How to use guppy/heapy for tracking down memory usage

This is a work in progress. It will grow a bit and it may not be entirely accurate everywhere.

Tutorial of sorts

All this was done on a checkout of marienz@gentoo.org-20060908111256-540d8fb3db5b337e, you should be able to check that out and follow along using something like:

```
bzr revert -rrevid:marienz@gentoo.org-20060908111256-540d8fb3db5b337e
```

in a pkgcore branch.

Heapy is powerful but has a learning curve. Problems are the documentation (http://guppy-pe.sourceforge.net/heapy_Use.html among others) is a bit unusual and there are various dynamic importing and other tricks in use that mean things like dir() are less helpful than they are on more "normal" python objects. This document's main purpose is to show you how to ask heapy various kinds of questions. It may or may not show a few cases where pkgcore uses more memory than it should too.

First, get an x86. Heapy currently does not like 64 bit archs much.

Emerge it:

```
emerge guppy
```

Fire up an interactive python prompt, set stuff up:

```
>>> from guppy import hpy
>>> from pkgcore.config import load_config
>>> c = load_config()
>>> hp = hpy()
```

Just to show how annoying heapy's internal tricks are:

```
>>> dir(hp)
['__doc__', '__getattr__', '__init__', '__module__', '__setattr__', '_hidir
>>> help(hp)
Help on class _GLUECLAMP_ in module guppy.etc.Glue:

_GLUECLAMP_ = <guppy.heapy.Use interface at 0x-484b8554>
```

This object is your "starting point", but as you can see the underlying machinery is not giving away any useful usage instructions.

Do everything that allocates some memory but is not the problem you are tracking down now. Then do:

```
>>> hp.setrelheap()
```

Everything allocated before this call will not be in the data sets you get later.
Now do your memory-intensive thing:

```
>>> l = list(x for x in c.repo["portdir"] if x.data)
```

Keep an eye on system memory consumption. You want to use up a lot but not all of your system ram for nicer statistics. The python process was eating about 109M res in top when the above stuff finished, which is pretty good (for my 512mb ram box).

```
>>> h = hp.heap()
```

The fun one. This object is basically a snapshot of what's reachable in ram (minus the stuff excluded through setrelheap earlier) which you can do various fun tricks with. Its str() is a summary:

```
>>> h
Partition of a set of 1449133 objects. Total size = 102766644 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0 985931  68 46300932  45  46300932  45 str
     1  24681   2 22311624  22  68612556  67 dict of pkgcore.ebuild.ebuild_
     2  49391   3 21311864  21  89924420  88 dict (no owner)
     3 115974   8  3776948   4  93701368  91 tuple
     4 152181  11  3043616   3  96744984  94 long
     5  36009   2  1584396   2  98329380  96 weakref.KeyedRef
     6  11328   1  1540608   1  99869988  97 dict of pkgcore.ebuild.ebuild_
     7  24702   2   889272   1 100759260  98 types.MethodType
     8  11424   1   851840   1 101611100  99 list
     9  24681   2   691068   1 102302168 100 pkgcore.ebuild.ebuild_src.pack
<54 more rows. Type e.g. '_.more' to view.>
```

(You might want to keep an eye on ram usage: heapy made the process grow another dozen mb here. It gets painfully slow if it starts swapping, so if that happens reduce your data set).

Notice the "Total size" in the top right: about 100M. That's what we need to compare later numbers with.

So here we can see that (surprise!) we have a ton of strings in memory. We also have various kinds of dicts. Dicts are treated a bit specially: the "dict of pkgcore.ebuild.ebuild_src.package" simply means "all the dicts that are __dict__ attributes of instances of that class". "dict (no owner)" are all the dicts that are not used as __dict__ attribute.

You probably guessed what you can use "index" for:

```
>>> h[0]
Partition of a set of 985931 objects. Total size = 46300932 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0 985931 100 46300932 100  46300932 100 str
```

Ok, that looks pretty useless, but it really is not. The "sets" heapy gives you (like "h" and "h[0]") are a bunch of objects, grouped together by an "equivalence relation". The default one (with the crazy name "Clodo" for "Class or dict owner") groups together all objects of the same class and dicts with the same owner. We can also partition the sets by a different equivalence relation. Let's do a silly example first:

```
>>> h.bytype
Partition of a set of 1449133 objects. Total size = 102766644 bytes.
 Index  Count   %     Size   % Cumulative  % Type
     0 985931  68 46300932  45  46300932  45 str
     1  85556   6 45226592  44  91527524  89 dict
     2 115974   8  3776948   4  95304472  93 tuple
     3 152181  11  3043616   3  98348088  96 long
     4  36009   2  1584396   2  99932484  97 weakref.KeyedRef
     5  24702   2   889272   1 100821756  98 types.MethodType
     6  11424   1   851840   1 101673596  99 list
     7  24681   2   691068   1 102364664 100 pkgcore.ebuild.ebuild_src.pack
     8  11328   1   317184   0 102681848 100 pkgcore.ebuild.ebuild_src.Thro
     9    408   0    26112   0 102707960 100 types.CodeType
<32 more rows. Type e.g. '_.more' to view.>
```

As you can see this is the same thing as the default view, but with all the dicts lumped together. A more useful one is:

```
>>> h.byrcs
Partition of a set of 1449133 objects. Total size = 102766644 bytes.
 Index  Count   %     Size   % Cumulative  % Referrers by Kind (class / dic
     0 870779  60 43608088  42  43608088  42 dict (no owner)
     1  24681   2 22311624  22  65919712  64 pkgcore.ebuild.ebuild_src.pack
     2 221936  15 20575932  20  86495644  84 dict of pkgcore.ebuild.ebuild_
     3 242236  17  8588560   8  95084204  93 tuple
     4      6   0  1966736   2  97050940  94 dict of weakref.WeakValueDicti
     5  36009   2  1773024   2  98823964  96 dict (no owner), dict of
                                             pkgcore.ebuild.ebuild_src.pack
     6  11328   1  1540608   1 100364572  98 pkgcore.ebuild.ebuild_src.Thro
     7  26483   2   800432   1 101165004  98 list
     8  11328   1   724992   1 101889996  99 dict of pkgcore.ebuild.ebuild_
     9      3   0   393444   0 102283440 100 dict of pkgcore.repository.pro
<132 more rows. Type e.g. '_.more' to view.>
```

What this does is:

- for every object, find all its referrers
- Classify those referrers using the "Clodo" relation you saw earlier
- Create a set of those classifiers of referrers. That means a set of things like "tuple, dict of someclass", *not* of actual referring objects.
- Group together all the objects with the same set of classifiers of referrers.

So now we know that we have a lot of objects referenced *only* by one or more dicts (still not very useful) and also a lot of them referenced by one "normal" dict, referenced by the dict of (meaning "an attribute of") ebuild_src.package, and referenced by a WeakRef. Hmm, I wonder what those are. But let's store this view of the data first, since it took a while to generate ("_" is a feature of the python interpreter, it's always the last result):

```
>>> byrcs = _
>>> byrcs[5]
Partition of a set of 36009 objects. Total size = 1773024 bytes.
 Index  Count   %     Size   % Cumulative  % Referrers by Kind (class / dic
     0  36009 100  1773024 100   1773024 100 dict (no owner), dict of
                                            pkgcore.ebuild.ebuild_src.pack
```

Erm, yes, we knew that already. If you look in the top right of the table you can see it is still grouping the items by the kind of their referrer, which is not very useful here. To get more information we can change what they are grouped by:

```
>>> byrcs[5].byclodo
Partition of a set of 36009 objects. Total size = 1773024 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0  36009 100  1773024 100   1773024 100 str
>>> byrcs[5].bysize
Partition of a set of 36009 objects. Total size = 1773024 bytes.
 Index  Count   %     Size   % Cumulative  % Individual Size
     0  10190  28   489120  28    489120  28        48
     1   7584  21   394368  22    883488  50        52
     2   7335  20   322740  18   1206228  68        44
     3   3947  11   221032  12   1427260  80        56
     4   3364   9   134560   8   1561820  88        40
     5   1903   5   114180   6   1676000  95        60
     6    877   2    56128   3   1732128  98        64
     7    285   1    19380   1   1751508  99        68
     8    451   1    16236   1   1767744 100        36
     9     57   0     4104   0   1771848 100        72
```

This took the set of objects with that odd set of referrers and redisplayed them grouped by "clodo". So now we know they're all strings. Most of them are pretty small too. To get some idea of what we're dealing with we can pull some random examples out:

```
>>> byrcs[5].byid
Set of 36009 <str> objects. Total size = 1773024 bytes.
 Index     Size   %   Cumulative  %   Representation (limited)
     0       80   0.0         80   0.0 'media-plugin...re20051219-r1'
     1       76   0.0        156   0.0 'app-emulatio...4.20041102-r1'
     2       76   0.0        232   0.0 'dev-php5/ezc...hemaTiein-1.0'
     3       76   0.0        308   0.0 'games-misc/f...wski-20030120'
     4       76   0.0        384   0.0 'mail-client/...pt-viewer-0.8'
     5       76   0.0        460   0.0 'media-fonts/...-100dpi-1.0.0'
     6       76   0.0        536   0.0 'media-plugin...gdemux-0.10.4'
     7       76   0.0        612   0.0 'media-plugin...3_pre20051219'
     8       76   0.0        688   0.0 'media-plugin...3_pre20051219'
     9       76   0.0        764   0.0 'media-plugin...3_pre20060502'
>>> byrcs[5].byid[0].theone
'media-plugins/vdr-streamdev-server-0.3.3_pre20051219-r1'
```

A pattern emerges! (sets with one item have a "theone" attribute with the actual item, all sets have a "nodes" attribute that returns an iterator yielding the items).

We could have used another heapy trick to get a better idea of what kind of string this was:

```
>>> byrcs[5].byvia
Partition of a set of 36009 objects. Total size = 1773024 bytes.
 Index  Count   %     Size   % Cumulative  % Referred Via:
     0      1   0       80   0        80   0 "['cpvstr']", '.key', '.keys()
     1      1   0       76   0       156   0 "['cpvstr']", '.key', '.keys()
     2      1   0       76   0       232   0 "['cpvstr']", '.key', '.keys()
     3      1   0       76   0       308   0 "['cpvstr']", '.key', '.keys()
     4      1   0       76   0       384   0 "['cpvstr']", '.key', '.keys()
     5      1   0       76   0       460   0 "['cpvstr']", '.key', '.keys()
     6      1   0       76   0       536   0 "['cpvstr']", '.key', '.keys()
     7      1   0       76   0       612   0 "['cpvstr']", '.key', '.keys()
     8      1   0       76   0       688   0 "['cpvstr']", '.key', '.keys()
     9      1   0       76   0       764   0 "['cpvstr']", '.key', '.keys()
<35999 more rows. Type e.g. '_.more' to view.>
```

Ouch, 36009 total rows for 36009 objects. What this did is similar to what "byrcs" did: for every object in the set it determined how they can be reached through their referrers, then groups objects that can be reached in the same ways together. Unfortunately it is grouping everything reachable as a dictionary key differently, so this is not very useful.

XXX WTF XXX

It is not likely this accomplishes anything, but let's assume we want to know if there are any objects in this set *not* reachable as the "key" attribute. Heapy can tell us (although this is *very* slow... there might be a better way but I do not know it yet):

```
>>> nonkeys = byrcs[5] & hp.Via('.key').alt('<')
>>> nonkeys.byrcs
hp.Nothing
```

(remember "hp" was our main entrance into heapy, the object that gave us the set of all objects we're interested in earlier).

What does this do? "hp.Via('.key')" creates a "symbolic set" of "all objects reachable *only* as the 'key' attribute of something" (it's a "symbolic set" because there are no actual objects in it). The "alt" method gives us a new symbolic set of everything reachable via "less than" this way. We then intersect this with our set and discover there is nothing left.

A similar construct that does not do what we want is:

```
>>> nonkeys = byrcs[5] & ~hp.Via('.key')
```

The "~" operator inverts the symbolic set, giving a set matching everything not reachable *exactly* as a "key" attribute. The key word here is "exactly": since everything in our set was also reachable in two other ways this intersection matches everything.

Ok, let's get back to the stuff actually eating memory:

```
>>> h[0].byrcs
Index  Count   %     Size   % Cumulative  % Referrers by Kind (class / dict
    0 670791  68 31716096  68  31716096  68 dict (no owner)
    1 139232  14  6525856  14  38241952  83 tuple
```

```
     2 136558  14  6042408  13  44284360  96 dict of pkgcore.ebuild.ebuild_s
     3  36009   4  1773024   4  46057384  99 dict (no owner), dict of
                                             pkgcore.ebuild.ebuild_src.packa
     4   1762   0   107772   0  46165156 100 list
     5    824   0    69476   0  46234632 100 types.CodeType
     6    140   0    31312   0  46265944 100 function, tuple
     7    194   0    11504   0  46277448 100 dict of module
     8     30   0     6284   0  46283732 100 dict of type
     9     55   0     1972   0  46285704 100 dict of module, tuple
```

Remember h[0] gave us all str objects, so this is all string objects grouped by the kind(s) of their referrers. Also notice index 3 here is the same set of stuff we saw earlier:

```
>>> h[0].byrcs[3] ^ byrcs[5]
hp.Nothing
```

Most operators do what you would expect, & intersects for example.

"We have a lot of strings in dicts" is not that useful either, let's see if we can narrow that down a little:

```
>>> h[0].byrcs[0].referrers.byrcs
Partition of a set of 44124 objects. Total size = 18636768 bytes.
 Index  Count   %     Size   % Cumulative  % Referrers by Kind (class / dic
     0  24681  56 12834120  69  12834120  69 dict of pkgcore.ebuild.ebuild_
     1  19426  44  5371024  29  18205144  98 dict (no owner)
     2      1   0   393352   2  18598496 100 dict of pkgcore.repository.pro
     3      1   0     6280   0  18604776 100 __builtin__.set
     4      1   0     6280   0  18611056 100 dict of module, guppy.heapy.he
     5      1   0     6280   0  18617336 100 dict of pkgcore.ebuild.eclass_
     6      1   0     6280   0  18623616 100 dict of
                                             pkgcore.repository.prototype.P
     7      4   0     5536   0  18629152 100 type
     8      4   0     3616   0  18632768 100 dict of type
     9      1   0     1672   0  18634440 100 dict of module, dict of os._Er
```

(Broken down: h[0].byrcs[0] is the set of all str objects referenced only by dicts, h[0].byrcs[0].referrers is the set of those dicts, and the final .byrcs displays those dicts grouped by *their* referrers)

Keep an eye on the size column. We have over 12M worth of just dicts (not counting the stuff in them) referenced only as attribute of ebuild_src.package. If we include the stuff kept alive by those dicts we're talking about a big chunk of the 100MB total here:

```
>>> t = _
>>> t[0].domisize
61269552
```

60M out of our 100M would be deallocated if we killed those dicts. So let's ask heapy what dicts that are:

```
>>> t[0].byvia
Partition of a set of 24681 objects. Total size = 12834120 bytes.
 Index  Count   %     Size   % Cumulative  % Referred Via:
     0  24681 100 12834120 100  12834120 100 "['data']"
```

(it is easy to get confused by the "byrcs" view of our "t". t[0] is *not* a bunch of "dict of ebuild_src.package". It is a bunch of dicts with strings in them, namely those that are *referred to* by the dict of ebuild_src.package, and not by anything else. So the byvia output means those dicts with strings in them are all "data" attributes of ebuild_src.package instances).

(sidenote: earlier we saw byvia say ".key", now it says "['data']". It's different because the previous type used __slots__ (so there was no "dict of" involved) and this type does not (so there is a "dict of" and our dicts are the "data" key in it).

So what is in the dicts:

```
>>> t[0].referents
Partition of a set of 605577 objects. Total size = 34289392 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0 556215  92 27710068  81  27710068  81 str
     1  24681   4  6085704  18  33795772  99 dict (no owner)
     2  24681   4   493620   1  34289392 100 long
>>> _.byvia
Partition of a set of 605577 objects. Total size = 34289392 bytes.
 Index  Count   %     Size   % Cumulative  % Referred Via:
     0  24681   4  6085704  18   6085704  18 "['_eclasses_']"
     1  21954   4  3742976  11   9828680  29 "['DEPEND']"
     2  22511   4  3300052  10  13128732  38 "['RDEPEND']"
     3  24202   4  2631304   8  15760036  46 "['SRC_URI']"
     4  24681   4  1831668   5  17591704  51 "['DESCRIPTION']"
     5  24674   4  1476680   4  19068384  56 "['HOMEPAGE']"
     6  24681   4  1297680   4  20366064  59 "['KEYWORDS']"
     7  24681   4   888516   3  21254580  62 '.keys()[3]'
     8  24681   4   888516   3  22143096  65 '.keys()[9]'
     9  24681   4   810108   2  22953204  67 "['LICENSE']"
<32 more rows. Type e.g. '_.more' to view.>
```

Strings, nested dicts and longs, and most size eaten up by the "_eclasses_" values. There is also a significant amount eaten up by keys values, which is a bit odd, so let's investigate:

```
>>> refs = t[0].referents
>>> i=iter(refs.byvia[7].nodes)
>>> i.next()
'DESCRIPTION'
>>> i.next()
'DESCRIPTION'
>>> i.next()
'DESCRIPTION'
>>> i.next()
'DESCRIPTION'
>>> i.next()
'DESCRIPTION'
```

Eep!

```
>>> refs.byvia[7].bysize
Partition of a set of 24681 objects. Total size = 888516 bytes.
```

```
Index  Count   %     Size   % Cumulative  % Individual Size
    0  24681 100   888516 100    888516 100        36
```

It looks like we have 24681 identical strings here, using up about 1M of memory. The other odd entry is the '_eclasses_' string apparently.

Extra stuff for c extension developers

To provide accurate statistics if your code uses extension types you must provide heapy with a way to get the following data for your custom types:

- How large is a certain instance?
- What objects does an instance contain?
- How does the instance refer to a contained object?

You provide these through a NyHeapDef struct, defined in heapdef.h in the guppy source. This header is not installed, so you should just copy it into your source tree. It is a good idea to read this header file side by side with the following descriptions, since it contains details omitted here. The stdtypes.c file contains implementations for the basic python types which you can read for inspiration.

The NyHeapDef struct provides heapy with three function pointers:

SizeGetter

To answer "how large is an instance" you provide a NyHeapDef_SizeGetter function that is called with a PyObject* and returns an int: the number of bytes the object occupies. If you do not provide this function heapy uses a default that looks at the tp_basicsize and tp_itemsize fields of the type. This means that if you do not allocate any extra memory for non-python objects (e.g. for c strings) you do not need to provide this function.

Traverser

To answer "What objects does an instance contain" you provide a traversal function (NyHeapDef_Traverser). This is called with a pointer to a "visit procedure", an instance of your extension type and some other stuff. You should then call the visit procedure for every python object contained in your object.

This might sound familiar: to support the python garbage collector you provide a very similar function (tp_traverse). Actually heapy will use tp_traverse if you do not provide a heapy-specific traverse function. Doing this makes sense if you do not support the garbage collector for some reason, or if you contain objects that are irrelevant to the garbage collector.

An example would be a type that contains a single python string object (that no other code can get a reference to). If this object does not have references to other python objects it cannot be involved in cycles so supporting gc would be useless. However you do still want heapy to know about the memory occupied by the contained string. You could do that by adding that size in your NyHeapDef_SizeGetter function but it is probably easier to tell heapy about the string through the traversal function (so you do not have to calculate the memory occupied by the string).

If the above type would also have a reference to some arbitrary (non-private) python object it should support gc, but it does not need to tell gc about the contained string. So you would have two traversal functions, one for heapy that visits the string and one for gc that does not.

RelationGetter

The last function heapy wants tells it in what way your instance refers to some contained object. It is used to provide the "byvia" view. This calls a visit function once for each way your instance refers to a target object, telling it what kind of reference it is.

Providing the heapdef struct to heapy

Once you have the needed function pointers in a struct you need to pass this to heapy somehow. This is done through a standard cpython mechanism called "cobjects". From python these look like rather stupid objects you cannot do anything with, but from c you can pull out a void* that was put in when the object was constructed. You can wrap an arbitrary pointer in a CObject, make it available as attribute of your module, then import it from some other module, pull the void* back out and cast it to the original type.

heapy looks for a _NyHeapDefs_ attribute on all loaded modules. If this attribute exists and is a CObject the pointer in it is used as a pointer to an array of NyHeapDef struct (terminated with a struct with only nulls). Example code doing this is in sets.c in the guppy source.