The only reason that this isn't a struct with private internals is for efficiency.

Limit CodedInputStream::PushLimit(
int byte_limit)

Places a limit on the number of bytes that the stream may read, starting from the current position.

Once the stream hits this limit, it will act like the end of the input has been reached until PopLimit() is called.

As the names imply, the stream conceptually has a stack of limits. The shortest limit on the stack is always enforced, even if it is not the top limit.

The value returned by PushLimit() is opaque to the caller, and must be passed unchanged to the corresponding call to PopLimit().

---

void CodedInputStream::PopLimit(
Limit limit)

Pops the last limit pushed by PushLimit().

The input must be the value returned by that call to PushLimit().

```
void CodedInputStream::SetTotalBytesLimit(
        int total_bytes_limit,
        int warning_threshold)
```

Sets the maximum number of bytes that this CodedInputStream will read before refusing to continue.

To prevent integer overflows in the protocol buffers implementation, as well as to prevent servers from allocating enormous amounts of memory to hold parsed messages, the maximum message length should be limited to the shortest length that will not harm usability. The theoretical shortest message that could cause integer overflows is 512MB. The default limit is 64MB. Apps should set shorter limits if possible. If warning_threshold is not -1, a warning will be printed to stderr after warning_threshold bytes are read. An error will always be printed to stderr if the limit is reached.

This is unrelated to PushLimit()/PopLimit().

Hint: If you are reading this because your program is printing a warning about dangerously large protocol messages, you may be confused about what to do next. The best option is to change your design such that excessively large messages are not necessary. For example, try to design file formats to consist of many small messages rather than a single large one. If this is infeasible, you will need to increase the limit. Chances are, though, that your code never constructs a CodedInputStream on which the limit can be set. You probably parse messages by calling things like Message::ParseFromString(). In this case, you will need to change your code to instead construct some sort of ZeroCopyInputStream (e.g. an ArrayInputStream), construct a CodedInputStream around that, then call Message::ParseFromCodedStream() instead. Then you can adjust the limit. Yes, it's more work, but you're doing something unusual.

```
bool CodedInputStream::IncrementRecursionDepth()
```

Increments the current recursion depth.

Returns true if the depth is under the limit, false if it has gone over.

class CodedOutputStream

```
#include <google/protobuf/io/coded_stream.h>
namespace google::protobuf::io
```

Class which encodes and writes binary data which is composed of varint- encoded integers and fixed-width pieces.

Wraps a ZeroCopyOutputStream. Most users will not need to deal with CodedOutputStream.

Most methods of CodedOutputStream which return a bool return false if an underlying I/O error occurs. Once such a failure occurs, the CodedOutputStream is broken and is no longer useful.

Members

explicit	**CodedOutputStream**(ZeroCopyOutputStream * output) *Create an CodedOutputStream that writes to the given ZeroCopyOutputStream.*
	~CodedOutputStream() *Destroy the CodedOutputStream and position the underlying ZeroCopyOutputStream immediately after the last byte written.*
bool	**WriteRaw**(const void * buffer, int size) *Write raw bytes, copying them from the given buffer.*
bool	**WriteString**(const string & str) *Equivalent to WriteRaw(str.data(), str.size()).*
bool	**WriteLittleEndian32**(uint32 value) *Write a 32-bit little-endian integer.*
bool	**WriteLittleEndian64**(uint64 value)

	Write a 64-bit little-endian integer.
bool	**WriteVarint32**(uint32 value) *Write an unsigned integer with Varint encoding. more...*
bool	**WriteVarint64**(uint64 value) *Write an unsigned integer with Varint encoding.*
bool	**WriteVarint32SignExtended**(int32 value) *Equivalent to WriteVarint32() except when the value is negative, in which case it must be sign-extended to a full 10 bytes.*
bool	**WriteTag**(uint32 value) *This is identical to WriteVarint32(), but optimized for writing tags. more...*
int	**ByteCount**() const *Returns the total number of bytes written since this object was created.*
static int	**VarintSize32**(uint32 value) *Returns the number of bytes needed to encode the given value as a varint.*
static int	**VarintSize64**(uint64 value) *Returns the number of bytes needed to encode the given value as a varint.*
static int	**VarintSize32SignExtended**(int32 value) *If negative, 10 bytes. Otheriwse, same as VarintSize32(). more...*

---

bool CodedOutputStream::WriteVarint32(uint32 value)

Write an unsigned integer with Varint encoding.

Writing a 32-bit value is equivalent to casting it to uint64 and writing it as a 64-bit value, but may be more efficient.

---

bool CodedOutputStream::WriteTag(uint32 value)

This is identical to WriteVarint32(), but optimized for writing tags.

In particular, if the input is a compile-time constant, this method compiles down to a couple instructions. Always inline because otherwise the aformentioned optimization can't work, but GCC by default doesn't want to inline this.

---

static int CodedOutputStream::VarintSize32SignExtended(int32 value)

If negative, 10 bytes. Otheriwse, same as VarintSize32().

< TODO(kenton): Make this a symbolic constant.

printer.h

```
#include <google/protobuf/io/printer.h>
namespace google::protobuf::io
```

Utility class for writing text to aZeroCopyOutputStream

Classes in this file

`Printer` *This simple utility class assists in code generation.*

class Printer

```
#include <google/protobuf/io/printer.h>
namespace google::protobuf::io
```

This simple utility class assists in code generation.

It basically allows the caller to define a set of variables and then output some text with variable substitutions. Example usage:

```
Printer printer(output, '$');
map<string, string> vars;
vars["name"] = "Bob";
printer.Print(vars, "My name is $name$.");
```

The above writes "My name is Bob." to the output stream.

Printer aggressively enforces correct usage, crashing (with assert failures) in the case of undefined variables. This helps greatly in debugging code which uses it. This class is not intended to be used by production servers.

Members

	`Printer(ZeroCopyOutputStream * output, char variable_delimiter)` *Create a printer that writes text to the given output stream. more...*
	`~Printer()`
`void`	`Print(const map< string, string > & variables, const char * text)` *Print some text after applying variable substitutions. more...*
`void`	`Print(const char * text)` *Like the first Print(), except the substitutions are given as parameters.*
`void`	`Print(const char * text, const char * variable, const string & value)` *Like the first Print(), except the substitutions are given as parameters.*
`void`	`Print(const char * text, const char * variable1, const string & value1, const char * variable2, const string & value2)` *Like the first Print(), except the substitutions are given as parameters.*
`void`	`Indent()` *Indent text by two spaces. more...*

void	Outdent()	
	Reduces the current indent level by two spaces, or crashes if the indent level is zero.	
bool	failed() const	
	True if any write to the underlying stream failed. more...	

```
Printer::Printer(
        ZeroCopyOutputStream * output,
        char variable_delimiter)
```

Create a printer that writes text to the given output stream.

Use the given character as the delimiter for variables.

```
void Printer::Print(
        const map< string, string > & variables,
        const char * text)
```

Print some text after applying variable substitutions.

If a particular variable in the text is not defined, this will crash. Variables to be substituted are identified by their names surrounded by delimiter characters (as given to the constructor). The variable bindings are defined by the given map.

```
void Printer::Indent()
```

Indent text by two spaces.

After calling Indent(), two spaces will be inserted at the beginning of each line of text. Indent() may be called multiple times to produce deeper indents.

```
bool Printer::failed() const
```

True if any write to the underlying stream failed.

(We don't just crash in this case because this is an I/O failure, not a programming error.)

tokenizer.h

```
#include <google/protobuf/io/tokenizer.h>
namespace google::protobuf::io
```

Class for parsing tokenized text from a ZeroCopyInputStream

Classes in this file

ErrorCollector *Abstract interface for an object which collects the errors that occur during parsing.*
Tokenizer *This class converts a stream of raw text into a stream of tokens for the protocol definition parser to parse.*
Tokenizer::Token *Structure representing a token read from the token stream.*

class ErrorCollector

```
#include <google/protobuf/io/tokenizer.h>
namespace google::protobuf::io
```

Abstract interface for an object which collects the errors that occur during parsing.

A typical implementation might simply print the errors to stdout.

Members

	ErrorCollector()
virtual	**~ErrorCollector**()
virtual void	**AddError**(int line, int column, const string & message) = 0 *Indicates that there was an error in the input at the given line and column numbers.* more...

```
virtual void ErrorCollector::AddError(
        int line,
        int column,
        const string & message) = 0
```

Indicates that there was an error in the input at the given line and column numbers.

The numbers are zero-based, so you may want to add 1 to each before printing them.

class Tokenizer

```
#include <google/protobuf/io/tokenizer.h>
namespace google::protobuf::io
```

This class converts a stream of raw text into a stream of tokens for the protocol definition parser to parse.

The tokens recognized are similar to those that make up the C language; see the TokenType enum for precise descriptions. Whitespace and comments are skipped. By default, C- and C++-style comments are recognized, but other styles can be used by callingset_comment_style().

Members	
enum	**TokenType** *more...*
	Tokenizer(ZeroCopyInputStream * input, ErrorCollector * error_collector) *Construct a Tokenizer that reads and tokenizes text from the given input stream and writes errors to the given error_collector. more...*
	~Tokenizer()
const Token &	**current**() *Get the current token. more...*
bool	**Next**() *Advance to the next token. more...*
Options	
enum	**CommentStyle** *Valid values for set_comment_style(). more...*
void	**set_allow_f_after_float**(bool value) *Set true to allow floats to be suffixed with the letter 'f'. more...*
void	**set_comment_style**(CommentStyle style) *Sets the comment style.*
Parse helpers	
static double	**ParseFloat**(const string & text) *Parses a TYPE_FLOAT token. more...*
static void	**ParseString**(const string & text, string * output) *Parses a TYPE_STRING token. more...*
static bool	**ParseInteger**(const string & text, uint64 max_value, uint64 * output) *Parses a TYPE_INTEGER token. more...*

```
enum Tokenizer::TokenType {
  TYPE_START,
  TYPE_END,
  TYPE_IDENTIFIER,
  TYPE_INTEGER,
  TYPE_FLOAT,
  TYPE_STRING,
  TYPE_SYMBOL
}
```

TYPE_START	Next() has not yet been called.
TYPE_END	End of input reached. "text" is empty.
TYPE_IDENTIFIER	A sequence of letters, digits, and underscores, not starting with a digit.

	It is an error for a number to be followed by an identifier with no space in between.
TYPE_INTEGER	A sequence of digits representing an integer. Normally the digits are decimal, but a prefix of "0x" indicates a hex number and a leading zero indicates octal, just like with C numeric literals. A leading negative sign is NOT included in the token; it's up to the parser to interpret the unary minus operator on its own.
TYPE_FLOAT	A floating point literal, with a fractional part and/or an exponent. Always in decimal. Again, never negative.
TYPE_STRING	A quoted sequence of escaped characters. Either single or double quotes can be used, but they must match. A string literal cannot cross a line break.
TYPE_SYMBOL	Any other printable character, like '!' or '+'. Symbols are always a single character, so "!+$%" is four tokens.

```
Tokenizer::Tokenizer(
        ZeroCopyInputStream * input,
        ErrorCollector * error_collector)
```

Construct a Tokenizer that reads and tokenizes text from the given input stream and writes errors to the given error_collector.

The caller keeps ownership of input and error_collector.

```
const Token & Tokenizer::current()
```

Get the current token.

This is updated when Next() is called. Before the first call to Next(), current() has type TYPE_START and no contents.

```
bool Tokenizer::Next()
```

Advance to the next token.

Returns false if the end of the input is reached.

```
enum Tokenizer::CommentStyle {
  CPP_COMMENT_STYLE,
  SH_COMMENT_STYLE
}
```

Valid values for set_comment_style().

CPP_COMMENT_STYLE	Line comments begin with "//", block comments are delimited by "/*" and "* /".
SH_COMMENT_STYLE	Line comments begin with "#". No way to write block comments.

```
void Tokenizer::set_allow_f_after_float(
        bool value)
```

Set true to allow floats to be suffixed with the letter 'f'.

Tokens which would otherwise be integers but which have the 'f' suffix will be forced to be interpreted as floats. For all other purposes, the 'f' is ignored.

```
static double Tokenizer::ParseFloat(
        const string & text)
```

Parses a TYPE_FLOAT token.

This never fails, so long as the text actually comes from a TYPE_FLOAT token parsed by Tokenizer. If it doesn't, the result is undefined (possibly an assert failure).

```
static void Tokenizer::ParseString(
        const string & text,
        string * output)
```

Parses a TYPE_STRING token.

This never fails, so long as the text actually comes from a TYPE_STRING token parsed by Tokenizer. If it doesn't, the result is undefined (possibly an assert failure).

```
static bool Tokenizer::ParseInteger(
        const string & text,
        uint64 max_value,
        uint64 * output)
```

Parses a TYPE_INTEGER token.

Returns false if the result would be greater than max_value. Otherwise, returns true and sets *output to the result. If the text is not from a Token of type TYPE_INTEGER originally parsed by a Tokenizer, the result is undefined (possibly an assert failure).

struct Tokenizer::Token

```
#include <google/protobuf/io/tokenizer.h>
namespace google::protobuf::io
```

Structure representing a token read from the token stream.

Members	
TokenType	**type**
string	**text** *The exact text of the token as it appeared in the input. more...*
int	**line** *"line" and "column" specify the position of the first character of the token within the input stream. more...*
int	**column**

stringToken::text

The exact text of the token as it appeared in the input.

e.g. tokens of TYPE_STRING will still be escaped and in quotes.

---

intToken::line

"line" and "column" specify the position of the first character of the token within the input stream.

They are zero-based.

zero_copy_stream.h

`#include <google/protobuf/io/zero_copy_stream.h>`
`namespace google::protobuf::io`

This file contains the ZeroCopyInputStream and ZeroCopyOutputStream interfaces, which represent abstract I/O streams to and from which protocol buffers can be read and written.

For a few simple implementations of these interfaces, see zero_copy_stream_impl.h.

These interfaces are different from classic I/O streams in that they try to minimize the amount of data copying that needs to be done. To accomplish this, responsibility for allocating buffers is moved to the stream object, rather than being the responsibility of the caller. So, the stream can return a buffer which actually points directly into the final data structure where the bytes are to be stored, and the caller can interact directly with that buffer, eliminating an intermediate copy operation.

As an example, consider the common case in which you are reading bytes from an array that is already in memory (or perhaps an mmap()ed file). With classic I/O streams, you would do something like:

```
char buffer[BUFFER_SIZE];
input->Read(buffer, BUFFER_SIZE);
DoSomething(buffer, BUFFER_SIZE);
```

Then, the stream basically just calls memcpy() to copy the data from the array into your buffer. With a ZeroCopyInputStream, you would do this instead:

```
const void* buffer;
int size;
input->Next(&buffer, &size);
DoSomething(buffer, size);
```

Here, no copy is performed. The input stream returns a pointer directly into the backing array, and the caller ends up reading directly from it.

If you want to be able to read the old-fashion way, you can create a CodedInputStream or CodedOutputStream wrapping these objects and use their ReadRaw()/WriteRaw() methods. These will, of course, add a copy step, but Coded*Stream will handle buffering so at least it will be reasonably efficient.

ZeroCopyInputStream example:

```
// Read in a file and print its contents to stdout.
int fd = open("myfile", O_RDONLY);
ZeroCopyInputStream* input = new FileInputStream(fd);

const void* buffer;
int size;
while (input->Next(&buffer, &size)) {
  cout.write(buffer, size);
}

delete input;
close(fd);
```

ZeroCopyOutputStream example:

```
// Copy the contents of "infile" to "outfile", using plain read() for
// "infile" but a ZeroCopyOutputStream for "outfile".
int infd = open("infile", O_RDONLY);
int outfd = open("outfile", O_WRONLY);
ZeroCopyInputStream* output = new FileOutputStream(outfd);

void* buffer;
int size;
while (output->Next(&buffer, &size)) {
  int bytes = read(infd, buffer, size);
  if (bytes < size) {
    // Reached EOF.
    output->BackUp(size - bytes);
    break;
  }
}

delete output;
close(infd);
close(outfd);
```

Classes in this file

`ZeroCopyInputStream` *Abstract interface similar to an input stream but designed to minimize copying.*
`ZeroCopyOutputStream` *Abstract interface similar to an output stream but designed to minimize copying.*

class ZeroCopyInputStream

`#include <google/protobuf/io/zero_copy_stream.h>`
`namespace google::protobuf::io`

Abstract interface similar to an input stream but designed to minimize copying.

Known subclasses:

- `ArrayInputStream`
- `ConcatenatingInputStream`
- `CopyingInputStreamAdaptor`
- `FileInputStream`
- `IstreamInputStream`
- `LimitingInputStream`

Members

	`ZeroCopyInputStream``()`
`virtual`	**`~ZeroCopyInputStream`**`()`
`virtual bool`	**`Next`**`(const void ** data, int * size) = 0` *Obtains a chunk of data from the stream. more...*
`virtual void`	**`BackUp`**`(int count) = 0` *Backs up a number of bytes, so that the next call to Next() returns data again that was already returned by the last call to Next(). more...*
`virtual bool`	**`Skip`**`(int count) = 0` *Skips a number of bytes. more...*
`virtual`	**`ByteCount`**`() const = 0`

int64	*Returns the total number of bytes read since this object was created.*

```
virtual bool ZeroCopyInputStream::Next(
        const void ** data,
        int * size) = 0
```

Obtains a chunk of data from the stream.

Preconditions:

- "size" and "data" are not NULL.

Postconditions:

- If the returned value is false, there is no more data to return or an error occurred. All errors are permanent.
- Otherwise, "size" points to the actual number of bytes read and "data" points to a pointer to a buffer containing these bytes.
- Ownership of this buffer remains with the stream, and the buffer remains valid only until some other method of the stream is called or the stream is destroyed.
- It is legal for the returned buffer to have zero size, as long as repeatedly calling Next() eventually yields a buffer with non-zero size.

```
virtual void ZeroCopyInputStream::BackUp(
        int count) = 0
```

Backs up a number of bytes, so that the next call to Next() returns data again that was already returned by the last call to Next().

This is useful when writing procedures that are only supposed to read up to a certain point in the input, then return. If Next() returns a buffer that goes beyond what you wanted to read, you can use BackUp() to return to the point where you intended to finish.

Preconditions:

- The last method called must have been Next().
- count must be less than or equal to the size of the last buffer returned by Next().

Postconditions:

- The last "count" bytes of the last buffer returned by Next() will be pushed back into the stream. Subsequent calls to Next() will return the same data again before producing new data.

```
virtual bool ZeroCopyInputStream::Skip(
        int count) = 0
```

Skips a number of bytes.

Returns false if the end of the stream is reached or some input error occurred. In the end-of-stream case, the stream is advanced to the end of the stream (so ByteCount() will return the total size of the stream).

class ZeroCopyOutputStream

```
#include <google/protobuf/io/zero_copy_stream.h>
namespace google::protobuf::io
```

Abstract interface similar to an output stream but designed to minimize copying.

Known subclasses:

- `ArrayOutputStream`
- `CopyingOutputStreamAdaptor`
- `FileOutputStream`
- `OstreamOutputStream`
- `StringOutputStream`

Members	
	ZeroCopyOutputStream()
virtual	**~ZeroCopyOutputStream**()
virtual bool	**Next**(void ** data, int * size) = 0 *Obtains a buffer into which data can be written. more...*
virtual void	**BackUp**(int count) = 0 *Backs up a number of bytes, so that the end of the last buffer returned by Next() is not actually written. more...*
virtual int64	**ByteCount**() const = 0 *Returns the total number of bytes written since this object was created.*

```
virtual bool ZeroCopyOutputStream::Next(
        void ** data,
        int * size) = 0
```

Obtains a buffer into which data can be written.

Any data written into this buffer will eventually (maybe instantly, maybe later on) be written to the output.

Preconditions:

- "size" and "data" are not NULL.

Postconditions:

- If the returned value is false, an error occurred. All errors are permanent.
- Otherwise, "size" points to the actual number of bytes in the buffer and "data" points to the buffer.
- Ownership of this buffer remains with the stream, and the buffer remains valid only until some other method of the stream is called or the stream is destroyed.
- Any data which the caller stores in this buffer will eventually be written to the output (unless BackUp() is called).
- It is legal for the returned buffer to have zero size, as long as repeatedly calling Next() eventually yields a buffer with non-zero size.

```
virtual void ZeroCopyOutputStream::BackUp(
        int count) = 0
```

Backs up a number of bytes, so that the end of the last buffer returned by Next() is not actually written.

This is needed when you finish writing all the data you want to write, but the last buffer was bigger than you needed. You don't want to write a bunch of garbage after the end of your data, so you use BackUp() to back up.

Preconditions:

- The last method called must have been Next().
- count must be less than or equal to the size of the last buffer returned by Next().
- The caller must not have written anything to the last "count" bytes of that buffer.

Postconditions:

- The last "count" bytes of the last buffer returned by Next() will be ignored.

zero_copy_stream_impl.h

`#include <google/protobuf/io/zero_copy_stream_impl.h>`
`namespace google::protobuf::io`

This file contains common implementations of the interfaces defined in zero_copy_stream.h.

These implementations cover I/O on raw arrays, strings, and file descriptors. Of course, many users will probably want to write their own implementations of these interfaces specific to the particular I/O abstractions they prefer to use, but these should cover the most common cases.

Classes in this file

`ArrayInputStream`
A ZeroCopyInputStream backed by an in-memory array of bytes.

`ArrayOutputStream`
A ZeroCopyOutputStream backed by an in-memory array of bytes.

`StringOutputStream`
A ZeroCopyOutputStream which appends bytes to a string.

`CopyingInputStream`
A generic traditional input stream interface.

`CopyingInputStreamAdaptor`
A ZeroCopyInputStream which reads from a CopyingInputStream.

`CopyingOutputStream`
A generic traditional output stream interface.

`CopyingOutputStreamAdaptor`
A ZeroCopyOutputStream which writes to a CopyingOutputStream.

`FileInputStream`
A ZeroCopyInputStream which reads from a file descriptor.

`FileOutputStream`
A ZeroCopyOutputStream which writes to a file descriptor.

`IstreamInputStream`
A ZeroCopyInputStream which reads from a C++ istream.

`OstreamOutputStream`
A ZeroCopyOutputStream which writes to a C++ ostream.

`ConcatenatingInputStream`
A ZeroCopyInputStream which reads from several other streams in sequence.

`LimitingInputStream`
A ZeroCopyInputStream which wraps some other stream and limits it to a particular byte count.

class ArrayInputStream: public ZeroCopyInputStream

`#include <google/protobuf/io/zero_copy_stream_impl.h>`

`namespace google::protobuf::io`

A ZeroCopyInputStream backed by an in-memory array of bytes.

Members	
	`ArrayInputStream(const void * data, int size, int block_size = -1)` *Create an InputStream that returns the bytes pointed to by "data". more...*
	`~ArrayInputStream()`
implements ZeroCopyInputStream	
`virtual bool`	`Next(const void ** data, int * size)` *Obtains a chunk of data from the stream. more...*
`virtual void`	`BackUp(int count)` *Backs up a number of bytes, so that the next call to Next() returns data again that was already returned by the last call to Next(). more...*
`virtual bool`	`Skip(int count)` *Skips a number of bytes. more...*
`virtual int64`	`ByteCount() const` *Returns the total number of bytes read since this object was created.*

```
ArrayInputStream::ArrayInputStream(
        const void * data,
        int size,
        int block_size = -1)
```

Create an InputStream that returns the bytes pointed to by "data".

"data" remains the property of the caller but must remain valid until the stream is destroyed. If a block_size is given, calls to Next() will return data blocks no larger than the given size. Otherwise, the first call to Next() returns the entire array. block_size is mainly useful for testing; in production you would probably never want to set it.

```
virtual bool ArrayInputStream::Next(
        const void ** data,
        int * size)
```

Obtains a chunk of data from the stream.

Preconditions:

- "size" and "data" are not NULL.

Postconditions:

- If the returned value is false, there is no more data to return or an error occurred. All errors are permanent.
- Otherwise, "size" points to the actual number of bytes read and "data" points to a pointer to a buffer containing these bytes.
- Ownership of this buffer remains with the stream, and the buffer remains valid only until some other method of the stream is called or the stream is destroyed.
- It is legal for the returned buffer to have zero size, as long as repeatedly calling Next() eventually yields a buffer with non-zero size.

```
virtual void ArrayInputStream::BackUp(
        int count)
```

Backs up a number of bytes, so that the next call to Next() returns data again that was already returned by the last call to Next().

This is useful when writing procedures that are only supposed to read up to a certain point in the input, then return. If Next() returns a buffer that goes beyond what you wanted to read, you can use BackUp() to return to the point where you intended to finish.

Preconditions:

- The last method called must have been Next().
- count must be less than or equal to the size of the last buffer returned by Next().

Postconditions:

- The last "count" bytes of the last buffer returned by Next() will be pushed back into the stream. Subsequent calls to Next() will return the same data again before producing new data.

```
virtual bool ArrayInputStream::Skip(
        int count)
```

Skips a number of bytes.

Returns false if the end of the stream is reached or some input error occurred. In the end-of-stream case, the stream is advanced to the end of the stream (so ByteCount() will return the total size of the stream).

class ArrayOutputStream: public ZeroCopyOutputStream

```
#include <google/protobuf/io/zero_copy_stream_impl.h>
namespace google::protobuf::io
```

A ZeroCopyOutputStream backed by an in-memory array of bytes.

Members	
	`ArrayOutputStream(void * data, int size, int block_size = -1)` *Create an OutputStream that writes to the bytes pointed to by "data". more...*
	`~ArrayOutputStream()`
implements ZeroCopyOutputStream	
`virtual bool`	`Next(void ** data, int * size)` *Obtains a buffer into which data can be written. more...*
`virtual void`	`BackUp(int count)` *Backs up a number of bytes, so that the end of the last buffer returned by Next() is not actually written. more...*
`virtual int64`	`ByteCount() const` *Returns the total number of bytes written since this object was created.*

```
ArrayOutputStream::ArrayOutputStream(
        void * data,
        int size,
        int block_size = -1)
```

Create an OutputStream that writes to the bytes pointed to by "data".

"data" remains the property of the caller but must remain valid until the stream is destroyed. If a block_size is given, calls to Next() will return data blocks no larger than the given size. Otherwise, the first call to Next() returns the entire array. block_size is mainly useful for testing; in production you would probably never want to set it.

```
virtual bool ArrayOutputStream::Next(
        void ** data,
        int * size)
```

Obtains a buffer into which data can be written.

Any data written into this buffer will eventually (maybe instantly, maybe later on) be written to the output.

Preconditions:

- "size" and "data" are not NULL.

Postconditions:

- If the returned value is false, an error occurred. All errors are permanent.
- Otherwise, "size" points to the actual number of bytes in the buffer and "data" points to the buffer.
- Ownership of this buffer remains with the stream, and the buffer remains valid only until some other method of the stream is called or the stream is destroyed.
- Any data which the caller stores in this buffer will eventually be written to the output (unless BackUp() is called).
- It is legal for the returned buffer to have zero size, as long as repeatedly calling Next() eventually yields a buffer with non-zero size.

```
virtual void ArrayOutputStream::BackUp(
        int count)
```

Backs up a number of bytes, so that the end of the last buffer returned by Next() is not actually written.

This is needed when you finish writing all the data you want to write, but the last buffer was bigger than you needed. You don't want to write a bunch of garbage after the end of your data, so you use BackUp() to back up.

Preconditions:

- The last method called must have been Next().
- count must be less than or equal to the size of the last buffer returned by Next().
- The caller must not have written anything to the last "count" bytes of that buffer.

Postconditions:

- The last "count" bytes of the last buffer returned by Next() will be ignored.

class StringOutputStream: public ZeroCopyOutputStream

```
#include <google/protobuf/io/zero_copy_stream_impl.h>
namespace google::protobuf::io
```

A ZeroCopyOutputStream which appends bytes to a string.

Members	
explicit	`StringOutputStream(string * target)` *Create a StringOutputStream which appends bytes to the given string. more...*
	`~StringOutputStream()`
implements ZeroCopyOutputStream	
virtual	`Next(void ** data, int * size)`

bool	*Obtains a buffer into which data can be written. more...*
virtual void	**BackUp**(int count) *Backs up a number of bytes, so that the end of the last buffer returned by Next() is not actually written. more...*
virtual int64	**ByteCount**() const *Returns the total number of bytes written since this object was created.*

---

explicit StringOutputStream::StringOutputStream(string * target)

Create a StringOutputStream which appends bytes to the given string.

The string remains property of the caller, but it MUST NOT be accessed in any way until the stream is destroyed.

Hint: If you call target->reserve(n) before creating the stream, the first call to Next() will return at least n bytes of buffer space.

---

virtual bool StringOutputStream::Next(void ** data, int * size)

Obtains a buffer into which data can be written.

Any data written into this buffer will eventually (maybe instantly, maybe later on) be written to the output.

Preconditions:

- "size" and "data" are not NULL.

Postconditions:

- If the returned value is false, an error occurred. All errors are permanent.
- Otherwise, "size" points to the actual number of bytes in the buffer and "data" points to the buffer.
- Ownership of this buffer remains with the stream, and the buffer remains valid only until some other method of the stream is called or the stream is destroyed.
- Any data which the caller stores in this buffer will eventually be written to the output (unless BackUp() is called).
- It is legal for the returned buffer to have zero size, as long as repeatedly calling Next() eventually yields a buffer with non-zero size.

---

virtual void StringOutputStream::BackUp(int count)

Backs up a number of bytes, so that the end of the last buffer returned by Next() is not actually written.

This is needed when you finish writing all the data you want to write, but the last buffer was bigger than you needed. You don't want to write a bunch of garbage after the end of your data, so you use BackUp() to back up.

Preconditions:

- The last method called must have been Next().
- count must be less than or equal to the size of the last buffer returned by Next().
- The caller must not have written anything to the last "count" bytes of that buffer.

Postconditions:

- The last "count" bytes of the last buffer returned by Next() will be ignored.

class CopyingInputStream

```
#include <google/protobuf/io/zero_copy_stream_impl.h>
namespace google::protobuf::io
```

A generic traditional input stream interface.

Lots of traditional input streams (e.g. file descriptors, C stdio streams, and C++ iostreams) expose an interface where every read involves copying bytes into a buffer. If you want to take such an interface and make a ZeroCopyInputStream based on it, simply implement CopyingInputStream and then use CopyingInputStreamAdaptor.

CopyingInputStream implementations should avoid buffering if possible. CopyingInputStreamAdaptor does its own buffering and will read data in large blocks.

Members	
`virtual`	`~CopyingInputStream()`
`virtual int`	`Read(void * buffer, int size) = 0` *Reads up to "size" bytes into the given buffer. more...*
`virtual int`	`Skip(int count)` *Skips the next "count" bytes of input. more...*

```
virtual int CopyingInputStream::Read(
        void * buffer,
        int size) = 0
```

Reads up to "size" bytes into the given buffer.

Returns the number of bytes read. Read() waits until at least one byte is available, or returns zero if no bytes will ever become available (EOF), or -1 if a permanent read error occurred.

```
virtual int CopyingInputStream::Skip(
        int count)
```

Skips the next "count" bytes of input.

Returns the number of bytes actually skipped. This will always be exactly equal to "count" unless EOF was reached or permanent read error occurred.

The default implementation just repeatedly calls Read() into a scratch buffer.

class CopyingInputStreamAdaptor: public ZeroCopyInputStream

```
#include <google/protobuf/io/zero_copy_stream_impl.h>
namespace google::protobuf::io
```

A ZeroCopyInputStream which reads from a CopyingInputStream.

This is useful for implementing ZeroCopyInputStreams that read from traditional streams. Note that this class is not really zero-copy.

If you want to read from file descriptors or C++ istreams, this is already implemented for you: use FileInputStream or IstreamInputStream respectively.

Members

explicit	**CopyingInputStreamAdaptor**(CopyingInputStream * copying_stream, int block_size = -1) *Creates a stream that reads from the given CopyingInputStream. more...*
	~CopyingInputStreamAdaptor()
void	**SetOwnsCopyingStream**(bool value) *Call SetOwnsCopyingStream(true) to tell the CopyingInputStreamAdaptor to delete the underlying CopyingInputStream when it is destroyed.*

implements ZeroCopyInputStream

virtual bool	**Next**(const void ** data, int * size) *Obtains a chunk of data from the stream. more...*
virtual void	**BackUp**(int count) *Backs up a number of bytes, so that the next call to Next() returns data again that was already returned by the last call to Next(). more...*
virtual bool	**Skip**(int count) *Skips a number of bytes. more...*
virtual int64	**ByteCount**() const *Returns the total number of bytes read since this object was created.*

---

explicit CopyingInputStreamAdaptor::CopyingInputStreamAdaptor(CopyingInputStream * copying_stream, int block_size = -1)

Creates a stream that reads from the given CopyingInputStream.

If a block_size is given, it specifies the number of bytes that should be read and returned with each call to Next(). Otherwise, a reasonable default is used. The caller retains ownership of copying_stream unless SetOwnsCopyingStream(true) is called.

---

virtual bool CopyingInputStreamAdaptor::Next(const void ** data, int * size)

Obtains a chunk of data from the stream.

Preconditions:

- "size" and "data" are not NULL.

Postconditions:

- If the returned value is false, there is no more data to return or an error occurred. All errors are permanent.
- Otherwise, "size" points to the actual number of bytes read and "data" points to a pointer to a buffer containing these bytes.
- Ownership of this buffer remains with the stream, and the buffer remains valid only until some other method of the stream is called or the stream is destroyed.
- It is legal for the returned buffer to have zero size, as long as repeatedly calling Next() eventually yields a buffer with non-zero size.

---

virtual void CopyingInputStreamAdaptor::BackUp(int count)

Backs up a number of bytes, so that the next call to Next() returns data again that was already returned by the last call to Next().

This is useful when writing procedures that are only supposed to read up to a certain point in the input, then return. If Next() returns a buffer that goes beyond what you wanted to read, you can use BackUp() to return to the point where you intended to finish.

Preconditions:

- The last method called must have been Next().
- count must be less than or equal to the size of the last buffer returned by Next().

Postconditions:

- The last "count" bytes of the last buffer returned by Next() will be pushed back into the stream. Subsequent calls to Next() will return the same data again before producing new data.

```
virtual bool CopyingInputStreamAdaptor::Skip(
        int count)
```

Skips a number of bytes.

Returns false if the end of the stream is reached or some input error occurred. In the end-of-stream case, the stream is advanced to the end of the stream (so ByteCount() will return the total size of the stream).

class CopyingOutputStream

```
#include <google/protobuf/io/zero_copy_stream_impl.h>
namespace google::protobuf::io
```

A generic traditional output stream interface.

Lots of traditional output streams (e.g. file descriptors, C stdio streams, and C++ iostreams) expose an interface where every write involves copying bytes from a buffer. If you want to take such an interface and make a ZeroCopyOutputStream based on it, simply implement CopyingOutputStream and then use CopyingOutputStreamAdaptor.

CopyingOutputStream implementations should avoid buffering if possible. CopyingOutputStreamAdaptor does its own buffering and will write data in large blocks.

Members	
virtual	**~CopyingOutputStream**()
virtual bool	**Write**(const void * buffer, int size) = 0 *Writes "size" bytes from the given buffer to the output. more...*

```
virtual bool CopyingOutputStream::Write(
        const void * buffer,
        int size) = 0
```

Writes "size" bytes from the given buffer to the output.

Returns true if successful, false on a write error.

class CopyingOutputStreamAdaptor: public ZeroCopyOutputStream

```
#include <google/protobuf/io/zero_copy_stream_impl.h>
namespace google::protobuf::io
```

A ZeroCopyOutputStream which writes to a CopyingOutputStream.

This is useful for implementing ZeroCopyOutputStreams that write to traditional streams. Note that this class is not really zero-copy.

If you want to write to file descriptors or C++ ostreams, this is already implemented for you: use FileOutputStream or OstreamOutputStream respectively.

Members	
explicit	`CopyingOutputStreamAdaptor(CopyingOutputStream * copying_stream, int block_size = -1)` *Creates a stream that writes to the given Unix file descriptor. more...*
	`~CopyingOutputStreamAdaptor()`
bool	`Flush()` *Writes all pending data to the underlying stream. more...*
void	`SetOwnsCopyingStream(bool value)` *Call SetOwnsCopyingStream(true) to tell the CopyingOutputStreamAdaptor to delete the underlying CopyingOutputStream when it is destroyed.*
implements ZeroCopyOutputStream	
virtual bool	`Next(void ** data, int * size)` *Obtains a buffer into which data can be written. more...*
virtual void	`BackUp(int count)` *Backs up a number of bytes, so that the end of the last buffer returned by Next() is not actually written. more...*
virtual int64	`ByteCount() const` *Returns the total number of bytes written since this object was created.*

```
explicit CopyingOutputStreamAdaptor::CopyingOutputStreamAdaptor(
        CopyingOutputStream * copying_stream,
        int block_size = -1)
```

Creates a stream that writes to the given Unix file descriptor.

If a block_size is given, it specifies the size of the buffers that should be returned by Next(). Otherwise, a reasonable default is used.

```
bool CopyingOutputStreamAdaptor::Flush()
```

Writes all pending data to the underlying stream.

Returns false if a write error occurred on the underlying stream. (The underlying stream itself is not necessarily flushed.

```
virtual bool CopyingOutputStreamAdaptor::Next(
        void ** data,
        int * size)
```

Obtains a buffer into which data can be written.

Any data written into this buffer will eventually (maybe instantly, maybe later on) be written to the output.

Preconditions:

- "size" and "data" are not NULL.

Postconditions:

- If the returned value is false, an error occurred. All errors are permanent.
- Otherwise, "size" points to the actual number of bytes in the buffer and "data" points to the buffer.
- Ownership of this buffer remains with the stream, and the buffer remains valid only until some other method of the stream is called or the stream is destroyed.
- Any data which the caller stores in this buffer will eventually be written to the output (unless BackUp() is called).
- It is legal for the returned buffer to have zero size, as long as repeatedly calling Next() eventually yields a buffer with non-zero size.

```
virtual void CopyingOutputStreamAdaptor::BackUp(
        int count)
```

Backs up a number of bytes, so that the end of the last buffer returned by Next() is not actually written.

This is needed when you finish writing all the data you want to write, but the last buffer was bigger than you needed. You don't want to write a bunch of garbage after the end of your data, so you use BackUp() to back up.

Preconditions:

- The last method called must have been Next().
- count must be less than or equal to the size of the last buffer returned by Next().
- The caller must not have written anything to the last "count" bytes of that buffer.

Postconditions:

- The last "count" bytes of the last buffer returned by Next() will be ignored.

class FileInputStream: public ZeroCopyInputStream

```
#include <google/protobuf/io/zero_copy_stream_impl.h>
namespace google::protobuf::io
```

A ZeroCopyInputStream which reads from a file descriptor.

FileInputStream is preferred over using an ifstream with IstreamInputStream. The latter will introduce an extra layer of buffering, harming performance. Also, it's conceivable that FileInputStream could someday be enhanced to use zero-copy file descriptors on OSs which support them.

Members	
explicit	**FileInputStream**(int file_descriptor, int block_size = -1) *Creates a stream that reads from the given Unix file descriptor. more...*
	~FileInputStream()
bool	**Close**() *Flushes any buffers and closes the underlying file. more...*
void	**SetCloseOnDelete**(bool value) *By default, the file descriptor is not closed when the stream is destroyed. more...*
int	**GetErrno**() *If an I/O error has occurred on this file descriptor, this is the errno from that error. more...*
implements ZeroCopyInputStream	
virtual bool	**Next**(const void ** data, int * size) *Obtains a chunk of data from the stream. more...*
virtual	**BackUp**(int count)

void	*Backs up a number of bytes, so that the next call to Next() returns data again that was already returned by the last call to Next(). more...*
virtual bool	**Skip**(int count) *Skips a number of bytes. more...*
virtual int64	**ByteCount**() const *Returns the total number of bytes read since this object was created.*

`explicit FileInputStream::FileInputStream( int file_descriptor, int block_size = -1)`

Creates a stream that reads from the given Unix file descriptor.

If a block_size is given, it specifies the number of bytes that should be read and returned with each call to Next(). Otherwise, a reasonable default is used.

`bool FileInputStream::Close()`

Flushes any buffers and closes the underlying file.

Returns false if an error occurs during the process; use GetErrno() to examine the error. Even if an error occurs, the file descriptor is closed when this returns.

`void FileInputStream::SetCloseOnDelete( bool value)`

By default, the file descriptor is not closed when the stream is destroyed.

Call SetCloseOnDelete(true) to change that. WARNING: This leaves no way for the caller to detect if close() fails. If detecting close() errors is important to you, you should arrange to close the descriptor yourself.

`int FileInputStream::GetErrno()`

If an I/O error has occurred on this file descriptor, this is the errno from that error.

Otherwise, this is zero. Once an error occurs, the stream is broken and all subsequent operations will fail.

`virtual bool FileInputStream::Next( const void ** data, int * size)`

Obtains a chunk of data from the stream.

Preconditions:

- "size" and "data" are not NULL.

Postconditions:

- If the returned value is false, there is no more data to return or an error occurred. All errors are permanent.
- Otherwise, "size" points to the actual number of bytes read and "data" points to a pointer to a buffer containing

these bytes.

- Ownership of this buffer remains with the stream, and the buffer remains valid only until some other method of the stream is called or the stream is destroyed.
- It is legal for the returned buffer to have zero size, as long as repeatedly calling Next() eventually yields a buffer with non-zero size.

```
virtual void FileInputStream::BackUp(
        int count)
```

Backs up a number of bytes, so that the next call to Next() returns data again that was already returned by the last call to Next().

This is useful when writing procedures that are only supposed to read up to a certain point in the input, then return. If Next() returns a buffer that goes beyond what you wanted to read, you can use BackUp() to return to the point where you intended to finish.

Preconditions:

- The last method called must have been Next().
- count must be less than or equal to the size of the last buffer returned by Next().

Postconditions:

- The last "count" bytes of the last buffer returned by Next() will be pushed back into the stream. Subsequent calls to Next() will return the same data again before producing new data.

```
virtual bool FileInputStream::Skip(
        int count)
```

Skips a number of bytes.

Returns false if the end of the stream is reached or some input error occurred. In the end-of-stream case, the stream is advanced to the end of the stream (so ByteCount() will return the total size of the stream).

class FileOutputStream: public ZeroCopyOutputStream

```
#include <google/protobuf/io/zero_copy_stream_impl.h>
namespace google::protobuf::io
```

A ZeroCopyOutputStream which writes to a file descriptor.

FileInputStream is preferred over using an ofstream with OstreamOutputStream. The latter will introduce an extra layer of buffering, harming performance. Also, it's conceivable that FileInputStream could someday be enhanced to use zero-copy file descriptors on OSs which support them.

Members

explicit	**FileOutputStream**(int file_descriptor, int block_size = -1) *Creates a stream that writes to the given Unix file descriptor. more...*
	~FileOutputStream()
bool	**Close**() *Flushes any buffers and closes the underlying file. more...*
void	**SetCloseOnDelete**(bool value) *By default, the file descriptor is not closed when the stream is destroyed. more...*
int	**GetErrno**() *If an I/O error has occurred on this file descriptor, this is the errno from that error. more...*

implements ZeroCopyOutputStream

virtual bool	**Next**(void ** data, int * size) *Obtains a buffer into which data can be written. more...*
virtual void	**BackUp**(int count) *Backs up a number of bytes, so that the end of the last buffer returned by Next() is not actually written. more...*
virtual int64	**ByteCount**() const *Returns the total number of bytes written since this object was created.*

```
explicit FileOutputStream::FileOutputStream(
        int file_descriptor,
        int block_size = -1)
```

Creates a stream that writes to the given Unix file descriptor.

If a block_size is given, it specifies the size of the buffers that should be returned by Next(). Otherwise, a reasonable default is used.

```
bool FileOutputStream::Close()
```

Flushes any buffers and closes the underlying file.

Returns false if an error occurs during the process; use GetErrno() to examine the error. Even if an error occurs, the file descriptor is closed when this returns.

```
void FileOutputStream::SetCloseOnDelete(
        bool value)
```

By default, the file descriptor is not closed when the stream is destroyed.

Call SetCloseOnDelete(true) to change that. WARNING: This leaves no way for the caller to detect if close() fails. If detecting close() errors is important to you, you should arrange to close the descriptor yourself.

```
int FileOutputStream::GetErrno()
```

If an I/O error has occurred on this file descriptor, this is the errno from that error.

Otherwise, this is zero. Once an error occurs, the stream is broken and all subsequent operations will fail.

```
virtual bool FileOutputStream::Next(
        void ** data,
        int * size)
```

Obtains a buffer into which data can be written.

Any data written into this buffer will eventually (maybe instantly, maybe later on) be written to the output.

Preconditions:

- "size" and "data" are not NULL.

Postconditions:

- If the returned value is false, an error occurred. All errors are permanent.
- Otherwise, "size" points to the actual number of bytes in the buffer and "data" points to the buffer.
- Ownership of this buffer remains with the stream, and the buffer remains valid only until some other method of the stream is called or the stream is destroyed.
- Any data which the caller stores in this buffer will eventually be written to the output (unless BackUp() is called).
- It is legal for the returned buffer to have zero size, as long as repeatedly calling Next() eventually yields a buffer with non-zero size.

```
virtual void FileOutputStream::BackUp(
        int count)
```

Backs up a number of bytes, so that the end of the last buffer returned by Next() is not actually written.

This is needed when you finish writing all the data you want to write, but the last buffer was bigger than you needed. You don't want to write a bunch of garbage after the end of your data, so you use BackUp() to back up.

Preconditions:

- The last method called must have been Next().
- count must be less than or equal to the size of the last buffer returned by Next().
- The caller must not have written anything to the last "count" bytes of that buffer.

Postconditions:

- The last "count" bytes of the last buffer returned by Next() will be ignored.

class IstreamInputStream: public ZeroCopyInputStream

```
#include <google/protobuf/io/zero_copy_stream_impl.h>
namespace google::protobuf::io
```

A ZeroCopyInputStream which reads from a C++ istream.

Note that for reading files (or anything represented by a file descriptor), FileInputStream is more efficient.

Members	
explicit	`IstreamInputStream(istream * stream, int block_size = -1)` *Creates a stream that reads from the given C++ istream. more...*
	`~IstreamInputStream()`
implements ZeroCopyInputStream	
virtual bool	`Next(const void ** data, int * size)` *Obtains a chunk of data from the stream. more...*
virtual void	`BackUp(int count)` *Backs up a number of bytes, so that the next call to Next() returns data again that was already returned by the last call to Next(). more...*
virtual bool	`Skip(int count)` *Skips a number of bytes. more...*
virtual int64	`ByteCount() const` *Returns the total number of bytes read since this object was created.*

```
explicit IstreamInputStream::IstreamInputStream(
        istream * stream,
        int block_size = -1)
```

Creates a stream that reads from the given C++ istream.

If a block_size is given, it specifies the number of bytes that should be read and returned with each call to Next(). Otherwise, a reasonable default is used.

```
virtual bool IstreamInputStream::Next(
        const void ** data,
        int * size)
```

Obtains a chunk of data from the stream.

Preconditions:

- "size" and "data" are not NULL.

Postconditions:

- If the returned value is false, there is no more data to return or an error occurred. All errors are permanent.
- Otherwise, "size" points to the actual number of bytes read and "data" points to a pointer to a buffer containing these bytes.
- Ownership of this buffer remains with the stream, and the buffer remains valid only until some other method of the stream is called or the stream is destroyed.
- It is legal for the returned buffer to have zero size, as long as repeatedly calling Next() eventually yields a buffer with non-zero size.

```
virtual void IstreamInputStream::BackUp(
        int count)
```

Backs up a number of bytes, so that the next call to Next() returns data again that was already returned by the last call to Next().

This is useful when writing procedures that are only supposed to read up to a certain point in the input, then return. If Next() returns a buffer that goes beyond what you wanted to read, you can use BackUp() to return to the point where you intended to finish.

Preconditions:

- The last method called must have been Next().
- count must be less than or equal to the size of the last buffer returned by Next().

Postconditions:

- The last "count" bytes of the last buffer returned by Next() will be pushed back into the stream. Subsequent calls to Next() will return the same data again before producing new data.

```
virtual bool IstreamInputStream::Skip(
        int count)
```

Skips a number of bytes.

Returns false if the end of the stream is reached or some input error occurred. In the end-of-stream case, the stream is advanced to the end of the stream (so ByteCount() will return the total size of the stream).

class OstreamOutputStream: public ZeroCopyOutputStream

```
#include <google/protobuf/io/zero_copy_stream_impl.h>
namespace google::protobuf::io
```

A ZeroCopyOutputStream which writes to a C++ ostream.

Note that for writing files (or anything represented by a file descriptor), FileOutputStream is more efficient.

Members	
`explicit`	`OstreamOutputStream(ostream * stream, int block_size = -1)` *Creates a stream that writes to the given C++ ostream. more...*
	`~OstreamOutputStream()`
implements ZeroCopyOutputStream	
`virtual bool`	`Next(void ** data, int * size)` *Obtains a buffer into which data can be written. more...*
`virtual void`	`BackUp(int count)` *Backs up a number of bytes, so that the end of the last buffer returned by Next() is not actually written. more...*
`virtual int64`	`ByteCount() const` *Returns the total number of bytes written since this object was created.*

```
explicit OstreamOutputStream::OstreamOutputStream(
        ostream * stream,
        int block_size = -1)
```

Creates a stream that writes to the given C++ ostream.

If a block_size is given, it specifies the size of the buffers that should be returned by Next(). Otherwise, a reasonable default is used.

```
virtual bool OstreamOutputStream::Next(
        void ** data,
        int * size)
```

Obtains a buffer into which data can be written.

Any data written into this buffer will eventually (maybe instantly, maybe later on) be written to the output.

Preconditions:

- "size" and "data" are not NULL.

Postconditions:

- If the returned value is false, an error occurred. All errors are permanent.
- Otherwise, "size" points to the actual number of bytes in the buffer and "data" points to the buffer.
- Ownership of this buffer remains with the stream, and the buffer remains valid only until some other method of the stream is called or the stream is destroyed.
- Any data which the caller stores in this buffer will eventually be written to the output (unless BackUp() is called).
- It is legal for the returned buffer to have zero size, as long as repeatedly calling Next() eventually yields a buffer with non-zero size.

```
virtual void OstreamOutputStream::BackUp(
```

```
        int count)
```

Backs up a number of bytes, so that the end of the last buffer returned by Next() is not actually written.

This is needed when you finish writing all the data you want to write, but the last buffer was bigger than you needed. You don't want to write a bunch of garbage after the end of your data, so you use BackUp() to back up.

Preconditions:

- The last method called must have been Next().
- count must be less than or equal to the size of the last buffer returned by Next().
- The caller must not have written anything to the last "count" bytes of that buffer.

Postconditions:

- The last "count" bytes of the last buffer returned by Next() will be ignored.

class ConcatenatingInputStream: public ZeroCopyInputStream

```
#include <google/protobuf/io/zero_copy_stream_impl.h>
namespace google::protobuf::io
```

A ZeroCopyInputStream which reads from several other streams in sequence.

ConcatenatingInputStream is unable to distinguish between end-of-stream and read errors in the underlying streams, so it assumes any errors mean end-of-stream. So, if the underlying streams fail for any other reason, ConcatenatingInputStream may do odd things. It is suggested that you do not use ConcatenatingInputStream on streams that might produce read errors other than end-of-stream.

Members	
	ConcatenatingInputStream(ZeroCopyInputStream *const streams, int count) *All streams passed in as well as the array itself must remain valid until the ConcatenatingInputStream is destroyed.*
	~ConcatenatingInputStream()
implements ZeroCopyInputStream	
virtual bool	**Next**(const void ** data, int * size) *Obtains a chunk of data from the stream. more...*
virtual void	**BackUp**(int count) *Backs up a number of bytes, so that the next call to Next() returns data again that was already returned by the last call to Next(). more...*
virtual bool	**Skip**(int count) *Skips a number of bytes. more...*
virtual int64	**ByteCount**() const *Returns the total number of bytes read since this object was created.*

```
virtual bool ConcatenatingInputStream::Next(
        const void ** data,
        int * size)
```

Obtains a chunk of data from the stream.

Preconditions:

- "size" and "data" are not NULL.

Postconditions:

- If the returned value is false, there is no more data to return or an error occurred. All errors are permanent.

- Otherwise, "size" points to the actual number of bytes read and "data" points to a pointer to a buffer containing these bytes.
- Ownership of this buffer remains with the stream, and the buffer remains valid only until some other method of the stream is called or the stream is destroyed.
- It is legal for the returned buffer to have zero size, as long as repeatedly calling Next() eventually yields a buffer with non-zero size.

```
virtual void ConcatenatingInputStream::BackUp(
        int count)
```

Backs up a number of bytes, so that the next call to Next() returns data again that was already returned by the last call to Next().

This is useful when writing procedures that are only supposed to read up to a certain point in the input, then return. If Next() returns a buffer that goes beyond what you wanted to read, you can use BackUp() to return to the point where you intended to finish.

Preconditions:

- The last method called must have been Next().
- count must be less than or equal to the size of the last buffer returned by Next().

Postconditions:

- The last "count" bytes of the last buffer returned by Next() will be pushed back into the stream. Subsequent calls to Next() will return the same data again before producing new data.

```
virtual bool ConcatenatingInputStream::Skip(
        int count)
```

Skips a number of bytes.

Returns false if the end of the stream is reached or some input error occurred. In the end-of-stream case, the stream is advanced to the end of the stream (so ByteCount() will return the total size of the stream).

class LimitingInputStream: public ZeroCopyInputStream

```
#include <google/protobuf/io/zero_copy_stream_impl.h>
namespace google::protobuf::io
```

A ZeroCopyInputStream which wraps some other stream and limits it to a particular byte count.

Members	
	LimitingInputStream(ZeroCopyInputStream * input, int64 limit)
	~LimitingInputStream()
implements ZeroCopyInputStream	
virtual bool	**Next**(const void ** data, int * size) *Obtains a chunk of data from the stream. more...*
virtual void	**BackUp**(int count) *Backs up a number of bytes, so that the next call to Next() returns data again that was already returned by the last call to Next(). more...*
virtual bool	**Skip**(int count) *Skips a number of bytes. more...*
virtual	**ByteCount**() const

int64	*Returns the total number of bytes read since this object was created.*


```
virtual bool LimitingInputStream::Next(
        const void ** data,
        int * size)
```

Obtains a chunk of data from the stream.

Preconditions:

- "size" and "data" are not NULL.

Postconditions:

- If the returned value is false, there is no more data to return or an error occurred. All errors are permanent.
- Otherwise, "size" points to the actual number of bytes read and "data" points to a pointer to a buffer containing these bytes.
- Ownership of this buffer remains with the stream, and the buffer remains valid only until some other method of the stream is called or the stream is destroyed.
- It is legal for the returned buffer to have zero size, as long as repeatedly calling Next() eventually yields a buffer with non-zero size.

```
virtual void LimitingInputStream::BackUp(
        int count)
```

Backs up a number of bytes, so that the next call to Next() returns data again that was already returned by the last call to Next().

This is useful when writing procedures that are only supposed to read up to a certain point in the input, then return. If Next() returns a buffer that goes beyond what you wanted to read, you can use BackUp() to return to the point where you intended to finish.

Preconditions:

- The last method called must have been Next().
- count must be less than or equal to the size of the last buffer returned by Next().

Postconditions:

- The last "count" bytes of the last buffer returned by Next() will be pushed back into the stream. Subsequent calls to Next() will return the same data again before producing new data.

```
virtual bool LimitingInputStream::Skip(
        int count)
```

Skips a number of bytes.

Returns false if the end of the stream is reached or some input error occurred. In the end-of-stream case, the stream is advanced to the end of the stream (so ByteCount() will return the total size of the stream).

code_generator.h

```
#include <google/protobuf/compiler/code_generator.h>
namespace google::protobuf::compiler
```

Defines the abstract interface implemented by each of the language-specific code generators.

Classes in this file

Classes in this file
CodeGenerator *The abstract interface to a class which generates code implementing a particular proto file in a particular language.*
OutputDirectory *CodeGenerators generate one or more files in a given directory.*

class CodeGenerator

```
#include <google/protobuf/compiler/code_generator.h>
namespace google::protobuf::compiler
```

The abstract interface to a class which generates code implementing a particular proto file in a particular language.

A number of these may be registered with CommandLineInterface to support various languages.

Known subclasses:

- CppGenerator
- JavaGenerator
- Generator

Members

	CodeGenerator()
virtual	**~CodeGenerator**()
virtual bool	**Generate**(const FileDescriptor * file, const string & parameter, OutputDirectory * output_directory, string * error) const = 0 *Generates code for the given proto file, generating one or more files in the given output directory.* more...

```
virtual bool CodeGenerator::Generate(
        const FileDescriptor * file,
        const string & parameter,
        OutputDirectory * output_directory,
        string * error) const = 0
```

Generates code for the given proto file, generating one or more files in the given output directory.

A parameter to be passed to the generator can be specified on the command line. This is intended to be used by Java and similar languages to specify which specific class from the proto file is to be generated, though it could have other uses as well. It is empty if no parameter was given.

Returns true if successful. Otherwise, sets *error to a description of the problem (e.g. "invalid parameter") and returns false.

class OutputDirectory


```
#include <google/protobuf/compiler/code_generator.h>
namespace google::protobuf::compiler
```

CodeGenerators generate one or more files in a given directory.

This abstract interface represents the directory to which the CodeGenerator is to write.

Members

	OutputDirectory()
virtual	**~OutputDirectory**()
virtual io::ZeroCopyOutputStream *	**Open**(const string & filename) = 0 *Opens the given file, truncating it if it exists, and returns a ZeroCopyOutputStream that writes to the file. more...*

```
virtual io::ZeroCopyOutputStream *
    OutputDirectory::Open(
        const string & filename) = 0
```

Opens the given file, truncating it if it exists, and returns a ZeroCopyOutputStream that writes to the file.

The caller takes ownership of the returned object. This method never fails (a dummy stream will be returned instead).

The filename given should be relative to the root of the source tree. E.g. the C++ generator, when generating code for "foo/bar.proto", will generate the files "foo/bar.pb2.h" and "foo/bar.pb2.cc"; note that "foo/" is included in these filenames. The filename is not allowed to contain "." or ".." components.

command_line_interface.h

```
#include <google/protobuf/compiler/command_line_interface.h>
namespace google::protobuf::compiler
```

Implements the Protocol Compiler front-end such that it may be reused by custom compilers written to support other languages.

Classes in this file

CommandLineInterface
This class implements the command-line interface to the protocol compiler.

class CommandLineInterface

```
#include <google/protobuf/compiler/command_line_interface.h>
namespace google::protobuf::compiler
```

This class implements the command-line interface to the protocol compiler.

It is designed to make it very easy to create a custom protocol compiler supporting the languages of your choice. For example, if you wanted to create a custom protocol compiler binary which includes both the regular C++ support plus support for your own custom output "Foo", you would write a class "FooGenerator" which implements the CodeGenerator interface, then write a main() procedure like this:

```
int main(int argc, char* argv[]) {
  google::protobuf::compiler::CommandLineInterface cli;

  // Support generation of C++ source and headers.
  google::protobuf::compiler::cpp::CppGenerator cpp_generator;
  cli.RegisterGenerator("--cpp_out", &cpp_generator,
    "Generate C++ source and header.");

  // Support generation of Foo code.
  FooGenerator foo_generator;
  cli.RegisterGenerator("--foo_out", &foo_generator,
    "Generate Foo file.");

  return cli.Run(argc, argv);
}
```

The compiler is invoked with syntax like:

```
protoc --cpp_out=outdir --foo_out=outdir --proto_path=src src/foo.proto
```

For a full description of the command-line syntax, invoke it with --help.

Members

	CommandLineInterface()
	~CommandLineInterface()
void	**RegisterGenerator**(const string & flag_name, CodeGenerator * generator,

	`const string & help_text)` *Register a code generator for a language.* more...
`int`	**`Run`**`(int argc, const char *const argv)` *Run the Protocol Compiler with the given command-line parameters.* more...
`void`	**`SetInputsAreProtoPathRelative`**`(bool enable)` *Call SetInputsAreCwdRelative(true) if the input files given on the command line should be interpreted relative to the proto import path specified using --proto_path or -I flags.* more...
`void`	**`SetVersionInfo`**`(const string & text)` *Provides some text which will be printed when the --version flag is used.* more...

---

void CommandLineInterface::RegisterGenerator(const string & flag_name, CodeGenerator * generator, const string & help_text)

Register a code generator for a language.

Parameters:

- flag_name: The command-line flag used to specify an output file of this type. The name must start with a '-'. If the name is longer than one letter, it must start with two '-'s.
- generator: The CodeGenerator which will be called to generate files of this type.
- help_text: Text describing this flag in the --help output.

Some generators accept extra parameters. You can specify this parameter on the command-line by placing it before the output directory, separated by a colon:

```
protoc --foo_out=enable_bar:outdir
```

The text before the colon is passed to CodeGenerator::Generate() as the "parameter".

---

int CommandLineInterface::Run(int argc, const char *const argv)

Run the Protocol Compiler with the given command-line parameters.

Returns the error code which should be returned by main().

It may not be safe to call Run() in a multi-threaded environment because it calls strerror(). I'm not sure why you'd want to do this anyway.

---

void CommandLineInterface::SetInputsAreProtoPathRelative(bool enable)

Call SetInputsAreCwdRelative(true) if the input files given on the command line should be interpreted relative to the proto import path specified using --proto_path or -I flags.

Otherwise, input file names will be interpreted relative to the current working directory (or as absolute paths if they start with '/'), though they must still reside inside a directory given by --proto_path or the compiler will fail. The latter mode is generally more intuitive and easier to use, especially e.g. when defining implicit rules in Makefiles.

```
void CommandLineInterface::SetVersionInfo(
        const string & text)
```

Provides some text which will be printed when the --version flag is used.

The version of libprotoc will also be printed on the next line after this text.

importer.h

```
#include <google/protobuf/compiler/importer.h>
namespace google::protobuf::compiler
```

This file is the public interface to the .proto file parser.

Classes in this file

`SourceTreeDescriptorDatabase` *An implementation of DescriptorDatabase which loads files from a SourceTree and parses them.*
`Importer` *Simple interface for parsing .proto files.*
`MultiFileErrorCollector` *If the importer encounters problems while trying to import the proto files, it reports them to a MultiFileErrorCollector.*
`SourceTree` *Abstract interface which represents a directory tree containing proto files.*
`DiskSourceTree` *An implementation of SourceTree which loads files from locations on disk.*

class SourceTreeDescriptorDatabase: public DescriptorDatabase

```
#include <google/protobuf/compiler/importer.h>
namespace google::protobuf::compiler
```

An implementation of DescriptorDatabase which loads files from a SourceTree and parses them.

Note: This class is not thread-safe since it maintains a table of source code locations for error reporting. However, when a DescriptorPool wraps a DescriptorDatabase, it uses mutex locking to make sure only one method of the database is called at a time, even if the DescriptorPool is used from multiple threads. Therefore, there is only a problem if you create multiple DescriptorPools wrapping the same SourceTreeDescriptorDatabase and use them from multiple threads.

Note: This class does not implement FindFileContainingSymbol() or FindFileContainingExtension(); these will always return false.

Members

	`SourceTreeDescriptorDatabase(SourceTree * source_tree)`
	`~SourceTreeDescriptorDatabase()`
`void`	`RecordErrorsTo(MultiFileErrorCollector * error_collector)` *Instructs the SourceTreeDescriptorDatabase to report any parse errors to the given MultiFileErrorCollector. more...*
`DescriptorPool::ErrorCollector *`	`GetValidationErrorCollector()` *Gets a DescriptorPool::ErrorCollector which records errors to the MultiFileErrorCollector specified with RecordErrorsTo(). more...*

implements DescriptorDatabase

virtual bool	**FindFileByName**(const string & filename, FileDescriptorProto * output) *Find a file by file name. more...*
virtual bool	**FindFileContainingSymbol**(const string & symbol_name, FileDescriptorProto * output) *Find the file that declares the given fully-qualified symbol name. more...*
virtual bool	**FindFileContainingExtension**(const string & containing_type, int field_number, FileDescriptorProto * output) *Find the file which defines an extension extending the given message type with the given field number. more...*

```
void SourceTreeDescriptorDatabase::RecordErrorsTo(
        MultiFileErrorCollector * error_collector)
```

Instructs the SourceTreeDescriptorDatabase to report any parse errors to the given MultiFileErrorCollector.

This should be called before parsing. error_collector must remain valid until either this method is called again or the SourceTreeDescriptorDatabase is destroyed.

```
DescriptorPool::ErrorCollector *
    SourceTreeDescriptorDatabase::GetValidationErrorCollector()
```

Gets a DescriptorPool::ErrorCollector which records errors to the MultiFileErrorCollector specified with RecordErrorsTo().

This collector has the ability to determine exact line and column numbers of errors from the information given to it by the DescriptorPool.

```
virtual bool SourceTreeDescriptorDatabase::FindFileByName(
        const string & filename,
        FileDescriptorProto * output)
```

Find a file by file name.

Fills in in *output and returns true if found. Otherwise, returns false, leaving the contents of *output undefined.

```
virtual bool SourceTreeDescriptorDatabase::FindFileContainingSymbol(
        const string & symbol_name,
        FileDescriptorProto * output)
```

Find the file that declares the given fully-qualified symbol name.

If found, fills in *output and returns true, otherwise returns false and leaves *output undefined.

```
virtual bool SourceTreeDescriptorDatabase::FindFileContainingExtension
```

```
        const string & containing_type,
        int field_number,
        FileDescriptorProto * output)
```

Find the file which defines an extension extending the given message type with the given field number.

If found, fills in *output and returns true, otherwise returns false and leaves *output undefined. containing_type must be a fully-qualified type name.

class Importer

```
#include <google/protobuf/compiler/importer.h>
namespace google::protobuf::compiler
```

Simple interface for parsing .proto files.

This wraps the process of opening the file, parsing it with a Parser, recursively parsing all its imports, and then cross-linking the results to produce a FileDescriptor.

This is really just a thin wrapper around SourceTreeDescriptorDatabase. You may find that SourceTreeDescriptorDatabase is more flexible.

TODO(kenton): I feel like this class is not well-named.

Members	
	`Importer(SourceTree * source_tree, MultiFileErrorCollector * error_collector)`
	`~Importer()`
`const FileDescriptor *`	`Import(const string & filename)` *Import the given file and build a FileDescriptor representing it. more...*
`const DescriptorPool *`	`pool() const` *The DescriptorPool in which all imported FileDescriptors and their contents are stored.*

```
const FileDescriptor *
    Importer::Import(
        const string & filename)
```

Import the given file and build a FileDescriptor representing it.

If the file is already in the DescriptorPool, the existing FileDescriptor will be returned. The FileDescriptor is property of the DescriptorPool, and will remain valid until it is destroyed. If any errors occur, they will be reported using the error collector and Import() will return NULL.

A particular Importer object will only report errors for a particular file once. All future attempts to import the same file will return NULL without reporting any errors. The idea is that you might want to import a lot of files without seeing the same errors over and over again. If you want to see errors for the same files repeatedly, you can use a separate Importer object to import each one (but use the same DescriptorPool so that they can be cross-linked).

class MultiFileErrorCollector

```
#include <google/protobuf/compiler/importer.h>
namespace google::protobuf::compiler
```

If the importer encounters problems while trying to import the proto files, it reports them to a MultiFileErrorCollector.

Members	
	`MultiFileErrorCollector()`

virtual	**~MultiFileErrorCollector**()
virtual void	**AddError**(const string & filename, int line, int column, const string & message) = 0 *Line and column numbers are zero-based. more...*

```
virtual void MultiFileErrorCollector::AddError(
        const string & filename,
        int line,
        int column,
        const string & message) = 0
```

Line and column numbers are zero-based.

A line number of -1 indicates an error with the entire file (e.g. "not found").

class SourceTree

```
#include <google/protobuf/compiler/importer.h>
namespace google::protobuf::compiler
```

Abstract interface which represents a directory tree containing proto files.

Used by the default implementation of Importer to resolve import statements Most users will probably want to use the DiskSourceTree implementation, below.

Known subclasses:

- DiskSourceTree

Members	
	SourceTree()
virtual	**~SourceTree**()
virtual io::ZeroCopyInputStream *	**Open**(const string & filename) = 0 *Open the given file and return a stream that reads it, or NULL if not found. more...*

```
virtual io::ZeroCopyInputStream *
    SourceTree::Open(
        const string & filename) = 0
```

Open the given file and return a stream that reads it, or NULL if not found.

The caller takes ownership of the returned object. The filename must be a path relative to the root of the source tree and must not contain "." or ".." components.

class DiskSourceTree: public SourceTree

```
#include <google/protobuf/compiler/importer.h>
namespace google::protobuf::compiler
```

An implementation of SourceTree which loads files from locations on disk.

Multiple mappings can be set up to map locations in the DiskSourceTree to locations in the physical filesystem.

Members

enum	**DiskFileToVirtualFileResult** *Return type for DiskFileToVirtualFile(). more...*
	DiskSourceTree()
	~DiskSourceTree()
void	**MapPath**(const string & virtual_path, const string & disk_path) *Map a path on disk to a location in the SourceTree. more...*
DiskFileToVirtualFileResult	**DiskFileToVirtualFile**(const string & disk_file, string * virtual_file, string * shadowing_disk_file) *Given a path to a file on disk, find a virtual path mapping to that file. more...*

implements SourceTree

virtual io::ZeroCopyInputStream *	**Open**(const string & filename) *Open the given file and return a stream that reads it, or NULL if not found. more...*

```
enum DiskSourceTree::DiskFileToVirtualFileResult {
  SUCCESS,
  SHADOWED,
  CANNOT_OPEN,
  NO_MAPPING
}
```

Return type for DiskFileToVirtualFile()

SUCCESS	
SHADOWED	
CANNOT_OPEN	
NO_MAPPING	

```
void DiskSourceTree::MapPath(
        const string & virtual_path,
        const string & disk_path)
```

Map a path on disk to a location in the SourceTree.

The path may be either a file or a directory. If it is a directory, the entire tree under it will be mapped to the given virtual location. To map a directory to the root of the source tree, pass an empty string for virtual_path.

If multiple mapped paths apply when opening a file, they will be searched in order. For example, if you do:

```
MapPath("bar", "foo/bar");
MapPath("", "baz");
```

and then you do:

```
Open("bar/qux");
```


the DiskSourceTree will first try to open foo/bar/qux, then baz/bar/qux, returning the first one that opens successfuly.

disk_path may be an absolute path or relative to the current directory, just like a path you'd pass to open().

DiskFileToVirtualFileResult
DiskSourceTree::DiskFileToVirtualFile(
const string & disk_file,
string * virtual_file,
string * shadowing_disk_file)

Given a path to a file on disk, find a virtual path mapping to that file.

The first mapping created with MapPath() whose disk_path contains the filename is used. However, that virtual path may not actually be usable to open the given file. Possible return values are:

- SUCCESS: The mapping was found. *virtual_file is filled in so that calling Open(*virtual_file) will open the file named by disk_file.
- SHADOWED: A mapping was found, but using Open() to open this virtual path will end up returning some differer file. This is because some other mapping with a higher precedence also matches this virtual path and maps it to a different file that exists on disk. *virtual_file is filled in as it would be in the SUCCESS case. *shadowing_disk_file is filled in with the disk path of the file which would be opened if you were to call Open(*virtual_file).
- CANNOT_OPEN: The mapping was found and was not shadowed, but the file specified cannot be opened. When this value is returned, errno will indicate the reason the file cannot be opened. *virtual_file will be set to the virtual path as in the SUCCESS case, even though it is not useful.
- NO_MAPPING: Indicates that no mapping was found which contains this file.

---

virtual io::ZeroCopyInputStream *
DiskSourceTree::Open(
const string & filename)

Open the given file and return a stream that reads it, or NULL if not found.

The caller takes ownership of the returned object. The filename must be a path relative to the root of the source tree and must not contain "." or ".." components.

parser.h

```
#include <google/protobuf/compiler/parser.h>
namespace google::protobuf::compiler
```

Implements parsing of .proto files to FileDescriptorProtos.

Classes in this file

Parser *Implements parsing of protocol definitions (such as .proto files).*
SourceLocationTable *A table mapping (descriptor, ErrorLocation) pairs -- as reported by DescriptorPool when validating descriptors -- to line and column numbers within the original source code.*

class Parser

```
#include <google/protobuf/compiler/parser.h>
namespace google::protobuf::compiler
```

Implements parsing of protocol definitions (such as .proto files).

Note that most users will be more interested in the Importer class. Parser is a lower-level class which simply converts a single .proto file to a FileDescriptorProto. It does not resolve import directives or perform many other kinds of validation needed to construct a complete FileDescriptor.

Members

	Parser()
	~Parser()
bool	**Parse**(io::Tokenizer * input, FileDescriptorProto * file) *Parse the entire input and construct a FileDescriptorProto representing it. more...*
void	**RecordSourceLocationsTo**(SourceLocationTable * location_table) *Requests that locations of certain definitions be recorded to the given SourceLocationTable while parsing. more...*
void	**RecordErrorsTo**(io::ErrorCollector * error_collector) *Requsets that errors be recorded to the given ErrorCollector while parsing. more...*
const string &	**GetSyntaxIndentifier**() *Returns the identifier used in the "syntax = " declaration, if one was seen during the last call to Parse(), or the empty string otherwise.*
void	**SetRequireSyntaxIdentifier**(bool value) *If set true, input files will be required to begin with a syntax identifier. more...*

```
bool Parser::Parse(
        io::Tokenizer * input,
```

```
        FileDescriptorProto * file)
```

Parse the entire input and construct a FileDescriptorProto representing it.

Returns true if no errors occurred, false otherwise.

```
void Parser::RecordSourceLocationsTo(
        SourceLocationTable * location_table)
```

Requests that locations of certain definitions be recorded to the given SourceLocationTable while parsing.

This can be used to look up exact line and column numbers for errors reported by DescriptorPool during validation. Set to NULL (the default) to discard source location information.

```
void Parser::RecordErrorsTo(
        io::ErrorCollector * error_collector)
```

Requsets that errors be recorded to the given ErrorCollector while parsing.

Set to NULL (the default) to discard error messages.

```
void Parser::SetRequireSyntaxIdentifier(
        bool value)
```

If set true, input files will be required to begin with a syntax identifier.

Otherwise, files may omit this. If a syntax identifier is provided, it must be 'syntax = "proto2";' and must appear at the top of this file regardless of whether or not it was required.

class SourceLocationTable

```
#include <google/protobuf/compiler/parser.h>
namespace google::protobuf::compiler
```

A table mapping (descriptor, ErrorLocation) pairs -- as reported by DescriptorPool when validating descriptors -- to line and column numbers within the original source code.

Members	
	SourceLocationTable()
	~SourceLocationTable()
bool	**Find**(const Message * descriptor, DescriptorPool::ErrorCollector::ErrorLocation location, int * line, int * column) const *Finds the precise location of the given error and fills in *line and column with the line and column numbers. more...*
void	**Add**(const Message * descriptor, DescriptorPool::ErrorCollector::ErrorLocation location, int line, int column) *Adds a location to the table.*
void	**Clear**() *Clears the contents of the table.*

```
bool SourceLocationTable::Find(
        const Message * descriptor,
        DescriptorPool::ErrorCollector::ErrorLocation location,
        int * line,
        int * column) const
```

Finds the precise location of the given error and fills in *line and column with the line and column numbers.

If not found, sets *line to -1 and *column to 0 (since line = -1 is used to mean "error has no exact location" in the ErrorCollector interface). Returns true if found, false otherwise.

cpp_generator.h

```
#include <google/protobuf/compiler/cpp/cpp_generator.h>
namespace google::protobuf::compiler::cpp
```

Generates C++ code for a given .proto file.

Classes in this file

CppGenerator

CodeGenerator implementation which generates a C++ source file and header.

class CppGenerator: public CodeGenerator

```
#include <google/protobuf/compiler/cpp/cpp_generator.h>
namespace google::protobuf::compiler::cpp
```

CodeGenerator implementation which generates a C++ source file and header.

If you create your own protocol compiler binary and you want it to support C++ output, you can do so by registering an instance of this CodeGenerator with the CommandLineInterface in your main() function.

Members

	CppGenerator()
	~CppGenerator()

implements CodeGenerator

virtual bool	**Generate**(const FileDescriptor * file, const string & parameter, OutputDirectory * output_directory, string * error) const *Generates code for the given proto file, generating one or more files in the given output directory. more...*

```
virtual bool CppGenerator::Generate(
        const FileDescriptor * file,
        const string & parameter,
        OutputDirectory * output_directory,
        string * error) const
```

Generates code for the given proto file, generating one or more files in the given output directory.

A parameter to be passed to the generator can be specified on the command line. This is intended to be used by Java and similar languages to specify which specific class from the proto file is to be generated, though it could have other uses as well. It is empty if no parameter was given.

Returns true if successful. Otherwise, sets *error to a description of the problem (e.g. "invalid parameter") and returns false.

java_generator.h

```
#include <google/protobuf/compiler/java/java_generator.h>
namespace google::protobuf::compiler::java
```

Generates Java code for a given .proto file.

Classes in this file

JavaGenerator

CodeGenerator implementation which generates Java code.

class JavaGenerator: public CodeGenerator

```
#include <google/protobuf/compiler/java/java_generator.h>
namespace google::protobuf::compiler::java
```

CodeGenerator implementation which generates Java code.

If you create your own protocol compiler binary and you want it to support Java output, you can do so by registering an instance of this CodeGenerator with the CommandLineInterface in your main() function.

Members

	JavaGenerator()
	~JavaGenerator()

implements CodeGenerator

virtual bool	**Generate**(const FileDescriptor * file, const string & parameter, OutputDirectory * output_directory, string * error) const *Generates code for the given proto file, generating one or more files in the given output directory. more...*

```
virtual bool JavaGenerator::Generate(
        const FileDescriptor * file,
        const string & parameter,
        OutputDirectory * output_directory,
        string * error) const
```

Generates code for the given proto file, generating one or more files in the given output directory.

A parameter to be passed to the generator can be specified on the command line. This is intended to be used by Java and similar languages to specify which specific class from the proto file is to be generated, though it could have other uses as well. It is empty if no parameter was given.

Returns true if successful. Otherwise, sets *error to a description of the problem (e.g. "invalid parameter") and returns false.

python_generator.h

```
#include <google/protobuf/compiler/python/python_generator.h>
namespace google::protobuf::compiler::python
```

Generates Python code for a given .proto file.

Classes in this file

Generator *CodeGenerator implementation for generated Python protocol buffer classes.*

class Generator: public CodeGenerator

```
#include <google/protobuf/compiler/python/python_generator.h>
namespace google::protobuf::compiler::python
```

CodeGenerator implementation for generated Python protocol buffer classes.

If you create your own protocol compiler binary and you want it to support Python output, you can do so by registering an instance of this CodeGenerator with the CommandLineInterface in your main() function.

Members

	Generator()
virtual	**~Generator**()
virtual bool	**Generate**(const FileDescriptor * file, const string & parameter, OutputDirectory * output_directory, string * error) const *CodeGenerator methods.*