C++ Generated Code

This page describes exactly what C++ code the protocol buffer compiler generates for any given protocol definition. Any differences between proto2 and proto3 generated code are highlighted - note that these differences are in the generated code as described in this document, not the base message classes/interfaces, which are the same in both versions. You should read the proto2 language guide (https://developers.google.com/protocol-buffers/docs/proto) and/or proto3 language guide (https://developers.google.com/protocol-buffers/docs/proto3) 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`** (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.message.html#Message). The class is a concrete class; no pure-virtual methods are left unimplemented. Methods that are virtual in **`Message`** (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.message.html#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`](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.message_lite.html#MessageLite) (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.message_lite.html#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`](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.message.html#Message) (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.message.html#Message) interface defines methods that let you check, manipulate, read, or write the entire message, including parsing from and serializing to binary strings.

- `bool ParseFromString(const string& data)`: Parse the message from the given serialized binary string (also known as wire format).
- `bool SerializeToString(string* output) const`: Serialize the given message to a binary string.
- string DebugString(): Return a string giving the `text_format` representation of the proto (should only be used for debugging).

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`](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.unknown_field_set.html#UnknownFieldSet) (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.unknown_field_set.html#UnknownFieldSet)

& unknown_fields() const: Returns the set of unknown fields encountered while parsing this message.

- [**UnknownFieldSet**](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.unknown_field_set.html#UnknownFieldSet) (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.unknown_field_set.html#UnknownFieldSet) *** mutable_unknown_fields()**: Returns a pointer to the mutable set of unknown fields encountered while parsing this message.

The class also defines the following static methods:

- **static const** [**Descriptor**](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.descriptor) (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.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](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.message.html#Message.Reflection) (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.message.html#Message.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()**](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.message.html#Message.New) (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.message.html#Message.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;`.

For field accessors returning a `const` reference, that reference may be invalidated when the next modifying access is made to the message. This includes calling any non-`const` accessor of any field, calling any non-`const` method inherited from `Message` or modifying the message through other ways (for example, by using the message as the argument of `Swap()`). Correspondingly, the address of the returned reference is only guaranteed to be the same across different invocations of the accessor if no modifying access was made to the message in the meantime.

For field accessors returning a pointer, that pointer may be invalidated when the next modifying or non-modifying access is made to the message. This includes, regardless of constness, calling any accessor of any field, calling any method inherited from `Message` or accessing the message through other ways (for example, by copying the message using the copy constructor). Correspondingly, the value of the returned pointer is never guaranteed to be the same across two different invocations of the accessor.

Singular Numeric Fields (proto2)

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 (https://developers.google.com/protocol-buffers/docs/proto.html#scalar).

Singular Numeric Fields (proto3)

For this field definition:

```
int32 foo = 1;
```

The compiler will generate the following accessor methods:

- **int32 foo() const**: Returns the current value of the field. If the field is not set, returns 0.
- **void set_foo(int32 value)**: Sets the value of the field. After calling this, **foo()** will return **value**.
- **void clear_foo()**: Clears the value of the field. After calling this, **foo()** will return 0.

For other numeric field types (including **bool**), **int32** is replaced with the corresponding C++ type according to the scalar value types table (https://developers.google.com/protocol-buffers/docs/proto3.html#scalar).

Singular String Fields (proto2)

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(string&& value)** (C++11 and beyond): Sets the value of the field, moving from the passed string. 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 pointer to the mutable `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.
- `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`. The message is free to delete the allocated `string` object at any time, so references to the object may be invalidated. 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 String Fields (proto3)

For any of these field definitions:

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

The compiler will generate the following accessor methods:

- `const string& foo() const`: Returns the current value of the field. If the field is not set, returns the empty string/empty bytes.
- `void set_foo(const string& value)`: Sets the value of the field. After calling this, `foo()` will return a copy of `value`.
- `void set_foo(string&& value)` (C++11 and beyond): Sets the value of the field, moving from the passed string. After calling this, `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, `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 pointer to the mutable `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. After calling this, `foo()` will return whatever value is written into the given string.
- `void clear_foo()`: Clears the value of the field. After calling this, `foo()` will return the empty string/empty bytes.
- `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. The message is free to delete the allocated `string` object at any time, so references to the object may be invalidated. 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 and `foo()` will return the empty string/empty bytes.

Singular Enum Fields (proto2)

Given the enum type:

```
enum Bar {
  BAR_VALUE = 0;
  OTHER_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 Enum Fields (proto3)

Given the enum type:

```
enum Bar {
  BAR_VALUE = 0;
  OTHER_VALUE = 1;
}
```

For this field definitions:

```
Bar foo = 1;
```

The compiler will generate the following accessor methods:

- **`Bar foo() const`**: Returns the current value of the field. If the field is not set, returns the default value (0).
- **`void set_foo(Bar value)`**: Sets the value of the field. After calling this, `foo()` will return `value`.
- **`void clear_foo()`**: Clears the value of the field. After calling this, `foo()` will return the default value.

Singular Embedded Message Fields

Given the message type:

```
message Bar {}
```

For any of these field definitions:

```
//proto2
optional Bar foo = 1;
required Bar foo = 1;
```

```
//proto3
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 pointer to the mutable `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`.
- `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. Calling this method with index outside of [0, foo_size()) yields undefined behavior.
- `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` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) `<int32>& foo() const`: Returns the underlying `RepeatedField` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) that stores the field's elements. This container class provides STL-like iterators and other methods.
- `RepeatedField` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) `<int32>* mutable_foo()`: Returns a pointer to the underlying mutable `RepeatedField` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) 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 (https://developers.google.com/protocol-buffers/docs/proto.html#scalar).

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. Calling this method with index outside of [0, foo_size()) yields undefined behavior.
- `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 pointer to the mutable `string` object that stores the value of the element at the given zero-based index. Calling this method with index outside of [0, foo_size()) yields undefined behavior.
- `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.
- `void clear_foo()`: Removes all elements from the field. After calling this, `foo_size()` will return zero.
- `const RepeatedPtrField` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) `<string>& foo() const`: Returns the underlying `RepeatedPtrField` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) that stores the field's elements. This container class provides STL-like iterators and other methods.
- `RepeatedPtrField` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) `<string>* mutable_foo()`: Returns a pointer to the underlying mutable `RepeatedPtrField` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) 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 = 0;
```

```
  OTHER_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. Calling this method with index outside of [0, foo_size()) yields undefined behavior.
- `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` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) `<int>& foo() const`: Returns the underlying `RepeatedField` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) that stores the field's elements. This container class provides STL-like iterators and other methods.
- `RepeatedField` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) `<int>* mutable_foo()`: Returns a pointer to the underlying mutable `RepeatedField` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) 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. Calling this method with index outside of [0, foo_size()) yields undefined behavior.
- `Bar* mutable_foo(int index)`: Returns a pointer to the mutable `Bar` object that stores the value of the element at the given zero-based index. Calling this method with index outside of [0, foo_size()) yields undefined behavior.
- `Bar* add_foo()`: Adds a new element and returns a pointer to it. The returned `Bar` is mutable and will have none of its fields set (i.e. it will be identical to a newly-allocated `Bar`).
- `void clear_foo()`: Removes all elements from the field. After calling this, `foo_size()` will return zero.
- `const RepeatedPtrField` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) `<Bar>& foo() const`: Returns the underlying `RepeatedPtrField` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) that stores the field's elements. This container class provides STL-like iterators and other methods.
- `RepeatedPtrField` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) `<Bar>* mutable_foo()`: Returns a pointer to the underlying mutable `RepeatedPtrField` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) that stores the field's elements. This container class provides STL-like iterators and other methods.

Oneof Numeric Fields

For this oneof (#oneof) field definition:

```
oneof oneof_name {
    int32 foo = 1;
    ...
}
```

The compiler will generate the following accessor methods:

- `bool has_foo() const` (proto2 only): Returns `true` if oneof case is `kFoo`.
- `int32 foo() const`: Returns the current value of the field if oneof case is `kFoo`. Otherwise, returns the default value.
- `void set_foo(int32 value)`:
    - If any other oneof field in the same oneof is set, calls `clear_oneof_name()`.
    - Sets the value of this field and sets the oneof case to `kFoo`.
    - `has_foo()` (proto2 only) will return true, `foo()` will return `value`, and `oneof_name_case()` will return `kFoo`.
- `void clear_foo()`:
    - Nothing will be changed if oneof case is not `kFoo`.
    - If oneof case is `kFoo`, clears the value of the field and oneof case. `has_foo()` (proto2 only) will return `false`, `foo()` will return the default value and `oneof_name_case()` will return `ONEOF_NAME_NOT_SET`.

For other numeric field types (including `bool`), `int32` is replaced with the corresponding C++ type according to the scalar value types table (https://developers.google.com/protocol-buffers/docs/reference/language.shtml#scalar).

Oneof String Fields

For any of these oneof (#oneof) field definitions:

```
oneof oneof_name {
    string foo = 1;
    …
}
oneof oneof_name {
    bytes foo = 1;
    ….
}
```

The compiler will generate the following accessor methods:

- `bool has_foo() const` (proto2 only): Returns `true` if the oneof case is `kFoo`.
- `const string& foo() const`: Returns the current value of the field if the oneof case is `kFoo`. Otherwise, returns the default value.
- `void set_foo(const string& value)`:
    - If any other oneof field in the same oneof is set, calls `clear_oneof_name()`.
    - Sets the value of this field and sets the oneof case to `kFoo`.
    - `has_foo()` (proto2 only) will return `true`, `foo()` will return a copy of `value` and `oneof_name_case()` will return `kFoo`.
- `void set_foo(const char* value)`:
    - If any other oneof field in the same oneof is set, calls `clear_oneof_name()`.
    - Sets the value of the field using a C-style null-terminated string and set the oneof case to `kFoo`.
    - `has_foo()` (proto2 only) will return `true`, `foo()` will return a copy of `value` and `oneof_name_case()` will return `kFoo`.
- `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()`:
    - If any other oneof field in the same oneof is set, calls `clear_oneof_name()`.
    - Sets the oneof case to `kFoo` and returns a pointer to the mutable string object that stores the field's value. If the oneof case was not `kFoo` prior to the call, then the returned string will be empty (not the default value).
    - `has_foo()` (proto2 only) will return `true`, `foo()` will return whatever value is written into the given string and `oneof_name_case()` will return `kFoo`.
- `void clear_foo()`:
    - If the oneof case is not `kFoo`, nothing will be changed .
    - If the oneof case is `kFoo`, frees the field and clears the oneof case . `has_foo()` (proto2 only) will return `false`, `foo()` will return the default value, and `oneof_name_case()` will return `ONEOF_NAME_NOT_SET`.
- `void set_allocated_foo(string* value)`:
    - Calls `clear_oneof_name()`.
    - If the string pointer is not `NULL`: Sets the string object to the field and sets the oneof case to `kFoo`. The message takes ownership of the allocated string

object, `has_foo()` (proto2 only) will return `true` and `oneof_name_case()` will return `kFoo`.

    - If the string pointer is `NULL`, `has_foo()` (proto2 only) will return `false` and `oneof_name_case()` will return `ONEOF_NAME_NOT_SET`.

- `string* release_foo()`:
    - Returns `NULL` if oneof case is not `kFoo`.
    - Clears the oneof case, 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()` (proto2 only) will return false, `foo()` will return the default value, and `oneof_name_case()` will return `ONEOF_NAME_NOT_SET`.

Oneof Enum Fields

Given the enum type:

```
enum Bar {
  BAR_VALUE = 0;
  OTHER_VALUE = 1;
}
```

For the oneof (#oneof) field definition:

```
oneof oneof_name {
    Bar foo = 1;
    ...
}
```

The compiler will generate the following accessor methods:

- `bool has_foo() const` (proto2 only): Returns `true` if oneof case is `kFoo`.
- `Bar foo() const`: Returns the current value of the field if oneof case is `kFoo`. Otherwise, returns the default value.
- `void set_foo(Bar value)`:
    - If any other oneof field in the same oneof is set, calls `clear_oneof_name()`.
    - Sets the value of this field and sets the oneof case to `kFoo`.
    - `has_foo()` (proto2 only) will return `true`, `foo()` will return `value` and `oneof_name_case()` will return `kFoo`.

    - 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()`:
    - Nothing will be changed if the oneof case is not `kFoo`.
    - If the oneof case is `kFoo`, clears the value of the field and the oneof case. `has_foo()` (proto2 only) will return `false`, `foo()` will return the default value and `oneof_name_case()` will return `ONEOF_NAME_NOT_SET`.

Oneof Embedded Message Fields

Given the message type:

```
message Bar {}
```

For the oneof (#oneof) field definition:

```
oneof oneof_name {
    Bar foo = 1;
    ...
}
```

The compiler will generate the following accessor methods:

- `bool has_foo() const`: Returns true if oneof case is `kFoo`.
- `const Bar& foo() const`: Returns the current value of the field if oneof case is `kFoo`. Otherwise, returns `Bar::default_instance()`.
- `Bar* mutable_foo()`:
    - If any other oneof field in the same oneof is set, calls `clear_oneof_name()`.
    - Sets the oneof case to `kFoo` and returns a pointer to the mutable Bar object that stores the field's value. If the oneof case was not `kFoo` 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`, `foo()` will return a reference to the same instance of `Bar` and `oneof_name_case()` will return `kFoo`.
- `void clear_foo()`:
    - Nothing will be changed if the oneof case is not `kFoo`.
    - If the oneof case equals `kFoo`, frees the field and clears the oneof case. `has_foo()` will return `false`, `foo()` will return the default value and

  `oneof_name_case()` will return `ONEOF_NAME_NOT_SET`.

- `void set_allocated_foo(Bar* bar)`:
  - Calls `clear_oneof_name()`.
  - If the `Bar` pointer is not `NULL`: Sets the `Bar` object to the field and sets the oneof case to `kFoo`. The message takes ownership of the allocated `Bar` object, has_foo() will return true and oneof_name_case() will return `kFoo`.
  - If the pointer is `NULL`, `has_foo()` will return `false` and `oneof_name_case()` will return `ONEOF_NAME_NOT_SET`. (The behavior is like calling `clear_oneof_name()`)
- `Bar* release_foo()`:
  - Returns `NULL` if oneof case is not `kFoo`.
  - If the oneof case is `kFoo`, clears the oneof case, 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`, `foo()` will return the default value and `oneof_name_case()` will return `ONEOF_NAME_NOT_SET`.

Map Fields

For this map field definition:

```
map<int32, int32> weight = 1;
```

The compiler will generate the following accessor methods:

- `const google::protobuf::Map<int32, int32>& weight();`: Returns an immutable `Map`.
- `google::protobuf::Map<int32, int32>* mutable_weight();`: Returns a mutable `Map`.

A `google::protobuf::Map` is a special container type used in protocol buffers to store map fields. As you can see from its interface below, it uses a commonly-used subset of `std::map` and `std::unordered_map` methods.

```
template<typename Key, typename T> {
class Map {
  // Member types
  typedef Key key_type;
  typedef T mapped_type;
  typedef MapPair< Key, T > value_type;
```

```
  // Iterators
  iterator begin();
  const_iterator begin() const;
  const_iterator cbegin() const;
  iterator end();
  const_iterator end() const;
  const_iterator cend() const;
  // Capacity
  int size() const;
  bool empty() const;

  // Element access
  T& operator[](const Key& key);
  const T& at(const Key& key) const;
  T& at(const Key& key);

  // Lookup
  int count(const Key& key) const;
  const_iterator find(const Key& key) const;
  iterator find(const Key& key);

  // Modifiers
  pair<iterator, bool> insert(const value_type& value);
  template<class InputIt>
  void insert(InputIt first, InputIt last);
  size_type erase(const Key& Key);
  iterator erase(const_iterator pos);
  iterator erase(const_iterator first, const_iterator last);
  void clear();

  // Copy
  Map(const Map& other);
  Map& operator=(const Map& other);
}
```

The easiest way to add data is to use normal map syntax, for example:

```
std::unique_ptr<ProtoName> my_enclosing_proto(new ProtoName);
(*my_enclosing_proto->mutable_weight())[my_key] = my_value;
```

`pair<iterator, bool> insert(const value_type& value)` will implicitly cause a deep copy of the **`value_type`** instance. The most efficient way to insert a new value into a **`google::protobuf::Map`** is as follows:

```
T& operator[](const Key& key): map[new_key] = new_mapped;
```

Using `google::protobuf::Map` with standard maps

`google::protobuf::Map` supports the same iterator API as `std::map` and `std::unordered_map`. If you don't want to use `google::protobuf::Map` directly, you can convert a `google::protobuf::Map` to a standard map by doing the following:

```
std::map<int32, int32> standard_map(message.weight().begin(),
                                    message.weight().end());
```

Note that this will make a deep copy of the entire map.

You can also construct a `google::protobuf::Map` from a standard map as follows:

```
google::protobuf::Map<int32, int32> weight(standard_map.begin(), standar
```

Parsing unknown values

On the wire, a .proto map is equivalent to a map entry message for each key/value pair, while the map itself is a repeated field of map entries. Like ordinary message types, it's possible for a parsed map entry message to have unknown fields: for example a field of type `int64` in a map defined as `map<int32, string>`.

If there are unknown fields in the wire format of a map entry message, they will be discarded.

If there is an unknown enum value in the wire format of a map entry message, it's handled differently in proto2 and proto3. In proto2, the whole map entry message is put into the unknown field set of the containing message. In proto3, it is put into a map field as if it is a known enum value.

Any

Given an Any (https://developers.google.com/protocol-buffers/docs/proto3#any) field like this:

```
import "google/protobuf/any.proto";

message ErrorStatus {
  string message = 1;
  google.protobuf.Any details = 2;
}
```

In our generated code, the getter for the **`details`** field returns an instance of **`google::protobuf::Any`**. This provides the following special methods to pack and unpack the **`Any`**'s values:

```
class Any {
 public:
  // Packs the given message into this Any using the default type URL
  // prefix "type.googleapis.com".
  void PackFrom(const google::protobuf::Message& message);

  // Packs the given message into this Any using the given type URL
  // prefix.
  void PackFrom(const google::protobuf::Message& message,
                const string& type_url_prefix);

  // Unpacks this Any to a Message. Returns false if this Any
  // represents a different protobuf type or parsing fails.
  bool UnpackTo(google::protobuf::Message* message) const;

  // Returns true if this Any represents the given protobuf type.
  template<typename T> bool Is() const;
}
```

Oneof

Given a oneof definition like this:

```
oneof oneof_name {
    int32 foo_int = 4;
    string foo_string = 9;
    ...
}
```

The compiler will generate the following C++ enum type:

```
enum OneofNameCase {
  kFooInt = 4,
  kFooString = 9,
  ONEOF_NAME_NOT_SET = 0
}
```

In addition, it will generate these methods:

- `OneofNameCase oneof_name_case() const`: Returns the enum indicating which field is set. Returns `ONEOF_NAME_NOT_SET` if none of them is set.
- `void clear_oneof_name()`: Frees the object if the oneof field set uses a pointer (Message or String), and sets the oneof case to `ONEOF_NAME_NOT_SET`.

Enumerations

Given an enum definition like:

```
enum Foo {
  VALUE_A = 0;
  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` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.descriptor.html#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 0, 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", &some_foo)` would return true and set `some_foo` to 1234.
- `const Foo Foo_MIN`: the smallest valid value of the enum (VALUE_A in the example).
- `const Foo Foo_MAX`: the largest valid value of the enum (VALUE_C in the example).
- `const int Foo_ARRAYSIZE`: always defined as `Foo_MAX + 1`.

Be careful when casting integers to proto2 enums. If an integer is cast to a proto2 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 proto2 message to an invalid value may cause an assertion failure. If an invalid enum value is read when parsing a proto2 message, it will be treated as an unknown field (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.unknown_field_set). These semantics have been changed in proto3. It's safe to cast any integer to a proto3 enum value as long as it fits into int32. Invalid enum values will also be kept when parsing a proto3 message and returned by enum field accessors.

Be careful when using proto3 enums in switch statements. Proto3 enums are open enum types with possible values outside the range of specified symbols. Unrecognized enum values will be kept when parsing a proto3 message and returned by the enum field accessors. A switch statement on a proto3 enum without a default case will not be able to catch all cases even if all the known fields are listed. This could lead to unexpected behavior including data corruption and runtime crashes. **Always add a default case or explicitly call `Foo_IsValid(int)` outside of the switch to handle unknown enum values.**

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 (proto2 only)

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 = 123;
  repeated int32 repeated_bar = 124;
}
```

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);
```

Arena Allocation

Arena allocation is a C++-only feature that helps you optimize your memory usage and improve performance when working with protocol buffers. Enabling arena allocation in your `.proto` adds additional code for working with arenas to your C++ generated code. You can find out more about the arena allocation API in the Arena Allocation Guide (https://developers.google.com/protocol-buffers/docs/reference/arenas).

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 (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.compiler.plugin.pb) 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 (https://developers.google.com/protocol-buffers/docs/r
                 FooResponse* response, Closure (https://developers.google.com/protocol
```

The parameters are equivalent to the parameters of `Service::CallMethod()` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#Service.CallMethod.details), 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()` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#RpcController.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` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#Service) interface. The protocol buffer compiler automatically generates implementations of the methods of `Service` as follows:

- `GetDescriptor` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#Service.GetDescriptor) : Returns the service's `ServiceDescriptor` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.descriptor.html#ServiceDescriptor) .
- `CallMethod` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#Service.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` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#Service.GetRequestPrototype) and `GetResponsePrototype` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#Service.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` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.descriptor.html#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` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#RpcChannel) `* channel)`: Constructs a new stub which sends requests on the given channel.
- `Foo_Stub(RpcChannel` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#RpcChannel) `* channel, ChannelOwnership` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#Service.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` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#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()` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#RpcChannel.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` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#RpcChannel) and `RpcController` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#RpcController) . See the documentation for `service.h` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service) for more information.

Plugin Insertion Points

Code generator plugins (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.compiler.plugin.pb) 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.

上次更新日期：五月 13, 2019

Downloads
Protocol buffers downloads and instructions

GitHub
The latest protocol buffers code and releases