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 be 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`. In this case, you...:

   ```
   protoc -I=$SRC_DIR --cpp_out=$DST_DIR $SRC_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() cons
t;
  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 which directly 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;
    }
  }

  // Optional:  Delete all global objects allocated by libprotobuf.
  google::protobuf::ShutdownProtobufLibrary();

  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.

Also notice the call to `ShutdownProtobufLibrary()` at the end of the program. All this does is delete any global objects that were allocated by the Protocol Buffer library. This is unnecessary for most programs, since the process is just going to exit anyway and the OS will take care of reclaiming all of its memory. However, if you use a memory leak checker that requires

that every last object be freed, or if you are writing a library which may be loaded and unloaded multiple times by a single process, then you may want to force Protocol Buffers to clean up everything.

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);

  // Optional:  Delete all global objects allocated by libprotobuf.
  google::protobuf::ShutdownProtobufLibrary();

  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 not specified 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.

Optimization Tips

The C++ Protocol Buffers library is extremely heavily optimized. However, proper usage can improve performance even more. Here are some tips for squeezing every last drop of speed out of the library:

- Reuse message objects when possible. 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,

objects can become bloated over time, especially if your messages vary in "shape" or if you occasionally construct a message that is much larger than usual. You should monitor the sizes of your message objects by calling the `SpaceUsed` method and delete them once they get too big.

- Your system's memory allocator may not be well-optimized for allocating lots of small objects from multiple threads. Try using Google's tcmalloc instead.

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.

Last updated 四月 *2, 2012.*