An introduction to genstruct

A final year project discussion paper (number two)

Table of Contents

About the author

Michael's student number is 964076. He can be contacted at mikal@stillhq.com. This submission is 1129words, not including the code snippets.

Abstract

This paper discusses **genstruct** a structure serialization methodology for C and C++ development. It has been developed by Dr Andrew Tridgell, who currently works for Quantum. **genstruct** is extremely powerful, and the ability to serialize in memory data structures has some interesting side effects – for instance the ability to upgrade code bases without incurring a service outage.

The meeting

genstruct is a C based data dumping package which Dr Andrew Tridgell discussed at the June 2002 Canberra Linux User's Group meeting on 27 June 2002. It finds it's roots in the perl **Data::Dumper** and Java string marshaling methodologies, and is an extremely powerful tool for development. Dr Tridgell originally developed **genstruct** for the **Samba** project. In this paper I discuss points Dr Tridgell raised in his talk in further detail.

What is genstruct?

As previously mentioned, **genstruct** is very similar in some respects to the perl **Data::Dumper**. It is therefore useful to provide a brief introduction to that functionality before moving onto **genstruct**.

Perl's Data::Dumper in a nutshell

Perl's **Data::Dumper** is quite simple to use:

Which produces...

Genstruct

genstruct is a perl program which is run at compile time. It parses the c header files for the program you want to use **genstruct** with, using tags that you have to embed into the header file. For example, the sample which comes with **genstruct** is as follows:

```
GENSTRUCT enum fruit {APPLE, ORANGE=2, PEAR,
      RASBERRY, PEACH};

GENSTRUCT
```

```
struct test2
{
int x1;
char *foo;
char fstring[20]; _NULLTERM
int dlen;
char *dfoo; _LEN(dlen)
enum fruit fvalue;
struct test2 *next;
};

GENSTRUCT struct test1 {
char foo[100];
char *foo2[20];
int xlen;
int *iarray; _LEN(xlen);
unsigned slen;
char **strings; _LEN(slen);
char *s2[5];
double d1, d2, d3;
struct test2 *test2;
int alen;
struct test2 *test2_array; _LEN(alen);
struct test2 *test2_fixed[2];
int plen;
struct test2 **test2_parray; _LEN(plen)
};
```

Code: /home/mikal/external/cvs.samba.org/junkcode/genstruct/test.h

In this example you can see that structures which should have **genstruct** enabled have the **GENSTRUCT** attribute associated with them, you are therefore not required to have all of the data structures in your code exportable. **GENSTRUCT** is merely an empty #define, which the **genstruct** header file parser can search for.

To create a string representation of a data structure, simply:

```
char *s;
struct test1 t;

// ... we need to populate t with data here ...

s = gen_dump(pinfo_test1, (char *) &t, 0);
```

In this code, we define a structure, fill it with data, and then use the **gen_dump** function to create a string representation of that structure. The arguments to **gen_dump** are:

```
char *gen_dump(const struct parse_struct *pinfo,
               const char *data,
               unsigned indent);
```

- *const struct parse_struct *pinfo*: is generated at compile time by **genstruct**, and is located in the output to that command.

- *const char *data*: is the data to dump to the string representation (simply cast your structure to a char * before passing it.
- *unsigned indent*: is the starting indent for ease of recursive calling. Set it to zero.
- **Returns**: a string representation of the structure.

The most interesting thing here is the *pinfo* structure which is the first argument to this **gen_dump** function. The *pinfo_test1* in this example looks like:

```
static const struct parse_struct pinfo_test1[] = {
{"foo", 0, sizeof(char), offsetof(struct test1, foo), 100, NULL,
  0, gen_dump_char, gen_parse_char},
{"foo2", 1, sizeof(char), offsetof(struct test1, foo2), 20, NULL,
  0, gen_dump_char, gen_parse_char},
{"xlen", 0, sizeof(int), offsetof(struct test1, xlen), 0, NULL,
  0, gen_dump_int, gen_parse_int},
{"iarray", 1, sizeof(int), offsetof(struct test1, iarray), 0,
  "xlen", 0, gen_dump_int, gen_parse_int},
// ... and so on ...
{NULL, 0, 0, 0, 0, NULL, 0, NULL, NULL}};
```

This table might seem a bit daunting at first, but readers need to remember that they're not expected to be able to read, generate, or use these tables. They are created solely for the use of **genstruct**.

The output of the **gen_dump** function call will be something like:

```
foo = {hello foo}
foo2 = 1:{foo2 \7d you}, 2:{foo2 you 2}
xlen = 6
iarray = 0:9, 1:4, 2:3, 3:9, 4:7
slen = 3
strings = 0:{test string 48}, 1:{test string 69}, 2:{test string 36}
s2 = 2:{t2 string 0}, 3:{t2 string 74}
d1 = 3.5
d3 = -7
test2 = {
        x1 = 4
        foo = {hello \7b there}
        fstring = {blah 1}
        dlen = 12
        dfoo = {q\a9\08z\faO\ca\e3\1d\b2M\88}
        fvalue = APPLE
        next = {
                x1 = 5
                foo = {hello \7b there}
                fstring = {blah 1}
                dlen = 28
                dfoo = {\e8\8f\dc\1c\0e\c7)'\ea\da\07e\ca\042\ce\078?\b0@\ba\ab\90\84\8
                fvalue = APPLE
                next = {
                        x1 = 6
                        foo = {hello \7b there}
                        fstring = {blah 1}
                        dlen = 27
                        dfoo = {5r\c3\c4O\e0\d2\16      \f9\01\e3\01f\ad\05\98\7b^L\d0\
                        fvalue = APPLE
                        next = {
                                x1 = 7
                                foo = {hello \7b there}
                                fstring = {blah 1}
                                dlen = 14
                                dfoo = {N/\d1\83\a2\94G\f1t\1a\07\7d\13\08}
                                fvalue = APPLE
                        }
                }
        }
}
```

This string representation can then be stored for later use.

The other side of the equation

The only reason you would use a package such as **Data::Dumper** or **genstruct** is so that you can read the information back in later. This is done with the **eval** function in perl. **genstruct**'s equivalent is **gen_parse**, which takes your string representation and recreates the data structures as stored. The arguments to **gen_parse** are:

```
int gen_parse(const struct parse_struct *pinfo,
              char *data,
              const char *str0);
```

- *const struct parse_struct *pinfo*: is the same parse structure that was used in the **gen_dump** call.
- *char *data*: is a pointer to the location that the structure should be created at. This memory should already have been allocated (for the main structure).
- *const char *str0*: the string representation to use.
- **Returns**: non zero if there was an error.

A sample usage is:

```
char *s;
struct test1 t1, t2;

// ... we need to populate t1 with data here ...

s = gen_dump(pinfo_test1, (char *) &t1, 0);
memset(&t2, 0, sizeof(t2));

if(gen_parse(pinfo_test1, (char *) &t2, s) != 0){
  printf("Parse failed!\n");
  exit(1);
  }
```

Additional features

genstruct also has the following additional features:

- It is smart enough to neatly handle the addition and removal of members of the structure.
- The output is presented in a human readable form, which might make editing of the contents of a structure easier.

Limitations

genstruct doesn't currently handle structures which contain file handles properly. The file handle itself (which is just an int) will be recovered, but the state of the file handle will not be restored.

Uses

There are several reasons that developers will be interested in **genstruct**. The main one is that it is a very convenient way of storing information between instances of a program – for example you could save information about the session that the user had just completed on program exit (for instance the size of the main window and the last five documents opened) into a structure, and then generate a string representation of this structure to save to a file. On start up, you can read the string back into the structure, and go from there. This was the original reason that the code was developed for the Samba project [1]

Interestingly, **genstruct** has other uses. For instance, if you have a long run server process such as Samba, then it is possible that you might want to change some of the internal data structures whilst the program is running. An example is when its the middle of the night and you only want to be backing up files, then you could save the normal structures to a string, and then reload them into a different set of structures which is optimized for large file reads. This would rely on **genstruct**'s ability to provide default values for keys which didn't exist when the string was created. Dr Tridgell expressed this as a major advantage of **genstruct** for the Samba team.

Obtaining genstruct

genstruct can be downloaded from Dr Tridgell's website at http://junkcode.samba.org

Response to the talk

Dr Tridgell is obviously a very talented software engineer, and his enthusiasm is contagious. He is also very obviously experienced at talking to audiences (he gives a very large number of conference presentations), and he normally speaks off the cuff.

Notes

1. The Samba team needs to store a large amount of information about user connections. This ranges from the name of all logged on users, to the locks on files that those users currently hold. At the moment this information is stored in a series of (key, value) **TDB** databases (**TDB** is outside the scope of this paper. A brief summary is that it is a very powerful GDBM like (key, value) pair database which supports concurrent access and locking. It has also had a SQL implementation built on top of it, although the Samba team doesn't use this), with several keys for each user's information. It is now possible to simply save a string representation of a structure describing the user, and recover that structure each time it is needed.