

Garbage Collection

This document contains the following sections:

[1.0 Garbage collection introduction](#)

[1.1 The garbage collection scheme](#)

[1.2 How newspace is managed with scavenges](#)

[1.3 The system knows the age of objects in newspace](#)

[1.4 Objects that survive long enough are tenured to oldspace](#)

[1.5 Placement of newspace and oldspace: the simple model](#)

[1.6 Oldspace, newspace, and Lisp will grow when necessary](#)

[1.7 The almost former gap problem](#)

[2.0 User control over the garbage collector](#)

[2.1 Switches and parameters](#)

[2.2 Triggering a scavenge](#)

[2.3 Triggering a global gc](#)

[2.4 Getting information on memory management using cl:room](#)

[3.0 Tuning the garbage collector](#)

[3.1 How do I find out when scavenges happen?](#)

[3.2 How many bytes are being tenured?](#)

[3.3 When there is a global gc, how many bytes are freed up?](#)

[3.4 How many old areas are there after your application is loaded?](#)

[3.5 Can other things be changed while running?](#)

[4.0 Initial sizes of old and new spaces](#)

[5.0 System parameters and switches](#)

[5.1 Parameters that control generations and tenuring](#)

[5.2 Parameters that control minimum size](#)

[5.3 Parameters that control how big newly allocated spaces are](#)

[5.4 Gsgc switches](#)

[5.5 Gsgc functions and variables](#)

[6.0 Global garbage collection](#)

[6.1 The tenuring macro](#)

[7.0 Gc cursors](#)

[8.0 Other gc information](#)

[9.0 gc errors](#)

[9.1 Storage-condition errors](#)

[9.2 Gsgc failure](#)

[10.0 Weak vectors, finalizations, static arrays, etc.](#)

[10.1 Weak arrays and hashtables](#)

[10.2 Finalizations](#)

[10.3 Example of weak vectors and finalizations](#)

[10.4 Static arrays](#)

1.0 Garbage collection introduction

Lisp does its own memory management. The *garbage collector* is part of the memory management system. It disposes of objects that are no longer needed, freeing up the space that they occupied. While the garbage collector is working, no other work can be done. Therefore, we have made the garbage collector as fast and unobtrusive as possible. Allegro CL uses a version of the generation-scavenging method of garbage collection. Because optimal performance of generation-scavenging garbage collection depends on the application, you have a great deal of control over how the garbage collector works. In this section, we will describe the user interface to the garbage collector, and suggest how to tune its performance for an application. In what follows, the generation-scavenging garbage collection system will be abbreviated *gsgc*, and the act of garbage collecting will be abbreviated *gc*.

1.1 The garbage collection scheme

The Allegro CL garbage collector is a two-space, generation-scavenging system. The two spaces are called newspace and oldspace. Note that, as we describe below, newspace is divided into two pieces, called areas, and oldspace may be divided into a number of pieces, also called areas. Generally, when we say newspace, we mean both newspace areas and when we say oldspace, we mean all oldspace areas. We try to use the word *area* when we want to refer to a single area, but please note that this naming convention is new and you may run into text that uses ``oldspace'` to refer to an oldspace area. Usually, the context should make this clear.

1.2 How newspace is managed with scavenges

The two pieces of newspace are managed as a stop-and-copy garbage collector system. The two areas are the same size. At any one time, one area is active and the other is not. A newspace area is filled from one end to the other. Imagine, for example, a book shelf. Existing books are packed together on the right side. Each new book is placed just to the left of the leftmost book. It may happen that books already placed are removed, leaving gaps, but these gaps are ignored, with each new book still being placed to the left of the location of the last new book. When the shelf fills up, the other shelf (newspace area) is used. First, all books remaining are moved to the other shelf, packed tight to one side, and new books are placed in the next free location.

So with Lisp objects in newspace areas. All existing objects are packed together on one side of the area and new objects are placed in the free space next to the existing objects, with gaps left by objects which are no longer alive being ignored. When the area fills up, Lisp stops and copies all live objects to the other area, packing them tight. Then the process is repeated.

The process of copying objects from one newspace area to the other is called a *scavenge*. We will discuss the speed of scavenges below, but scavenges are supposed to be so fast that humans usually barely notice them.

1.3 The system knows the age of objects in newspace

The system keeps track of the age of objects in newspace by counting the number of scavenges that the object has survived. The number of scavenges survived is called the generation of an object. When objects are created, they have generation 1, and the generation is increased by 1 at each scavenge.

Of course, many objects become garbage as time passes. (An object is garbage when there are no pointers to it from any other live object. If there are no pointers to an object, nothing can reference or access it and so it is guaranteed never to be looked at

again. Thus, it is garbage.) The theory of a generation scavenging garbage collector is that most objects that will ever become garbage will do so relatively quickly and so will not survive many scavenges.

1.4 Objects that survive long enough are tenured to oldspace

The problem with a stop-and-copy system is that objects that survive have to be moved and moving objects takes time. If an object is going to be around for a while (or for the entire Lisp session), it should be moved out of newspace to some place where it does not have to be moved (or is moved much less often). This is where the other half of the generation scavenging algorithm comes into play. Once an object has survived enough scavenges, it is assumed to be long-lived and is moved to oldspace. Oldspace is not touched during scavenges and so objects in oldspace are not moved during scavenges, thus saving considerable time over a pure stop-and-copy system.

Part of a scavenge is checking the age (generation) of surviving objects and moving those that are old enough to oldspace. The remaining objects are moved to the other newspace area. The age at which objects are tenured is user-settable. Its initial value is 4 and that seems to work for many applications. We will discuss below how changing that (and many other) settings can affect gc performance.

The process of moving an object to oldspace is called *tenuring* and the object moved is said to be tenured. At one point, oldspace was also called tenured space and you may see that term occasionally in Allegro CL documents.

Note the assumption: objects that survive a while are likely to survive a long while.

If one could know exactly how long an object is going to survive, one could provide the best possible garbage collection scheme. But that knowledge is not available. Objects are created all the time by different actions and users and even application writers typically do not know what actions create objects or how long those objects will live. Indeed, that information often depends on future events that are hard to control -- such as the behavior of the person running the application.

So the algorithm makes that assumption: if an object survives for a while, it is likely to survive for a long while, perhaps forever (forever means the length of the Lisp session). Of course, for many objects this assumption is wrong: the object may become garbage soon after it is tenured. However, as we said above, scavenges (which are automatic and cannot be prevented by a user although they can be triggered by a user) do not touch oldspace. In order to clear garbage from oldspace, a global garbage collection (global gc) must be done. An interface for automating global gc's is provided in Allegro CL and different interfaces are easy to implement (see below for more information), but the two important points about global gc's are:

1. in most cases, they should be done from time to time; and
 2. when and if they happen can be wholly under your control.
-

1.5 Placement of newspace and oldspace: the simple model

The Lisp heap grows upwards (to higher addresses). Oldspaces are at low addresses and newspace occupies addresses higher than any oldspace area. This means that newspace can grow without affecting oldspace and oldspace can grow (usually by creating a new oldspace area) by having newspace move up as far as necessary.

Why might newspace grow? Suppose, for example, a newspace area is 600 Kbytes and you want to allocate a 1 Mbyte array. Newspace has to grow to accommodate this.

Why might oldspace grow? As objects are tenured to oldspace, it slowly fills up. Even with regular global gc's, it can fill up.

When it does, newspace moves up and a new old area is created. (New areas are created rather than a single area being expanded for various technical reasons. We discuss below how to reduce sizes dynamically. See [Section 3.5 Can other things be changed while running?](#).)

We will not describe the internal algorithms of the garbage collector because they cannot be changed or modified by users in any way. But let us consider how newspace might be moved, as this might make the process clearer. Suppose the current scavenge is about to move live objects to the high address area. Before anything is moved, Lisp can compute how much space the live objects need, how much space objects waiting to be allocated need, and how much space a new old area needs. From that information, it can compute the highest address of the high address newspace area. It requests from the Operating System that the area be allocated (using **malloc** or **sbrk**), and once the Operating System confirms the allocation, starts copying the live objects to that high address, filling toward lower addresses. When all live objects have been moved and new objects are allocated in the high address newspace area, the new oldspace area (if one is required) can be created and the location of the low address newspace area can be determined. Recall that high address newspace area is active so the low address newspace area does not contain anything of importance.

1.6 Oldspace, newspace, and Lisp will grow when necessary

A consequence of what we just said about newspace moving when it has to grow or when a new oldspace area is needed is that the size of the Lisp image can grow while it is running. This is usually normal, indeed what you want. It allows images to start small and grow as much (but no more) than they need. It also allows the same image to run effectively on machines with different configurations.

But, sometimes growth can be unexpected and the image can want to grow to a size larger than the Operating System can handle (usually because there is not enough swap space).

The growth is often necessary, because of the type of application being run. What is important is that the growth be managed and be no more than is really needed.

1.7 The almost former gap problem

In earlier releases, space for foreign code loaded into the image, space for foreign objects, and direct calls to **malloc** all could cause a gap to be placed above newspace. If a new oldspace or a larger newspace was needed, it had to be placed above the gap, causing in some cases a small need for additional space to result in a multimegabyte increase in image size. Now, malloc space is placed away from the new and old spaces and so the Lisp heap (new and old spaces together) are unaffected and can grow incrementally as needed. There is a Lisp heap size specified by the *lisp-heap-size* argument to **build-lisp-image** (see *building-images.htm*). The OS will try to reserve this space when Lisp starts up. If more space is needed, Lisp will request it from the OS but it is possible more space will not be available. If this happens, you might increase the original request.

The space reserved in a running Lisp is reported as 'resrve' on the 'Lisp heap' line of the output of `(room t)`. If the heap grows larger than that size, gaps may appear. If you see gaps in your application, you should consider starting with an image with a larger heap size.

2.0 User control over the garbage collector

Application writers and users can control the behavior of the garbage collector in order to make their programs run more efficiently. This is not always easy, since getting optimal behavior depends on knowing how your application behaves and that information may be difficult to determine. Also, there are various paths to improvement, some of which work better than others (but different paths work better for different applications).

One thing to remember is that (unless the image needs to grow larger than available swap space), things will work whether or not they work optimally. You cannot expect optimal gc behavior at the beginning of the development process. Instead, as you gather information about your application and gc behavior, you determine ways to make it work better.

2.1 Switches and parameters

The automated gc system is controlled by switches and parameters (they are listed in [Section 5.0 System parameters and switches](#) below). There is not much difference between a switch and a parameter (a switch is usually true or false, a parameter usually has a value) and there probably should not be a distinction, but these things are hard to change after they are implemented. The functions **gsgc-switch** and **gsgc-parameter** can be used to poll the current value and (with **setf**) to set the value of switches and parameters.

The function **gsgc-parameters** prints out the values of all switches and parameters:

```
cl-user(14): (sys:gsgc-parameters)
:generation-spread 4 :current-generation 4 :tenure-limit 0

:free-bytes-new-other 131072 :free-percent-new 25
:free-bytes-new-pages 131072 :expansion-free-percent-new 35
:expansion-free-percent-old 35
:quantum 32

(switch :auto-step) t (switch :use-remap) t
(switch :hook-after-gc) t (switch :clip-new) nil
(switch :gc-old-before-expand) nil (switch :next-gc-is-global) nil
(switch :print) t (switch :stats) t
(switch :verbose) nil (switch :dump-on-error) nil
cl-user(15):
```

gsgc-switch can poll and set switches while **gsgc-parameter** can poll and set parameters: Here we poll and set the `:print` switch.

```
cl-user(15): (setf (sys:gsgc-switch :print) nil)
nil
cl-user(16): (sys:gsgc-switch :print)
nil
cl-user(17): (setf (sys:gsgc-switch :print) t)
t
cl-user(18): (sys:gsgc-switch :print)
t
cl-user(19):
```

The **gc** function can be used to toggle some of the switches.

2.2 Triggering a scavenge

The system will cause a scavenge whenever it determines that one is necessary. There is no way to stop scavenges from occurring at all or even to stop them from occurring for a specified period of time.

However, you can cause a scavenge by calling the `gc` function with no arguments:

```
(excl:gc) ;; triggers a scavenge
```

You can also cause a scavenge and have all live objects tenured by calling the `gc` function with the argument `:tenure`, like this

```
(excl:gc :tenure)
```

2.3 Triggering a global gc

Global gc's (a gc of old and new space) are not triggered automatically (but triggering can be automated). You can trigger a global gc by calling `gc` with the argument `t`:

```
(excl:gc t) ;; triggers a global gc
```

See section [Section 6.0 Global garbage collection](#) for information on other ways to trigger a global gc and ways to automate global gc's.

2.4 Getting information on memory management using `cl:room`

The function `room` provides information on current usage (it identifies oldspaces and newspace and free and used space in each). Setting the `:print` switch and perhaps the `:stats` and `:verbose` switches causes the system to print information while gc's are occurring. See [Section 5.4 Gsgc switches](#) and [Section 3.1 How do I find out when scavenges happen?](#).

Evaluating `(room t)` provides the most information about the current state of memory management. Here is a `(room t)` output from a Allegro CL image that has been doing a fair amount of work. This is from a UNIX machine and was done immediately after a global gc, so some of the oldspaces have significant free space.

```
CL-USER(1): (room t)
area area address(bytes)          cons          other bytes
# type                          8 bytes each
                                (free:used)      (free:used)
Top #x106a0000
New #x10598000(1081344)  134:13113    340128:582984
New #x10490000(1081344)  -----
1 Old #x10290000(2097152)  0:0         2095952:0
0*Old #x10000e80(2683264)  0:68273     0:2124792
Tot (Old Areas)          0:68273     2095952:2124792
* = closed old area
```

Root pages: 61

Lisp heap: #x10000000 pos: #x106a0000 resrve: #x10fa0000
 C heap: #x64000000 pos: #x64011000 resrve: #x640fa000
 Pure space: #x2d1aa000 end: #x2d747ff8

code	type	items	bytes	
112:	(SIMPLE-ARRAY T)	6586	930920	28.3%
1:	CONS	80502	644016	19.6%
8:	FUNCTION	8432	520192	15.8%
7:	SYMBOL	17272	414528	12.6%
117:	(SIMPLE-ARRAY CHARACTER)	2259	259984	7.9%
96:	(SHORT-SIMPLE-ARRAY T)	17498	151736	4.6%
18:	BIGNUM	2966	139928	4.3%
125:	(SIMPLE-ARRAY (UNSIGNED-BYTE 8))	31	87816	2.7%
12:	STANDARD-INSTANCE	3291	52656	1.6%
9:	CLOSURE	2301	39688	1.2%
15:	STRUCTURE	666	24944	0.8%
127:	(SIMPLE-ARRAY (UNSIGNED-BYTE 32))	9	9920	0.3%
108:	(SHORT-SIMPLE-ARRAY CODE)	16	7368	0.2%
10:	HASH-TABLE	108	3456	0.1%
17:	DOUBLE-FLOAT	120	1920	0.1%
111:	(SHORT-SIMPLE-ARRAY FOREIGN)	51	1216	0.0%
16:	SINGLE-FLOAT	141	1128	0.0%
118:	(SIMPLE-ARRAY BIT)	11	296	0.0%
20:	COMPLEX	11	176	0.0%
80:	(ARRAY T)	7	168	0.0%
11:	READTABLE	8	128	0.0%
123:	(SIMPLE-ARRAY (SIGNED-BYTE 32))	1	88	0.0%
13:	SYSVECTOR	3	48	0.0%
85:	(ARRAY CHARACTER)	1	24	0.0%

total bytes = 3292344

aclmalloc arena:

	max size	free bytes	used bytes	total
	112	3472	112	3584
	496	3472	496	3968
	1008	2016	2016	4032
	2032	0	12192	12192
	4080	0	8160	8160
	9200	18400	18400	36800
total bytes:		27360	41376	68736

CL-USER(2):

Newspace and oldspaces

Newspace is divided into two equal size parts (only one of which is used at any time). There can be numerous oldspaces Two are shown in the example, but many more are common after Lisp has run for a while). Oldspaces are numbered. The `gsgc-` parameter `:open-old-area-fence` takes such a number as an argument (see [Section 5.1 Parameters that control generations and tenuring](#) for information on `gsgc-` parameters).

The 0th old area in the output is closed, as indicated by the asterisk. If there are no closed old areas (i.e. (`sys:gsgc-parameter :open-old-area-fence`) returns 0) then no asterisks show up and the "*" = closed old area" note isn't given. When asterisks are shown, they denote any old areas that are closed.

Root pages

Root pages: 61

Root pages contain information about pointers from oldspace to newspace.

Lisp heap, C heap, and Pure space

```
Lisp heap:  #x10000000  pos: #x106a0000  resrve: #x10fa0000
C heap:     #x64000000  pos: #x64011000  resrve: #x640fa000
Pure space: #x2d1aa000  end: #x2d747ff8
```

The first value is the starting address of the specified heap in memory. The 'Pure space' line only appears in Lisps which use a pll file (see **pll-file**), showing where the pll files is mapped.

In the first two lines, **pos** (position) is the highest-use location; this is one byte larger than the highest memory used by the lisp. Some operating systems will only commit (assign physical pages) to memory between base (inclusive) and position (exclusive). This is a hexadecimal address value. **resrve** (reserved) is the number of bytes lisp thinks is reserved to it in virtual memory space. On some operating systems which support it, addresses greater than position, but less than starting location+reserved, will not be overwritten by shared-libraries, other memory mapping operations, etc.

The Lisp heap reserved size is a true limit only for certain free products. With paid license images (and some free products), this value is important only because if the heap grows larger than this limit, gaps in the heap may appear. See [Section 1.7 The almost former gap problem](#) for more information. This value is not a limit in any sense on how big the image can grow.

The name "C heap" is something of a misnomer. It is actually the **aclmalloc** area used for space allocated by the **aclmalloc** function. That function differs from **malloc()** in that it ensures that Lisp will remember the location of **aclmalloc**'ed allocations and preserve it through **dumplisp** and restarts, thus guaranteeing that **aclmalloc** addresses remain valid.

More information on **aclmalloc** and regular **malloc()**:

- **aclmalloc** space must be located at the same spot as it starts out in a **generate-application** or **build-lisp-image** (where it is specified by the *c-heap-start* argument). If the c-heap cannot be located at the same place as before, the Lisp startup will fail. This is actually comforting, because it means that addresses used for **aclmalloc** locations are guaranteed to always be valid and unchanged.
- **malloc()** space is always started fresh when a Lisp starts up, so addresses used by **malloc** calls must never be used across **dumplisp**s.
- The lisp interface to **aclmalloc()** is **aclmalloc**. There is no Lisp interface to **malloc()** (there is an internal function called **excl::malloc**, but it calls **aclmalloc()**, and not **malloc()**, which is why it is not exported). Foreign definitions for **malloc()** and **free()** can be made in order to gain access to these functions. However, calling **malloc()** in a Lisp environment is dangerous, and should only be done after careful consideration of the above.

Type counts

The type counts are as if printed by **print-type-counts**:

code	type	items	bytes
112:	(SIMPLE-ARRAY T)	6586	930920 28.3%
1:	CONS	80502	644016 19.6%
8:	FUNCTION	8432	520192 15.8%

7: SYMBOL	17272	414528	12.6%
117: (SIMPLE-ARRAY CHARACTER)	2259	259984	7.9%
96: (SHORT-SIMPLE-ARRAY T)	17498	151736	4.6%
18: BIGNUM	2966	139928	4.3%
125: (SIMPLE-ARRAY (UNSIGNED-BYTE 8))	31	87816	2.7%
12: STANDARD-INSTANCE	3291	52656	1.6%
9: CLOSURE	2301	39688	1.2%
15: STRUCTURE	666	24944	0.8%
127: (SIMPLE-ARRAY (UNSIGNED-BYTE 32))	9	9920	0.3%
108: (SHORT-SIMPLE-ARRAY CODE)	16	7368	0.2%
10: HASH-TABLE	108	3456	0.1%
17: DOUBLE-FLOAT	120	1920	0.1%
111: (SHORT-SIMPLE-ARRAY FOREIGN)	51	1216	0.0%
16: SINGLE-FLOAT	141	1128	0.0%
118: (SIMPLE-ARRAY BIT)	11	296	0.0%
20: COMPLEX	11	176	0.0%
80: (ARRAY T)	7	168	0.0%
11: READTABLE	8	128	0.0%
123: (SIMPLE-ARRAY (SIGNED-BYTE 32))	1	88	0.0%
13: SYSVECTOR	3	48	0.0%
85: (ARRAY CHARACTER)	1	24	0.0%

total bytes = 3292344

Aclmalloc arena

The aclmalloc arena describes allocation of space for **acmmallocs** and foreign data. It is divided into chunks of various sizes to allow allocation of requests of various sizes without fragmentation. (Space allocated by **acmmalloc** is freed by **acmfree**.)

acmmalloc arena:

max size	free bytes	used bytes	total
112	3472	112	3584
496	3472	496	3968
1008	2016	2016	4032
2032	0	12192	12192
4080	0	8160	8160
9200	18400	18400	36800
total bytes:	27360	41376	68736

3.0 Tuning the garbage collector

As a user, or as an application writer, how can you get the garbage collector to work best for you? At first, you do not have to do anything. The system is set up to work as delivered. You will not run out of space, global gc's will happen from time to time (as described below, see [Section 6.0 Global garbage collection](#)), the image will grow as necessary, and assuming you do not run out of swap space, everything will work.

Of course, it will not necessarily work as well as it could. As delivered, the garbage collector is set to work best with what we assume is a typical application: objects, none of which are too big, are created as needed. Most objects that survive a while are likely to survive a long while or perhaps forever, and so on. If your application's use of Lisp has different behavior, performance

may be suboptimal.

So what to do? One problem is that optimizing gc behavior is a multidimensional problem. Factors that affect it include

1. things internal to Lisp (the initial sizes of oldspace and newspace, the various products included in the image, whether shared libraries or static text is used for certain constants or whether the constants are part of the Lisp heap);
2. environmental factors (amount of physical memory, amount of swap space, usual running environment -- i.e. runs with lots of other programs or runs on a dedicated machine, whether paging is done over the network); and
3. details of what your code or application does.

Optimization in a multidimensional environment is always complicated.

The first step is always to gather the information necessary to do the tuning. Information like:

[Section 3.1 How do I find out when scavenges happen?](#)

[Section 3.2 How many bytes are being tenured?](#)

[Section 3.3 When there is a global gc, how many bytes are freed up?](#)

[Section 3.4 How many old areas are there after your application is loaded?](#)

[Section 3.5 Can other things be changed while running?](#)

3.1 How do I find out when scavenges happen?

There are three gsgc switches (these control the behavior of the garbage collector) that affect printing information about the garbage collector: `:print`, `:stats`, and `:verbose`. Doing

```
(setf (sys:gsgc-switch :print) t)
```

will cause a short message to be printed whenever a scavenge happens. Unless the `:print` switch is `t`, no message will be printed.

The `:stats` and `:verbose` switches control the amount of information printed. If the `:stats` switch is true, the message contains more information but the information is compact. If the `:verbose` switch is also true, a longer, more easily understood message is printed.

```
;; In this example, we cause a scavenge with all flags off,
;; then with :print true, then :print and :stats true,
;; and finally :print, :stats, and :verbose all true.
```

```
cl-user(5): (gc)
cl-user(6): (setf (sys:gsgc-switch :print) t)
t
cl-user(7): (gc)
gc: done
cl-user(8): (setf (sys:gsgc-switch :stats) t)
t
cl-user(9): (gc)
gc: E=17% N=17536 O+=0
cl-user(10): (setf (sys:gsgc-switch :verbose) t)
t
cl-user(11): (gc)
```

```
scavenging...done eff: 15%, copy new: 1664 + old: 16064 = 17728
Page faults: gc = 0 major + 2 minor
cl-user(12):
```

With just `:print true`, a very short message is printed. With `:stats true`, the message contains much more information, but it is coded -- E means Efficiency, N means bytes copied in newspace, O means bytes copied to oldspace. With `:verbose` also true, the same information is displayed in expanded form and additional information (about page faults) is provided.

Efficiency is defined as the ratio of cpu time not associated with gc to total cpu time. Efficiency should typically be 75% or higher, but the efficiencies in the example are low because we triggered gc's without doing anything else of significance.

It is usually desirable to have `:print` and `:stats` true while developing software. This allows you to monitor gc behavior and see if there seems to be a problem.

3.2 How many bytes are being tenured?

That information is shown when the `:print` and `:stats` switches are true, but perhaps the real question is whether things are being tenured that would be better left in newspace (because they will soon become garbage). This often happens when a complex operation (like a compile of a large file) is being carried out. This question, in combination with the next can tell you if that is the case.

In the following, copied from above, 0 bytes are tenured in the first gc (`O+=0`) and 16064 in the second (`old: 16064`):

```
cl-user(9): (gc)
gc: E=17% N=17536 O+=0
cl-user(10): (setf (sys:gsgc-switch :verbose) t)
t
cl-user(11): (gc)
scavenging...done eff: 15%, copy new: 1664 + old: 16064 = 17728
Page faults: gc = 0 major + 2 minor
cl-user(12):
```

3.3 When there is a global gc, how many bytes are freed up?

If the `:print` and `:stats` switches are true, the amount of space freed by a global gc is printed at the end of the report. Here is an example. The form `(gc t)` triggers a global gc.

```
cl-user(13): (gc t)
gc: Mark Pass...done(1,583+66), marked 128817 objects, max depth = 17, cut 0 xfers.
Weak-vector Pass...done(0+0).
Cons-cell swap...done(0+67), 346 cons cells moved
Symbol-cell swap...done(17+0), 1 symbol cells moved
Oldarea break chain...done(83+0), 40 holes totaling 6816 bytes
Page-compactation data...done(0+0).
Address adjustment...done(1,400+67).
Compacting other objects...done(150+0).
Page compactation...done(0+0), 0 pages moved
```

```
New rootset...done(667+0), 20 rootset entries
Building new pagemap...done(83+0).
Merging empty oldspaces...done, 0 oldspaces merged.
global gc recovered 9672 bytes of old space.
gc: E=0% N=1504 O+=0 pfg=54+187
cl-user(14):
```

The next to last line reports on what was recovered from oldspace (9672 bytes). The value is often much higher. It is low in this example because we have not in fact done anything significant other than test gc operations.

There is plenty of other information but we will not describe its meaning in detail. It is typically useful in helping us help you work out complicated gc problems.

The amount of space freed is a rough measure of how many objects are being tenured that perhaps should be left for a while longer in newspace. If the number is high, perhaps things are being tenured too quickly (increasing the value of the : generation-spread switch will keep objects in newspace longer, as will a larger newspace).

3.4 How many old areas are there after your application is loaded?

The output printed by **room** shows the two newspace areas and the various oldspace areas. Here is an example of **room** output. (**room** takes an argument to indicate how much information should be displayed). The following is the output of (cl:room t), which causes the most information to be displayed.

```
cl-user(3): (room t)
area area  address(bytes)          cons          other bytes
#  type                                8 bytes each
                                     (free:used)    (free:used)
Top #x569a000
New #x5134000(5660672)          5:95781      597040:4239616
New #x4bce000(5660672)          -----
7 Old #x498e000(2359296)       458:18903    31904:2170416
6 Old #x494e000(262144)        0:1019       0:253648
5 Old #x478e000(1835008)       0:41779     0:1498064
4 Old #x474e000(262144)        0:23437     0:73424
3 Old #x45ce000(1572864)       0:27513     0:1350736
2 Old #x454e000(524288)        0:7133      0:466512
1 Old #x448e000(786432)        0:4076      0:753104
0 Old #x4000d00(4772608)       0:97824     0:3983672
Tot (Old Areas)                458:221684   31904:10549576
Root pages: 158
Lisp heap: #x4000000 pos: #x569a000 resrve: 23699456
C heap:    #x5400000 pos: #x54027000 resrve: 1024000
```

```
code  type                items    bytes
96: (simple-array t)      76658   3864816 22.8%
108: (simple-array code)  8699    3608136 21.3%
1: cons                   314901  2519208 14.9%
99: (simple-array (unsigned-byte 16)) 10938   2242320 13.2%
101: (simple-array character) 38383   1632920 9.6%
8: function               21721   1284216 7.6%
```

7: symbol	36524	876576	5.2%
107: (simple-array (signed-byte 32))	264	264336	1.6%
12: standard-instance	14244	227904	1.3%
9: closure	8854	145448	0.9%
98: (simple-array (unsigned-byte 8))	44	105184	0.6%
97: (simple-array bit)	49	103952	0.6%
15: structure	830	33144	0.2%
100: (simple-array (unsigned-byte 32))	12	10264	0.1%
10: hash-table	225	7200	0.0%
18: bignum	410	4480	0.0%
16: single-float	505	4040	0.0%
111: (simple-array foreign)	103	2464	0.0%
17: double-float	124	1984	0.0%
64: (array t)	22	528	0.0%
65: (array bit)	13	312	0.0%
13: sysvector	14	224	0.0%
20: complex	12	192	0.0%
11: readtable	7	112	0.0%
69: (array character)	1	24	0.0%

total bytes = 16939984

aclmalloc arena:

max size	free bytes	used bytes	total
48	3024	48	3072
496	3968	0	3968
1008	4032	0	4032
2032	2032	2032	4064
4080	8160	36720	44880
5104	10208	10208	20416
9200	27600	9200	36800
20464	20464	20464	40928
total bytes:	79488	78672	158160

cl-user(4):

The output shows the two equal size newspace areas, only one of which is being used. It also shows eight oldspaces and provides information about what is in the oldspaces. Then information is printed about other objects such as the number of root pages (a root page keeps information on pointers from oldspace to newspace -- these pointers must be updated after a scavenge), and the locations of the Lisp and C heaps. Then, there is a table showing the types and numbers of objects. Finally, used and available malloc space is displayed.

3.5 Can other things be changed while running?

Yes. The function **resize-areas** can be used to rearrange things while running. It is typically useful to call this function, for example, after loading a large application. If you know desirable old- and newspace sizes for your application, it is preferable to build an image with those sizes (using the `:oldspace` and `:newspace` arguments to **build-lisp-image**, see *building-images.htm* for more information). However, you may not know until runtime what the best sizes are, in which case you can call **resize-areas** on application startup. Be warned that it may take some time.

Another use of **resize-areas** is when you wish to dump an image (with **dumplisp**) into which your application has been loaded. You call **resize-areas** just before **dumplisp** in that case.

4.0 Initial sizes of old and new spaces

The initial sizes of newspace and oldspace are determined when the image is built with **build-lisp-image**. See *building-images.htm* (where **build-lisp-image** is fully documented -- the page on it is brief to avoid two sources for the same complex discussion) The `:newspace` argument to **build-lisp-image** controls the initial size of newspace and the `:oldspace` argument the initial size of oldspace.

An image dumped with **dumplisp** inherits new and oldspace sizes from the dumping image. See *dumplisp.htm*.

resize-areas will restructure old and newspace sizes in a running image.

The garbage collector will automatically resize old and newspace when it needs to. The amount of resizing depends on space required to allocate or move to oldspace live objects, and also on the parameters that relate to sizes.

5.0 System parameters and switches

The parameters and switches described under the next set of headings control the action of the garbage collector. You may change them during run time to optimize the performance of the Lisp process. All parameters and switches values may be set with **setf**. However, some values should not be changed by you. The descriptions of the parameters say whether you should change their values. By default, the system does automatically increase the generation number. You may find that it is useful to step it yourself at appropriate times with a call to **gsgc-step-generation**.

There is really no difference between a parameter and a switch other than the value of switches is typically `t` or `nil` while parameters often have numeric values. However, once both were implemented, it became difficult to redo the design.

The function **gsgc-parameters** prints the values of all parameters and switches. **gsgc-switch** and **gsgc-parameter** retrieve the value of, respectively, a single switch or a single parameter, and with **setf** can set the value as follows.

```
(setf (sys:gsgc-parameter parameter) value)  
      (setf (sys:gsgc-switch switch) value)
```

Switches and parameters are named by keywords.

5.1 Parameters that control generations and tenuring

The first three parameters relate to the generation number and when objects are tenured. Please note that of the three parameters, you should only set the value of `:generation-spread`. The fourth parameter, which is setf'able, allows closing off some old areas, meaning that no objects will be tenured to them. Old areas are now numbered, allowing for some to be closed off.

:current-generation	The value of this parameter is a 16 bit unsigned integer. New objects are created with this generation tag. Its initial value is 1, and it is incremented when the generation is stepped. The system may change this value after a scavenge. Users should not set this value. Note: Both the current generation number and the generation of an individual object are managed in a non-intuitive way. While it is conceptually correct that the generation number increases, the actual implementation works quite differently, often resetting the generation number toward 0.
:tenure-limit	The value of this parameter is a 16 bit integer. During a scavenge, objects whose generation exceeds this value are not tenured and all the rest are tenured. Users should not set this value. Its initial value is 0, and it is constantly being reset appropriately by the system.
:generation-spread	The value of this parameter is the number of distinct generations that will remain in newspace after garbage collection. Note: objects that are marked for tenuring and objects that are to stay in newspace permanently do not belong to a specific generation. Setting the value of this parameter to 0 will cause all data to be tenured immediately. This is one of the most important parameters for users to set. Its initial value is 4 and its maximum effective value is 25.
:open-old-area-fence	<p>The value of this parameter is always a non-negative integer which is the number of the oldest old area that is open (not closed). Old areas are numbered with 0 as the oldest old area.</p> <p>This parameter is settable, either with the number of the old-area that is desired to be the first open-old-area, or with a negative number, for which the old-areas are counted backward to set the fence. For example, <code>(setf (sys:gsgc-parameter :open-old-area-fence) -1)</code> will close all old areas except for the newest one, no matter how many old areas exist. Restrictions: At least one old area will be open at any time. Attempts to set the fence to a higher or lower number than can be set will result in silent saturation (setting the fence to the nearest possibility), however the return value from such a <code>setf</code> will always indicate the actual open-old-area-fence. Obviously from these restrictions, one must have at least two old-areas in order to close one.</p> <p>See the note on closed old areas just after this table for more information.</p>

Further information on closed old areas

Old areas can be marked as closed. When an old area is closed, no objects are newly tenured into a closed old-area; it is as if the area is full. Also, no dead object in a closed old area is collected while the area is closed, and data pointed to by that object is also not collected.

See the description of the `:open-old-area-fence` in the table just above for details on how to specify old areas as closed.

The intended usage model for closing old areas is this: a programmer with an application, such as a VAR, will load up their application, perform a `global-gc` and possibly a **resize-areas**, and then close most of the old-areas, leaving room for their users' data models to be loaded into the open-areas. When the user is done with the data model, it can be thrown away and a fast `global-gc` performed, making way for the next data model.

5.2 Parameters that control minimum size

The following parameters control the minimum size of newspace and when the system will allocate a new newspace. At the end of a scavenge, at least `:free-bytes-new-pages` plus `:free-bytes-new-other` bytes must be free, and at least `:free-percent-new` percent of newspace must be free. If these conditions are not met, the system will allocate a new newspace, large enough for these conditions to be true after allocating the object that caused the scavenge, if there is one. (Unless explicitly called by the user, a scavenge occurs when the system is unable to allocate a new object.) Note that there is no system reason why there are two parameters, `:free-bytes-new-pages` and `:free-bytes-new-other`. Their values are added to get total free space required.

<code>:quantum</code>	The value of this parameter is a 32-bit integer which represents the minimum amount of space (in 8 Kbyte pages) that will be requested for a new new or old space and the granularity of space requested (that is space will be requested in multiples of <code>:quantum</code> pages). Its initial value is 32. This parameter value is overshadowed by the other size-related parameters described immediately below, and for that reason, we do not recommend that you change this value.
<code>:free-bytes-new-other</code>	This is one of the parameters which determine the minimum free space which must be available after a scavenge. Its initial value is 131072.
<code>:free-bytes-new-pages</code>	This is one of the parameters which determine the minimum free space which must be available after a scavenge. Its initial value is 131072.
<code>:free-percent-new</code>	This parameter specifies the minimum fraction of newspace which must be available after a scavenge, or else new newspace will be allocated. Its initial value is 25.

5.3 Parameters that control how big newly allocated spaces are

The final two parameters control how large new newspace (and new oldspace) will be. If newspace is expanded or a new oldspace is allocated, then at least the percentage specified by the appropriate parameter shall be free, after, in the case of newspace, the object that caused the scavenge has been allocated, and after, in the case of oldspace, all objects due for tenuring have been allocated. There are different concerns for the newspace parameter and the oldspace parameter.

Let us consider the oldspace parameter first. In the case where no foreign code is loaded, then oldspaces are carved out of newspace, and newspace grows up into memory as needed. If each new oldspace is just large enough, the next time an object is tenured, another oldspace, again just large enough, will be created, and the result will be a bunch of small oldspaces, rather than a few larger ones. This problem will not occur if there is foreign code, since some oldspaces will be as large as previous newspaces. If the function room shows a bunch of little oldspaces, you might try increasing the `:expansion-free-percent-old` parameter to cure the problem. However, **resize-areas** can be used instead to coalesce the oldspaces into one.

The newspace parameter is more complicated, since newspace can grow incrementally (assuming growth is not blocked by foreign code). Since growing newspace takes time, you want to ensure that when newspace grows, it grows enough. Therefore, it is essential that `:expansion-free-percent-new` be larger than `:free-percent-new`. Otherwise, you might find newspace growing just enough to satisfy `:free-percent-new`, and then having to grow again at the next scavenge, since allocating a new object again reduced the free space below the `:free-percent-new` threshold.

<code>:expansion-free-percent-new</code>	At least this percentage of newspace must be free after allocating new newspace. The system will allocate sufficient extra space to guarantee that this condition is met. Its initial value is 35.
--	--

`:expansion-free-percent-old`

At least this percentage of space must be free in newly allocated oldspace (note: not the total oldspace). Its initial value is 35.

5.4 Gsgc switches

There are several switches which control the action of `gsgc`. The value of a switch must be `t` or `nil`. The function **`gsgc-switch`** takes a switch name as an argument and returns its value or with **`setf`**, sets its value. **`gsgc-parameters`** also prints out their values. The switches can be set by evaluating the form

```
(setf (sys:gsgc-switch switch-name) nil-or-non-nil)
```

<code>:gc-old-before-expand</code>	If this switch is set true, then before expanding oldspace, the system will do a global garbage collection (that is, it will gc oldspace) to see if the imminent expansion is necessary. If enough space is free after the garbage collection of oldspace the expansion will not occur. Initially <code>nil</code> .
<code>:print</code>	If true, print a message when a gc occurs. Can be set by <code>excl:gc</code> . The length of the message is determined by the next two switches. If both are <code>nil</code> , a minimal message is printed. See just below the table for examples. Note that the messages are printed to the Console window on Windows and are not seen in the Integrated Development Environment Debug window.
<code>:stats</code>	If true and <code>:print</code> is true, print statistics (such as how many bytes tenured) about the gc. See just below the table for examples.
<code>:verbose</code>	If true, make the message printed (when <code>:print</code> is true) more English-like. <code>:clip-new</code> messages are only printed when this and <code>:print</code> are true. See just below the table for examples.
<code>:auto-step</code>	This is the most important of the switches. If true, which is its initial value, <code>gsgc-step-generation</code> is effectively called after every scavenge. Thus (with the default <code>:generation-spread</code>) an object is tenured after surviving four scavenges.
<code>:hook-after-gc</code>	If this switch is true, the function object bound to the variable <code>*gc-after-hook*</code> will be funcalled after every scavenge or global gc.
<code>:next-gc-is-global</code>	<p>If this switch is set true, the next gc will be a global gc (that is both newspace and oldspace will be gc'ed). After the global gc, the system will reset this switch to <code>nil</code>. This will happen even if you cause the global gc by evaluating the form <code>(excl:gc t)</code>.</p> <p>The difference between setting this switch and causing a global gc explicitly with the function <code>excl:gc</code> is that setting this switch causes the system to wait until a scavenge is necessary before doing the global gc while calling the function causes the global gc to occur at once. The system uses this switch under certain circumstances.</p>

:clip-new	<p>The scavenger maintains a new logical pool of memory in newspace called `reserved'. When the <code>:clip-new</code> switch is true, memory will be transferred between `free' and `reserved' to try to maintain the freespace in use at the levels specified by the <code>gsgc</code> size parameters. If newspace grows to 5 meg because of a flurry of allocation, for example, a subsequent scavenge that finds only 200K of live data would put lots of space into the `reserved' bucket. The next scavenge would then happen much sooner. This is supposed to improve locality. Note that if there is sufficient RAM to hold the image, the performance improvement caused by improved locality of reference will most likely be offset by the need for more frequent scavenges.</p> <p>If <code>:print</code> and <code>:verbose</code> are both true, information about the action triggered by this switch is printed. The information refers to `hiding' (moving space to the reserved bucket) and `revealing' (moving space to the free bucket).</p>
:use-remap	<p>If this switch is set true and the operating system on which Allegro CL is running supports it, then physical memory pages that are no longer used after a garbage collection are given back to the operating system in such a fashion that paging is improved.</p> <p>Specifically, when this switch is true and the currently used half of newspace is copied to the unused half, the following things are done with the previously-used memory area: (1) the operating system is advised to ignore the page reference behavior of those addresses, and (2) the memory is unmapped and then is remapped, after being filled with zeros. The zero filling is necessary for security reasons, since the memory given back to the operating system will potentially be given to another process that requests virtual memory, without first being cleared. If it were not for (2), then remapping would always be advantageous and there would be no switch to control this behavior. As it is, there may be certain situations where zero filling will be too expensive, especially on machines which have a very large amount of physical memory and the decrease in locality does not effect the runtime performance of the Allegro CL application, or where the <code>mmap()</code> implementation is flawed.</p>
:dump-on-error	<p>If this switch is set true, then a core dump is automatically taken when an internal garbage collection error occurs. The core dump will fail, however, if (1) there is a system imposed limit on the size of a core dump and dumping the image would exceed this limit or (2) there is some other system impediment to dumping core, such as the existence of a directory named <code>core</code>. We assume that you can prevent the second limitation. Here are a few more words on the first limitation. In the C shell, the <code>limit</code> command and its associates can be used to set a higher limit or no limit for the maximum size of a core dump.</p> <p>If the value of this switch is <code>nil</code>, you will be asked if you want a core dump when there is a <code>gsgc</code> error. Note that whatever the value, you have the opportunity to get a core dump. Set the switch to true when, for example, you intend to leave a program running all night in order to trigger a failure.</p>

These examples show the effect on gc messages of `:stats` and `:verbose` being true. When `:print` is true but both `:stats` and `:verbose` are `nil`, a message like the following is printed during a scavenge:

```
gc: done
```

When `:verbose` is also true but `:stats nil`, the message is:

```
scavenging...done
```

When `:stats` is true and `:verbose nil`, the message is similar to:

```
gc: E=34% N=30064 O+=872 pfu=0+101 pfg=1+0
```

The same message with `:verbose true` would be:

```
scavenging...done eff: 34%, copy new: 30064 + old 872 = 30936
Page faults: non-gc = 0 major + 101 minor, gc = 1 major + 0 minor
```

When `:verbose` is `nil`, abbreviations are used. Their meanings are explained when `:verbose` is true.

E or eff. is efficiency: the ratio of non-gc time and all time (the efficiency is low in our example because we forced gc's in order to produce the example; as we say elsewhere, efficiencies of less than 75% are a cause for concern when the gc is triggered by the system).

The copy figures are the number of bytes copied within newspace and to oldspace.

X means "expanding", so XO means "expanding oldspace" and XN means "expanding newspace". XMN means "expanding and moving newspace".

Page faults are divided between user (pfu or non-gc) caused and gc (pfg or gc) caused. See the Unix man page for **getrusage** for a description of the difference between major and minor page faults.

Here are a couple of more examples (with `:verbose` on and off in a fresh Lisp each time):

```
cl-user(1): (gc :print)
t
cl-user(2): (setf (sys:gsgc-switch :verbose ) t)
t
[...]
cl-user(7): (defconstant my-array (make-array 10000000))
scavenging...expanding new space...expanding and moving new space...done
  eff: 36%, copy new: 7533984 + old: 85232 = 7619216
  Page faults: non-gc = 1 major + 0 minor
my-array
```

;; And in a fresh image with `:verbose off`:

```
cl-user(1): (gc :print)
t
cl-user(2): (defconstant my-array (make-array 10000000))
gc: XN-XMN-E=32% N=7522488 O+=85632 pfu=4+0
my-array
```

5.5 Gsgc functions and variables

Function or variable	Arguments of functions	Brief Description
gsgc-step-generation		Calling this function, which returns the new value of <code>:current-generation</code> , increases the current generation number and, if necessary, the value of <code>:tenure-limit</code> as well.
gc	&optional <i>action</i>	Called with no arguments, perform a scavenge; called with argument <code>t</code> , perform a global gc. Other arguments cause setting of switches and the printing of information. Argument <code>:help</code> shows other arguments and their effects as does the gc page.
print-type-counts	&optional (<i>location t</i>)	Prints a list of quantities and sizes of lisp objects in the specified location in the heap, along with type names and type codes of each object type printed. See the print-type-counts page for location details.
lispval-storage-type	<i>object</i>	Returns a keyword denoting where object is stored. See the lispval-storage-type page for interpretation of the returned value and examples. (In earlier releases, the function pointer-storage-type performed this function. It is still supported, but its use is deprecated. lispval-storage-type is more flexible and should be used instead.)
resize-areas	&key <i>verbose old old-symbols new global-gc tenure expand sift-old-spaces pack-heap</i>	This function resizes old and newspaces, perhaps coalescing oldspaces, according to the arguments. See the resize-areas page for details.
<code>*gc-after-hook*</code>		If the gsgc switch <code>:hook-after-gc</code> is true, then the value of this symbol, if true, will be funcalled immediately after a scavenge. See the description of <code>*gc-after-hook*</code> for details.
gc-after-c-hooks		Returns a list of addresses of C functions that will be called after a gc. See gc-after-c-hooks for details.
gc-before-c-hooks		Returns a list of addresses of C functions that will be called before a gc. See gc-before-c-hooks for details.

6.0 Global garbage collection

In a global garbage collection (global gc), objects in oldspace are garbage collected. Doing so frees up space in oldspace for newly tenured objects. Global gc's are time consuming (they take much longer than scavenges) and they are not necessary for Lisp to run.

The effect of never doing a global gc is the Lisp process will slowly grow larger. The rate of growth depends on what you are

doing. The costs of growth are that the paging overhead increases and, if the process grows too much, swap space is exhausted, perhaps causing Lisp to stop or fail.

You have complete control over global gc's. The system will keep track of how many bytes have been tenured since the last global gc. You can choose one of these options for global gc:

1. you can cause the system to print a message when a specified number of bytes have been tenured recommending that you cause a global gc;
2. you can cause the system to trigger a global gc automatically when that number of bytes have been tenured;
3. you can have the system trigger a global gc and print a message (the default behavior); or
4. you can have the system do nothing at all about global garbage collection.

The function that records how many bytes have been tenured since the last global gc is the default value of the variable `*gc-after-hook*`. If you set that variable (whose value must be a function or `nil`) to `nil` or a function that does not keep records of bytes tenured, you will not get the behavior described here. (See the description of `*gc-after-hook*` for information on defining a function that does what you want and records bytes tenured correctly.)

If `*gc-after-hook*` has as its value its initial value or a function that records bytes tenured correctly, global gc behavior is controlled by the global variable `*global-gc-behavior*`.

The variable `*tenured-bytes-limit*` is used in conjunction with `*global-gc-behavior*`. The number of bytes tenured (moved to oldspace) since the last global gc is remembered and the `*global-gc-behavior*` depends on when `*tenured-bytes-limit*` is exceeded.

6.1 The tenuring macro

The **tenuring** macro causes the immediate tenuring (moving to oldspace) of all objects allocated while within the scope of its body. This is normally used when loading files, or performing some other operation where the objects created by forms will not become garbage in the short term. This macro is very useful for preventing newspace expansion.

7.0 Gc cursors

It is useful if possible to provide some sort of cue while garbage collections are occurring. This allows users to know that a pause is caused by a gc (and not by an infinite loop or some other problem). Typical cues include changing the mouse cursor, printing a message, or displaying something in a buffer as Emacs does when the emacs-lisp interface is running.

Unfortunately, providing such a cue for every scavenge is a difficult problem and if it is done wrong, the consequences to Lisp can be fatal. However, we have provided an interface for the brave. The functions **gc-before-c-hooks** and **gc-after-c-hooks** return settable lists of addresses of C functions to be called before and after a gc.

Luckily, scavenges are usually fast and so failing to provide a cue may not be noticeable. Global gc's, however can be slow but it is possible to provide a cue for a global gc even without using C functions. There are two strategies: (1) determine when a global gc is necessary and either schedule it when convenient or warn the user that one is imminent; (2) do not give advance warning but provide a cue when it happens. Looking at these examples, you can probably craft your own method of warning or cue.

Note that in these examples, we replace the value of `*gc-after-hook*` with a new value, destroying the current value

(which provides for the default automated behavior). The default function is named by the (internal) symbol `excl::default-gc-after-hook`.

One way to implement the first strategy is to set a flag when a global gc is needed and then have code that acts on that flag. This code can be run at your choosing -- but be sure that it is run at some point. You might do this:

```
(defvar *my-gc-count* 0)
(defvar *time-for-gc-p* nil)
(defun my-gc-after-hook (global-p to-new to-old eff to-be-alloc)
  (declare (ignore eff to-new to-be-alloc))
  (if global-p
      (progn (setq *my-gc-count* 0)
             (setq *time-for-gc-p* nil))
      (progn (setq *my-gc-count* (+ *my-gc-count* to-old))
             (if (> *my-gc-count* excl:*tenured-bytes-limit*)
                 (setq *time-for-gc-p* t))))))
```

Make sure you compile **my-gc-after-hook** before making it the value of `*gc-after-hook*`. Now, define a function that triggers a global gc (calls `(gc t)`) when `*time-for-gc-p*` is true. This function can be called by a user of your application, or when your application is about to do something that the user expects to wait for anyway, or whenever, so long as it is called at some point.

In the second strategy, we provide some cue to the user that a global gc is occurring. We have not included the code for the cue (you should supply that) and notice we have gone to some pains to avoid a recursive error (where the garbage collector calls itself).

```
(defvar *my-gc-count* 0)
(defvar *prevent-gc-recursion-problem* nil)
(defun my-gc-after-hook (global-p to-new to-old eff to-be-alloc)
  (declare (ignore eff to-new to-be-alloc))
  (when (null *prevent-gc-recursion-problem*)
    (if global-p (setq *my-gc-count* 0)
        (progn (setq *my-gc-count* (+ *my-gc-count* to-old))
               (if (> *my-gc-count* excl:*tenured-bytes-limit*)
                   (excl:without-interrupts
                    (setq *prevent-gc-recursion-problem* t)
                    ;; (<change the cursor, print a warning,
whatever>)
                    (gc t)
                    ;; (<reset the cursor if necessary>)
                    (setq *my-gc-count* 0)
                    (setq *prevent-gc-recursion-problem* nil))))))))
```

Note:

- Because the after hook is called whenever a gc (including a global gc) occurs, one wants to be sure that the code doesn't get into some kind of recursive loop. `*prevent-gc-recursion-problem*` achieves that.
- You replace these lines:

```
;; (<change the cursor, print a warning, whatever>)
;; (<reset the cursor if necessary>)
```

with whatever code you want, but be careful that there is no possibility of waiting (for user input, e.g.) or going into an infinite loop because you are in a **without-interrupts** form and waiting is wrong and an infinite loop is fatal in that case.

8.0 Other gc information

The following list contains information and advice concerning gsgc. Some of the information has already been provided above, but is given again here for emphasis.

1. While a garbage collection is happening, the handling of asynchronous signals (such as results from entering the Unix interrupt character or from certain mouse actions in a window system) is blocked. The signals are queued (with duplicated signals thrown out) until the garbage collection has completed.
 2. Do not set the `:auto-step` switch to `nil` unless some other method of stepping the generation is enabled (including specific action by you). If objects are not tenured, newspace will grow, filling up with long-lived objects, and performance will degrade significantly.
 3. There is no way for you directly to cause a specific new object to be tenured immediately except by calling `gc` with the argument `:tenure` (which will cause *all* live objects to be tenured). There is no way to prevent a specific object from ever being tenured except by disabling generation stepping and thus preventing all objects from being tenured.
 4. Users should not set the value of the current generation or the tenure limit. The behavior of these values is algorithm-dependent in a non-intuitive way. Control is exercised with generation stepping and setting `:generation-spread`.
-

9.0 gc errors

It is not easy to cause a gsgc error. Such errors are usually catastrophic (often Lisp dies either without warning or with a brief message that some unrecognizable object was discovered). Once the garbage collector becomes confused, it cannot be straightened out.

Such errors can be caused when Lisp code is compiled with the compiler optimizations set so that normal argument and type checking is disabled. For example, if a function is compiled with the values of speed and safety such that the compiler:verify-argument-count-switch is `nil`, and that function is passed the wrong number of arguments (usually too few), it can trigger a fatal gsgc error. Before you report a gsgc error as a bug (and please do report them), please recompile any code where checking was disabled with settings of speed and safety which allow checking. See if the error repeats itself. See *Declarations and optimizations* in *compiling.htm* for more information on compiler optimization switches and values of speed and safety.

Garbage collector errors may also be caused by foreign code signal handlers. Note that foreign code signal handlers should not call `lisp_call_address` or `lisp_value`. See *foreign-functions.htm* for more information on signals.

See the information on the `:dump-on-error` gsgc-switch in [Section 5.4 Gsgc switches](#).

9.1 Storage-condition errors

The Allegro CL image will grow as necessary while running. If it needs to grow and it cannot, a storage-condition type error is signaled (storage-condition is a standard Common Lisp condition type). While these errors might arise from insufficient swap

space, the typical cause is a conflict in the virtual address space. That is, something else (a program or a library location) has grabbed virtual address space in a range that Lisp needs to grow the heap. (Allegro CL does not allow discontinuous virtual address ranges.)

Whatever the cause, the error is triggered by a request for space which cannot be fulfilled. Here we show the error when a largish array is created (this example is contrived in order to show the error: a request for such an array does not typically cause a problem).

```
CL-USER(25): (setq my-array (make-array 100000))
Error: An allocation request for 483032 bytes caused a need for 12320768 more
bytes of heap.
The operating system will not make the space available.
[condition type: STORAGE-CONDITION]
[1] CL-USER(26):
```

A global gc may free up enough space within Lisp to continue without growing. Killing processes other than Lisp may free enough space for Lisp to grow. But it may be the case that other allocations of virtual address space conflicts with Lisp usage. Please contact Franz customer support for assistance in determining whether this is the case if the problem persists.

You trigger a global gc by evaluating (see **gc**):

```
(gc t)
```

9.2 Gsgc failure

When the garbage collector gets confused, usually by following what it believes to be a valid pointer, but one that does not point to an actual Lisp object, Lisp fails with a two part message. The first part is a brief description of the specific problem. The second part is a general statement about the gc failures and the fact they cannot be recovered from, along with an opportunity of get a core file (which may be useful for later analysis). Here are some examples of the second part:

```
The gc's internal control tables may have been corrupted and Lisp
execution cannot continue.
```

[or]

```
The internal data structures in the running Lisp image have been
corrupted and execution cannot continue.
```

[then]

```
Check all foreign functions
and any Lisp code that was compiled with high speed and/or low safety,
as these are two common sources of this failure. If you cannot find
anything incorrect in your code you should contact technical support
for Allegro Common Lisp, and we will try to help determine whether
this is a coding error or an internal bug.
```

```
Would you like to dump core for debugging before exiting(y or n)?
```

Here are some examples of the first part:

```
system error (gsgc): Unknown object type at (0xc50ec9a)
```

```
system error (gsgc):
  Object already pointing to target newspace half: 0x42c43400
```

```
system error (gsgc): Scavenger invoked itself recursively.
```

As the text says in the second part, there is no recovery.

The causes of such errors can be one or more of the following:

- a bad pointer introduced from foreign code
- foreign code inappropriately altering Lisp data
- an inappropriate call back from foreign code into Lisp
- incorrectly declared code when declarations are trusted
- a gc bug

Diagnosing and fixing the problem can be difficult. Here are some initial steps to take where possible:

- Determine if the problem is reliably repeatable (do you get a failure every time or most times).
- Recompile with compiler settings speed 1 and safety and debug 3 (see *Declarations and optimizations in compiling.htm*). This will suppress optimizations and most type declarations. If a variable has been improperly typed (so, when declarations are believed, Lisp thinks it is an object of one type when it is in fact another type), the failure should go away. If it does, the problem reduces to finding the false declaration.
- Turn on gc printing, so you have a record of gc's up to the failure. (Evaluate `(gc :print)`, see **gc**.)
- Look at foreign code called by Lisp: are there call backs into Lisp, are Lisp values or pointers passed to the foreign code, or does the foreign code modify Lisp data. All these things are supported, but if done wrong, can cause problems.
- Make sure that you have all the available patches (see **sys:update-allegro**). There may be patches for gc bugs, and one may fix your problem.

If you cannot quickly determine the cause of the problem and a solution for it, contact Franz Inc. technical support at support@franz.com. Be sure to include the output of a call to **print-system-state**, and provide any information about foreign code, optimizations, etc. that you can. We may ask you for a core file (which, as said above, can be optionally generated when there is a failure).

10.0 Weak vectors, finalizations, static arrays, etc.

A Lisp object becomes garbage when nothing points to or references it. The way the garbage collector works is it finds and identifies live objects (and, typically, moves them somewhere). Whatever is left is garbage. Finalizations and weak vectors allow pointers to objects which will not, however, keep them alive. If one of these pointers exists, the garbage collector will see the item and (depending on the circumstances), either keep it alive (by moving it) or abandon it.

It is useful to distinguish two actions of the garbage collector. When the only pointers to an object are weak pointers or finalizations, the garbage collector follows those pointers and 'identifies the object as garbage'. If it decides not to keep the object alive, it 'scavenges the object away'. Note that any Lisp object can be scavenged away - that just means that the memory it used is freed and (eventually) overwritten with new objects. Only objects pointed to by a weak vector or a finalization can be identified as garbage, however. Live objects are not garbage and objects with nothing pointing to them are not even seen by the garbage collector.

10.1 Weak arrays and hashtables

Weak arrays are created with the standard Common Lisp function **make-array** (using the non-standard *weak* keyword argument) or (if a vector is desired) with the function **weak-vector**. When you create a weak array by specifying a true value for the *weak* keyword argument to **make-array**, you cannot also:

- Create a specialized array (`:element-type` should be `t` or unspecified).
- Specify a value for the `:displaced-to` keyword argument (weak arrays cannot be displaced into other arrays).
- Specify an `:allocation` other than `:new` (the non-standard *allocation* keyword argument allows specifying where the arrays will be created, with choices including foreign space, non-gc'ed Lisp space, or the Lisp heap, called for by `:new`, which is also the default).

When **make-array** successfully accepts a true value for the *weak* keyword argument, the object that is weak is always the underlying simple-vector; if the resultant array is non-simple or is multidimensional, then the array itself is not marked as weak, but objects in the array will still be dropped by the gc when not otherwise referenced, because the simple-array that is the data portion of the array is itself weak.

See the discussion of extensions to *make-array* (in *implementation.htm*) for further details on the Allegro CL implementation of **make-array** and the *weak* keyword argument.

As we said in the brief description above, the most important feature about weak arrays is that being pointed to by a weak array does not prevent an object from being identified as garbage and scavenged away. When an object is scavenged away, the entry in a weak array that points to the object will be replaced with `nil`.

Weak arrays allow you to keep track of objects and to discover when they have become garbage and disposed of. An application of weak arrays might be determining when some resource external to Lisp can be flushed. Suppose that all references to a file external to Lisp are made through a specific pathname. It may be that once all live references to that pathname are gone (i.e. the pathname has become garbage) the file itself is no longer needed and it can be removed from the filesystem. If you have a weak array which points to the pathname, when the reference is replaced by `nil` by the garbage collector, you can tell the system to kill the file.

It is important to remember that objects which have been tenured will not be identified as garbage unless a global gc is performed. If you use weak arrays, you should either arrange that global gc's are done regularly or that you do an explicit global gc before checking the status of an element of the weak array.

We provide a simple example of weak vectors (weak arrays differ from weak vectors only in having more dimensions) and finalizations in [Section 10.3 Example of weak vectors and finalizations](#).

Weak hashtables are also supported. See *implementation.htm* for information on an extension to **make-hash-table** that creates weak hashtables.

10.2 Finalizations

A finalization associates an object with a function. If the object is identified as garbage by the garbage collector, the function is called with the object as its single argument before the object is scavenged away. Multiple finalizations can be scheduled for the same object; all are run if and when the gc identifies the object as garbage. The order of the running of multiple finalizations is unspecified.

The timing is important here. While the garbage collector is running, nothing else can be done in Lisp. In particular, functions (except those internal to the garbage collector itself) cannot be called. Therefore, the process of running a finalization and the eventual reclamation of the finalized object occurs over two successive invocations of the garbage collector. During the first gc, the object is identified as garbage. The garbage collector sees the finalization and so, instead of immediately scavenging the object away, it keeps it alive and makes a note to call the finalization function as soon as it finishes the current scavenge (or global gc). The finalization is removed from the object after the function is run so that during the next garbage collection, the object is either garbage or, if a weak vector points to it, again identified as garbage and (since it no longer has a finalization) scavenged away. See the example in [Section 10.3 Example of weak vectors and finalizations](#).

A finalization is not a type of Lisp object. Finalizations are implemented as vectors (but that may change in a later release). You should not write any code which depends on the fact that finalizations are implemented as vectors.

You schedule a finalization with the function **schedule-finalization**. The function **unschedule-finalization** removes the finalization from the object. Finalizations are only run once, immediately after the garbage collection which identified the object as garbage. The object is not scavenged away during the garbage collection in which it is identified as garbage (since it must be around to be the argument to the finalization function).

10.3 Example of weak vectors and finalizations

On Windows, load the file after the transcript example. Typing the example into the Debug window in the Integrated Development Environment on Windows does not work as described, because tools in the IDE cause pointers to the objects to be preserved, and, as a result, the example does not work as described. On Unix, you can type the example directly to a prompt.

```
;; This example can be typed to a prompt on a UNIX platform.
;; We define a weak vector and a function to be called by finalizations.
CL-USER(10): (setq wv (weak-vector 1))
#(nil)
CL-USER(11): (defun foo (x) (format t "I am garbage!~%" x))
foo
;; We create an object and put it in the weak vector:
CL-USER(12): (setq a (cons 1 2))
(1 . 2)
CL-USER(13): (setf (aref wv 0) a)
(1 . 2)
;; We schedule a finalization.
CL-USER(14): (schedule-finalization a 'foo)
#((1 . 2) foo nil)
;; We remove the reference to the cons by setting the value of A to NIL.
CL-USER(15): (setq a nil)
nil
;; We evaluate unrelated things to ensure that the object
;; is not the value of *, **, or ***.
CL-USER(16): 1
1
CL-USER(17): 2
2
CL-USER(18): 3
3
;; We trigger a scavenge. The finalization function is called.
```

```
;; Note: an automatic gc might occur while you are entering the
;; forms shown. If it does, you might see the message printed by
;; our finalization sooner.
```

```
CL-USER(19): (gc)
```

```
I am garbage!
```

```
;; At this point, the weak vector still points to the object:
```

```
CL-USER(20): wv
```

```
#((1 . 2))
```

```
;; The next gc causes the value in the weak vector to
;; be changed.
```

```
CL-USER(21): (gc)
```

```
CL-USER(22): wv
```

```
#(nil)
```

On Windows, put this in a file `finalization.cl`:

```
(in-package :cg-user)
```

```
(setq wv (weak-vector 1))
```

```
(defun foo (x) (format t "I am garbage!~%"))
```

```
(setq a (cons 1 2))
```

```
(setf (aref wv 0) a)
```

```
(schedule-finalization a 'foo)
```

```
(setq a nil)
```

and do this in the Debug Window (the transcript shows the **I am garbage!** message occurring after `wv` is evaluated at **cl-user (11)** but if a gc was triggered by the loading of `finalization.cl`, you may see instead the **I am garbage!** message sooner):

```
cl-user(10): :ld finalization.cl
```

```
Loading finalization.cl
```

```
cl-user(11): wv
```

```
#((1 . 2))
```

```
cl-user(12): (gc)
```

```
I am garbage!
```

```
cl-user(13): 1
```

```
1
```

```
cl-user(14): 2
```

```
2
```

```
cl-user(15): 3
```

```
3
```

```
cl-user(16): (gc)
```

```
cl-user(17): wv
```

```
#(nil)
```

10.4 Static arrays

The data in a static array is placed in an area that is *not* garbage collected. This means that the data is never moved and, therefore, pointers to it can be safely stored in foreign code. (Generally, Lisp objects are moved about by the garbage collector so a pointer passed to foreign code is only guaranteed to be valid during the call. See *foreign-functions.htm* for more information on this point.) Only certain element types are permitted (listed below). General arrays (whose elements can be any Lisp object) cannot be created as static arrays.

The primary purpose of static arrays is for use with foreign code. They may also be useful with pure Lisp code but only if the array is needed for the whole Lisp run and the array never has to be significantly changed. The problem is the array is not garbage collected and there is no way (within Lisp) to free up the space.

Static arrays are returned by **make-array** with the (Allegro CL-specific) *allocation* keyword argument specified. The data is stored in an area which is not garbage collected but the header (if any) is stored in standard Lisp space. The following element types are supported in static arrays.

```

bit
(signed-byte 8)
(unsigned-byte 4)
(unsigned-byte 8)
(signed-byte 16)
(unsigned-byte 16)
(signed-byte 32)
(unsigned-byte 32)
character
single-float
double-float
(complex single-float)
(complex double-float)

```

See *implementation.htm* for information on this extension to *make-array*. See **lispval-other-to-address** for information on freeing static arrays.