# Overview

## What Is Protobuf?

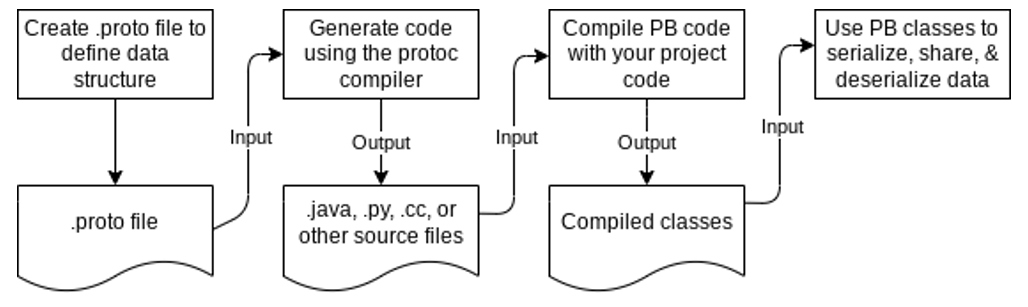
Google Protocol (or protocol buffers) is a language-neutral and platform-neutral mechanism for **serializing structured data** – think of XML and JSON, but smaller, faster, and simpler. Also, it’s free and open-source.

You define how you want your data to be structured once, then you can **use generated source code to easily write and read your structured data** to and from various data streams and using various languages.

Currently, protobuf supports C++, C#, Dart, Go, Java, Kotlin, Objective-C, Python, Ruby, PHP. And it can works in Linux, MacOS and Windows.

## How Does Protobuf Work?

### Message Definition Files



Protocol buffer **messages are described by .proto files**.

Example:

message Person {

  optional string name = 1;

  optional int32 id = 2;

  optional string email = 3;

}

The proto compiler is invoked at build time on these files to generate code to manipulate the corresponding protocol buffer. Each generated class contains getters and setters for each field, as well as methods to **serialize the whole structure to raw bytes**.

Compiling.proto files creates a Builder class that you can use to create new instances, as in Java code:

Person john = Person.newBuilder()

    .setId(1234)

    .setName("John Doe")

    .setEmail("jdoe@example.com")

    .build();

output = new FileOutputStream(args[0]);

john.writeTo(output);

You can then **deserialize raw bytes** using generated methods in other languages, like C++:

Person john;

fstream input(argv[1], ios::in | ios::binary);

john.ParseFromIstream(&input);

int id = john.id();

std::string name = john.name();

std::string email = john.email();

### Data Encoding With Wire Format

If you create a Person class and make the object in JSON format, it would look like this. This data size is 82 bytes if you remove spaces.

{

    "userName": "Martin",

    "favouriteNumber": 1337,

    "interests": ["daydreaming", "hacking"]

}

If protocol buffers are used, the above would be expressed as follows:

message Person {

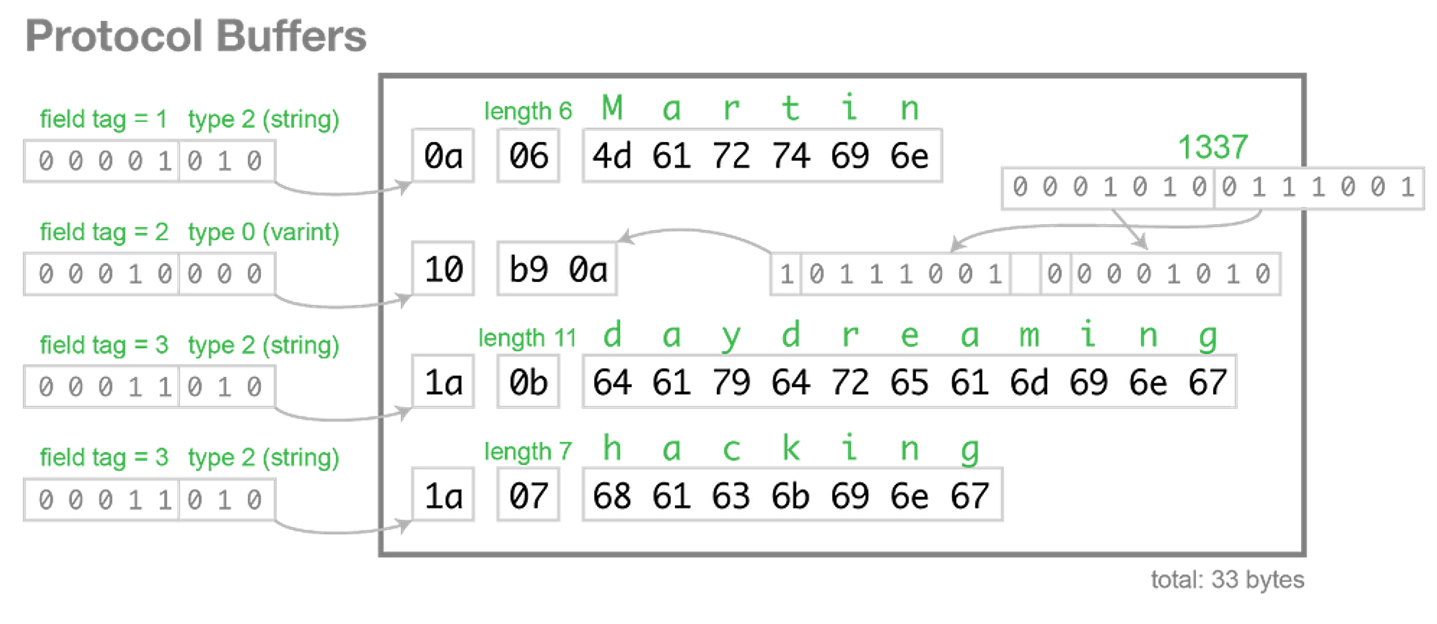
    required string user\_name           = 1;

    optional int64  favourite\_number    = 2;

    repeated string interests           = 3;

}

What protobuf does is to replace attribute values (Martin, 1337, daydreaming, hacking) with integer values (1, 2, and 3 in this case).



Let’s take the field userName as an example. It is divided into 8 bits:

* 5 bits (0b00001) represent the field tag (number) attached to the proto file
* 3 bits (0b010) represent the field data type

Also, the first byte (0x0a) contains meta information (field tag and type), the second byte (0x06) indicates the length of data, and the rest (0x4d, 0x61, 0x72, 0x74, 0x69, 0x6e) is the value of data.

Using this principle, **the whole structure can be expressed with only 33 bytes (which is almost 3x smaller than it in JSON)**.

For more details, check:

<https://protobuf.dev/programming-guides/encoding/>

<https://github.com/square/wire>

## Advantages and Disadvantages

### Advantages

#### Compact Data Storage

As describe in the above session.

#### Fast Parsing

It is common to parse data in JSON format back into an object and use it as an object.

But with protobuf, **when a byte comes, it’s used in memory as it is**, and it ends when the object reference points to it. Thus, there is no need (or very little) for parsing data.

3 - 10x smaller and 20 - 100x faster than XML.

#### Fast Data Transfer Speed

As the data parsing takes very short time, you can send data very fast.

Also, as the bytes that represent the data are smaller, you can send it even faster.

#### Cross-Language Compatibility

**The same messages can be read by code written in two different languages**. For example, you can have a Java program on one platform capturing data from a cental software system, serialize it based on a .proto definition, and then extract specific values from that serialized data in a separate Python program running on another platform.

#### Cross-Project Compatibility

You can **use protobuf across projects by defining message types in** .proto **files** that reside outside of a specific project’s code base.

#### No Code Update

As long as you follow some simple practices when **updating** .proto **definitions**, old code will read new messages without issues, ignoring any newly added fields. To the old code, fields that were deleted will have their default value, and deleted repeated fields will be empty.

New code will also transparently read old messages. New fields will not be present in old messages; in these cases protocol buffers provide a reasonable default value.

### Disadvantages

#### Not All Data Work

Protobuf does not fit all data. In particular:

* Protocol buffers assume that **entire messages can be loaded into memory at once** **and are not larger than an object graph**. For data bigger than a few megabytes, consider a different solution. When working with larger data, you may effectively end up with several copies of the data due to serialized copies, which can **cause surprising spikes in memory usage**.
* When protocol buffers are serialized, **the same data can have many different binary serializations**. You cannot compare two messages for equality without fully parsing them.
* Protocol buffer **messages are not compressed**.
* Protocol buffer messages are less than maximally efficient in both size and speed for many uses that involve **large, multi-dimensional arrays of floating point numbers**. For these applications, FITS and similar formats have less overhead.
* Protocol buffers are **not well supported in non-object-oriented languages**, such as Fortran and IDL.
* Protocol buffer messages don’t inherently self-describe their data, but they have a fully reflective schema that you can use to implement self-description. That is, you cannot fully interpret one without access to its corresponding .proto file.
* Protocol buffers are **not a formal standard of any organization**. This makes them unsuitable for use in environments with legal or other requirements to build on top of standards.

#### Discomfort in reading

In the case of JSON, it is easy to read the data. However, **the data written by protobuf is more difficult to read**.

If you don't have a .proto file, you don't know what it means.

This makes it problematic to be used as an external API (since every client needs to have a .proto file). Thus, protobuf is mainly used for data exchange between internal services.

#### Re-Compilation of Code

**If a .proto file is changed, the application that using that file must be re-compiled**.

# Installation

## Download

<https://protobuf.dev/downloads/>

Basicially, the release package of protobuf contains:

* A executable file – protoc
* Several .proto files which define the structure and semantics of standard Google Protocol Buffers messages and types. They’re required when working with protobuf.

This is a quick look to files in protobuf release package in Linux:

.

│ readme.txt

│

├───bin

│ protoc

│

└───include

└───google

└───protobuf

│ any.proto

│ api.proto

│ descriptor.proto

│ duration.proto

│ empty.proto

│ field\_mask.proto

│ source\_context.proto

│ struct.proto

│ timestamp.proto

│ type.proto

│ wrappers.proto

│

└───compiler

plugin.proto

## Installation

### Linux

<https://grpc.io/docs/protoc-installation/>

### Windows

<https://www.geeksforgeeks.org/how-to-install-protocol-buffers-on-windows/>

## CLI

**Usage: protoc.exe [OPTION] PROTO\_FILES**

Parse PROTO\_FILES and generate output based on the options given:

**-IPATH, --proto\_path=PATH** Specify the directory in which to search for imports. May be specified multiple times;

directories will be searched in order.

If not given, the current working directory is used.

If not found in any of the these directories, the --descriptor\_set\_in descriptors will be

checked for required proto file.

**--version** Show version info and exit.

**-h, --help** Show this text and exit.

**--encode=MESSAGE\_TYPE** Read a text-format message of the given type from standard input and write it in binary

to standard output. The message type must be defined in PROTO\_FILES or their imports.

**--deterministic\_output** When using --encode, ensure map fields are deterministically ordered. Note that this order

is not canonical, and changes across builds or releases of protoc.

**--decode=MESSAGE\_TYPE** Read a binary message of the given type from standard input and write it in text format

to standard output. The message type must be defined in PROTO\_FILES or their imports.

**--decode\_raw** Read an arbitrary protocol message from standard input and write the raw tag/value

pairs in text format to standard output. No PROTO\_FILES should be given when using this

flag.

**--descriptor\_set\_in=FILES** Specifies a delimited list of FILES each containing a FileDescriptorSet

(a protocol buffer defined in descriptor.proto).

The FileDescriptor for each of the PROTO\_FILES

provided will be loaded from these FileDescriptorSets. If a FileDescriptor

appears multiple times, the first occurrence will be used.

**-oFILE,** Writes a FileDescriptorSet (a protocol buffer,

**--descriptor\_set\_out=FILE** defined in descriptor.proto) containing all of

the input files to FILE.

**--include\_imports** When using --descriptor\_set\_out, also include

all dependencies of the input files in the

set, so that the set is self-contained.

**--include\_source\_info** When using --descriptor\_set\_out, do not strip SourceCodeInfo from the FileDescriptorProto.

This results in vastly larger descriptors that include information about the original

location of each decl in the source file as well as surrounding comments.

**--retain\_options** When using --descriptor\_set\_out, do not strip any options from the FileDescriptorProto.

This results in potentially larger descriptors

that include information about options that were

only meant to be useful during compilation.

**--dependency\_out=FILE** Write a dependency output file in the format expected by make. This writes the transitive

set of input file paths to FILE

**--error\_format=FORMAT** Set the format in which to print errors.

FORMAT may be 'gcc' (the default) or 'msvs'

(Microsoft Visual Studio format).

**--fatal\_warnings** Make warnings be fatal (similar to -Werr in gcc). This flag will make protoc return

with a non-zero exit code if any warnings are generated.

**--print\_free\_field\_numbers** Print the free field numbers of the messages

defined in the given proto files. Extension ranges are counted as occupied fields numbers.

**--enable\_codegen\_trace** Enables tracing which parts of protoc are responsible for what codegen output. Not supported

by all backends or on all platforms.

**--plugin=EXECUTABLE** Specifies a plugin executable to use.

Normally, protoc searches the PATH for plugins, but you may specify additional

executables not in the path using this flag.

Additionally, EXECUTABLE may be of the form NAME=PATH, in which case the given plugin name

is mapped to the given executable even if the executable's own name differs.

**--cpp\_out=OUT\_DIR** Generate C++ header and source.

**--csharp\_out=OUT\_DIR** Generate C# source file.

**--java\_out=OUT\_DIR** Generate Java source file.

**--kotlin\_out=OUT\_DIR** Generate Kotlin file.

**--objc\_out=OUT\_DIR** Generate Objective-C header and source.

**--php\_out=OUT\_DIR** Generate PHP source file.

**--pyi\_out=OUT\_DIR** Generate python pyi stub.

**--python\_out=OUT\_DIR** Generate Python source file.

**--ruby\_out=OUT\_DIR** Generate Ruby source file.

**--rust\_out=OUT\_DIR** Generate Rust sources.

**@<filename>** Read options and filenames from file. If a relative file path is specified, the file

will be searched in the working directory.

The --proto\_path option will not affect how this argument file is searched. Content of

the file will be expanded in the position of @<filename> as in the argument list.

Note that shell expansion is not applied to the

content of the file (i.e., you cannot use quotes, wildcards, escapes, commands, etc.).

Each line corresponds to a single argument, even if it contains spaces.

# Syntax

## Field Properties

### Field Names

Every field in a message must have a name.

Notes:

* When setting field names, remember:
* It can sometimes be **difficult, or even impossible, to change field names after** they’ve been used in production.
* Field names cannot contain dashes. For more on coding rules, check [here](https://protobuf.dev/programming-guides/style/).
* Use pluralized names for repeated fields.

### Field Labels

A message field can be one of the following:

* optional: It has one of possible states:
  + The field is set: It contains a value that was explicitly set or parsed from the wire. It will be serialized to the wire.
  + The field is unset: It return the default value. It will not be serialized to the wire.
  + The default value of string is an empty string
  + The default value of bool is false
  + The default value of number is 0
  + The default value of enum is its first value.
  + For more details, check [Default Values](https://protobuf.dev/programming-guides/proto3/#default) session.
* repeated: It can be repeated zero or more times in a well-formed message – think of dynamically-sized arrays. The order of the repeated values will be preserved.
* map: This is a key/value field type.
* If no explicit field label is applied, the default field label, called "implicit field presence" is assumed. (You cannot explicitly set a field to this state.) A well-formed message can have zero or one of this field (but not more than one). You also cannot determine whether a field of this type was parsed from the wire. An implicit presence field will be serialized to the wire unless it is the default value. For more on this subject, see [Field Presence](https://protobuf.dev/programming-guides/field_presence).

Note: The required label from proto1 and prot2 is removed from proto3 as it’s strongly discouraged. To further explain, required is forever. 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.

Example:

message Person {

    required string name = 1;

    optional int32 id = 2 [default = 10];

    repeated string email = 3;

}

### Field Tags (Numbers)

Each field in a message must be defined with an integer number between 1 and 536,870,911 with the following restrictions.

* The given number must be **unique** among all fields for that message.
* Field numbers 19,000 to 19,999 are reserved for the protocol buffers implementation. The proto compiler will complain if you use one of these numbers.
* You cannot use any previously reserved field numbers or any field numbers that have been allocated to extensions.

Notes:

* Field number **should not be changed** once a message is in use because it identifies the field in the message wire format.
* Field numbers **should never be reused**. Never take a field number out of the reserved list for reuse with a new field definition. If you delete a field, you should reserve its field number to prevent someone from accidentally reusing the number. For more detail, check [Deleting Fields](https://protobuf.dev/programming-guides/proto3/#deleting) session and [Reserved Fields](https://protobuf.dev/programming-guides/proto3/#fieldreserved) session.

## Field Data Types

### Primitive Types

Protobuf supports primitive data types, such as integers, booleans, and floats. For the full list, see [Scalar Value Types](https://protobuf.dev/programming-guides/proto3#scalar).

### Complex Types

A field can also be of:

* A message type to nest parts of the definition, such as for repeating sets of data.
* An enum type to specify a set of values to choose from.
* A oneof type to be used when a message has many optional fields and at most one field will be set at the same time.
* A map type to add key-value pairs to your definition.

Tips:

* From proto2, **messages can allow extensions to define fields outside of the message**, itself. For example, the protobuf library’s internal message schema allows extensions for custom, usage-specific options.

#### Message

#### Enum

In the following example we’ve added an enum called Corpus with all the possible values, and a field of type Corpus:

enum Corpus {

  CORPUS\_UNSPECIFIED = 0;

  CORPUS\_UNIVERSAL = 1;

  CORPUS\_WEB = 2;

  CORPUS\_IMAGES = 3;

  CORPUS\_LOCAL = 4;

  CORPUS\_NEWS = 5;

  CORPUS\_PRODUCTS = 6;

  CORPUS\_VIDEO = 7;

}

message SearchRequest {

  string query = 1;

  int32 page\_number = 2;

  int32 results\_per\_page = 3;

  Corpus corpus = 4;

}

Notes:

* Values must be in the range of a 32-bit integer.
* Value of the first element in every enum must be 0.

#### One of

<https://protobuf.dev/programming-guides/proto3/#oneof>

#### Map

<https://protobuf.dev/programming-guides/proto3/#maps>

## Reuse Messages

You can use other messages as field types. For example:

message Result {

  string url = 1;

  string title = 2;

  repeated string snippets = 3;

}

message SearchResponse {

  repeated Result results = 1;

}

## Nested Messages

You can define and **use message inside another message**. For example:

message SearchResponse {

  message Result {

    string url = 1;

    string title = 2;

    repeated string snippets = 3;

  }

  repeated Result results = 1;

}

message Outer {       // Level 0

  message MiddleAA {  // Level 1

    message Inner {   // Level 2

      int64 ival = 1;

      bool  booly = 2;

    }

  }

  message MiddleBB {  // Level 1

    message Inner {   // Level 2

      int32 ival = 1;

      bool  booly = 2;

    }

  }

}

If you want to **reuse a message outside its parent message**, you refer to it as \_Parent\_.\_Type\_:

message SomeOtherMessage {

  SearchResponse.Result result = 1;

}

## Adding Comments

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

/\* SearchRequest represents a search query, with pagination options to

 \* indicate which results to include in the response. \*/

message SearchRequest {

  string query = 1;

  int32 page\_number = 2;   // Which page number do we want?

  int32 results\_per\_page = 3;  // Number of results to return per page.

}

## Importing Definitions

In the earlier example, the Result message is defined in the same file as SearchResponse. But what if it in another .proto file? You can import it.

To import another .proto’s definitions, add an import statement to the top of your file:

import "myproject/other\_protos.proto";

# Third-Party

<https://github.com/protocolbuffers/protobuf/blob/main/docs/third_party.md>

# Protobuf with C++

The following example runs on Windows. The same technique and steps can be applied to other OSs.

## Notes

### Incompatibe Version

A **newer version of protoc might require a newer standard of C++**. For example, protoc [v3.5.0](https://github.com/protocolbuffers/protobuf/releases/tag/v3.5.0) has a strict C++11 requirement, while 3.5.0 doesn’t. Also, because there is **no option to choose C++ standard in protoc**, you should pick up a suitable version of protobuf to use in your project.

## Workflow

### Step 1: Prepare a .proto file

Each .proto file is a protocol format for your program.

The below addressbook.proto has most data types you can think of, including message, enum, string, integer, array.

syntax = "proto2";

package tutorial;

message Person {

  optional string name = 1;

  optional int32 id = 2;

  optional string email = 3;

  enum PhoneType {

    PHONE\_TYPE\_UNSPECIFIED = 0;

    PHONE\_TYPE\_MOBILE = 1;

    PHONE\_TYPE\_HOME = 2;

    PHONE\_TYPE\_WORK = 3;

  }

  message PhoneNumber {

    optional string number = 1;

    optional PhoneType type = 2 [default = PHONE\_TYPE\_HOME];

  }

  repeated PhoneNumber phones = 4;

}

message AddressBook {

  repeated Person people = 1;

}

### Step 2: Compile .proto file to code

Now that you have the addressbook.proto, the next step is to generate classes you’ll need to read and write AddressBook (and hence Person and PhoneNumber) messages.

To do this, run the protobuf compiler protoc on the addressbook.proto:

$ protoc -I=$SRC\_DIR --cpp\_out=$DST\_DIR $SRC\_DIR/addressbook.proto

Where:

- SRC\_DIR: Path to the source directory (where your application’s source code lives)

- DST\_DIR: Path to the destination directory (where you want the generated code to go, often the same as SRC\_DIR)

- Because you want C++ code will be generated, you use the --cpp\_out option.

This generates the following files in your specified destination directory:

* addressbook.pb.h: Header file which declares your generated classes.
* addressbook.pb.cc: Implementation file which contains the definition of your classes.

To make thing organized, let’s put generation command to a Windows batch script:

echo off

:: Path to proto compiler

set PROTOC=protoc-25.1-win64\bin\protoc.exe

:: Path to your available code

set SRC\_DIR=.

:: Path to the generated code (often same as SRC\_DIR)

set DST\_DIR=.

%PROTOC% -I=%SRC\_DIR% --cpp\_out=%DST\_DIR% %SRC\_DIR%/addressbook.proto

### Step 3: Check your generated code

You don’t need to do this step. But because this is your first time, let’s take a look at the generated code structure.

// optional string 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();

// optional int32 id

inline bool has\_id() const;

inline void clear\_id();

inline int32\_t id() const;

inline void set\_id(int32\_t value);

// optional string 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();

// repeated PhoneNumber phones

inline int phones\_size() const;

inline void clear\_phones();

inline const ::google::protobuf::RepeatedPtrField< ::tutorial::Person\_PhoneNumber >& phones() const;

inline ::google::protobuf::RepeatedPtrField< ::tutorial::Person\_PhoneNumber >\* mutable\_phones();

inline const ::tutorial::Person\_PhoneNumber& phones(int index) const;

inline ::tutorial::Person\_PhoneNumber\* mutable\_phones(int index);

inline ::tutorial::Person\_PhoneNumber\* add\_phones();

Firstly, accessor methods:

* Getter methods name(), id(), email(), phones(): Have exactly name as the field in lowercase, which gets value for the field.
* Setter methods set\_name(), set\_id(), set\_email(): Start with set\_ prefix, which sets value for the field. Note that repeated field phones has two mutable\_ methods (either with index param or not) but not a set\_ method.
* Check methods has\_name(), has\_id(), has\_email(): For singular (required or optional) field which return true if that field has been set.
* Clear method clear\_name(), clear\_id(), clear\_email(), clear\_phones(): Start with clear\_ prefix, which un-set the field back to its empty state.
* Mutable getter methods mutable\_name(), mutable\_email(), mutable\_phones(): Start with mutable\_ prefix, which returns a direct pointer to the string. Note that you can call mutable\_name() or mutable\_email() even if name or email is not already set; it will be initialized to an empty string automatically.
* Size method phones\_size(): For repeated fields, which return the number of items in the repeated field.
* Add method add\_phones(): For repeated fields, which add another item to the repeated field.

Secondly, standard message methods:

* bool IsInitialized() const: Checks if all the required fields have been set.
* std::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.

Thirdly, serialization and deserialization (or parsing) methods:

* bool SerializeToString(std::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 std::string class as a convenient container.
* bool ParseFromString(const std::string& data): Deserializes (or parses) a message from the given string.
* bool SerializeToOstream(std::ostream\* output) const: Serializes the message and stores the bytes in the given C++ ostream.
* bool ParseFromIstream(std::istream\* input): Deserializes (or parses) a message from the given C++ istream.

These are just a couple of the options provided for parsing and serialization. See the [Message API reference](https://protobuf.dev/reference/cpp/api-docs/google.protobuf.message#Message) for a complete list.

### Step 4: Write a message

Now let’s try using your generated protobuf classes. We will write personal details to the address book file.

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.

#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\_phones();

        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::PHONE\_TYPE\_MOBILE);

        } else if (type == "home") {

            phone\_number->set\_type(tutorial::Person::PHONE\_TYPE\_HOME);

        } else if (type == "work") {

            phone\_number->set\_type(tutorial::Person::PHONE\_TYPE\_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\_people());

    {

        // Write the new address book back to file.

        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;

}

### Step 5: Read a message

Of course, an address book wouldn’t be much use if you couldn’t get any information out of it.

Here is a program which reads the address book file 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.people\_size(); i++) {

        const tutorial::Person& person = address\_book.people(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.phones\_size(); j++) {

            const tutorial::Person::PhoneNumber& phone\_number = person.phones(j);

            switch (phone\_number.type()) {

                case tutorial::Person::PHONE\_TYPE\_MOBILE:

                    cout << "Mobile phone #: ";

                    break;

                case tutorial::Person::PHONE\_TYPE\_HOME:

                    cout << "Home phone #: ";

                    break;

                case tutorial::Person::PHONE\_TYPE\_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;

}

### Step 6: Compile your project code

Now you have all the code you need, it’s time to build it. You can use g++, Clang, MSBuild or any compiler.

Just make sure:

* Add static libraries of protobuf and POSIX thread with option -lprotobuf and -pthread to the build command.
* Choose the correct C++ standard with option --std=c++xx.
* Get protobuf header files. You can get here: <https://github.com/protocolbuffers/protobuf/tree/main/src/google/protobuf>. Note that this is the main branch, and you should select a specific release branch depending on the protoc version.

### Step 7 [optional]: Extending a Protocol Buffer

Sooner or later after you release the code, you might 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 field 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 field numbers (that is, field numbers that were never used in this protocol buffer, not even by deleted fields).

There are [some exceptions](https://protobuf.dev/programming-guides/proto#updating) 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 field number. 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.

**Refs**: <https://protobuf.dev/getting-started/cpptutorial/>