C++ Arena Allocation Guide

Arena allocation is a C++-only feature that helps you optimize your memory usage and improve performance when working with protocol buffers. This page describes exactly what C++ code the protocol buffer compiler generates in addition to the code described in the C++ Generated Code Guide (https://developers.google.com/protocol-buffers/docs/reference/cpp-generated) when arena allocation is enabled. It assumes that you are familiar with the material in the language guide (https://developers.google.com/protocol-buffers/docs/proto) and the C++ Generated Code Guide (https://developers.google.com/protocol-buffers/docs/reference/cpp-generated).

Why use arena allocation?

Memory allocation and deallocation constitutes a significant fraction of CPU time spent in protocol buffers code. By default, protocol buffers performs heap allocations for each message object, each of its subobjects, and several field types, such as strings. These allocations occur in bulk when parsing a message and when building new messages in memory, and associated deallocations happen when messages and their subobject trees are freed.

Arena-based allocation has been designed to reduce this performance cost. With arena allocation, new objects are allocated out of a large piece of preallocated memory called the arena. Objects can all be freed at once by discarding the entire arena, ideally without running destructors of any contained object (though an arena can still maintain a "destructor list" when required). This makes object allocation faster by reducing it to a simple pointer increment, and makes deallocation almost free. Arena allocation also provides greater cache efficiency: when messages are parsed, they are more likely to be allocated in continuous memory, which makes traversing messages more likely to hit hot cache lines.

To get these benefits you'll need to be aware of object lifetimes and find a suitable granularity at which to use arenas (for servers, this is often per-request). You can find out more about how to get the most from arena allocation in Usage patterns and best practices (#usage).

Getting started

You enable arena allocation on a per-`.proto` basis. To do this, add the following `option` to your `.proto` file:

```
option cc_enable_arenas = true;
```

This tells the protocol buffer compiler to generate the additional code you need to use arena allocation for the messages in your file, as used in the following example.

```
#include <google/protobuf/arena.h>
{
  google::protobuf::Arena arena;
  MyMessage* message = google::protobuf::Arena::CreateMessage<MyMessage>(&are
  // ...
}
```

The message object created by `CreateMessage()` will exist for as long as `arena` exists, and you should not `delete` the returned message pointer. All of the message object's internal storage (with a few exceptions[1] (#fn1)) and submessages (for example, submessages in a repeated field within `MyMessage`) are allocated on the arena as well.

For the most part, the rest of your code will be the same as if you weren't using arena allocation.

We'll look at the arena API in more detail in the following sections, and you can see a more extensive example (#example) at the end of the document.

1. Currently, string fields store their data on the heap even when the containing message is on the arena. Unknown fields are also heap-allocated.↩ (#ref1)

Arena class API

You create message objects on the arena using the `google::protobuf::Arena` (https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.arena.html)

class. This class implements the following public methods.

Constructors

- `Arena()`: Creates a new arena with default parameters, tuned for average use cases.
- `Arena(const ArenaOptions& options)`: Creates a new arena that uses the specified allocation options. The options available in `ArenaOptions` include the ability to use an initial block of user-provided memory for allocations before resorting to the system allocator, control over the initial and maximum request sizes for blocks of memory, and allowing you to pass in custom block allocation and deallocation function pointers to build freelists and others on top of the blocks.

Allocation methods

- `template<typename T> static T* CreateMessage(Arena* arena)`: Creates a new protocol buffer object of message type `T` on the arena. This message type must be defined in a `.proto` file with `option cc_enable_arenas = true;` otherwise, a compile error will result.

  If `arena` is not NULL, the returned message object is allocated on the arena, its internal storage and submessages (if any) will be allocated on the same arena, and its lifetime is the same as that of the arena. The object must not be deleted/freed manually: the arena owns the message object for lifetime purposes.

  If `arena` is NULL, the returned message object is allocated on the heap, and the caller owns the object upon return.
- `template<typename T> static T* Create(Arena* arena, args...)`: Similar to `CreateMessage()` but lets you create an object of any class on the arena, not just protocol buffer message types with `option cc_enable_arenas = true;`: you can use a protocol buffer message class from a file that doesn't have arena support enabled, or an arbitrary C++ class. For example, let's say you have this C++ class:

  ```
  class MyCustomClass {
      MyCustomClass(int arg1, int arg2);
      // ...
  };
  ```

  ...you can create an instance of it on the arena like this:

  ```
  void func() {
      // ...
  ```

```
    google::protobuf::Arena arena;
    MyCustomClass* c = google::protobuf::Arena::Create<MyCustomClass>(&ar
    // ...
}
```

- `template<typename T> static T* CreateArray(Arena* arena, size_t n)`: If `arena` is not NULL, this method allocates raw storage for `n` elements of type `T` and returns it. The arena owns the returned memory and will free it on its own destruction. If `arena` is NULL, this method allocates storage on the heap and the caller receives ownership.

  `T` must have a trivial constructor: constructors are not called when the array is created on the arena.

"Owned list" methods

The following methods let you specify that particular objects or destructors are "owned" by the arena, ensuring that they are deleted or called when the arena itself is deleted

- `template<typename T> void Own(T* object)`: Adds `object` to the arena's list of owned heap objects. When the arena is destroyed, it traverses this list and frees each object using operator delete, i.e., the system memory allocator. This method is useful in cases when an object's lifetime should be tied to the arena but, for whatever reason, the object itself cannot be or was not already allocated on the arena.
- `template<typename T> void OwnDestructor(T* object)`: Adds the destructor of `object` to the arena's list of destructors to call. When the arena is destroyed, it traverses this list and calls each destructor in turn. It does not attempt to free the underlying memory of object. This method is useful when an object is embedded in arena-allocated storage but its destructor will not otherwise be called, for example because its containing class is a protobuf message whose destructor won't be called, or because it was manually constructed in a block allocated by `AllocateArray()`.

Other methods

- `uint64 SpaceUsed() const`: Returns the total size of the arena, which is the sum of the sizes of the underlying blocks. This method is thread-safe; however, if there are concurrent allocations from multiple threads this method's return value may not include the sizes of those new blocks.
- `uint64 Reset()`: Destroys the arena's storage, first calling all registered destructors and freeing all registered heap objects and then discarding all arena blocks. This

teardown procedure is equivalent to that which occurs when the arena's destructor runs, except the arena is reusable for new allocations after this method returns. Returns the total size used by the arena: this information is useful for tuning performance.

- `template<typename T> Arena* GetArena()`: Returns a pointer to this arena. Not directly very useful but allows `Arena` to be used in template instantiations that expect `GetArena()` methods to be present.

Thread safety

`google::protobuf::Arena`'s allocation methods are thread-safe, and the underlying implementation goes to some length to make multithreaded allocation fast. The `Reset()` method is *not* thread-safe: the thread performing the arena reset must synchronize with all threads performing allocations or using objects allocated from that arena first.

Generated message class

The following message class members are changed or added when you enable arena allocation.

Message class methods

- `Message(Message&& other)`: If the source message is not on arena, the move constructor efficiently *moves* all fields from one message to another without making copies or heap allocations (the time complexity of this operation is `O(number-of-declared-fields)`). However, if the source message is on arena, it performs a *deep copy* of the underlying data. In both cases the source message is left in a valid but unspecified state.
- `Message& operator=(Message&& other)`: If both messages are not on arena or are on the *same* arena, the move-assignment operator efficiently *moves* all fields from one message to another without making copies or heap allocations (the time complexity of this operation is `O(number-of-declared-fields)`). However, if only one message is on arena, or the messages are on different arenas, it performs a *deep copy* of the underlying data. In both cases the source message is left in a valid but unspecified state.
- `void Swap(Message* other)`: If both messages to be swapped are not on arenas or are on the *same* arena, `Swap()` (https://developers.google.com/protocol-buffers/docs/reference/cpp-generated.html#message)

behaves as it does without having arena allocation enabled: it efficiently swaps the message objects' contents, usually via cheap pointer swaps and avoiding copies at all costs. However, if only one message is on an arena, or the messages are on different arenas, `Swap()` performs *deep copies* of the underlying data. This new behavior is necessary because otherwise the swapped sub-objects could have differing lifetimes, leading potentially to use-after-free bugs.

- `Message* New(Arena* arena)`: An alternate override for the standard `New()` method. It allows a new message object of this type to be created on the given arena. Its semantics are identical to `Arena::CreateMessage<T>(arena)` if the concrete message type on which it is called is generated with arena allocation enabled. If the message type is not generated with arena allocation enabled, then it is equivalent to an ordinary allocation followed by `arena->Own(message)` if `arena` is not NULL.
- `Arena* GetArena()`: Returns the arena on which this message object was allocated, if any.
- `void UnsafeArenaSwap(Message* other)`: Identical to `Swap()`, except it assumes both objects are on the same arena (or not on arenas at all) and always uses the efficient pointer-swapping implementation of this operation. Using this method can improve performance as, unlike `Swap()`, it doesn't need to check which messages live on which arena before performing the swap. As the `Unsafe` prefix suggests, you should only use this method if you are sure the messages you want to swap aren't on different arenas; otherwise this method could have unpredictable results.

Embedded message fields

When you allocate a message object on an arena, its embedded message field objects (submessages) are automatically owned by the arena as well. How these message objects are allocated depends on where they are defined:

- If the message type is also defined in a `.proto` file with arena allocation enabled, the object is allocated on the arena directly.
- If the message type is from another `.proto` without arena allocation enabled, the object is heap-allocated but is "owned" by the parent message's arena. This means that when the arena is destroyed, the object will be freed along with the objects on the arena itself.

For either of these field definitions:

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

The following methods are added or have some special behavior when arena allocation is enabled. Otherwise, accessor methods just use the default behavior (https://developers.google.com/protocol-buffers/docs/reference/cpp-generated.html#embeddedmessage).

- `Bar* mutable_foo()`: Returns a mutable pointer to the submessage instance. If the parent object is on an arena then the returned object will be as well.
- `void set_allocated_foo(Bar* bar)`: Takes a new object and adopts it as the new value for the field. Arena support adds additional copying semantics to maintain proper ownership when objects cross arena/arena or arena/heap boundaries:
  - If the parent object is on the heap and `bar` is on the heap, or if the parent and message are on the same arena, this method's behavior is unchanged.
  - If the parent is on an arena and `bar` is on the heap, the parent message adds `bar` to its arena's ownership list with `arena->Own()`.
  - If the parent is on an arena and `bar` is on a different arena, this method makes a copy of message and takes the copy as the new field value.
- `Bar* release_foo()`: Returns the existing submessage instance of the field, if set, or a NULL pointer if not set, releasing ownership of this instance to the caller and clearing the parent message's field. Arena support adds additional copying semantics to maintain the contract that the returned object is always *heap-allocated*:
  - If the parent message is on an arena, this method will make a copy of the submessage on the heap, clear the field value, and return the copy.
  - If the parent message is on the heap, the method behavior is unchanged.
- `void unsafe_arena_set_allocated_foo(Bar* bar)`: Identical to `set_allocated_foo`, but assumes both parent and submessage are on the same arena. Using this version of the method can improve performance as it doesn't need to check whether the messages are on a particular arena or the heap. This method should only be used when the parent message is on the arena and the submessage is on the same arena, or an arena with equivalent lifetime.
- `Bar* unsafe_arena_release_foo()`: Similar to `release_foo()`, but assumes the parent message is on the arena, and returns an *arena-allocated* object that should not be deleted directly. This method should only be used when the parent message is on the arena.

String fields

Currently, string fields store their data on the heap even when their parent message is on the arena. Because of this, string accessor methods largely use the default behavior (https://developers.google.com/protocol-buffers/docs/reference/cpp-generated.html#string) even when arena allocation is enabled, with a couple of exceptions.

For any of these field definitions:

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

The following methods are added or have some special/notable behavior when arena allocation is enabled.

- **`void set_allocated_foo(string* value)`** and **`void unsafe_arena_set_allocated_foo(string* value)`**: Similar to the embedded message field methods (#arenaembeddedmessage) described above where the child message is on the heap. As well as setting the field value to **`value`**, calling **`set_allocated_foo()`** when the parent is on the arena adds **`value`** to the parent message's arena's ownership list with **`arena->Own()`**. The **`unsafe_arena`** version just sets the value without adding it to the ownership list.
- **`string* release_foo()`** and **`string* unsafe_arena_release_foo()`**: Again, these are similar to the embedded message field methods (#arenaembeddedmessage) described above where the child message is on the heap. Calling **`release_foo()`** when the parent is on the arena makes a copy of the string value, clears the field, and returns a pointer to the copy. Calling the **`unsafe_arena`** version just releases the field and returns a pointer to the still **`Own()`**-ed string.

Repeated fields

Repeated fields allocate their internal array storage on the arena when the containing message is arena-allocated, and also allocate their elements on the arena when these elements are separate objects retained by pointer (messages or strings). At the message-class level, generated methods for repeated fields do not change. However, the **`RepeatedField`** and **`RepeatedPtrField`** objects that are returned by accessors do have new methods and modified semantics when arena support is enabled.

Repeated numeric fields

`RepeatedField` objects that contain primitive types (#repeatednumeric) have the following new/changed methods when arena allocation is enabled:

- `void UnsafeArenaSwap(RepeatedField* other)`: Performs a swap of `RepeatedField` contents without validating that this repeated field and other are on the same arena. If they are not, the two repeated field objects must be on arenas with equivalent lifetimes. The case where one is on an arena and one is on the heap is checked and disallowed.
- `void Swap(RepeatedField* other)`: Checks each repeated field object's arena, and if one is on an arena while one is on the heap or if both are on arenas but on different ones, the underlying arrays are copied before the swap occurs. This means that after the swap, each repeated field object holds an array on its own arena or heap, as appropriate.

Repeated embedded message fields

`RepeatedPtrField` objects that contain messages (#repeatedmessage) have the following new/changed methods when arena allocation is enabled.

- `void UnsafeArenaSwap(RepeatedPtrField* other)`: Performs a swap of `RepeatedPtrField` contents without validating that this repeated field and other have the same arena pointer. If they do not, the two repeated field objects must have arena pointers with equivalent lifetimes. The case where one has a non-NULL arena pointer and one has a NULL arena pointer is checked and disallowed.
- `void Swap(RepeatedPtrField* other)`: Checks each repeated field object's arena pointer, and if one is non-NULL (contents on arena) while one is NULL (contents on heap) or if both are non-NULL but have different values, the underlying arrays and their pointed-to objects are copied before the swap occurs. This means that after the swap, each repeated field object holds an array on its own arena or on the heap, as appropriate.
- `void AddAllocated(SubMessageType* value)`: Checks that the provided message object is on the same arena as the repeated field's arena pointer. If it is on the same arena, then the object pointer is added directly to the underlying array. Otherwise, a copy is made, the original is freed if it was heap-allocated, and the copy is placed on the array. This maintains the invariant that all objects pointed to by a repeated field are in the same ownership domain (heap or specific arena) as indicated by the repeated field's arena pointer.
- `SubMessageType* ReleaseLast()`: Returns a heap-allocated message equivalent to the last message in the repeated field, removing it from the repeated field. If the

repeated field itself has a NULL arena pointer (and thus, all of its pointed-to messages are heap-allocated), then this method simply returns a pointer to the original object. Otherwise, if the repeated field has a non-NULL arena pointer, this method makes a copy that is heap-allocated and returns that copy. In both cases, the caller receives ownership of a heap-allocated object and is responsible for deleting the object.

- `void UnsafeArenaAddAllocated(SubMessageType* value)`: Like `AddAllocated()`, but does not perform heap/arena checks or any message copies. It adds the provided pointer directly to the internal array of pointers for this repeated field. The caller must guarantee that the provided object is heap-allocated if the repeated field has a NULL arena pointer, or arena-allocated (on the same arena or on one with identical lifetime) if the repeated field has a non-NULL arena pointer.
- `SubMessageType* UnsafeArenaReleaseLast()`: Like `ReleaseLast()` but performs no copies, even if the repeated field has a non-NULL arena pointer. Instead, it directly returns the pointer to the object as it was in the repeated field. The returned object is thus on the heap if the repeated field's arena pointer is NULL and on an arena if the repeated field has a non-NULL arena pointer. The caller receives ownership if the object was heap-allocated. If the object was arena-allocated, the caller must not attempt to delete the returned object.
- `void ExtractSubrange(int start, int num, SubMessageType** elements)`: Removes `num` elements from the repeated field, starting from index `start`, and returns them in `elements` if it is not NULL. If the repeated field is on an arena, and elements are being returned, the elements are copied to the heap first. In both cases (arena or no arena), the caller owns the returned objects on the heap.
- `void UnsafeArenaExtractSubrange(int start, int num, SubMessageType** elements)`: Removes `num` elements from the repeated field, starting from index `start`, and returns them in `elements` if it is not NULL. Unlike `ExtractSubrange()`, this method never copies the extracted elements.

Repeated string fields

Repeated fields of strings have the same new methods and modified semantics as repeated fields of messages, because they also maintain their underlying objects (namely, strings) by pointer reference.

Usage patterns and best practices

When using arena-allocated messages, several usage patterns can result in unintended copies or other negative performance effects. You should be aware of the following

common patterns that may need to be altered when adapting code for arenas. (Note that we have taken care in the API design to ensure that correct behavior still occurs — but higher-performance solutions may require some reworking.)

Unintended copies

Several methods that never create object copies when not using arena allocation may end up doing so when arena support is enabled. These unwanted copies can be avoided if you make sure that your objects are allocated appropriately and/or use provided arena-specific method versions, as described in more detail below.

Set Allocated/Add Allocated/Release

By default, the `release_`*`field`*`()` and `set_allocated_`*`field`*`()` methods (for singular message and string fields), and the `ReleaseLast()` and `AddAllocated()`() methods (for repeated message and string fields) all allow user code to directly attach and detach sub-objects, passing ownership of pointers without copying any data.

However, when the parent message is on an arena, these methods now sometimes need to copy the passed in or returned object to maintain compatibility with existing ownership contracts. More specifically, methods that take ownership (`set_allocated_`*`field`*`()` and `AddAllocated()`) may copy data if the parent is on an arena and the new subobject is not, or vice versa, or they are on different arenas. Methods that release ownership (`release_`*`field`*`()` and `ReleaseLast()`) may copy data if the parent is on the arena, because the returned object must be on the heap, by contract.

To avoid such copies, we have added corresponding "unsafe arena" versions of these methods where copies are **never performed**: `unsafe_arena_set_allocated_`*`field`*`()`, `unsafe_arena_release_`*`field`*`()`, `UnsafeArenaAddAllocated()`, and `UnsafeArenaRelease()` for singular and repeated fields, respectively. These methods should be used only when you know they are safe to do so and that the parent and child objects are allocated as expected; otherwise, for example, you could end up with parent and child objects with differing lifetimes, leading potentially to use-after-free bugs.

Here's an example of how you can avoid unnecessary copies with these methods. Let's say you have created the following messages on an arena.

```
Arena* arena = new google::protobuf::Arena();
MyFeatureMessage* arena_message_1 =
  google::protobuf::Arena::CreateMessage<MyFeatureMessage>(arena);
arena_message_1->mutable_nested_message()->set_feature_id(11);
```

```
MyFeatureMessage* arena_message_2 =
  google::protobuf::Arena::CreateMessage<MyFeatureMessage>(arena);
```

The following code makes inefficient usage of the `release_...()` API:

```
arena_message_2->set_allocated_nested_message(arena_message_1->release_nested

arena_message_1->release_message(); // returns a copy of the underlying neste
```

Using the "unsafe arena" version instead avoids the copy:

```
arena_message_2->set_allocated_nested_message(
   arena_message_1->unsafe_arena_release_nested_message());
```

You can find out more about these methods in the Embedded message fields (#arenaembeddedmessage) and String fields (#arenastring) sections above.

Swap

When two messages' contents are swapped with `Swap()`, the underlying subobjects may be copied if the two messages live on different arenas, or if one is on the arena and the other is on the heap. If you want to avoid this copy and either (i) know that the two messages are on the same arena or different arenas but the arenas have equivalent lifetimes, or (ii) know that the two messages are on the heap, you can use a new method, `UnsafeArenaSwap()`. This method both avoids the overhead of performing the arena check and avoids the copy if one would have occurred.

For example, the following code incurs a copy in the `Swap()` call:

```
MyFeatureMessage* message_1 =
  google::protobuf::Arena::CreateMessage<MyFeatureMessage>(arena);message_1->r

MyFeatureMessage* message_2 = new MyFeatureMessage;
message_2->mutable_nested_message()->set_feature_id(22);

message_1->Swap(message_2); // Inefficient swap!
```

To avoid the copy in this code, you allocate `message_2` on the same arena as `message_1`:

```
MyFeatureMessage* message_2 =
   google::protobuf::Arena::CreateMessage<MyFeatureMessage>(arena);
```

Embedded message fields and arena-enable options

Each `.proto` file has its own "feature switch" for arena support. If `cc_enable_arenas` is not set in a given `.proto` file, the types defined in that file will not be stored on the arena, even if some other type includes a submessage that has a type defined in that file. In other words, `cc_enable_arenas` is not transitive. Rather, submessages of an arena-capable message that do not themselves have arena support will always be stored on the heap, and will be added to the parent message's arena's `Own()` list so that their lifetimes are tied to the arena's lifetime.

The reason for this restriction is that adding arena support adds some overhead in the case that arenas are not used because of the extra generated code, so we choose (for now) not to enable arena support globally. Furthermore, for type- and API-compatibility reasons, we can have only one C++ generated class per proto message type, so we cannot generate with-arena and without-arena versions of a class. In the future, after further optimization, we may be able to lift this restriction and globally enable arena support. For now, though, it should be enabled for as many submessage types as possible to improve performance.

Granularity

We have found in most application server use cases that an "arena-per-request" model works well. You may be tempted to divide arena use further, either to reduce heap overhead (by destroying smaller arenas more often) or to reduce perceived thread-contention issues. However, the use of more fine-grained arenas may lead to unintended message copying, as we describe above. We have also spent effort to optimize the `Arena` implementation for the multithreaded use-case, so a single arena should be appropriate for use throughout a request lifetime even if multiple threads process that request.

Example

Here's a simple complete example demonstrating some of the features of the arena allocation API.

```
// my_feature.proto

syntax = "proto2";
import "nested_message.proto";

package feature_package;

option cc_enable_arenas = true;
```

```
// NEXT Tag to use: 4
message MyFeatureMessage {
  optional string feature_name = 1;
  repeated int32 feature_data = 2;
  optional NestedMessage nested_message = 3;
};


// nested_message.protofset

syntax = "proto2";

package feature_package;

// add cc_enable_arenas on each submessage for
// the best performance when using arenas.
option cc_enable_arenas = true;

// NEXT Tag to use: 2
message NestedMessage {
  optional int32 feature_id = 1;
};
```

Message construction and deallocation:

```
#include <google/protobuf/arena.h>

Arena arena;

MyFeatureMessage* arena_message =
   google::protobuf::Arena::CreateMessage<MyFeatureMessage>(&arena);

arena_message->set_feature_name("Proto2 Arena");
arena_message->mutable_feature_data()->Add(2);
arena_message->mutable_feature_data()->Add(4);
arena_message->mutable_nested_message()->set_feature_id(247);
```

上次更新日期：八月 22, 2017

Downloads
Protocol buffers downloads and instructions

GitHub
The latest protocol buffers code and releases