Users' Guide to the Multi-Process Garbage Collector (MPGC)

The Managed Data Structures Project
Analytics Lab, Hewlett Packard Labs

Monday, September 19, 2016

Contents

tl;dr

- Heap files: heaps/gc_heap, heaps/managed_heap
  - Create by truncate -s*size*
  - managed_heap 10% of gc_heap
- Create object: make_gc<*T*>(…)
- Crate array: make_gc_array<*T*>(…)
- Three types of data: GC-allocated, GC-friendly, GC-unfriendly
- Garbage collected arrays: *gc_array<T>*.
- Pointers within MPGC heap, on stack: *gc_ptr<T>*
- Pointers everyplace else: *external_gc_ptr<T>*
- Pointers to data members: *gc_sub_ptr<T>*
- Both work with *std::atomic*
- Sharing data between processes: persistent_roots()
- Turning GC-friendly data into GC-allocated: *gc_wrapped<T>*
- Useful provided classes:
  - *gc_vector<T>*
  - *gc_string*
- GC-allocated data:
  - Derive from *gc_allocated*
  - No virtual functions
  - Only GC-friendly members
  - Define descriptor() function
- GC-friendly data:
  - No virtual functions
  - Only GC-friendly members
  - Define descriptor() function unless already known.
- Fake virtual functions with *gc_allocated_with_virtuals*

Introduction

The Multi-Process Garbage Collector (MPGC) provides a managed, shared, non-volatile heap for C++ data. It was designed to support any C++ program[1] (or set of programs) that want to share data between processes (either running at the same time

[1] Or, via JNI, any Java program that wants to share off-Java-heap data.

or at different times) and which don't want to have to worry about figuring out when it is safe to free up shared memory or about corrupting the heap if a process crashes.

A Persistent Heap

MPGC was designed for use on The Machine, being developed by Hewlett Packard Enterprise, and it assumes that the memory it manages is non-volatile. In the current implementation, this is simulated by mapping files into the address space of each process. For many applications, this file-backed-DRAM implementation will be sufficient on its own.[2]

Since objects are created and manipulated directly in persistent memory, MPGC removes the need to read data from storage, create objects in DRAM, modify the objects, and then write data back out to storage. After a program terminates, if another program (or the same program run again) needs to use the same data, it's already there, in the format designed by the C++ programmer, ready to use.

A Shared Heap

Any number of processes can share an MPGC heap, and each of these processes can have any number of application threads. These processes need not know about one another, and they can start and exit at any time without needing to do anything (overt) to synchronize with one another. The heap itself doesn't provide any concurrency control, but it does support the *std::atomic<T>* class, including atomic GC-aware pointers.

An Automatically Memory-Managed Heap

Perhaps the most important feature of the MPGC heap is that it is fully garbage collected. Programs need not (in fact, they cannot) directly delete the objects on the heap. Rather, MPGC takes care of determining when to delete objects on the heap. It guarantees that an object will never be destroyed if there is a possibility that any process—currently running or that could run in the future—could ever access it. So the programmer need not worry about dangling pointers. And it guarantees that any object that isn't still reachable by any process will get destroyed in a timely manner. So the programmer need not worry about leaking memory. Unlike with reference-counting solutions like *std::shared_ptr<T>*, circular references between objects pose no problem, so the programmer need not worry about "breaking cycles" in order to allow objects to be destroyed.

A Fault-Tolerant Heap

The processes that share an MPGC heap cannot damage it by crashing. As long as a program follows the rules laid out in this document and doesn't go crazy writing data to random locations in the heap, there is little it can do to harm the integrity of the heap itself.[3]

---

[2] It is recommended for performance reasons that there be a sufficient amount of available DRAM to contain the entire heap backing file.

[3] The current implementation does, however, depend on processor caches being flushed, so unpexected machine failures (e.g., due to sudden loss of power) may cause loss of integrity.

An Efficient Heap

MPGC is an *on-the-fly* garbage collector. This means that it never decides that it needs to "stop the world" in order to determine which objects on the heap are still alive and which can be collected. Periodically, it will interrupt each application thread just long enough to walk its stack, but the pause will never be proportional to either the heap size or the number of live objects on the heap. The actual garbage collection activity takes place in parallel GC threads within each process.

Using MPGC

In this section, we'll talk about what you need to do to use MPGC. How to compile, how to link, and how to run.

Building and Installing MPGC

The MPGC repository contains a build directory that contains subdirectories for different build configurations (at the time of writing, intel-debug, intel-opt, arm-debug, and arm-opt). To build the library, cd to one of these configuration directories and type "make". To alter the configuration or to create your own, merely edit the Makefile. See build/BUILDING.md for details of all make targets and configuration variables that can be set.

To install the header files, libraries, and executable tools, use "make install" (or one of its more specialized sibling targets). By default, this copies files to an install directory that is a sibling of the MPGC repository directory.

Compiling Against MPGC

MPGC can be compiled with GCC version 4.9.2 or higher. It has been successfully compiled using GCC version 6.1. It is known to build on both Intel and ARM machines at optimization levels up to -O3. It has only been tested on Linux.

The compilation flags must include -pthread and -mcx16 (on Intel machines),

The include path must include the MPGC install dir's include directory.[4] Your source code should

```
#include "mpgc/gc.h"
```

Files must be included using the mpgc subdir. gc.h gets you most of what you need. Classes like *gc_vector* and *gc_string* have their own include files.

Linking Against MPGC

The library must be linked against both the mpgc and ruts libraries. These can be built as static or dynamic libraries and can be found in the lib directory of the install dir.[5] The program should also be linked against the standard C++ library and the pthreads library.

---

[4] If compiling against non-installed headers, it must include both include and ruts/include.

[5] If linking with non-installed libraries, they are in the lib and ruts/lib directories.

Building Using the tools directory

The easiest way to build your code is to put it in a subdirectory of the tools dir. As detailed in BUILDING.md, each such subdirectory will be built to an executable with the correct include path, library path, and configuration-specific build options, which can be overridden in the Makefile.

Building the MPGC heap

To actually run the program, you need to create the backing files for the heap. This only needs to be done when you want a clean heap.

There are two heap files to create. By default, the files are heaps/gc_heap and heaps/managed_heap in the program's current working directory. The directory the heap files are looked for in can be changed by setting the $MPGC_HEAPS_DIR environment variable, and the individual files can be pointed to by $MPGC_GC_HEAP and $MPGC_CONTROL_HEAP. Note that the last two point to the files directly, not to the file names within $MPGC_HEAPS_DIR.[6]

To create the heap files, run createheap in the bin directory of the install dir.[7] This takes an argument specifying the desired size of the GC heap file, as a number with an optional suffix, e.g.,

```
createheap 10GB
createheap 1.5T
createheap 800M
createheap 10
```

If no suffix is given, GB is assumed. The size of the control heap will be automatically determined. To set it explicitly, use the -s (--ctrl-size) option. To set the names of the files, use -f (--heap-path) and -c (--ctrl-path). Needed directories will be created automatically.

Running the Program

Once it's linked, the executable should run as normal. No background processes are necessary.

If the program crashes, it can be somewhat difficult to debug, as GDB can't find the mmapped MPGC heap from the core dump. To make this easier, if the $SUSPEND_ON_ABORT environment variable is set to anything that's reasonably a "true" value—true, yes, on, +, 1—then a crash will instead cause the process to suspend. You can then point GDB at it with

```
gdb --pid process-PID
```

---

[6] In the future, there will be only a single heap file.

[7] If using non-installed binaries, it is in build/<*config*>/tools.

Getting Information from the Running Program

If $GC_TRACE_CYCLES is set to a true value, MPGC will periodically (once per GC cycle) print information about the state of the GC heap.

More specific information can be obtained by calling memory_stats(),which returns a reference to an object that includes

- bytes_in_heap(): The number of bytes in the MPGC heap.
- bytes_in_use(): The number of bytes that survived the last collection cycle.
- bytes_free(): The number of bytes available after the last collection cycle.
- n_objects(): The number of objects traced in the last collection cycle.
- n_processes(): The number of processes using the MPGC heap.
- gc_cycle_number(): Incremented after collection every cycle.

Most of the numbers are currently static and refer to what was found in the last cycle, so they will be a bit out of date. The number of processes may include processes which terminated during or after the last cycle and have not yet been cleaned up.

Using a Shared Persistent Heap

Because the MPGC heap is both shared and persistent, it's important to remember that there are certain classes of mistake that will result in not only your process crashing, but actually corrupting the heap for other processes, and that this corruption won't go away simply by rebooting. We have made it as easy as we can to avoid these mistakes, but there are still a few rules to keep in mind.

The first is important enough that there's a whole section devoted to it below: **don't hide pointers**. The MPGC heap is a garbage collected heap, and if the garbage collector can't realize that you have a pointer to an object, it may decide that the object is garbage and allow its memory to be reallocated to another object (in your process or another one). When you attempt to use the hidden pointer, the results could be disastrous. If you use the provided smart pointer classes in the manner described below, this shouldn't be a problem.

The second rule is **don't be psychotic**. There's nothing stopping you from simply going crazy and writing into the MPGC heap memory. Don't. You won't like the results and neither will the processes your sharing the heap with.

The third can be a bit trickier. You have to remember that **old versions of data may stick around**. If you create a *Book* class, whose instances have an author and title, and you later change your class to add a page count, you need to remember that because it's a persistent heap, there may be instances of *Book* that were created under the old rule and don't have the page count field. We will likely provide some support for such class evolution in the future, but as of now, you're on your own to recognize this situation.

Finally, you need to remember that the MPGC heap only stores data and **each process is responsible for supplying its own methods**. This means that two different programs may have different sets of methods associated with the same data type. This is, in our opinion, a good thing, but it does mean that different programs may have different

assumptions about invariants on a class. It also means that when it comes to the pseudo-virtual member functions described in a later section, a program may wind up calling a method on an instance of a subclass it doesn't know about, and there's no way to know what the correct behavior should be.

Programming with MPGC

Programming with MPGC largely consists of three activities:

1. writing classes whose instances can go on the MPGC heap,
2. creating objects on the MPGC heap, and
3. working with pointers to objects on the MPGC heap.

The first activity will be described in detail in the next chapter.

*The **mpgc** Namespace*

All classes and functions provided by MPGC are in the **mpgc** namespace. To access them without having to fully-qualify all uses, simply say

```
using namespace mpgc;
```

Examples in this document will assume that and will not use fully-qualified names. For example, the pointer type will be written as gc_ptr<T>, not mpgc::gc_ptr<T>.

Creating Objects on the MPGC Heap

Once you have a class that can go on the MPGC heap, creating instances of that class is a piece of cake:

```
gc_ptr<Book> book = make_gc<Book>(author, title, nPages);
```

The make_gc<T> function takes arguments that are passed on to *T*'s constructor.[8]

What Can Go on the MPGC Heap

A program will that uses the MPGC heap will work with three flavors of data types:

- **GC-allocated** data is instances of classes specifically designed to live on the MPGC heap. As described below, such a class will inherit from *gc_allocated*, and its instances can *only* be created on the MPGC heap by make_gc.
  - Instances can't be created by **new** on the process's local heap, nor can they be created on the stack or as global, class-static, or function-static variables. They also cannot be thread-local.
- **GC-friendly** data is data that isn't GC-allocated, but that can be placed within GC-allocated objects. To be GC-friendly, it needs to be safe to simply drop the data on the floor, which means that its data type must be *trivially destructible*. It also must be the case that MPGC can find all of the GC pointers within the data, which means that the data type must have an associated *descriptor*. And it may not be a class that has virtual functions
  - Defining a descriptor will be described below, as will a workaround for the restriction on virtual functions.

---

[8] The extra gc_token argument (described below) is not passed in to make_gc<T>.

  - All primitive data types (numbers, **bool**, **char**) are GC-friendly.
  - *gc_vector<T>* is GC-friendly.
  - *gc_ptr<T>* is GC-friendly.
  - *gc_string* is GC-friendly
  - *std::pair<X,Y>* is GC-friendly if *X* and *Y* are.
  - *std::tuple* is expected to be GC-friendly if its components are, but this is not yet implemented.

  GC-friendly data can go anywhere—with the important exception that if it contains gc_ptrs, it can't go on the process-local heap or in static or global variables—but cannot be created by make_gc.
- Data that is neither GC-allocated nor GC-friendly is **GC-unfriendly**. GC-unfriendly data may not appear on the MPGC heap, either as a top-level object or inside of one.

Garbage-Collected Arrays

In addition to individual objects, you can also put fixed size arrays on the MPGC heap. Such arrays can contain either GC-allocated or GC-friendly objects. If the element type is a GC-allocated class, the elements will be *gc_ptr*s to the data objects, while if the element type is a GC-friendly class, the elements will be the data objects themselves.

GC-allocated arrays are instances of type *gc_array<T>*. To create an array of 100 integers, simply call

```
gc_ptr<gc_array<int>> array = make_gc<gc_array<int>>(100);
```

As a shortcut, this can be written as

```
gc_ptr<gc_array<int>> array = make_gc_array<int>(100);
```

and *gc_ptr<gc_array<T>>* can be written as *gc_array_ptr<T>*, so it can be further simplified to

```
gc_array_ptr<int> array = make_gc_array<int>(100);
```

If make_gc_array<*T*>() determines that the array would have a size of zero, it returns a null *gc_array_ptr<T>*.

The *gc_array* class has the following constructors:

1. gc_array(gc_token &, std::size_t n)
   Creates a *gc_array* containing exactly n default-constructed elements.
2. gc_array(gc_token &, std::size_t n, Iter from, std::size_t n_to_copy)
   Creates an array of size n, copying at most n_to_copy elements starting at from (inclusive). If n_to_copy is less than n, the remaining elements are default initialized (or left uninitialized if a zero value is equivalent to default initialization).

Make_gc_array<*T*>() has the following forms:

1. make_gc_array<*T*>(std::size_t n)
   Creates a *gc_array* containing exactly n default-constructed elements.
2. make_gc_array<*T*>(*Iter* from, *Iter* to)
   Creates a *gc_array* containing the elements in the range from from (inclusive) to to (exclusive).
3. make_gc_array<*T*>(std::size_t n, *Iter* from, *Iter* to)
   Creates a *gc_array* containing exactly n elements, with the initial elements taken from the range from from (inclusive) to to (exclusive). If there are remaining elements, they are default constructed. If the range contains too many elements, only the first n are copied.

gc_array<T> has an interface modeled after *std::array<T>*. Since a null *gc_array_ptr<T>* is treated as a reference to a zero-element array, to avoid requiring you to constantly check for null pointers, all methods on *gc_array<T>* are also available on *gc_array_ptr<T>*.[9] For example, instead of saying

```
array->size()
```

you can simply say

```
array.size()
```

and this has the advantage that it works even if array is a null pointer, in which case it returns zero for the size.

Given a *gc_array_ptr* like the one created above, you can read and modify elements

```
int x = array[5];
array[i] += 10;
```

If the provided index is out of bounds, an assertion failure will result. Alternatively, you can use checked lookup

```
int x = array.at(5);
array.at(i) += 10;
```

Here, if the provided index is out of bounds, *std::out_of_range* will be thrown.

You can also get references to the front() and back() of the array.

[9] *Most, but not all functions are mirrored.*

The results of these operations are of type *gc_array::const_reference* if the array is const and *gc_array::reference* otherwise. This value object holds onto the array object, so it is guaranteed to remain valid.[10]

gc_array<T> provides methods for querying the size of the array:

```
std::size_t n = array.size();
bool is_empty = array.empty();
```

You can clear() the array, setting all of its content to copies of default-constructed elements.

Finally, *gc_array<T>* provides a full set of functions for iterating over the content: begin(), end(), cbegin(), cend(), rbegin(), rend(), crbegin(), and crend(). These return instances of type *gc_array::iterator* or *gc_array::const_iterator*, as appropriate. Because begin() and end() are provided, *gc_array_ptr<T>* may be used in a range-for statement:

```
for (int n : array) {
    sum += n;
}
```

As with the reference classes, iterators prevent the array from being garbage collected,[11] so it is safe to hold onto them. Note, however, that they hold onto the entire array, so keeping iterators around may be wasteful if only a single element is needed.

gc_array iterators act as C++ random-access iterators, which means you can increment them, decrement them, add or subtract a number from them, take the difference between two of them, and compare them for equality or ordering. Comparing iterators into different arrays is undefined behavior and may result in an assertion failure. It is legal to create an iterator that points beyond the bounds of the array, but attempting to dereference such an iterator (either to get a *reference* or access a data member or member function) is undefined and will likely result in an assertion failure. It is, however, guaranteed that such errors will not corrupt the MPGC heap.

Finally, for programs that wish to use them in templated classes or functions, *gc_array<T>* defines

```
constexpr static bool holds_direct_values;
typename value_type;
```

---

[10] *This is currently not guaranteed. Until it is, reference objects should only be used as temporaries when it is sure that something else holds a reference to the array.*

[11] *In this case, they currently do.*

value_type is the type of value actually held by the array. It will be either *T* or *gc_ptr<T>*. holds_direct_values will be **true** if *value_type* is *T* and **false** otherwise.

Two Flavors of Smart Pointer

Because of the requirement that MPGC be able to find all of the pointers held by a process, there are actually two flavors of smart pointer used to refer to GC-allocated objects. Which is used depends on where the pointer will be found.

Pointers on the MPGC heap and the stack

To refer to a GC-allocated object from within another GC-allocated object or by a (non-static) function-local variable or temporary value, the class *gc_ptr<T>* (or *gc_array_ptr<T>*) is used. This is a garden-variety smart pointer and provides all of the expected conversions. Essentially, if an assignment or initialization would be valid for a *T** pointer, the corresponding assignment or initialization will be valid for a *gc_ptr<T>*. In particular,

```
gc_ptr<Derived> dp = …;
gc_ptr<Base> bp = dp;
```

is valid if *Base* is a (possibly indirect) base class of *Base*. Also, assigning a non-const pointer to a const pointer:

```
gc_ptr<T> ncp = …;
gc_ptr<const T> cp = ncp;
```

is valid. The other direction, assigning a const pointer to a non-const pointer, however, is not valid.

A default-constructed *gc_ptr<T>* does not refer to any object. The same effect can be obtained by initializing it to or assigning it from **nullptr**. Checking whether a *gc_ptr* is null can be done by calling is_null() on it, comparing it to **nullptr**, or converting it (implicitly) to **bool**.

To get a *gc_ptr<T>* to **this** from within a method of *T*, you can call this_as_gc_ptr(**this**). As a shortcut, the **GC_THIS** macro expands to that.[12]

If you are certain that it's safe to cast a bare pointer to a *gc_ptr*, you can use

```
gc_ptr_from_bare_ptr(ptr)
```

Yes, it's cumbersome. It's intended to be cumbersome. If you find yourself needing to do this often, sit back and think of whether what you're doing is reasonable.[13] Doing such a conversion if you aren't certain that the object it points to still exists (i.e., if you

---

[12] this_as_gc_ptr() is still useful for situations in which you've cast away constness to get yourself a non-const **this** pointer within a const method.

[13] MPGC currently uses it five times—three times within the allocator itself, once to convert an *external_gc_ptr* to the corresponding *gc_ptr*, and once in the definition of this_as_gc_ptr().

can't prove that there's a *gc_ptr* to it somewhere) will lead to the creation of a dangling *gc_ptr* and will corrupt the MPGC heap.

Except for *gc_array_ptr*s, which implicitly convert to *gc_array::iterator*s, *gc_ptr*s *are not iterators*. They can be compared for equality and inequality, but they cannot be incremented, you can't add a number to one, you can't take the difference between two of them, and you can't ask whether one is less than another.

To obtain a *T** pointer from a *gc_ptr<T>*, you can call

```
p.as_bare_pointer()
```

This should be used with great care, as it allows you to get a pointer to an object which may outlive the object. It is possible that this facility may go away in the future.

gc_ptr<T> also supports std::swap(), so you can say

```
std::swap(p1, p2);
```

to swap two pointer values. And it supports the standard static_pointer_cast, dynamic_pointer_cast, and const_pointer_cast to explicitly make conversions that are not implicitly allowed. To cast away constness, you can say

```
gc_ptr<const T> cp = …;
gc_ptr<T> ncp = std::const_ptr_cast<T>(p);
```

And you can cast down an inheritance hierarchy by

```
gc_ptr<Base> bp = …;
gc_ptr<Derived> dp = std::static_pointer_cast<Derived>(bp);
```

For the latter, **be very careful that you know that the object is actually of the destination type**. If you are wrong and you try to access fields of the object that don't exist, you can corrupt the MPGC heap. (Merely creating the incorrect pointer, however, cannot harm the MPGC heap.)

GC anchors

When you just need to ensure that there's a *gc_ptr* to an object, in order to ensure that it doesn't go away, the *gc_anchor* type, which is an alias for *gc_ptr<gc_allocated>* may be used.

Atomic GC pointers

Both flavors of smart pointer, *gc_ptr<T>* and *external_gc_ptr<T>* fully support the C++ *std::atomic<T>* class in a safe, fault-tolerant, and lock-free manner.

Pointers on the Process Heap, in Statics, and Global Variables

To refer to a GC-allocated object from outside of the MPGC heap and the thread's stack, a different type of smart pointer is used: *external_gc_ptr<T>* (and *external_gc_array_ptr<T>*). This smart pointer has essentially the same API as *gc_ptr<T>*.

The rules for when to use which are

- On the program stack (e.g., temporary values, function parameters, and non-static function local variables), either type of pointer can be used.
- In an object instantiated (directly or as part of another object) on the MPGC heap, *gc_ptr<T>* must be used, and trying to use *external_gc_ptr<T>* will cause a compiler error.
- Anyplace else, *external_gc_ptr<T>* must be used. This includes
  - in global variables
  - in class static variables
  - in function static variables
  - in objects created by **new** on the process heap

Unfortunately, we do not currently have any way to guarantee that *gc_ptr<T>* is not used in those last circumstances, so it's up to the programmer to be careful. If a *gc_ptr<T>* is put in any of those locations, the MPGC collector will not be able to find it, and if all such pointers to an object are hidden, the object will be collected as garbage, leading to the hidden pointer becoming a "dangling pointer". At that point, either dereferencing it or copying it someplace that the MPGC collector will find it will likely case corruption of the heap. So don't do it.

An *external_gc_ptr<T>* can be dereferenced directly by means of operator ->(), and an equivalent *gc_ptr<T>* can be obtained by calling value() on it. Going the other way, *gc_ptr<T>* is implicitly convertible to *external_gc_ptr<T>*.

Don't Hide Pointers

Read this section carefully!

As may be apparent from the above, it's very important that the garbage collector be able to find all of the pointers from your process in order to guarantee that it doesn't collect something that it didn't realize you had a pointer to squirreled away. Following a few simple rules will guarantee that that doesn't happen:

1. Obey the rules given above on when to use *gc_ptr<T>* and when to use *external_gc_ptr<T>*.
2. Do not, under any circumstances create a *std::shared_ptr<T>* or *std::unique_ptr<T>* pointing to a GC-allocated object.[14]
3. Don't get a bare *T** pointer to a GC-allocated object other than, perhaps, **this** within the object's methods if it will only be used during the method call.

[14] Well, okay, maybe under some circumstances. *external_gc_ptr<T>* actually uses *std::shared_ptr<T>* internally in a somewhat bizarre, but safe, manner. This falls under the dictum of "When you're ready to break a rule, you *know* you're ready: you don't need anyone else to tell you. If you're not that certain, then you're *not* ready." (Tom Phoenix)

4. If you do get a bare *T** pointer to a GC-allocated object, don't **reinterpret_cast** it to a number, change the number beyond recognition, and expect to get a valid pointer back by undoing your changes.
5. Similarly, don't do that sort of thing with the internal representation of a *gc_ptr<T>* or *external_gc_ptr<T>*.
    - If you feel you need to add flags or a version number to a *gc_ptr<T>*, check out the *versioned_gc_ptr<T>* class described below.
6. Don't get a bare *T** pointer to a field of a GC-allocated object. The object may get collected behind it, rendering the pointer invalid.
    - See the discussion of *gc_sub_ptr* below for how to do this right.
7. If, for some reason, you do decide to store away a *T** pointer, don't assume that it will be valid in another process. There is no guarantee that the MPGC heap will be mapped into the same part of the address space in another process.

An important and easily overlooked way to hide pointers is to place an instance of a GC-friendly (or-unfriendly) class **that contains** gc_ptrs in the process-local heap or in a static or global variable. This includes classes like gc_vector<T> or gc_string (see below), and it also includes standard C++ classes like std::vector<gc_ptr<T>> or std::set<gc_ptr<T>>, which allocate objects on the process-local heap, **even if the std::set or std::vector itself is placed on the stack**. In the latter case, it is safer to use std::vector<external_gc_ptr<T>> or std::set<external_gc_ptr<T>>. In te former, try to use a variant (e.g., external_gc_vector<T> or external_gc_string) that uses external_gc_ptrs rather than gc_ptrs internally.

When You Can Hide Pointers

That being said, whil the overhead to using *gc_ptr<T>* is small, it isn't non-existent, and there may be times where this is important. So the exception to all of those rules is:

> It's okay to use a bare *T** pointer or a pointer to a data member or a *T&* reference or a munged number you got from a pointer **IF** you are really, truly, completely, one hundred percent certain **that a *gc_ptr<T>* or *external_gc_ptr<T>* someplace else points to the same object**.

Note that this certainty extends to

1. certainty that no other thread could overwrite that pointer while you're still counting on it being there, and
2. certainty that, if it's on the stack, the compiler won't decide to remove it or overwrite it early (or never create it in the first place) because it can prove to itself that there's no way it could possibly be used.

That last point is very tricky. If you're not sure you understand it, you're probably better off not chancing it, especially if you're compiling with optimization on.

One common case in which it can be safe to use classes like std::vector<gc_ptr<T>>, which hide pointers, is when you can be sure that all of the objects they will contain are also contained in some other, non-hiding, object, like a gc_vector<T> or gc_array<T>. For example, it is safe to say

```
long n_unique(const gc_vector<Employee> &v) {
    std::unordered_set<gc_ptr<Employee>> unique(v.begin(), v.end());
    return unique.size();
}
```

as long as you can be certain that no other thread (in any process) can change v while you're working.

Persistent Root Objects

In addition to objects kept alive by pointers from running processes, MPGC also has the notion of *persistent root objects*. These are objects that are guaranteed to stick around even if no process is pointing to them. They can be viewed akin to the stable storage provided by a file system and are useful for communicating between processes, whether running concurrently or at different times.

The persistent root objects are accessed via mpgc::persistent_roots(), which returns a reference to a map-like object that can be accessed by:

```
gc_ptr<Graph> network =
    persistent_roots().lookup<Graph>("com.hpe.myproject.Network");
```

Note that lookup<*T*>() trusts you that the object stored under that key (if there is one) is actually of type *T*. **Don't be wrong**. To minimize the chances of being wrong,

- Choose a key that is very unlikely to be used by any other program for a different purpose.
- Instead of making lots of calls on persistent_roots(), create a single *control block* object for your application that can be used to access other root objects in a type-safe manner. That way, you only have to be right about one object.

The key used to look up persistent root objects can be, among other things, a string (either a C++ *char** or a *std::basic_string*) or a number.

To associate an object with a persistent root key, you can say

```
persistent_roots().store(my_key, my_value);
```

To atomically change the association and get a new value, you can say

```
gc_ptr<T> oldVal = persistent_roots().store(my_key, new_value);
```

As before, the system trusts you that the old value, if it exists is of the specified type.

To remove an association, use remove(key). Note that this is different from associating it with a null reference. To check whether the association exists, call contains(key). To add an association only if one didn't exist, you can use

```
gc_ptr<T> val = persistent_roots().store_new(my_key, new_value);
```

This value returned is the resulting value: the old one if one existed or the specified one, which became the associated value.

To do a compare-and-swap, setting the value only if the current one is what you expect, you can use

```
bool worked = persistent_roots().replace(key, expected, new_value);
```

Here, expected is a (C++) reference to a *gc_ptr<T>*. If the function returns **false**, expected is changed to the current value.

Finally, when multiple processes are sharing the same persistent root object, it may be the case that you want to ensure that they all see the same one and that whoever gets there first creates it and associates it, but it doesn't matter which process that is. For that, you can use

```
gc_ptr<Graph> network = persistent_roots().find_or<Graph>(key, fn, args …);
```

If the association exists, it is returned. Otherwise fn(args…) is called and stor_new() is called on the result. Note that multiple processes may call the function, but it is guaranteed that only one will make the association and all will see the same resulting value. Any extraneous objects created will be dropped on the floor and collected.

In the common case in which the work to be done is simply to construct the object using make_gc<T>, this can be shortened to

```
gc_ptr<Graph> network = persistent_roots().find_or_create<Graph>(key, args …);
```

where the arguments are passed to the constructor.

Pointers into Objects on the Heap

The *gc_sub_ptr<T>* class is used to get a GC-safe pointer to a data member of a GC-allocated object. In this case, *T* is a GC-friendly class. Consider the case in which a GC-allocated class wants to maintain a list of counters to increment upon notification of some event. The counters can be in any class; the notifier doesn't care. What it can do is create a method

```
void register_counter(gc_sub_ptr<int> counter);
```

and store the counters away, say in a *gc_vector<gc_sub_ptr<int>>*. Then, when it comes time to notify everybody all it has to do is say

```
void notify() {
    for (const gc_sub_ptr<int> &c : counters) {
        (*c)++;
    }
```

```
}
```

To create a *gc_sub_ptr<T>* you need to provide a *gc_ptr* to the object and either a pointer to the instance's data member or a pointer-to-data-member. If counter is, say, the n_orders field within a *Customer* class, then from within a *Customer* method, it could say

```
notifier.register_counter(gc_sub_ptr<int>(&n_orders, this_as_gc_ptr(this)));
```

If, on the other hand, you have a *gc_ptr<Customer>*, you can do it as

```
notifier.register_counter(gc_sub_ptr<int>(&Customer::n_orders, pCust);
```

In either case, the pointer to the integer data member held by the notifier is guaranteed to remain valid even if it's the only remaining reference to that particular *Customer* object.

The result of dereferencing a *gc_sub_ptr<T>* is a *gc_sub_ref<T>*, which also holds onto the enclosing object. Like a reference, it can be read and assigned to.

For situations in which an *external_gc_ptr* would be used, use *external_gc_sub_ptr* and *external_gc_sub_ref*.

Wrapping Non-GC-Allocated Classes

When you have a GC-friendly type that you want to make GC-allocated, a quick way to do this is to use *gc_wrapped<T>*. For example, *gc_vector<T>*, described below, is a GC-friendly class so that instances of it can sit inside of GC-allocated objects or be put on the stack, without requiring an explicit *make_gc<gc_vector<T>>* (which wouldn't actually work, since *gc_vector* is not GC-allocated). If, however, there's a need for, say, a *gc_vector*<**double**> to be shared between several objects, it could be created as

```
auto v = make_gc<gc_wrapped<gc_vector<double>>>(256);
```

Where v would be a *gc_ptr<gc_wrapped<gc_vector*<**double**>>>. The *gc_-wrapped* object implicitly converts to the thing it wraps, so a user could simply say

```
double sum = std::reduce(v->begin(), v->end());
```

A *gc_ptr<gc_wrapped<T>>* implicitly delegates to the content of the *gc_wrapped* object (i.e., the contained *T*). So if pw is such a pointer, *p will refer to the content and p->fn() will call a function on the content.

Writing Classes That Use Either Flavor of Pointer

Sometimes, you may want to write a class that needs to hold references to GC-allocated data but where you want the option of making instances GC-friendly, to put inside GC-allocated objects, but you also want to be able to make instances that you can safely put inside a *std::vector* or in a static variable. To simplify this, MPGC provides two *pointer class* classes:

```
struct internal_pointers {
    template <typename T> using ptr_t = gc_ptr<T>;
```

```
    template <typename T> using array_ptr_t = gc_array_ptr<T>;
    template <typename T> using sub_ptr_t = gc_sub_ptr<T>;
    constexpr static bool is_internal = true;
};
struct external_pointers {
    template <typename T> using ptr_t = external_gc_ptr<T>;
    template <typename T> using array_ptr_t = external_gc_array_ptr<T>;
    template <typename T> using sub_ptr_t = external_gc_sub_ptr<T>;
    constexpr static bool is_internal = false;
};
```

With these classes, you can write class templates like

```
template <typename PC>
class BudgetItem {
    template <typename P> friend class BudgetItem<P>;
    template <typename T> using ptr_t = PC::template ptr_t<T>;

    ptr_t<Department> _department;
    long _budget;
public:
    template <typename P>
    BudgetItem(const BudgetItem<P> &other)
        : _department(other.department), _budget(other.budget)
        {}

    template <typename P>
    BudgetItem(BudgetItem<P> &&other)
        : _department(std::move(other.department)), _budget(other.budget)
        {}
    …
};
```

The **friend** declaration is necessary to allow *BudgetItem*s with different pointer classes to see one another's private members.

Working with Multiple Threads

For multi-threaded programs that use MPGC, it's important that MPGC be aware of all threads that may use pointers to objects on the MPGC heap, even as temporary variables, whether *gc_ptr*s, *external_gc_ptr*s, or bare *T** pointers. In the future, we hope to be able to hook in threads automatically, but for now, make sure that each of your threads calls

```
mpgc::initialize_thread();
```

This is an inlined function that can be safely called any number of times (after the first time, it's essentially just a check of a thread-local boolean variable), so it can be used to guard any function that may use pointers to GC-allocated objects, but the easiest thing is often to just put it as the first line of the function used as the argument to the thread constructor.

As an alternative, the argument to the thread constructor can be wrapped by a call to mpgc::gc_safe(), as in

```
thread t(mpgc::gc_safe(worker_fn), arg1, arg2);
```

This simply returns a lambda function that calls initialize_thread() and then calls the provided function (taking the same arguments as the provided function).

If the program is single-threaded, initialize_thread() is guaranteed to be called before the program can obtain a pointer into the MPGC heap, so it need not be explicitly called.

Forking Processes

When using fork() to create a new process, the basic rule is…don't. The slightly more accurate rule is: Don't, unless you essentially immediately proceed to call exec() to load a new binary in the forked process, before you do anything using any GC pointers you may still have access to. The new process will not have an MPGC garbage collection thread running, but because of the way C++ static variables work, even if you call initialize_thread(), it won't realize that it needs to create one. If you're lucky, the new process will simply crash before corrupting the MPGC heap. But don't count on it.

Working with C++ Standard Library Classes

Unfortunately, most of the classes in the C++ standard library are not yet GC-friendly. One that is is *std::pair<X,Y>*, which is GC-friendly if *X* and *Y* are.[15]

The standard smart pointer classes (e.g., *unique_ptr<T>* and *shared_ptr<T>*) will probably never be GC-friendly, as they are too tied to an individual process.

Most of the container classes allocate their own storage by means of an *allocator* parameter, which, by default, allocates from the process's heap. We hope to develop our own allocator, which will allow most, if not all of the standard library to be able to allocate its data on the MPGC heap. Until then, we are adding GC-friendly replacements for classes one by one.

GC-Friendly Vectors

The *mpgc::gc_vector<T>* class is a GC-friendly replacement for *std::vector<T>* and presents the same interface. It allocates its backing storage on the MPGC heap. Like *std::vector*, *gc_vector* is not inherently thread-safe. The same operations that invalidate iterators for *vector* will do the same fore *gc_vector*. Unlike *vector*, however, when a *gc_vector*'s iterator becomes invalid because the vector replaced its backing array, the old backing array sticks around until there are no more iterators pointing to it. So writing through an invalid iterator won't corrupt the MPGC heap. Also, attempting to dereference an iterator that doesn't point into the vector (in cases where an *out_of_range* exception is not thrown) will result in an assertion failure rather than heap corruption.

gc_vector<T> contains a *gc_ptr<T>*, so it should only be used in situations in which it is safe to use a *gc_ptr* (i.e., in the MPGC heap or on the stack). In other situations, in which you would use an *external_gc_ptr<T>*, use *external_gc_vector<T>* instead. These two classes are freely interconvertible, including being move-

[15] *std::tuple<Types…> will be as well, but it's proving to be trickier than expected*

constructible from one another, so returning either by value does not result in any copies being made, even if the receiving context expects the other one. If you need to write template code that can take either type of vector, the actual class is

```
namespace mpgc {
    template <typename T, typename PointerClass>
    class gc_basic_vector;
}
```

and *gc_vector* and *external_gc_vector* are

```
namespace mpgc {
    template <typename T>
        using gc_vector = gc_basic_vector<T, internal_pointers>;
    template <typename >
        using external_gc_vector = gc_basic_vector<T, external_pointers>;
}
```

using the internal_pointers and external_pointers *pointer classes* described above.

GC-aware Strings

The *mpgc::gc_basic_string<CharT,Traits,PointerClass>* class is a GC-friendly replacement for *std::basic_string<CharT,Traits,Alloc>* and presents much the same interface. (It's actually a bit extended in that it also takes the *std::basic_string* as a parameter where it seems reasonable.) Like *basic_string*, *gc_basic_string* has aliases for common character types: *gc_string* (**char)**, *gc_wstring* (**wchar_t**), *gc_uc16string* (**char16_t**) and *gc_uc32string* (**char32_t**).

As with *gc_vector*, *gc_basic_string* is not inherently thread-safe, but its iterators hold onto the backing array.

Also as with *gc_vector*, there are "external" forms of *gc_string* (*external_gc_string*), etc., which are freely interconvertible. Use *gc_string* where you would use *gc_ptr<T>* and *external_gc_string* where you would use *external_gc_ptr<T>*.

One thing to watch out for. The data() and c_str() methods both return pointers to the underlying (null-terminated) character array as a bare pointer rather than as an *iterator*. This can sometimes be necessary in order to pass in to functions that expect such poiners, but these must be used with extreme care. If the *gc_string* is changed (even in another thread) and there are no iterators holding onto the backing array, these pointers will dangle, and writing through them can corrupt the MPGC heap. Unless you are absolutely positive that nobody will change the *gc_string* while you're using it, it's safer to call begin() to get an iterator and then call as_bare_pointer() on that to get the bare pointer. As long as you hold onto the iterator, the pointer will be valid, at least as long as you don't go past the first null character.

Designing GC-Allocated Classes

Of course, most programs will want to define some GC-allocated classes of their own. Luckily, this is straightforward, although there are some rules that have to be followed.

No Virtual Functions or Virtual Base Classes

The first rule is that a GC-allocated class (or a GC-friendly class, for that matter) cannot have virtual functions or virtual base classes. The reason for this is that all instances of a class with virtual functions or virtual base classes include a *virtual function table pointer*, and the compiler generates this to point to functions loaded from the program's binary. Unfortunately, if another process tries to follow this pointer and jump to code it is very unlikely that anything good will happen and quite likely that the program will immediately crash. (Worse, it might not crash…yet.)

We currently don't catch violations of this at compile time.

MPGC provides a (somewhat awkward, but usable) way to overcome this limitation, which will be described below.

No destructor

The destructor for a GC-allocated object will never be called, so it shouldn't have one. The same goes for a GC-friendly class used for members of a GC-allocated object. With templated classes, this may sometimes be difficult, but the rule is that if it's there, the destructor should be *trivial*. This is, with the template parameters used, it shouldn't actually do anything.

To enforce this, all GC-allocated classes must either satisfiy std::is_trivially_-destructible<*T*> or else declare a specialization of mpgc::is_collectible<*T*> that has returns **true** for value.

Derive from gc_allocated

The next rule is that all GC-allocated classes need to derive from *gc_allocated*. (GC-friendly classes may not.) This derivation may be indirect. That is, your GC-allocated class can derive from another GC-allocated class rather than deriving from *gc_allocated* directly. The derivation should be public, either by declaring your class as a **struct** or by declaring the base class as **public**.

```
class Person : public gc_allocated
```

If multiple inheritance is used, only the first base class may be *gc_allocated* (or another GC-allocated class). If the first base class is not GC-allocated, a compile-time error will be generated when you attempt to call make_gc<T>() on your class. All other base classes must be GC-friendly.

Add an Extra Parameter to Your Constructors

All constructors for your class must have an initial parameter of type *gc_token&*. This value should be passed up to your first base class, as in

```
Person(gc_token &token, const gc_string &name, int age)
    : gc_allocated(token), _name(name), _age(age)
    { }
```

Passing it up is the *only* thing that you should do with that token. It is generated by make_gc() and is specific to the particular allocation in progress. Storing it away or passing it in to somebody else's constructor is almost certainly an extraordinarily bad idea.

Make Your Constructors Visible to make_gc()

Any constructors that you want to allow to be used to create instances via calls to make_gc() need to be visible to it. This is easy for public constructors, but the rule applies even to calls to make_gc() from within your own class and subclasses. There are (at least) three ways to accomplish this.

First, you can make the constructors public. This will work, but protection is typically there for a reason.

Second, you can *friend* in the make_gc() function:

```
template <typename X, typuename …Args>
friend gc_ptr<X> make_gc(Args&&…args);
```

This will work, but it's essentially the same as making the constructor public. Anybody who wants can call make_gc() and get access to your hidden constructor.

To keep protection, use the third approach. Make all of your constructors public, but give them a parameter of a private type:

```
private:
    class private_ctor {};
public:
    Person(gc_token &token, private_ctor, long id)
        : gc_allocated(token)
    {
        load(id);
    }
```

Now, within your class, you can say

```
make_gc<Person>(private_ctor(), id);
```

But anybody outside the class won't be able to create the *private_ctor* parameter.

Use only GC-Friendly Members

For a GC-allocated class, all of the non-static data members must be GC-friendly. As described above, a GC-friendly data type:

- Does not derive from *gc_allocated*
- Has no virtual functions
- Has a trivial destructor (or no destructor)
- Has only GC-Friendly non-static data members and base classes
- Has a defined object descriptor (more on that below)

GC-friendly classes include

- all of the C++ primitive data types (numbers, characters, **bool**).
- all empty classes (or other classes too small to contain a pointer).
- *gc_ptr<T>* for GC-allocated *T*.
- *gc_array_ptr<T>* for GC-allocated or GC-friendly *T*.
- *gc_sub_ptr<T>* for GC-friendly *T*.

- *std::atomic<T>* for GC-friendly *T.*
- *std::pair<X,Y>* for GC-friendly *X* and *Y*.
- *gc_vector<T>* for GC-friendly or GC-allocated *T.*
- *gc_basic_string<C,T>* (*gc_string, gc_wstring*, *gc_u16string*, *gc_u32string*)

They also include bare pointers (*T**) and references (*T&*). Note that

- These are ignored for purposes of garbage collection.
- They are unlikely to work from any other process than the one that created the object (including another run of the same program).

It is not a good idea to put them in GC-allocated objects.

Define an Object Descriptor

Finally, each GC-allocated or GC-friendly class needs to tell MPGC how to find the GC pointers within its instances. There are two ways to do this.

The most common way is to define a public static descriptor() function within the class. Say we're defining an *Employee* class, which inherits from *Person* and has a *_department* field and a *_boss* field. The descriptor() function is defined formulaically as

```
public:
    static const auto &descriptor() {
        static gc_descriptor d =
            GC_DESC(Employee)
            .WITH_SUPER(Person)
            .WITH_FIELD(&Employee::_department)
            .WITH_FIELD(&Employee::_boss);
        return d;
    }
```

All descriptor() functions will look essentially identical: a call to GC_DESC() with the name of the class, calls to WITH_SUPER() for all superclasses other than, optionally, *gc_allocated*, and calls to WITH_FIELD() for all non-static data members. It doesn't matter what order the calls to WITH_SUPER() and WITH_FIELD() come in, but they all must be mentioned or you'll get a compilation error when an object of this class (or a class that contains it) is created using make_gc().

If the class is a template class, don't include the template parameters in the call to GC_DESC() and add the keyword **template** immediately before WITH_FIELD (after the dot).

If for some reason you can't add this function within the class itself, the descriptor can be defined externally. To do this, define a specialization of the *gc_traits<T>* class that includes the function:

```
template <>
struct gc_traits<Employee> {
    static const auto &descriptor() { … }
```

```
};
```

There are two types of compiler error that you're likely to run into when writing descriptor() functions. Unfortunately, they result in very long compiler errors.[16] Fortunately, it's not terribly hard to spot what went wrong.

The first type of error is mentioning a field that isn't GC-friendly. This will result in an error of the form

> incomplete type 'mpgc::gc_traits<*some type name*, void>' used in nested name specifier

The "required from here" line following this error will point to the descriptor() function that has a WITH_FIELD() or WITH_SUPER() that's not GC-friendly, and the "some type name" in the error message will tell you the type it couldn't make a descriptor for.

The second type of error is forgetting to mention a field. This will result in

```
static_assert(Mask == 0, "Not all fields covered");
```

Again, the "required from here" line, this time above the error message, will point to the descriptor() function that's missing something.

How to Fake Virtual Functions

As noted above, GC-allocated classes cannot have virtual functions, because this would result in the C++ compiler generating code that would store pointers into the process's local memory into the MPGC heap. But virtual functions are useful, so we provide an admittedly ugly way to get the same functionality.

Consider the following (GC-unfriendly) classes, perhaps from a graphics program:

```
class Shape {
    Color _color;
public:
    Shape(Color c) : _color(c) {}
    Color color() cont {
        return _color;
    }
    virtual double area() const = 0;
    virtual void grow(double factor) = 0;
};

class Rectangle : public Shape {
    double _width;
    double _height;
public:
    Rectangle(Color c, double w, double h)
        : Shape(c), _width(w), _height(h)
    {}
    double area() const override {
        return _width * _height;
```

[16] GCC 6.1 supports the new C++ *concepts* feature, which should make the error messages much more meaningful, but we don't yet require that version.

```
    }
    void grow(double factor) override {
        _width *= factor;
        _height *= factor;
    }
};
```

All *Shape*s have a non-virtual color() method and two virtual methods, a const area() method and a non-const grow() method. *Rectangle* is a shape and defines the two virtual methods in a straightforward way.

So how do we make this GC-allocated?

Derive from gc_allocated_with_-virtuals

The first thing to do is to make the base class, *Rectangle*, derive from a class called *gc_allocated_with_virtuals<Base,Disc>*, where *Base* is the root of this particular class hierarchy (in this case, *Rectangle*), and *Disc* is a *discriminator* type used to differentiate the leaf classes. Each leaf class must have a different value. Good choices for the discriminator type are **enum**s (when the complete set of classes is known ahead of time) or integers. If *Disc* isn't specified, it defaults to **size_t**. Note that these values will (currently) be used as the index into an array, so avoid the temptation to use large UUIDs.

In our case, we'll use an enum:

```
enum class Shapes : unsigned char { RECTANGLE, SQUARE, OVAL, CIRCLE };

class Shape : public gc_allocated_with_virtuals<Rectangle,Shapes> {
    using super = gc_allocated_with_virtuals<Rectangle,Shapes>;
    …
};
```

We give ourself an alias because that class name is a mouthful, and we'll need to type it a few times.

Identify the Discriminator Value

The next thing to do is to ensure that each leaf class has an identified discriminator value and that this value is passed up the constructor chain to *gc_allocated_with_virtuals*. By convention, the discriminator value is stored in a static variable called discrim:

```
class Shape : public gc_allocated_with_virtuals<Shape,Shapes> {
    using super = gc_allocated_with_virtuals<Shape,Shapes>;
    Color _color;
public:
    Shape(gc_token &gc, Color c, discriminator_type d)
        : super(gc, d),  _color(c) {}
    Color color() cont {
        return _color;
    }
    …
};

class Rectangle : public Shape {
    double _width;
    double _height;
```

```
public:
    constexpr static discriminator_type discrim = Shapes::RECTANGLE;
    Rectangle(Color c, double w, double h, discriminator_type d = discrim)
        : Shape(c, d), _width(w), _height(h)
    {}
    …
};
```

Only classes that can be instantiated need to have a discrim static member. By convention all constructors take a final argument of type *discriminator_type*, and for instantiable classes, this defaults to discrim. Users of this class aren't expected to pass this in, so a user might simply say

```
gc_ptr<Shape> s = make_gc<Rectangle>(Color::RED, 10, 5);
```

but if there was an instantiable subclass of *Rectangle*, say

```
class Square : public Rectangle {
public:
    constexpr static discriminator_type discrim = Shapes::SQUARE;
    Square(Color c, double lem, discriminator_type d = discrim)
        : Rectangle(c, len, len, d)
    {}
    …
};
```

Saying

```
gc_ptr<Shape> s = make_gc<Square>(Color::RED, 10);
```

would wind up calling *Rectangle*(*Color*::**RED**, 10, 10, *Shapes*::**SQUARE**), which would further call *Shape*(*Color*::**RED**, *Shapes*::**SQUARE**).

Mention the Superclass in the Hierarchy Root Class's Descriptor

As with all GC-allocated classes not inheriting directly from *gc_allocated*, the superclass needs to be mentioned in the root class's object descriptor:

```
class Shape : public gc_allocated_with_virtuals<Rectangle,Shapes> {
    using super = gc_allocated_with_virtuals<Rectangle,Shapes>;
    Color _color;
public:
    Shape(gc_token &gc, Color c, discriminator_type d)
        : super(gc, d),  _color(c) {}
    Color color() cont {
        return _color;
    }
    static const auto &descriptor() {
        static gc_descriptor d =
           GC_DESC(Shape)
           .WITH_SUPER(super)
           .WITH_FIELD(&Shape::_color);
        return d;
    }
    …
};
```

Declare the Virtual Functions

Now comes the fun part. Each virtual class in the hierarchy will have a nested class, by convention called *virtuals*, with the same inheritance hierarchy, the root class's inheriting from *virtuals_base* (declared in *gc_allocated_with_virtuals*). This class is not GC-friendly and can therefore contain virtual functions (in fact, it will contain nothing but virtual functions):

```
class Shape : public gc_allocated_with_virtuals<Rectangle,Shapes> {
    …
public:
    struct virtuals : virtuals_base {
        …
    };
    …
};

class Rectangle : public Shape {
    …
public:
    struct virtuals : Shape::virtuals {
        ..
    };
    …
};
```

Each virtual function will be the same as the desired virtual function, with an extra first argument that's a pointer to the (GC-allocated) class in which it was declared:

```
class Shape : public gc_allocated_with_virtuals<Rectangle,Shapes> {
    …
public:
    struct virtuals : virtuals_base {
        virtual double area(const Shape *self) = 0;
        virtual void grow(const Shape *self, double factor) = 0;
    };
    …
};

class Rectangle : public Shape {
    …
public:
    struct virtuals : Shape::virtuals {
        virtual double area(const Shape *self) override {
           return self->call_non_virtual(&Rectangle::area_impl);
        }
        virtual void grow(const Shape *self, double factor) override {
           self->call_non_virtual(&Rectangle::grow_impl, factor);
        }
    };
    …
};
```

Note that the self parameter in *Rectangle::virtuals::*area() is of type **const** *Shape**. It's **const** because the virtual function it mirrors is a **const** method, and it's *Shape** (rather than *Rectangle**) because it needs to match the declaration in *Shape::virtuals*.

The virtual functions are either pure virtual functions (signaled by the trailing = 0) or they should contain a single call to self->call_non_virtual(). This call should take a

reference to the implementation method of the GC-allocated class (a non-virtual member function) and the virtual function parameters.

The implementation methods are declared in the GC-allocated class, just as they would have been had we been able to declare virtual member functions:

```
class Rectangle : public Shape {
    …
public:
    struct virtuals : Shape::virtuals {
        virtual double area(const Shape *self) override {
          return self->call_non_virtual(&Rectangle::area_impl);
        }
        virtual void grow(const Shape *self, double factor) override {
          self->call_non_virtual(&Rectangle::grow_impl, factor);
        }
    };

    double area_impl() const {
        return _width * _height;
    }
    void grow(double factor) {
        _width *= factor;
        _height *= factor;
    }
    …
};
```

Then, in the class that declares a virtual member function, a non-virtual proxy function is declared:

```
class Shape : public gc_allocated_with_virtuals<Rectangle,Shapes> {
    …
public:
    struct virtuals : virtuals_base {
        virtual double area(const Shape *self) = 0;
        virtual void grow(const Shape *self, double factor) = 0;
    };

    double area() const {
        return call_virtual(this, &virtuals::area);
    }
    void grow(double factor) {
        call_virtual(this, &virtuals::grow. factor);
    }
    …
};
```

The net result of this is that if you say

```
gc_ptr<Shape> s = make_gc<Rectangle>(Color::RED, 10, 5);

s.grow(2);
```

the following sequence of calls takes place:

```
Shape::grow(2)
Shape::call_virtual(s, &Shape::virtuals::grow, 2)
```

```
Rectangle::virtuals::grow(s, 2)
Rectangle::call_non_virtual(s, &Rectangle::grow_impl, 2)
Rectangle::grow_impl(2)
```

When inlining is taken into account, this is very nearly as efficient as a single virtual function call.

Creating the Dispatch Table

Finally, we need to create the dispatch table that will be used to implement call_virtual(). This is done by defining an init_vf_table() function as

```
template <>
void gc_allocated_with_virtuals<Shape,Shapes>
::init_vf_table(vf_table &ctable)
{
    ctable.bind<Rectangle>();
    ctable.bind<Circle>();
    …
}
```

This registers the named classes with their respective discrim values. If you choose not to use a static member named discrim to hold the discriminator, you can pass the discriminator value for the class in as a parameter to the bind() call.

If you don't know all of the possible instantiable classes at one place in your code and want to build up the table dynamically, you can use the protected vtbl() method to get a reference to the table and call bind() on it to register, e.g.,

```
vtbl().bind<Rectangle>();
```

You still have to define the init_vf_table() function, but it needn't actually do any bindings. Note that

- Until a class is registered in the table, you won't be able to call virtual functions on pointers to objects with that class's discriminator.
- The bind() call is not thread-safe, so if there's a chance that two classes may try to call it at the same time, you should wrap the calls with a mutex.

Support Classes and Functions

Versioned Pointers

License

MPGC is distributed under the GNU Lesser General Public License (v. 3.0), with an exception that treats code generated by MPGC during the compilation of an application as falling under the license of the application rather than the license of MPGC.

LGPL v. 3.0

GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

"The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

- a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or
- b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if

the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

- a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.
- b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

- a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.
- b) Accompany the Combined Work with a copy of the GNU GPL and this license document.
- c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.
- d) Do one of the following:
    - 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.
    - 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.
- e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

- a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.
- b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.

Additional Exception

As an exception, the copyright holders of this Library grant you permission to (i) compile an Application with the Library, and (ii) distribute the Application containing code generated by the Library and added to the Application during this compilation process under terms of your choice, provided you also meet the terms and conditions of the Application license.