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` implements specialized versions of all methods for maximum speed. However, if the `.proto` file contains the line:

```
option optimize_for = CODE_SIZE;
```

then `Foo` will override only the minimum set of methods necessary to function and rely on reflection-based implementations of the rest. This significantly reduces the size of the generated code, but also reduces performance. Alternatively, if the `.proto` file contains:

```
option optimize_for = LITE_RUNTIME;
```

then `Foo` will include fast implementations of all methods, but will implement the `google::protobuf::MessageLite` interface, which only contains a subset of the methods of `Message`. In particular, it does not support descriptors or reflection. However, in this mode, the generated code only needs to link against `libprotobuf-lite.so` (`libprotobuf-lite.lib` on Windows) instead of `libprotobuf.so` (`libprotobuf.lib`). The "lite" library is much smaller than the full library, and is more appropriate for resource-constrained systems such as mobile phones.

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.

The `Message` interface defines methods that let you check, manipulate, read, or write the entire message, including parsing from and serializing to binary strings. In addition to these methods, 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.
- `void Swap(Foo* other)`: Swap content with another message.
- `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.

As well as accessor methods, the compiler generates an integer constant for each field containing its field number. The constant name is the letter `k`, followed by the field name converted to camel-case, followed by `FieldNumber`. For example, given the field `optional int32 foo_bar = 5;`, the compiler will generate the constant `static const int kFooBarFieldNumber = 5;`.

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`.
- `void set_foo(const char* value, int size)`: Like above, but the string size is given explicitly rather than determined by looking for a null-terminator byte.
- `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.
- `void set_allocated_foo(string* value)`: Sets the `string` object to the field and frees the previous field value if it exists. If the `string` pointer is not `NULL`, the message takes ownership of the allocated `string` object and `has_foo()` will return `true`. Otherwise, if the `value` is `NULL`, the behavior is the same as calling `clear_foo()`.
- `string* release_foo()`: Releases the ownership of the field and returns the pointer of the `string` object. After calling this, caller takes the ownership of the allocated `string` object, `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.
- `void set_allocated_foo(Bar* bar)`: Sets the `Bar` object to the field and frees the previous field value if it exists. If the `Bar` pointer is not `NULL`, the message takes ownership of the allocated `Bar` object and `has_foo()` will return `true`. Otherwise, if the `Bar` is `NULL`, the behavior is the same as calling `clear_foo()`.
- `Bar* release_foo()`: Releases the ownership of the field and returns the pointer of the `Bar` object. After calling this, caller takes the ownership of the allocated `Bar` object, `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.
- `void set_foo(int index, const char* value, int size)`: Like above, but the string size is given explicitly rather than determined by looking for a null-terminator byte.
- `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.
- `void add_foo(const char* value, int size)`: Like above, but the string size is given explicitly rather than determined by looking for a null-terminator byte.
- `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<int>& foo() const`: Returns the underlying `RepeatedField` that stores the field's elements. This container class provides STL-like iterators and other methods.
- `RepeatedField<int>*` 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.

- `const string& Foo_Name(int value)`: Returns the name for given numeric value. Returns an empty string if no such value exists. If multiple values have this number, the first one defined is returned. In the above example, `Foo_Name(5)` would return `"VALUE_B"`.
- `bool Foo_Parse(const string& name, Foo* value)`: If `name` is a valid value name for this enum, assigns that value into `value` and returns true. Otherwise returns false. In the above example, `Foo_Parse("VALUE_C", &someFoo)` would return true and set `someFoo` to 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()`, `AddExtension()`, `SetAllocatedExtension()` and `ReleaseExtension()`. 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 = 1;
  repeated int32 repeated_bar = 2;
}
```

For the singular extension field `bar`, 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.GetExtension(bar) == 1);
foo.ClearExtension(bar);
assert(!foo.HasExtension(bar));
```

Similarly, for the repeated extension field `repeated_bar`, the compiler generates an extension identifier called `repeated_bar`, which you can also use with `Foo`'s extension accessors:

```
Foo foo;
for (int i = 0; i < kSize; ++i) {
  foo.AddExtension(repeated_bar, i)
}
assert(foo.ExtensionSize(repeated_bar) == kSize)
for (int i = 0; i < kSize; ++i) {
  assert(foo.GetExtension(repeated_bar, i) == i)
}
```

(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

If the `.proto` file contains the following line:

```
option cc_generic_services = true;
```

Then the protocol buffer compiler will generate code based on the service definitions found in the file as described in this section. However, the generated code may be undesirable as it is not tied to any particular RPC system, and thus requires more levels of indirection that code tailored to one system. If you do NOT want this code to be generated, add this line to the file:

```
option cc_generic_services = false;
```

If neither of the above lines are given, the option defaults to `false`, as generic services are deprecated. (Note that prior to 2.4.0, the option defaults to `true`)

RPC systems based on `.proto`-language service definitions should provide plugins to generate code approriate for the system. These plugins are likely to require that abstract services are disabled, so that they can generate their own classes of the same names. Plugins are new in version 2.3.0 (January 2010).

The remainder of this section describes what the protocol buffer compiler generates when abstract services are enabled.

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.

Plugin Insertion Points

Code generator plugins which want to extend the output of the C++ code generator may insert code of the following types using the given insertion point names. Each insertion point appears in both the `.pb.cc` file and the `.pb.h` file unless otherwise noted.

- `includes`: Include directives.
- `namespace_scope`: Declarations that belong in the file's package/namespace, but not within any particular class. Appears after all other namespace-scope code.
- `global_scope`: Declarations that belong at the top level, outside of the file's namespace. Appears at the very end of the file.
- `class_scope:TYPENAME`: Member declarations that belong in a message class. `TYPENAME` is the full proto name, e.g. `package.MessageType`. Appears after all other public declarations in the class. This insertion point appears only in the `.pb.h` file.

> Do not generate code which relies on private class members declared by the standard code generator, as these implementation details may change in future versions of Protocol Buffers.