NAME

w_statistics_t − generic statistics structure

SYNOPSIS

```
#include <w.h>
#include <w_statistics.h>

class w_statistics_t {

    // members of interest to users
    w_statistics_t    *copy_brief() const;
    w_statistics_t    *copy_all() const;

    int               *int_val(NAMED_CONSTANT) const;
    static int        error_int; //returned if error

    uint              *uint_val(NAMED_CONSTANT) const;
    static unsigned int   error_int; //returned if error

    float             *float_val(NAMED_CONSTANT) const;
    static float      error_float; //returned if error

    char              typechar(NAMED_CONSTANT) const;
                      //returns 'v' for unsigned long
                      //        'l' for long
                      //        'i' for int
                      //        'u' for unsigned int
                      //        'f' for float

    const char        *string(NAMED_CONSTANT) const;
    const char        *module(NAMED_CONSTANT) const;

    // for arithmetic
    friend
    w_statistics_t   &
    operator+=(w_statistics_t &, const w_statistics_t &);

    friend
    w_statistics_t   &
    operator-=(w_statistics_t &, const w_statistics_t &);

       void                 zero(); // clears all stats
};

// For SDL users
static shrc Shore::gather_stats(w_statistics_t &, bool remote=false);
```

DESCRIPTION

This is a class for collecting and printing simple statistics, meaning integers, unsigned integers, and one-word floating-point numbers. Statistics are collected in modules (meant to correspond to software modules). Each module consists of a list of statistics, along with metadata describing the type and semantics of each statistic. Modules are distinguished by unique masks, which are manifest constants. (There is no convenient way to make sure the masks are unique.)

In order to reduce the effort required to read this manual page, we distinguish two kinds of

readers: those writing software that generates statistics (e.g., a value-added-server), and those writing software that uses statistics generated elsewhere (e.g., an application that uses statistics generated by the Shore Object Cache). After reading the section below, **MECHANISM**, you can skip a section that does not apply to you.

MECHANISM

This class allows local and remote statistics to be collected. In the local case, the modules in an instance of *w_statistics_t* store references to data structures that are updated *in place* by the software module that generates the statistics. For example, Object Cache statistics are generated by the Object Cache, and are stored in a data structure to which the application's *w_statistics_t* instance has direct access through a reference.

In the remote case, copies of the statistics and all their metadata are put in the *w_statistics_t* instance. The implications of this are that it can be confusing to copy and save statistics, because local (static) statistics and remote (malloc-ed) statistics have to be treated differently.

WRITING SOFTWARE THAT GENERATES STATISTICS

In Shore, we use a Perl script (in the source tree under) to generate the statistics modules for the various software modules. The section **GENERATING MODULES** , below describes the input to this script.

Any number of software modules can 'add' their statistics to a *w_statistics_t* instance for later printing by an application. The application can use generic methods to print the entire set of modules, or it can print only selected statistics according to its own formatting rules, or use selected statistics for its own purposes.

GENERATING MODULES

We use an (abbreviated) example from the Shore source tree to show how to use the Perls script to generate statistics modules. The input to the Perl script is as follows:

```
SM Storage manager = 0x00060000 sm_stats_info_t {
    // Record pinning:
    u_long rec_pin_cnt  Times records were pinned in the buffer pool
    u_long rec_unpin_cnt Times records were unpinned

    // Btree stats:
    u_long bt_find_cnt  Btree lookups (find_assoc())
    u_long bt_insert_cnt    Btree inserts (create_assoc)
    u_long bt_remove_cnt    Btree removes (destroy_assoc)
    u_long bt_scan_cnt      Btree scans started
    // ... we don't include all the stats
};
```

The first line identifies the module and some of the metadata to be associated with it. *SM* is a character string that will be a prefix for all the manifest constants generated by the script. What lies between *SM* and the equal sign (=) becomes a descriptive string for the module, for the purpose of printing the statistics. *0x00060000* is for distinguishing this module from others. *sm_stats_info_t* is the name of a C++ class that "owns" the statistics. This name is used to generate the file names for the output of The files generated in this example are:

```
sm_stats_info_t_def.i
sm_stats_info_t_msg.i
sm_stats_info_t_op.i
sm_stats_info_t_struct.i
```

The second line is a comment. C++ and C comments, and blank lines are acceptable.

The next lines defines a single statistic, whose type is *u_long*. Types can be any one of : **long, u_long, int, u_int, float.** After the type is a C/C++ struct member name. The class *sm_stats_info_t* will contain the members

```
unsigned long rec_pin_cnt;
```

```
unsigned long rec_unpin_cnt;
```
and so on. The list of members is generated by the script, and will be found in The script does not generate the entire definition for *sm_stats_info_t* because the author of the software may wish to make the statistics be only a small part of the class, and therefore define the class as follows:

```
class sm_stats_info_t {
    ... // stuff

#include "sm_stats_info_t_struct.i"

    ... // more stuff
};
```

Getting back to the input to the Perl script, the remainder of the third line is a string that describes the semantics of the statistic. It will be quoted by the Perl script. You should not quote it in your input file. The string should not be very long because it makes the output difficult to format nicely.

The file contains the definition of an output operator

```
w_statistics_t &
operator<<(w_statistics_t &s,const sm_stats_info_t &t);
```
This operator is declared to be a friend of your class *sm_stats_info_t* (by including The file also contains some metadata describing the types of the statistics, which are members of your class (by including

The file contains the list of descriptive strings for the module. These must be used as follows (sorry, this isn't automatically generated): in some single place (so it isn't multiply defined), do

```
// the strings:
const char *sm_stats_info_t ::stat_names[] = {

#include "sm_stats_info_t_msg.i"

};
```

The output file contains the manifest constants for the module, which are generated for (optional) use by the application (the program that prints the statistics).

GATHERING STATISTICS

Using the above example, the module of statistics called a *sm_stats_info_t* is added to a *w_statistics_t* instance with the operator

```
w_statistics_t &
operator<<(w_statistics_t &s,const sm_stats_info_t &t)
```
as follows:

```
w_statistics_t stats;
// assume the sm_stats_info_t is called ss_m::stats_info

stats << ss_m::stats_info;
```

MISCELLANEOUS METHODS

Copy_brief makes copies of the statistics, but copies pointers to the metadata. The result is mutable.

Copy_all makes copies of the statistics and the metadata. The result is mutable.

The methods **int_val uint_val,** and **float_val** return the integer, unsigned integer, or floating point value of the statistic. When an error occurs in evaluating the method, these functions return *error_int, error_uint,* and *error_float,* respectively. You can find out the type of a statistic with the method **typechar,** which returns 'v' for unsigned longs, and 'f' for floats.

String returns the printable, descriptive string for the statistic indicated by the manifest (named) constant. **Module** returns the printable, descriptive string for the module of which the statistic

is a member.

Operators **operator+=** and **operator-=** perform the indicated arithmetic on the corresponding statistics in the operands, which are instances of *w_statistics_t*. The operands must contain exactly the same statistics, and left-hand operand must be mutable, which means that it must be a copy of a local (static) instance, or it must be a remote (malloced) instance.

Zero sets all the values to 0 (or 0.0 for floats). It will fail on an immutable (static, local) instance.

WRITING SOFTWARE THAT USES GENERATED STATISTICS

GATHERING STATISTICS

Applications (users of SDL) will use the method **Shore::gather_stats.**

```
w_statistics_t      localstats;
SH_DO(Shore::gather_stats(localstats));

w_statistics_t      remotestats;
SH_DO(Shore::gather_stats(remotestats, true));
```

PRINTING ALL THE STATISTICS

A program can use the output operator to print all the statistics in an instance of the class *w_statistics_t*. The program does not need to have any compiled-in knowledge of any of the modules contained in the instance.

This operator does not print any information about statistics whose values are zero.

```
w_statistics_t     stats;
cout << stats << endl;
```

PRINTING SELECTED STATISTICS

In order to use selected statistics, a program must have compiled in the manifest constants for the modules of interest. For SDL users, these are included by `#include <ShoreStats.h>` See the following man pages for lists of the constants available for the various software layers: **statistics(oc), statistics(svas),** and **statistics(ssm).**

For example, to print the storage manager's count of the bytes of log generated:

```
w_statistics_t    current;
SH_DO(Shore::gather_stats(current, true));

cout << "Module "
    << current.module(SM_log_bytes_generated) << endl;

cout << ::form("%-30.30s %10.10d",
    current.string(SM_log_bytes_generated),
    current.int_val(SM_log_bytes_generated)) << endl;
```

The first print statement prints the name of the module; you can call the method **w_statistics_t::module** with the manifest constant for any statistic to get a descriptive name of the module (in this case, "Storage manager"). The second print statement formats the output as follows:

```
Bytes written to the log       0000000928
```

SAVING STATISTICS and COMPUTING DIFFERENCES

Statistics can be saved for later use in computing the costs of certain operations. The natural thing to want to do is to gather two entirely different copies of all the statistics, you can just gather twice, and compute the difference:

```
w_statistics_t      earlier;
SH_DO(Shore::gather_stats(earlier, true));
w_statistics_t      later;
SH_DO(Shore::gather_stats(later, true));

// DON'T DO THIS WITH LOCAL STATISICS
```

```
later -= earlier;
cout << later << endl;
```

With local statistics ...

This will not work for local statistics because the differences will always be zero! Each of the instances of *w_statistics_t* points directly to the current local statistics data structures for each module! It works fine for remote statistics (those gathered from the Shore server) because each of *earlier* and *later* is a complete copy of the statistics and metadata.

To save local statistics, you need to make a copy.

```
w_statistics_t      current;

SH_DO(Shore::gather_stats(current));
w_statistics_t         *saved = current.copy_brief();
```

Copy_brief copies only that values of the statistics, and it makes duplicate references to the metadata stored in *current* (rather than copying all the descriptive strings, for example). **IMPORTANT:** This means that you had better not let *current* go out of scope until you are finished with *saved !* Now, you'd like to just subtract one from the other:

```
// ERROR:
current -= *saved;
```

but that doesn't work because current is immutable. (Remember, it points into the current statistics.) You have to copy it also:

```
// OK:
w_statistics_t *cur = current.copy_brief();
*cur -= saved;
cout << *cur << endl;
// Don't forget to delete:
delete cur;
delete saved;
```

More about remote statistics ...

With remote statistics, you might wonder how you can save the expense of twice copying all the metadata from the server. Here's how:

w_statistics_t current; w_statistics_t *saved;

SH_DO(Shore::gather_stats(current, true)); w_statistics_t *saved = current.copy_brief(); // gather a current set SH_DO(Shore::gather_stats(current, true)); current -= *saved; cout << current << endl;

In this example, because
current
contains remote statistics (everything is malloced),
it is a writable instance of
w_statistics_t;
it can be overwritten and updated by the subtraction.

VERSION

This manual page applies to Version 2.0 of the Shore Storage Manager.

SPONSORSHIP

The Shore project is sponsored by the Advanced Research Project Agency, ARPA order number 018 (formerly 8230), monitored by the U.S. Army Research Laboratory under contract DAAB07-91-C-Q518. Further funding for this work was provided by DARPA through Rome Research Laboratory Contract No. F30602-97-2-0247.

COPYRIGHT

SEE ALSO

rc(fc), intro(fc), statistics(oc), statistics(svas), and **statistics(ssm).**