Introspection via Self Debugging

Russell Harmon

reh5586@cs.rit.edu

Rochester Institute of Technology

Computer Science

December 12, 2013

1 Introduction

The omnipresent support for introspection in modern programming languages indicates the usefulness of the tool. [4, 5, 19, 21] Unfortunately, C, which is one of the most pervasive programming languages and the foundation of nearly every modern operating system, does not support introspection.

By leveraging an existing debugger, an API entitled Ruminate brings introspection to the programming language C. Debuggers have long had access to the type information which is needed for introspection. On most UNIX platforms, this is accomplished by the debugger reading any debugging symbols which may be present in the target binary, and inspecting the state of the target program. These debugging symbols are not present by default and the compiler must be instructed to add the symbols at compile time. These techniques are leveraged to gain the information which is needed to create an introspection API and building on that, an API which can convert arbitrary types to and from JSON.

2 Motivation

One of the motivating factors for any language introducing introspection as a feature is the following use case:

> You are tasked with changing the save game format of a popular 1980s style terminal based game from a binary format composed of writing the structs which compose the game state to disk to a more flexible JSON format. After investigation, you discover that in order to do this, you can use the Jansson [17] C library to produce JSON. In order to do so, you invoke variants of the `json_object_set` function as given by the following prototype:
>
> ```
> int json_object_set(
>   json_t *object,
>   const char *key,
>   json_t *value
> );
> ```
>
> You observe that `json_object_set` takes as parameters the name and value of the field to be written necessitating the writing of a separate `json_object_set` call for every field of every aggregate type. After considering the literally thousands of fields across the nearly three hundred structs in the game you give up in frustration.

If a programmer were able to introspect types in C, they could write a generalized JSON conversion function which could determine the name of every aggregate type and aggregate member procedurally thereby significantly shortening the amount of code needed. A programmer could also use an introspective library for creation of platform independent binary structure representations for use in network communication. Clearly, it is a significant convenience to developers to be able to write code which is able to introspect upon data in a meta-programming style.

3 Introspection in Current Programming Languages

Introspection is found in many of the programming languages commonly used today including Java [19], Ruby [4], Python [21], Perl [5] and a limited form of introspection in C++ [29]. The various approaches to introspection differ in implementation details; some receiving introspection as a direct consequence of the way they implement objects while some provide it as part of the standard library. Despite this, they all provide approximately the same set of features. It is by these features that introspection can be defined, rather than the details of how the features are implemented.

Introspection implementations generally provide several different forms of introspection. A common form of introspection provided is *type* introspection. Specifically, a program leveraging type introspection is able to inspect the types of identifiers or values used in the program. Another form of introspection is *function* introspection. This form of introspection allows programs to retrieve information about functions which is not part of the type system, such as the function's name or argument's names. Finally, a third form of introspection is *stack* introspection. This allows a program to retrieve a list of stack frames at a given point in a program's execution, commonly referred to as a stacktrace or backtrace.

Existing attempts to add introspection to C or C++ frequently require a separate description of the object to be implemented which is generated using a separate parser [23], a complementary *metadata object* [2], or require specific code to be written that describes the type. All of these introspection implementations have the limitation that objects which come from external libraries cannot be introspected. Ruminate has neither this library boundary limitation nor requires external compile-time tools or hand written object descriptions in order to operate. Instead, Ruminate requires only that the library or executable to introspect contain debugging symbols.

4 Debugging in C

There already exist a number of tools for interactive debugging of C programs. Some of the more well known ones include GDB [9], WinDBG, Visual Studio's debugger and LLDB [25]. Traditionally, these debuggers have been used interactively via the command line where more recently debuggers such as the one embedded within Visual Studio integrate into an IDE.

An understanding of debugging in general, and about LLDB specifically are crucial to the understanding of this document, so some time will be spent explaining debugging.

Conceptually, a debugger is composed of two major components, a symbol parser and a process controller. Among other types of symbols in a binary, Linux usually uses DWARF [6] debugging symbols. These debugging symbols are intended for a debugger to parse and informs the debugger about some information which is not available otherwise from inspection of the compiled binary. This information includes the source file name(s), line number to compiled instruction mappings and type information. Interactive debugging using a debugger is possible without debugging symbols, but difficult. The other major piece of a debugger is the ability to control another process. This is necessary in order for the debugger to inspect or modify a debugee's runtime state, set break or watchpoints and intercept signals. In order to accomplish this, specific support must exist in the kernel which is hosting the process to be controlled. Across the various modern platforms, there exists several different implementations enabling one process to control another. On Linux, the API for process control is `ptrace(2)` [20].

An important aspect of the type information which is available to a debugger is that this information is almost entirely static. For instance, during an interactive debugging session when printing a variable the debugger knows only the type of the *variable* being displayed, rather than the type of the *data* itself. This is in stark contrast with other introspective languages where the type information is carried with the data and can be recovered without any additional context. An example of the result of this under LLDB is shown in Fig. 1. Notice that even though the value of `baz` is the string `"Hello World!"`, because the type of `baz` is `void *`, LLDB is unable to deduce the type.

4.1 LLDB

LLDB [25] is a debugger built on the LLVM [27] framework. Designed to be used as a library, it vends a public C++ API which is promised to be relatively stable, and has bindings to Python in which the LLDB authors have written its unit test suite [24].

Figure 2 shows a simple debugging session using LLDB. In it, a test program is launched and the value of a stack-local variable is printed. Take note that LLDB is aware that the type of `foo.bar` is `char *`. In fact regardless of the language most debuggers make available to their users a non-strict subset of the type information which is available to the programmer writing the original source file.

Under LLDB's public API, a type is represented by an SBType [18]. In order to get an instance of SBType, you can either retrieve the type by name, or retrieve the type of a variable by that variable's name

```
Process 12066 stopped
* thread #1: tid = 0x1c03, 0x0000000100000f64 a.out`main + 20 at a.c:3
    frame #0: 0x0000000100000f64 a.out`main + 20 at a.c:3
   1    int main() {
   2            void *baz = "Hello World!";
-> 3    }
(lldb) print baz
(void *) $0 = 0x0000000100000f66
```

Figure 1: Static Type Information in Debuggers

```
Current executable set to './a.out' (x86_64).
(lldb) breakpoint set -n main
Breakpoint created: 1: name = 'main', locations = 1
(lldb) run
Process 10103 launched: './a.out' (x86_64)
Process 10103 stopped
* thread #1: tid = 0x1c03, 0x0000000100000f60 a.out`main + 16 at a.c:6
    frame #0: 0x0000000100000f60 a.out`main + 16 at a.c:6
   3    };
   4    int main() {
   5            struct foo foo;
-> 6            foo.bar = "Hello World!";
   7    }
(lldb) next
Process 10103 stopped
* thread #1: tid = 0x1c03, 0x0000000100000f64 a.out`main + 20 at a.c:7
    frame #0: 0x0000000100000f64 a.out`main + 20 at a.c:7
   4    int main() {
   5            struct foo foo;
   6            foo.bar = "Hello World!";
-> 7    }
(lldb) print foo.bar
(char *) $0 = 0x0000000100000f66 "Hello World!"
```

Figure 2: Interactive Debugging with LLDB

with the debugee stopped at a breakpoint. Once that is accomplished, an SBType can give you much of the static type information about that variable which exists in the target's debugging symbols.

When an operation is performed on an SBType, LLDB lazily retrieves the type information needed to service that operation. Building on clang [3], LLDB uses the debugging symbols to generate a partial clang AST. This AST is then retained for future inspection of that type.

5 Related Work

A System for Runtime Type Introspection in C++ [2] discusses an approach to introspection for C++ whereby metadata objects are created using macros which are expected to be called at the definition of the object which is to be introspected.

The Seal C++ Reflection System [23] discusses an introspection system for C++ which uses a metadata generation tool to create descriptor files which contain the information needed for introspection.

C++, along with all the other languages supported by Microsoft's CLR can be reflected upon by leveraging features exposed by the CLR. [22]

Reflection for C++ [16] uses an approach very similar to the one proposed here, but instead of using a debugger to retrieve debugging information, it instead reads the debugging symbols directly. This limits the API to only leveraging information that it can retrieve from the debugging symbols themselves, as opposed to could be deduced dynamically. This limitation means that it does not support *stack introspection.* It also only supports introspection of classes - no support for enums, unions, bit fields, global or static functions and variables is available. It does however support a limited form of reflection wherein you can create instances of reflected classes and call reflected method pointers. Reflection for C++ also supports optionally describing a class explicitly in the class declaration using a series of macros provided by the library as an alternative to debugging symbol based introspection.

Ego [7, 8] is similar to the *Reflection for C++* project. It reads Stabs [28] symbols from a binary and allows static type introspection using those symbols. Like Reflection for C++, because Ego does not dynamically inspect the state of the program it does not support *stack introspection*, nor does it support retrieving the type of arbitrary expressions. Ego is also further limited in that it does not support introspection of third party libraries. Ego does however support retrieving information about the specific source or header files in which code is defined, and supports a more fully featured API for enumerating the different kinds of symbols defined in a project than that found in Ruminate wherein a programmer can fully traverse the structure of a specific symbol including the scope in which it is defined. Ego's documentation does describe a feature it calls "stack introspection," but this does not allow retrieval of a call stack. Instead, it allows retrieval of

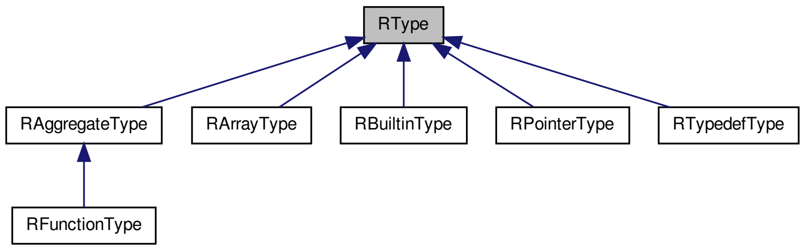


Figure 3: RType inheritance graph

variables from the current stack frame using stack specific symbol representations.

GObject Introspection [11], a part of the Gnome [10] project allows for introspection of instances of a `GObject`. These objects are plain C structs which are initialized following a specific convention in order to inform the `GObject` subsystem of the type of that object. GObject Introspection then uses that type information which was initialized at runtime in order to provide introspective fatures. It is not possible to introspect structs which were not initialized following the `GObject` initialization contract.

6 Ruminate

By leveraging LLDB, an API entitled Ruminate [14] was created which hides the complexities of self debugging and enables a programmer to introspect a C program. The intent is not to express every detail of C's type system, but instead to expose a subset complete enough to be useful while small enough to be determinable by a debugger. The DWARF standard for debugging symbols attempts to describe a common subset of features found in many procedural languages and Ruminate only attempts to provide introspective features which can be implemented leveraging DWARF debugging symbols.

After a call to `ruminate_init`, the programmer can call `ruminate_get_type` providing it with an expression. `ruminate_get_type` will return an instance of `RType` which represents the type of the expression passed. The programmer can also retrieve an instance of a `RType` with a call to `ruminate_get_types_by_name` passing in a string which will return a list of types which have that name. The name of a function, if extant, is not a part of its type and therefore is not available from an `RType`. A function's name can be retrieved by address with a call to `ruminate_get_function_name`. Section 10 documents the use of these and other Ruminate functions.

There are only a few methods defined on `RType` itself. Most of the functionality of introspection is found in the subclasses of `RType`. Figure 3 shows the inheritance hierarchy of `RType`. An instance of `RType` can be safely cast to the child type indicated by a call to `r_type_id`. The type retrieved may itself have sub types, for which there exists analogous functions to the `r_type_id` function, and the type retrieved may be further cast to the corresponding child type.

A simple example of the use of Ruminate can be seen in Fig. 4a wherein a struct is introspected. In this example, `ruminate_get_type` is called to introspect the struct `bar`. The `print_data` function then receives the resulting `RType` and prints its name, then for every member of `foo`, it prints the member name and recursively calls itself passing into the recursive invocation the member's type. The typedef `string_t` is then encountered and is dealt with specially as a string. Because the type system of C does not have a specific string type, it is impossible to determine procedurally whether any given `char *` is a string.[1] Finally the builtin `int` is printed. The output of this program is shown in Fig. 4b.

Since Ruminate is built on a debugger, it can also provide the programmer with stack introspection. A call to `ruminate_backtrace` will return a `RFrameList` representing all the stack frames in the call stack which resulted in the call to `ruminate_backtrace`. Shown in Fig. 5a, a simple `abort_with_stacktrace` function which calls `abort(3)` [1] after printing a full stack trace has been written to demonstrate the use of stack introspection. It's output is shown in Fig. 5b

Leveraging Ruminate, a library built for the conversion of C data structures into JSON [26] was created. An example of the use of this library is shown in Fig. 6. In it, the `json_serialize` function generates a `json_t` by inspecting the `RType` and associated value passed into it. The string member variable `s` of `MyStruct` is handled by registering a custom serializer for that type via a call to `json_state_add_serializer`. Unions and array pointers are not shown in this example, and can only be converted to JSON using custom serializers. This library has two output modes, a simple non-invertable mode and a more verbose invertable mode. These two output modes are shown in Fig. 6b and Fig. 6c respectively. The invertable mode's output can be converted from JSON back to its original type with a call to `json_deserialize`.

Ruminate also supports introspecting third party code. Fig. 7 shows the C standard library's `FILE *stdout` converted to JSON using Ruminate. Declarations found in a public header file make their way into the debug symbols of the file which includes that header file, making those declarations introspectable. Additionally, private types can be introspected if the library which contains that type has debugging symbols.

Building on this, a reference counted typed memory allocator was written. This memory allocator allows the creation of values which carry their type. After a call to one of `r_mem_malloc`, `r_mem_malloc_sized`, `r_mem_calloc` or `r_mem_calloc_sized`, a pointer to heap allocated memory is returned which carries with it the type of that pointer. The type of that memory can be retrieved with a call to `r_mem_type`. The fact

[1]See Section 8 for further discussion on why strings are special cased.

```c
#include <ruminate.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

typedef char *string_t;

struct foo {
  string_t str;
  int i;
};

void print_data( RType *type, const void *data ) {
  switch( r_type_id(type, NULL) ) {
    case R_TYPE_TYPEDEF:
      if( strcmp(r_string_bytes(r_type_name(type, NULL)), "string_t") == 0 )
        printf("(string_t) \"%s\"\n", *((const string_t *) data));
      break;
    case R_TYPE_BUILTIN:
      switch( r_builtin_type_id((RBuiltinType *) type, NULL) ) {
        case R_BUILTIN_TYPE_INT:
          printf("(int) %d\n", *((const int *) data));
          break;
      }
      break;
    case R_TYPE_AGGREGATE: {
      RAggregateType *agg = (RAggregateType *) type;
      if( r_aggregate_type_id(agg, NULL) == R_AGGREGATE_TYPE_STRUCT ) {
        printf("(%s) {\n", r_string_bytes(r_type_name(type, NULL)));
        for( size_t i = 0; i < r_aggregate_type_nmembers(agg, NULL); i++ ) {
          RAggregateMember *memb = r_aggregate_type_member_at(agg, i, NULL);
          printf("\t.%s = ", r_string_bytes(r_aggregate_member_name(memb, NULL)));
          RTypeMember *tmemb = (RTypeMember *) memb;
          off_t offset = r_type_member_offset(tmemb, NULL);
          print_data(r_type_member_type(tmemb, NULL), data + offset);
        }
        printf("}\n");
      }
      break;
    }
  }
}

int main( int argc, char *argv[] ) {
  (void) argc;
  ruminate_init(argv[0], NULL);
  struct foo bar = {
    .str = "Hello World!",
    .i = 6666
  };
  print_data(ruminate_get_type(bar, NULL), &bar);
}
```

(a) Introspective code

```
(foo) {
  .str = (string_t) "Hello World!"
  .i = (int) 6666
}
```

(b) Output from introspective code

Figure 4: Introspection using Ruminate

```c
#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>
#include <stdint.h>

#include <ruminate.h>

void abort_with_backtrace( const char *message ) {
  RFrameList *frames = ruminate_backtrace(NULL);

  fprintf(stderr, "abort(): %s\n", message == NULL ? "" : message);
  size_t frames_len = r_frame_list_size(frames, NULL);
  for( size_t i = 0; i < frames_len; i++ ) {
    RFrame *frame = r_frame_list_at(frames, i, NULL);
    RString *fname = r_frame_function_name(frame, NULL);
    RString *mname = r_frame_module_name(frame, NULL);
    RString *cuname = r_frame_compile_unit_name(frame, NULL);
    uint32_t line = r_frame_line(frame, NULL);
    fprintf(
      stderr,
      "\tat %s(%s, %s:%d)\n",
      r_string_bytes(fname),
      r_string_bytes(mname),
      r_string_bytes(cuname),
      line
    );
    r_string_unref(cuname);
    r_string_unref(mname);
    r_string_unref(fname);
  }

  r_frame_list_unref(frames);
  abort();
}

void bar( int i ) {
  if( i < 2 ) {
    bar(i + 1);
  } else {
    abort_with_backtrace("Hello World!");
  }
}

void foo() {
  bar(0);
}

int main( int argc, char *argv[] ) {
  ruminate_init(argv[0], NULL);
  foo();
}
```

(a) Introspective code

```
abort(): Hello World!
        at ruminate_hit_breakpoint(libruminate.so, ruminate.cpp:49)
        at ruminate_backtrace(libruminate.so, ruminate.cpp:257)
        at abort_with_backtrace(backtrace.exe, util.c:22)
        at bar(backtrace.exe, backtrace.c:11)
        at bar(backtrace.exe, backtrace.c:9)
        at bar(backtrace.exe, backtrace.c:9)
        at foo(backtrace.exe, backtrace.c:16)
        at main(backtrace.exe, backtrace.c:23)
        at __libc_start_main(libc.so.6, :0)
```

(b) Output from introspective code

Figure 5: Stack traces using Ruminate

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <glib.h>
#include <ruminate.h>
#include <jansson.h>
#include <ruminate-jansson.h>

typedef char *string;
struct MyStruct {
  int i; string s; int *p;
  union { char b; void *v; } *u;
  enum MyEnum { MY_ENUM_VALUE_1, MY_ENUM_VALUE_2 } e;
  char a[3];
};
static void *deserialize_string( JsonDeserializerArgs args, void *data, GError **error ) {
  const char *str = json_string_value(args.value);
  size_t str_len = strlen(str) + 1;
  char **ret = r_mem_malloc_fn(args.type, NULL);
  *ret = r_mem_malloc_sized(char *, str_len, NULL);
  memcpy(*ret, str, str_len);
  return ret;
}
static json_t *serialize_string( JsonSerializerArgs args, void *data, GError **error ) {
  return json_string(*((char **) args.value));
}
static JsonHook string_hook = { .serializer = serialize_string, .deserializer = deserialize_string };
int main( int argc, char *argv[] ) {
  ruminate_init(argv[0], NULL);
  int ipt = 2;
  struct MyStruct foo = { .i = 1, .u = NULL, .s = "hello world!", .e = MY_ENUM_VALUE_2,
                          .p = &ipt, .a = { 1, 2, 3 } };
  JsonState *st = json_state_new();
  json_state_add_hook(st, g_quark_from_static_string("string"), &string_hook);
  json_state_set_flags(st, JSON_FLAG_INVERTABLE);
  json_t *serialized = json_serialize(st, ruminate_get_type(foo, NULL), &foo, NULL);
  json_dumpf(serialized, stdout, 0);
  printf("\n");
  struct MyStruct *_foo = json_deserialize(st, serialized, NULL);
  printf("struct MyStruct {");
  printf(" .i = %d,", _foo->i);
  printf(" .u = %p,", _foo->u);
  printf(" .s = \"%s\",", _foo->s);
  printf(" .e = %d,", _foo->e);
  printf(" .p = %p (%d),", _foo->p, *_foo->p);
  printf(" .a = [%d, %d, %d]", _foo->a[0], _foo->a[1], _foo->a[2]);
  printf(" };\n");
  r_mem_unref(_foo->s), r_mem_unref(_foo->p), r_mem_unref(_foo);
}
```

(a) JSON Library Use

```
{"s": "hello world!", "i": 1, "p": 2, "e": 1, "a": [1, 2, 3]}
```

(b) Non-invertable JSON output

```
{"value":
   {"s": "hello world!",
    "i": 1,
    "p": 2,
    "e": 1,
    "a": [1, 2, 3]},
 "type": "MyStruct"}
struct MyStruct { .i = 1, .s = "hello world!", .e = 1,
                  .p = 0x15aedc8 (2), .a = [1, 2, 3] };
```

(c) Invertable JSON output

Figure 6: JSON Library Example

```
{
   "__pad1":null,
   "_IO_read_base":72,
   "_shortbuf":[0],
   "_IO_backup_base":null,
   "_vtable_offset":0,
   "_flags":-72537468,
   "_IO_read_ptr":72,
   "_IO_write_base":72,
   "_fileno":1,
   "_IO_buf_base":72,
   "_chain":{
      "__pad1":null,
      "_IO_read_base":null,
      "_shortbuf":[0],
      "_IO_backup_base":null,
      "_vtable_offset":0,
      "_flags":-72540024,
      "_IO_read_ptr":null,
      "_IO_write_base":null,
      "_fileno":0,
      "_IO_buf_base":null,
      "_chain":null,
      "_IO_write_ptr":null,
      "_IO_read_end":null,
      "_IO_save_end":null,
      "__pad3":null,
      "_IO_write_end":null,
      "_offset":-1,
      "_old_offset":-1,
      "_IO_save_base":null,
      "_flags2":0,
      "_IO_buf_end":null,
      "_markers":null,
      "_cur_column":0,
      "__pad4":null,
      "__pad5":0,
      "_mode":0,
      "_unused2":[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
   },
   "_IO_write_ptr":"",
   "_IO_read_end":72,
   "_IO_save_end":null,
   "__pad3":null,
   "_IO_write_end":72,
   "_offset":-1,
   "_old_offset":-1,
   "_IO_save_base":null,
   "_flags2":0,
   "_IO_buf_end":"",
   "_markers":null,
   "_cur_column":0,
   "__pad4":null,
   "__pad5":0,
   "_mode":-1,
   "_unused2":[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
}
```

Figure 7: `stdout` converted to JSON

```c
#include <ruminate.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( int argc, char *argv[] ) {
  (void) argc;

  ruminate_init(argv[0], NULL);

  const char *src_str = "Hello World!";
  size_t src_str_len = strlen(src_str) + 1;
  void *str = r_mem_malloc_sized(char *, src_str_len, NULL);
  memcpy(str, src_str, src_str_len);

  RType *str_type = r_mem_type(str);
  RString *str_type_name = r_type_name(str_type, NULL);

  printf("(%s) \"%s\"\n", r_string_bytes(str_type_name), str);

  r_string_unref(str_type_name);
  r_type_unref(str_type);
  r_mem_unref(str);
}
```

(a) Typed Value Use

```
(char *) "Hello World!"
```

(b) Typed Value Output

Figure 8: Typed Values

that the type of memory allocated using this allocator is carried with the value itself means that unlike any other type of value in C, the real type of e.x. a `void *` can be determined. Figure 8a shows an example of the use of this library to create a typed string. The program creates the string, prints to standard out its value and the name of its type via a call to `r_mem_type` and then exits. The output of this program is shown in Fig. 8b.

7 Implementation

Ruminate is architected as two major components, the primary application, hereafter referred to as the debugee, and a debugger control process. When initialized, Ruminate spawns a debugger control process tasked with controlling LLDB. This control process then attaches to the debugee and proceeds to wait for instructions over RPC from the debugee. This design was chosen because LLDB uses `ptrace` on Linux to control its debugee, and `ptrace` does not support controlling the calling process.

When `ruminate_init` is invoked, the debugger control process is started and an RPC connection is

```
char a[sizeof((*((int (*)[4]) NULL))[0])];
__typeof__(&a) ap = &a;
(int (*)[4]) ap
```

Figure 9: Array Introspection of an `int [4]`

negotiated between the debugger control process and the debugee. The debugger control process then attaches the debugger to the debugee and sets a breakpoint on the internal function `ruminate_hit_breakpoint`.

When `ruminate_get_type` is later invoked, an asynchronous RPC call is made to the debugger control process which instructs LLDB to retrieve type information about the variable which is being introspected. At the same time, `ruminate_hit_breakpoint` is called which stops the debugee and allows the debugger to locate the variable. LLDB accomplishes this by reading the debugging symbols in the binary and constructing an `SBType` [18] object to represent the type. Due to this, the type information available from LLDB and therefore available by Ruminate is limited to the type information found in the debugging symbols and any other information that can be inferred by inspecting the running program itself. The `SBType` constructed is then wrapped in an object which implements the RPC contract for a type. A proxy to this object is then returned to the debugee which further wraps this proxy in a `RType` which is finally returned to the programmer. Most calls to interrogate the returned `RType` are initially forwarded via RPC to the `SBType` held by the debugger control process, and then cached in the `RType` before being returned.

Measures have been taken to minimize the overhead involved in this form of introspection. The result of nearly all methods performed on a type is cached within the `RType`, so subsequent calls for that information will not need to perform RPC calls to the debugger control process. Strings are also wrapped in an `RString` type which is a read-only reference counted container for `char *` strings. All methods which return strings return an `RString` which is also cached internally.

Unlike every other type and despite the presence of relevant information in the debugging symbols, LLDB does not allow the programmer to retrieve information about arrays from an `SBType`. Only the value backed `SBValue` can retrieve this information. This provides a challenge since there may not be a value available to inspect when an array is introspected as an RType is not bound to the value which may have been used to generate it and the value of an array may have gone out of scope by the time it is introspected. The solution to this is to dynamically generate an array of a size large enough to hold one element of the array member type and use that as the value to inspect. This is accomplished by using a feature of LLDB which allows expression evaluation within the debugee. A C expression is generated which creates such an array and evaluates that expression in the context of the debugee, then retrieves an `SBValue` representing the array created and inspects that. For an array type `int [4]`, the expression evaluated in order to generate the array is shown in Fig. 9.

Typed values as returned by the typed memory allocation routines are implemented by padding the start of the memory to be allocated so it is large enough to store a pointer to the `RType` representing its type, its reference count and its size. The pointer returned to the programmer is actually offset into the object allocated so that the pointer returned can be used as bare memory.

8 Limitations

This style of introspection is more limited than the classic style of introspection whereby an object carries its own type information. Instead, the type of a *value* must be specified by the type of its *variable* or by name. As discussed in Section 4.1 very little type information can be determined at runtime. The result of this is that a call to `ruminate_get_type` on a variable whose type is **`void *`** will return an `RType` which represents a **`void *`** rather than the real type of the data.

Although not strictly a limitation, many programmers will likely want to introspect strings as such, rather than as the **`char *`** type. Unfortunately, since there is no difference in types between a C string and a pointer to one or more chars, Ruminate is unable to determine the difference between the two. The example code shown in Fig. 4a works around this issue by creating a typedef of **`char *`** to `string_t`. When interactively working with a debugger, it is assumed that a **`char *`** type points at a string which results in accessing arbitrary, sometimes uninitialized memory when that assumption is invalid.

There are two distinct kinds of arrays referred to by the C standard [15]. The first is the array *type* and the second is the array *object*. An array object is a contiguous block of memory representing one or more instances of some element type. Contrast this with the array type in that a pointer to an array object with element type *t* has type `t *` and is still said to be an array. An array *type* however with element type *t* has type `t []`. C provides for type coercion from array type to pointer type under most circumstances, and the array index operator `[]` rather than operating on array types instead operates on pointer types. This allows a programmer to use arrays and pointers nearly interchangeably.[1] This is troublesome for a programmer introspecting a pointer as a pointer type makes no distinction between array objects and non-array objects. Following with the design of C, Ruminate makes no differentiation between pointers to array objects and pointers to non array objects. It is the programmer's responsibility to deduce what kind of object is pointed to. In the JSON library built using Ruminate, pointer types are assumed to point at a single non-array object, which the programmer can change by installing custom serializers to handle array objects including strings.

LLDB does not support same-process debugging. This is due to that fact that in Linux, `ptrace` is used to control the debugee, and `ptrace` does not support tracing the calling process. This limitation necessitated

[1]Except in some applications of multi-dimensional arrays

the creation of the debugger control process. The IPC overhead added by the calls to `ptrace` is significant. The program shown in Fig. 4a takes approximately 1.6 seconds to run.

LLDB does not support debugging only a single thread of a multi-threaded application. This means that whenever the debugee must be stopped (which is a minimum of once per call to `ruminate_get_type`), all threads are stopped.

As discussed in Section 7, LLDB has only value backed representations for arrays. This means that in order to get type information about any array, the process must be stopped. Stopping the debugee is a significant cause of overhead due to the fact that not only is a context switch of the introspecting thread triggered, all threads in the debugee are stopped.

DWARF requires the names of function arguments be included in its internal type information. LLDB does not provide a means to access this information from its API. Therefore, introspection of the names of function arguments does not currently work. This limitation may be lifted in the future.

Ruminate can only currently be run using a patched LLDB. Several features which needed to be added to LLDB are not currently available in a stock LLDB installation. Specifically, an SBType did not support retrieving information about enums, for which support was added [12]. Also, LLDB did not support controlling its signal disposition thereby controlling whether LLDB stops and/or suppresses signals when the debugee is set to receive one, for which support was also added [13].

Ruminate has a rather severe failure case. Since the debugger control process is controlled by the debugee, if the debugee causes LLDB to stop it (e.x. via delivery of a signal) and the debugger control process does not properly handle the stop, the two processes may enter a dead lock state wherein both processes are waiting for each other. Signals are now handled correctly after support was added for controlling LLDB's signal disposition. The only currently known cause of this state is a race condition during deinitialization.

The DWARF debugging symbols provide both line number information and type information. Correct operation of Ruminate is severely hampered in cases where both are missing. The only type of introspection still available in the absence of all debugging symbols is stack introspection, and source file and line number information will not be available from the stack frames thus returned. For all other features of introspection, Ruminate requires that debugging symbols be present in those modules which are to be introspected. Ruminate does not require that all modules linked in an executable possess debugging information.

9 Future Work

The original plans for the future of Ruminate was to support other debuggers (e.x. `gdb`) including those on other platforms (e.x. `windbg` on Windows). The overhead added by the RPC is however quite significant and an alternative approach should be considered.

One possible approach is to modify LLDB to support same process debugging. This would likely not fit well with the design of LLDB as it would be difficult to implement some features which LLDB relies on, such as breakpoints.

Another approach is to modify LLDB to expose its internals as libraries and leverage those libraries in order to perform introspection without the need for the debugger control process. The functionality that these libraries would need to expose includes symbol parsing including DWARF debugging symbols, call stack traversal and expression evaluation.

Another direction this project could take is to support additional sources of type information. Some compilers support emitting their AST in binary form during compilation. If embedded inside the emitted binary, that AST could be used for complete type information. This would not however provide runtime information such as stack introspection and so some additional work to implement the features of a debugger would still be necessary.

LLDB integrates with clang and LLVM in order to provide support for "expression evaluation." This means that C code can be given to LLDB as a string, and it will compile, link and execute that code in the debugee. This could potentially be leveraged in order to add "eval" functionality to Ruminate whereby a program could invoke the expression evaluation subsystem of LLVM itself.

Bibliography

[1] *abort(3) - cause abnormal process termination. Linux Programmer's Manual.* 2013. URL: http://man7.org/linux/man-pages/man3/abort.3.html (visited on 11/12/2013).

[2] Maximilien de Bayser and Renato Cerqueira. "A System for Runtime Type Introspection in C++". In: *Proceedings of the 16th Brazilian conference on Programming Languages.* SBLP'12. Natal, Brazil: Springer-Verlag, 2012, pp. 102–116. ISBN: 978-3-642-33181-7. URL: http://dx.doi.org/10.1007/978-3-642-33182-4_9.

[3] *clang: a C language family frontend for LLVM.* URL: http://clang.llvm.org/.

[4] *Class: Object (Ruby 1.9.3).* URL: http://ruby-doc.org/core-1.9.3/Object.html#method-i-instance_variables (visited on 2/19/2013).

[5] *Class::MOP::Class.* URL: http://search.cpan.org/dist/Class-MOP/lib/Class/MOP/Class.pm (visited on 2/19/2013).

[6] DWARF Standards Committee. *The DWARF Debugging Standard.* URL: http://dwarfstd.org/ (visited on 10/20/2012).

[7] Ludovic Courtès. "Systèmes tolérant les fautes à base de support d'exécution réflexifs Capture en ligne de l'état d'applications". MA thesis. Université de Franche-Comté, 2003. URL: http://www.fdn.fr/~lcourtes/software/ego/laas-dea.pdf (visited on 11/12/2013).

[8] Ludovic Courtès. *The Ego Reference Manual.* 2004. URL: http://www.fdn.fr/~lcourtes/software/ego/ego-manual.html (visited on 11/12/2013).

[9] *GDB: The GNU Project Debugger.* URL: https://www.gnu.org/software/gdb/ (visited on 2/19/2013).

[10] *GNOME.* URL: http://www.gnome.org/ (visited on 11/12/2013).

[11] *GObject Introspection.* URL: http://wiki.gnome.org/GObjectIntrospection (visited on 11/12/2013).

[12] Russell Harmon. *Enumerating the members of an enum.* Submitted to the lldb-dev mailing list. Oct. 26, 2013. URL: `http://lists.cs.uiuc.edu/pipermail/lldb-dev/2013-October/002626.html` (visited on 11/12/2013).

[13] Russell Harmon. *Ignoring Signals via the API.* Submitted to the lldb-dev mailing list. July 27, 2013. URL: `http://lists.cs.uiuc.edu/pipermail/lldb-dev/2013-July/002108.html` (visited on 11/26/2013).

[14] Russell Harmon. *Ruminate. Type Introspection for C.* 2013. URL: `http://rus.har.mn/ruminate/` (visited on 11/12/2013).

[15] JTC1/SC22/WG14. *Programming languages — C.* ISO n1570. International Organization for Standardization, 2011.

[16] K. Knizhnik. *Reflection for C++.* URL: `http://www.garret.ru/cppreflection/docs/reflect.html` (visited on 2/19/2013).

[17] Petri Lehtinen. *Jansson.* URL: `http://www.digip.org/jansson/` (visited on 2/7/2013).

[18] *LLDB python API. Class SBType.* July 19, 2013. URL: `http://lldb.llvm.org/python_reference/lldb.SBType-class.html` (visited on 2013-11-12).

[19] *Package java.lang.reflect.* URL: `http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/package-summary.html` (visited on 2/19/2013).

[20] *ptrace(2) - process trace. Linux Programmer's Manual.* 2013. URL: `http://man7.org/linux/man-pages/man2/ptrace.2.html` (visited on 11/12/2013).

[21] *Python 2.7.3 » Documentation » The Python Standard Library.* Built-in Functions. URL: `http://docs.python.org/2/library/functions.html#dir` (visited on 2/19/2013).

[22] *Reflection in C++.* URL: `http://msdn.microsoft.com/en-us/library/y0114hz2(v=vs.80).aspx` (visited on 2/19/2013).

[23] S. Roiser and P. Mato. "The Seal C++ Reflection System". In: *Proceedings of CHEP 2004.* CHEP04. (Sept. 27–Oct. 1, 2004). CERN. Interlaken, Switzerland, 2004. URL: `http://indico.cern.ch/getFile.py/access?contribId=222&resId=0&materialId=paper&confId=0` (visited on 2/19/2013).

[24] LLDB Team. *LLDB python API.* URL: `http://lldb.llvm.org/python_reference/lldb-module.html`.

[25] LLVM Team. *The LLDB Debugger.* URL: `http://lldb.llvm.org/` (visited on 10/11/2012).

[26] *The JSON Data Interchange Format*. ECMA-404 (RFC 4627). First Edition. Rue du Rhône Genève, Switzerland: ECMA International, Oct. 2013. URL: `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf` (visited on 11/12/2013).

[27] *The LLVM Compiler Infrastructure*. URL: `http://llvm.org/` (visited on 11/12/2013).

[28] *The "stabs" representation of debugging information*. URL: `http://www.sourceware.org/gdb/download/onlinedocs/stabs.html` (visited on 18/12/2013).

[29] "Working Draft, Standard for Programming Language C++". In: *ISO/IEC 14882:2011* (2011), 99, §5.2.8.

10 Appendix

Ruminate

Generated by Doxygen 1.8.4

Thu Dec 12 2013 13:15:24

Contents

1 Main Page

Ruminate is an `introspective` library for C.

Usage

To start using it, see ruminate_get_type() in `ruminate.h`

Links

You can also get a lot of information about the project from my master's degree project report, found `here`

2 Todo List

Member r_type_size (RType ∗, GError ∗∗error)

Document this

Member RAggregateMember::r_aggregate_member_name (RAggregateMember ∗member, GError ∗∗error)

Function argument names return `""`. I'm not sure it's even possible to get these, so this feature might go away.

Member RTypedefType::r_typedef_type_canonical (RTypedefType ∗, GError ∗∗error)

Be able to single step through non-canonical types of an RTypedefType.

Member ruminate_backtrace (GError ∗∗error)

This method should return a GPtrArray rather than a custom list implementation.

Member ruminate_get_type_by_variable_name (const char ∗, GError ∗∗)

document

3 Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

4 Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

5 File Index

5.1 File List

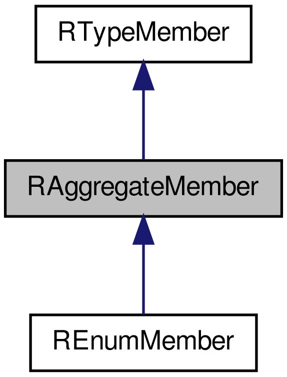
Here is a list of all documented files with brief descriptions:

6 Class Documentation

6.1 RAggregateMember Struct Reference

An opaque struct representing a aggregate member.

Inheritance diagram for RAggregateMember:



Public Member Functions

- RAggregateMemberId r_aggregate_member_id (RAggregateMember ∗member, GError ∗∗error)

  Get the real type identifier of this aggregate member.
- RString ∗ r_aggregate_member_name (RAggregateMember ∗member, GError ∗∗error)

  Get the name of this aggregate member.

6.1.1 Detailed Description

An opaque struct representing a aggregate member.

See Also

RAggregateType

6.1.2 Member Function Documentation

6.1.2.1 **RAggregateMemberId** r_aggregate_member_id (**RAggregateMember** ∗ *member,* GError ∗∗ *error*)

Get the real type identifier of this aggregate member.

Returns

the real type of this aggregate member

Parameters

`in`	*member*	the aggregate member to get the id of
`out`	*error*	see errors.h

6.1.2.2 RString ∗ r_aggregate_member_name (RAggregateMember ∗ *member,* GError ∗∗ *error*)

Get the name of this aggregate member.

Returns

a RString containing the name of this aggregate member

Todo Function argument names return `""`. I'm not sure it's even possible to get these, so this feature might go away.

Parameters

`in`	*member*	the aggregate member to get the name of
`in`	*error*	see errors.h

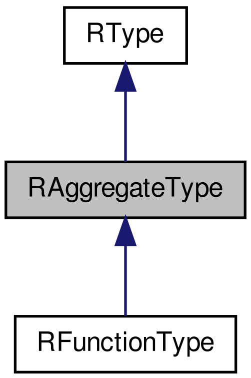
The documentation for this struct was generated from the following file:

- ruminate/aggregate_member.h

6.2 RAggregateType Struct Reference

An opaque struct representing a aggregate type.

Inheritance diagram for RAggregateType:



Public Member Functions

- RAggregateTypeId r_aggregate_type_id (RAggregateType ∗type, GError ∗∗error)

Get the aggregate type identifier of this aggregate type.

- size_t r_aggregate_type_nmembers (RAggregateType ∗type, GError ∗∗error)

  Get the number of members in this aggregate type.
- RAggregateMember ∗ r_aggregate_type_member_at (RAggregateType ∗type, size_t index, GError ∗∗error)

  Get an aggregate's member at a specified index.
- RAggregateMember ∗ r_aggregate_type_member_by_name (RAggregateType ∗type, const char ∗name, GError ∗∗error)

  Get an aggregate's member by name.

6.2.1 Detailed Description

An opaque struct representing a aggregate type.

This aggregate type can be safely cast to it's sub-type which can be determined by using r_aggregate_type_id()

6.2.2 Member Function Documentation

6.2.2.1 RAggregateTypeId r_aggregate_type_id (RAggregateType ∗ *type*, GError ∗∗ *error*)

Get the aggregate type identifier of this aggregate type.

The RAggregateTypeId of this RAggregateType represents the child type of this RAggregateType, and can be safely cast into that child type.

Returns

the child type of this aggregate type.

Parameters

in	*type*	the aggregate type to retrieve the id of
out	*error*	see errors.h

6.2.2.2 RAggregateMember ∗ r_aggregate_type_member_at (RAggregateType ∗ *type*, size_t *index*, GError ∗∗ *error*)

Get an aggregate's member at a specified index.

Returns

a RAggregateMember representing the member of this aggregate at index *index*.

Parameters

in	*type*	the aggregate type to retrieve a member of
in	*index*	the index of the member
out	*error*	see errors.h

6.2.2.3 RAggregateMember ∗ r_aggregate_type_member_by_name (RAggregateType ∗ *type*, const char ∗ *name*, GError ∗∗ *error*)

Get an aggregate's member by name.

Returns

a RAggregateMember representing the member of this aggregate with name *name*.

Parameters

`in`	*type*	the aggregate type to retrieve a member of
`in`	*name*	the name of the aggregate member
`out`	*error*	see errors.h

6.2.2.4 size_t r_aggregate_type_nmembers (RAggregateType ∗ *type,* GError ∗∗ *error*)

Get the number of members in this aggregate type.

Returns

the number of members in this aggregate type.

Parameters

`in`	*type*	the type to get the number of members of
`out`	*error*	see errors.h

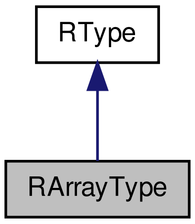
The documentation for this struct was generated from the following file:

- ruminate/aggregate_type.h

6.3 RArrayType Struct Reference

An opaque struct representing an array type.

Inheritance diagram for RArrayType:



Public Member Functions

- size_t r_array_type_size (RArrayType ∗type, GError ∗∗error)

  Get the size of this array.
- RTypeMember ∗ r_array_type_member_at (RArrayType ∗type, size_t index, GError ∗∗error)

  Get the type of a member of this array.

6.3.1 Detailed Description

An opaque struct representing an array type.

6.3.2 Member Function Documentation

6.3.2.1 RTypeMember ∗ r_array_type_member_at (RArrayType ∗ *type,* size_t *index,* GError ∗∗ *error*)

Get the type of a member of this array.

Returns

A RTypeMember representing the type of the argument at index *index*

Parameters

in	*type*	the type to get the member type of
in	*index*	the index in the array to get the member type of
out	*error*	see errors.h

6.3.2.2 size_t r_array_type_size (RArrayType ∗ *type,* GError ∗∗ *error*)

Get the size of this array.

Returns

the size of the array

Parameters

in	*type*	the type to get the size of
out	*error*	see errors.h

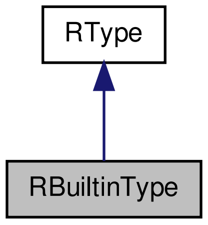
The documentation for this struct was generated from the following file:

- ruminate/array_type.h

6.4 RBuiltinType Struct Reference

An opaque struct representing a builtin type.

Inheritance diagram for RBuiltinType:



Public Member Functions

- RBuiltinTypeId r_builtin_type_id (RBuiltinType ∗type, GError ∗∗error)

  Get the builtin type identifier of this builtin type.
- bool r_builtin_type_is_signed (RBuiltinType ∗type, GError ∗∗error)

  Determine if this type is signed.
- bool r_builtin_type_is_unsigned (RBuiltinType ∗type, GError ∗∗error)

  Determine if this type is unsigned.

6.4.1 Detailed Description

An opaque struct representing a builtin type.

6.4.2 Member Function Documentation

6.4.2.1 RBuiltinTypeId r_builtin_type_id (RBuiltinType ∗ *type,* GError ∗∗ *error*)

Get the builtin type identifier of this builtin type.

The RBuiltinTypeId of this RBuiltinType represents the real type of this builtin type.

Returns

the real type of this builtin type

Parameters

`in`	*type*	the builtin type to retrieve the id of
`out`	*error*	see errors.h

6.4.2.2 bool r_builtin_type_is_signed (RBuiltinType ∗ *type,* GError ∗∗ *error*)

Determine if this type is signed.

Note that the `char` type can be neither signed nor unsigned.

Returns

whether or not this type is signed

Parameters

`in`	*type*	the type to determine the signedness of
`out`	*error*	see errors.h

6.4.2.3 bool r_builtin_type_is_unsigned (RBuiltinType ∗ *type,* GError ∗∗ *error*)

Determine if this type is unsigned.

Note that the `char` type can be neither signed nor unsigned.

Returns

whether or not this type is unsigned

Parameters

`in`	*type*	the type to determine the signedness of
`out`	*error*	see errors.h

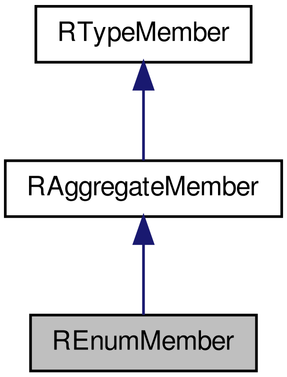
The documentation for this struct was generated from the following file:

- ruminate/builtin_type.h

6.5 REnumMember Struct Reference

An opaque struct representing an enum member.

Inheritance diagram for REnumMember:



Public Member Functions

- intmax_t r_enum_member_value_signed (REnumMember ∗member, GError ∗∗error)

    Get the signed value of this enum member.
- uintmax_t r_enum_member_value_unsigned (REnumMember ∗member, GError ∗∗error)

    Get the unsigned value of this enum member.

6.5.1 Detailed Description

An opaque struct representing an enum member.

6.5.2 Member Function Documentation

6.5.2.1 intmax_t r_enum_member_value_signed (REnumMember ∗ *member,* GError ∗∗ *error*)

Get the signed value of this enum member.

Returns

the signed value of this enum member

Parameters

`in`	*member*	the enum member to get the value of
`out`	*error*	see errors.h

6.5.2.2 uintmax_t r_enum_member_value_unsigned (**REnumMember** ∗ *member,* GError ∗∗ *error*)

Get the unsigned value of this enum member.

Returns

the unsigned value of this enum member

Parameters

`in`	*member*	the enum member to get the value of
`out`	*error*	see errors.h

The documentation for this struct was generated from the following file:

- ruminate/enum_member.h

6.6 RFrame Struct Reference

An opaque struct representing a call stack frame.

Public Member Functions

- void r_frame_ref (RFrame ∗frame)

  Increment the reference count of this RFrame.
- void r_frame_unref (RFrame ∗frame)

  Decrement the reference count of this RFrame.
- RString ∗ r_frame_function_name (RFrame ∗frame, GError ∗∗error)

  Get the name of the function this RFrame represents.
- RString ∗ r_frame_module_name (RFrame ∗frame, GError ∗∗error)

  Get the name of the module which contains this frame.
- RString ∗ r_frame_compile_unit_name (RFrame ∗frame, GError ∗∗error)

  Get the name of the compile unit which contains this frame.
- RType ∗ r_frame_function_type (RFrame ∗frame, GError ∗∗error)

  Get the type of this function.
- uintmax_t r_frame_line (RFrame ∗frame, GError ∗∗error)

  Get the line number that this frame is at.

6.6.1 Detailed Description

An opaque struct representing a call stack frame.

6.6.2 Member Function Documentation

6.6.2.1 RString ∗ r_frame_compile_unit_name (RFrame ∗ *frame,* GError ∗∗ *error*)

Get the name of the compile unit which contains this frame.

This is usually the name of the file which defined this function.

Returns

a RString containing the name of this compile unit.

Parameters

`in`	*frame*	the frame to get the compile unit name of
`out`	*error*	see errors.h

6.6.2.2 RString ∗ r_frame_function_name (RFrame ∗ *frame,* GError ∗∗ *error*)

Get the name of the function this RFrame represents.

Returns

a RString containing the name of this function.

Parameters

`in`	*frame*	the frame to get the name of
`out`	*error*	see errors.h

6.6.2.3 RType ∗ r_frame_function_type (RFrame ∗ *frame,* GError ∗∗ *error*)

Get the type of this function.

Returns

an RType representing the type of this function.

Parameters

`in`	*frame*	the frame to get the type of
`out`	*error*	see errors.h

6.6.2.4 uintmax_t r_frame_line (RFrame ∗ *frame,* GError ∗∗ *error*)

Get the line number that this frame is at.

Returns

the line number that this frame is at

Parameters

`in`	*frame*	the frame to get the line number of
`out`	*error*	see errors.h

6.6.2.5 RString ∗ r_frame_module_name (RFrame ∗ *frame,* GError ∗∗ *error*)

Get the name of the module which contains this frame.

This is usually the name of the executable or library.

Returns

a RString containing the name of this module.

Parameters

`in`	*frame*	the frame to get the module name of
`out`	*error*	see errors.h

6.6.2.6 void r_frame_ref (RFrame ∗ *frame*)

Increment the reference count of this RFrame.

Parameters

`in`	*frame*	the frame to increment the reference count of

6.6.2.7 void r_frame_unref (RFrame ∗ *frame*)

Decrement the reference count of this RFrame.

The RFrame will be freed if it's reference count drops to zero.

Parameters

`in`	*frame*	the frame to decrement the reference count of

The documentation for this struct was generated from the following file:

- ruminate/frame.h

6.7 RFrameList Struct Reference

An opaque struct representing a call stack.

Public Member Functions

- size_t r_frame_list_size (RFrameList ∗list, GError ∗∗error)

  Get the number of elements in this frame list.
- RFrame ∗ r_frame_list_at (RFrameList ∗list, size_t index, GError ∗∗error)

  Get an RFrame from this RFrameList.
- void r_frame_list_ref (RFrameList ∗list)

  Increment the reference count on this RFrameList.
- void r_frame_list_unref (RFrameList ∗list)

  Decrement the reference count of this RFrameList.

6.7.1 Detailed Description

An opaque struct representing a call stack.

A call stack is a list of one or more RFrame instances.

See Also

RFrame

6.7.2 Member Function Documentation

6.7.2.1 RFrame ∗ r_frame_list_at (RFrameList ∗ *list,* size_t *index,* GError ∗∗ *error*)

Get an RFrame from this RFrameList.

Returns

the RFrame at index *index*

Parameters

`in`	*list*	the frame list to get an element from
`in`	*index*	the index of the RFrame to get
`out`	*error*	see errors.h

6.7.2.2 void r_frame_list_ref (RFrameList ∗ *list*)

Increment the reference count on this RFrameList.

Parameters

`in`	*list*	the frame list to increment the reference count of

6.7.2.3 size_t r_frame_list_size (RFrameList ∗ *list,* GError ∗∗ *error*)

Get the number of elements in this frame list.

Returns

the number of elements in this frame list.

Parameters

`in`	*list*	the frame list to get the size of
`out`	*error*	see errors.h

6.7.2.4 void r_frame_list_unref (RFrameList ∗ *list*)

Decrement the reference count of this RFrameList.

The RFrameList will be freed if it's reference count drops to zero.

Parameters

in	*list*	the frame list to decrement the reference count of

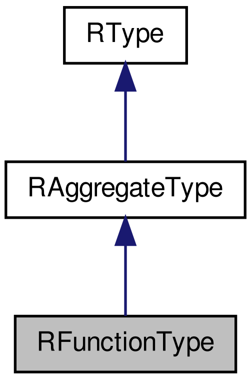
The documentation for this struct was generated from the following file:

- ruminate/frame.h

6.8 RFunctionType Struct Reference

An opaque struct representing a function.

Inheritance diagram for RFunctionType:



Public Member Functions

- RType ∗ r_function_type_return_type (RFunctionType ∗type, GError ∗∗error)

    Get the return type of this function.

6.8.1 Detailed Description

An opaque struct representing a function.

6.8.2 Member Function Documentation

6.8.2.1 RType ∗ r_function_type_return_type (RFunctionType ∗ *type*, GError ∗∗ *error*)

Get the return type of this function.

Returns

An RType representing the return type of this function.

Parameters

`in`	*type*	the function to get the return type of
`out`	*error*	see errors.h

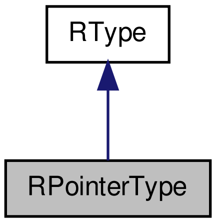
The documentation for this struct was generated from the following file:

- ruminate/function_type.h

6.9 RPointerType Struct Reference

An opaque struct representing a pointer to another type.

Inheritance diagram for RPointerType:



Public Member Functions

- RType $*$ r_pointer_type_pointee (RPointerType $*$type, GError $**$error)

  Get the type that this RPointerType points to.

6.9.1 Detailed Description

An opaque struct representing a pointer to another type.

6.9.2 Member Function Documentation

6.9.2.1 RType $*$ r_pointer_type_pointee (RPointerType $*$ *type,* GError $**$ *error*)

Get the type that this RPointerType points to.

Returns

The type that this RPointerType points to.

Parameters

in	*type*	the type to dereference
out	*error*	see errors.h

The documentation for this struct was generated from the following file:

- ruminate/pointer_type.h

6.10 RString Struct Reference

An opaque struct representing a string.

Public Member Functions

- RString ∗ r_string_ref (RString ∗string)

  Increase the reference count on this RString.
- void r_string_unref (RString ∗string)

  Decrease the reference count on this RString.
- const char ∗ r_string_bytes (RString ∗string)

  Get the C-style string (array of characters) backing this RString.
- size_t r_string_length (RString ∗string)

  Get the length of this RString.

6.10.1 Detailed Description

An opaque struct representing a string.

An RString is a reference counted array of characters.

6.10.2 Member Function Documentation

6.10.2.1 const char ∗ r_string_bytes (RString ∗ *string*)

Get the C-style string (array of characters) backing this RString.

The behavior is undefined if this array is modified.

Returns

the array of characters backing this RString

Parameters

in	*string*	the string to get the array of characters of

6.10.2.2 size_t r_string_length (RString ∗ *string*)

Get the length of this RString.

Returns

the length of this RString

Parameters

`in`	*string*	the string to get the length of

6.10.2.3 RString ∗ r_string_ref (RString ∗ *string*)

Increase the reference count on this RString.

Returns

string

Parameters

`in`	*string*	the string to increase the reference count of

6.10.2.4 void r_string_unref (RString ∗ *string*)

Decrease the reference count on this RString.

If the reference count of this RString drops to zero, the string will be freed.

Parameters

`in`	*string*	the string to decrease the reference count of

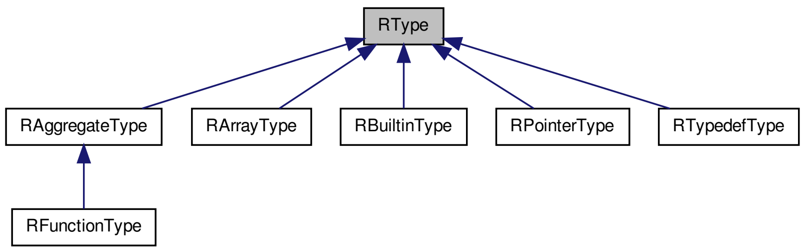
The documentation for this struct was generated from the following file:

- ruminate/string.h

6.11 RType Struct Reference

An opaque struct representing a type.

Inheritance diagram for RType:



Public Member Functions

- RTypeId r_type_id (RType ∗type, GError ∗∗error)

  Get the type identifier of this type.
- RString ∗ r_type_name (RType ∗type, GError ∗∗error)

Get the name of this type.

- RType ∗ r_type_ref (RType ∗type)

  Increase the reference count of this type.

- void r_type_unref (RType ∗type)

  Decrease the reference count of this.

- RType ∗ r_type_pointer (RType ∗type, GError ∗∗error)

  Get an RType representing a pointer to this type.

6.11.1 Detailed Description

An opaque struct representing a type.

This type can be safely cast to it's sub-type which can be determined using r_type_id().

6.11.2 Member Function Documentation

6.11.2.1 RTypeId r_type_id (RType ∗ *type*, GError ∗∗ *error*)

Get the type identifier of this type.

The RTypeId of this RType represents the child type of this RType, and can be safely cast into that child type.

Returns

the child type of this type.

Parameters

`in`	*type*	the type to retrieve the type id of
`out`	*error*	see errors.h

6.11.2.2 RString ∗ r_type_name (RType ∗ *type*, GError ∗∗ *error*)

Get the name of this type.

Returns

an RString containing the name of this type.

Parameters

`in`	*type*	the type to retrieve the name of
`out`	*error*	see errors.h

6.11.2.3 RType ∗ r_type_pointer (RType ∗ *type*, GError ∗∗ *error*)

Get an RType representing a pointer to this type.

Returns

An RType representing a pointer to *type*.

Parameters

in	*type*	the type to get a pointer to
out	*error*	see errors.h

6.11.2.4 RType $*$ r_type_ref (RType $*$ *type*)

Increase the reference count of this type.

Returns

type

Parameters

in	*type*	the type to increase the reference count of

6.11.2.5 void r_type_unref (RType $*$ *type*)

Decrease the reference count of this.

This RType will be freed if the reference count drops to zero.

Parameters

in	*type*	the type to decrease the reference count of

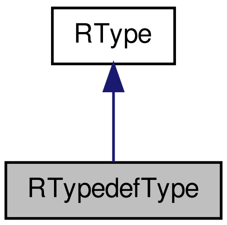
The documentation for this struct was generated from the following file:

- ruminate/type.h

6.12 RTypedefType Struct Reference

An opaque struct representing a typedef'ed type.

Inheritance diagram for RTypedefType:



Public Member Functions

- RType $*$ r_typedef_type_canonical (RTypedefType $*$, GError $**$error)

Get the canonical type of this type.

6.12.1 Detailed Description

An opaque struct representing a typedef'ed type.

6.12.2 Member Function Documentation

6.12.2.1 RType ∗ r_typedef_type_canonical (RTypedefType ∗ , GError ∗∗ *error*)

Get the canonical type of this type.

A canonical type strips away all typedefs contained within this type.

For example with the following code,

```
typedef char *String;
typedef String *StringArray;
```

calling r_typedef_type_canonical() on a RTypedefType representing a `StringArray` will return an RType representing `char **`.

Todo Be able to single step through non-canonical types of an RTypedefType.

Returns

An RType representing the canonical type of this RTypedefType.

Parameters

out	*error*	see errors.h

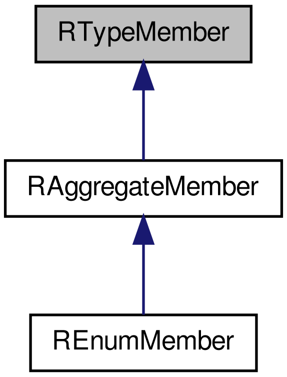
The documentation for this struct was generated from the following file:

- ruminate/typedef_type.h

6.13 RTypeMember Struct Reference

An opaque struct representing a type member.

Inheritance diagram for RTypeMember:



Public Member Functions

- RTypeMemberId r_type_member_id (RTypeMember ∗member, GError ∗∗error)

  Get the type member id of this RTypeMember.
- RType ∗ r_type_member_type (RTypeMember ∗member, GError ∗∗error)

  Get the type of this type member.
- ptrdiff_t r_type_member_offset (RTypeMember ∗member, GError ∗∗error)

  Get the offset of this type member into it's container.
- RTypeMember ∗ r_type_member_ref (RTypeMember ∗member)

  Increment the reference count of this RTypeMember.
- void r_type_member_unref (RTypeMember ∗member)

  Decrement the reference count of this RTypeMember.

6.13.1 Detailed Description

An opaque struct representing a type member.

6.13.2 Member Function Documentation

6.13.2.1 RTypeMemberId r_type_member_id (RTypeMember ∗ *member,* GError ∗∗ *error*)

Get the type member id of this RTypeMember.

Parameters

in	*member*	the type member to get the id of
out	*error*	see errors.h

6.13.2.2 ptrdiff_t r_type_member_offset (**RTypeMember** ∗ *member,* **GError** ∗∗ *error*)

Get the offset of this type member into it's container.

This is the number of bytes into this type member's container (either an RArrayType, or an RAggregateType) that this member is located.

See Also

RArrayType
RAggregateType

Parameters

in	*member*	the type member to get the offset of
out	*error*	see errors.h

6.13.2.3 RTypeMember ∗ r_type_member_ref (**RTypeMember** ∗ *member*)

Increment the reference count of this RTypeMember.

Parameters

in	*member*	the type member to increment the reference count of

6.13.2.4 RType ∗ r_type_member_type (**RTypeMember** ∗ *member,* **GError** ∗∗ *error*)

Get the type of this type member.

Parameters

in	*member*	the type member to get the type of
out	*error*	see errors.h

6.13.2.5 void r_type_member_unref (**RTypeMember** ∗ *member*)

Decrement the reference count of this RTypeMember.

If the reference count reaches zero, this RTypeMember will be freed.

Parameters

in	*member*	the type member to decrement the reference count of

The documentation for this struct was generated from the following file:

- ruminate/type_member.h

7 File Documentation

7.1 ruminate/aggregate_member.h File Reference

Aggregate members.

Enumerations

- enum RAggregateMemberId {
  R_AGGREGATE_MEMBER_BITFIELD,
  R_AGGREGATE_MEMBER_ENUM,
  R_AGGREGATE_MEMBER_OTHER }

  An identifier denoting the real type of this RAggregateMember.

7.1.1 Detailed Description

Aggregate members. A RAggregateMember represents a member of an aggregate type.

See Also

RAggregateMember
RAggregateType

7.1.2 Enumeration Type Documentation

7.1.2.1 enum **RAggregateMemberId**

An identifier denoting the real type of this RAggregateMember.

This identifier can be retrieved using r_aggregate_member_id().

Enumerator

R_AGGREGATE_MEMBER_BITFIELD a bitfield

R_AGGREGATE_MEMBER_ENUM an instance of REnumMember

See Also

REnumMember

R_AGGREGATE_MEMBER_OTHER a "normal" type (non enum-member nor bitfield)

7.2 ruminate/aggregate_type.h File Reference

Aggregate types.

Enumerations

- enum RAggregateTypeId {
  R_AGGREGATE_TYPE_STRUCT,
  R_AGGREGATE_TYPE_UNION,
  R_AGGREGATE_TYPE_ENUM,
  R_AGGREGATE_TYPE_FUNCTION,
  R_AGGREGATE_TYPE_UNKNOWN }

  An identifier denoting the child type of this RAggregateType.

7.2.1 Detailed Description

Aggregate types. A RAggregateType represents an aggregate type (`struct`, `union`, `enum` or `function`)

See Also

RAggregateType

7.2.2 Enumeration Type Documentation

7.2.2.1 enum RAggregateTypeId

An identifier denoting the child type of this RAggregateType.

This identifier can be retrieved using r_aggregate_type_id().

Enumerator

R_AGGREGATE_TYPE_STRUCT a `struct`

R_AGGREGATE_TYPE_UNION a `union`

R_AGGREGATE_TYPE_ENUM an `enum`

R_AGGREGATE_TYPE_FUNCTION a function type

See Also

RFunctionType

R_AGGREGATE_TYPE_UNKNOWN an unknown type

7.3 ruminate/builtin_type.h File Reference

Built-in types.

Enumerations

- enum RBuiltinTypeId {
  R_BUILTIN_TYPE_INT,
  R_BUILTIN_TYPE_LONG,
  R_BUILTIN_TYPE_DOUBLE,
  R_BUILTIN_TYPE_SHORT,
  R_BUILTIN_TYPE_CHAR,
  R_BUILTIN_TYPE_VOID,
  R_BUILTIN_TYPE_BOOL,
  R_BUILTIN_TYPE_UNKNOWN }

  An identifier denoting the real type of this RBuiltinType.

7.3.1 Detailed Description

Built-in types. A RBuintinType represents a built-in type (`int`, `double`, etc.)

See Also

RBuiltinType

7.3.2 Enumeration Type Documentation

7.3.2.1 enum RBuiltinTypeId

An identifier denoting the real type of this RBuiltinType.

This identifier can be retrieved using r_builtin_type_id().

Enumerator

R_BUILTIN_TYPE_INT	an int
R_BUILTIN_TYPE_LONG	a long
R_BUILTIN_TYPE_DOUBLE	a double
R_BUILTIN_TYPE_SHORT	a short
R_BUILTIN_TYPE_CHAR	a char
R_BUILTIN_TYPE_VOID	the void type
R_BUILTIN_TYPE_BOOL	a bool
R_BUILTIN_TYPE_UNKNOWN	an unknown type

7.4 ruminate/errors.h File Reference

Error handling facilities.

Macros

- #define RUMINATE_ERROR

  The error quark representing errors produced by this library.
- #define RUMINATE_ERRNO_ERROR

  The error quark representing errors produced by the standard C library.

Enumerations

- enum RuminateError {
  RUMINATE_ERROR_SB_INVALID,
  RUMINATE_ERROR_LLDB_ERROR,
  RUMINATE_ERROR_RANGE,
  RUMINATE_ERROR_NO_PRIMITIVE_TYPE,
  RUMINATE_ERROR_INVALID_TYPE,
  RUMINATE_ERROR_INCOMPLETE_TYPE,
  RUMINATE_ERROR_ICE,
  RUMINATE_ERROR_STDLIB,
  RUMINATE_ERROR_SHORT_READ,
  RUMINATE_ERROR_NO_PRGNAME,
  RUMINATE_ERROR_UNIMPLEMENTED,
  RUMINATE_ERROR_UNSPEC }

  The various errors produced by ruminate.

7.4.1 Detailed Description

Error handling facilities. Every function which takes as an argument a `GError **` reports errors through this pointer.

In brief, pass `NULL` as the `GError **` argument to functions or pass a pointer to a `NULL` `GError *` to recieve a `GError` when an error occurs.

See Also

`GError`

7.4.2 Macro Definition Documentation

7.4.2.1 #define RUMINATE_ERRNO_ERROR

The error quark representing errors produced by the standard C library.

This quark will be placed in the `domain` field of a `GError` produced when an error occurrs.

See Also

`GQuark`

7.4.2.2 #define RUMINATE_ERROR

The error quark representing errors produced by this library.

This quark will be placed in the `domain` field of a `GError` produced when an error occurrs.

See Also

`GQuark`

7.4.3 Enumeration Type Documentation

7.4.3.1 enum **RuminateError**

The various errors produced by ruminate.

These will be placed in the `code` field of a `GError` produced when an error occurrs.

7.5 ruminate/memory.h File Reference

Typed reference counted memory allocator.

Macros

- #define r_mem_malloc(type, error)

    Allocate typed memory.
- #define r_mem_malloc_sized(type, size, error)

    Allocate typed memory with size.
- #define r_mem_calloc(type, nmemb, error)

    Allocate zero'ed typed memory.
- #define r_mem_calloc_sized(type, size, nmemb, error)

    Allocate zero'ed typed memory, with size.

Functions

- void ∗ r_mem_malloc_fn (RType ∗type, GError ∗∗error)

  Allocate typed memory.
- void ∗ r_mem_malloc_sized_fn (RType ∗type, size_t size, GError ∗∗error)

  Allocate typed memory with size.
- void ∗ r_mem_calloc_fn (RType ∗type, size_t nmemb, GError ∗∗error)

  Allocate zero'ed typed memory.
- void ∗ r_mem_calloc_sized_fn (RType ∗type, size_t size, size_t nmemb, GError ∗∗error)

  Allocate zero'ed typed memory, with size.
- size_t r_mem_size (void ∗mem)

  Retrieve the allocated size of typed memory.
- void ∗ r_mem_ref (void ∗mem)

  Increment the reference count of typed memory.
- void r_mem_unref (void ∗mem)

  Decrease the reference count of typed memory.
- RType ∗ r_mem_type (void ∗mem)

  Retrieve the type of typed memory.

7.5.1 Detailed Description

Typed reference counted memory allocator. These functions provide facilities for allocating dynamic memory which you can retrieve the type of via a call to r_mem_type(). This allows you to retrieve the real type of e.x. a `void *`.

This memory is also reference counted via calls to r_mem_ref() and r_mem_unref().

7.5.2 Macro Definition Documentation

7.5.2.1 #define r_mem_calloc(*type*, *nmemb*, *error*)

Allocate zero'ed typed memory.

This macro allocates enough memory for *nmemb* instances of *type* sized elements and sets the allocated memory to zero.

Parameters

`in`	*type*	the type of the memory to allocate
`in`	*nmemb*	the number of *type* sized members to allocate.
`out`	*error*	see errors.h

Returns

a pointer to dynamically allocated memory

7.5.2.2 #define r_mem_calloc_sized(*type*, *size*, *nmemb*, *error*)

Allocate zero'ed typed memory, with size.

This macro allocates enough memory for *nmemb* instances of *size* sized elements and sets the allocated memory to zero.

Parameters

`in`	*type*	the type of the memory to allocate
`in`	*size*	the size of the memory to allocate. This must be at least as large as the type represented by *type*
`in`	*nmemb*	the number of *type* sized members to allocate.
`out`	*error*	see errors.h

Returns

a pointer to dynamically allocated memory

7.5.2.3 #define r_mem_malloc(*type, error*)

Allocate typed memory.

This macro allocates memory of size `sizeof(type)` and with type *type*.

This memory must be freed via a call to r_mem_unref().

Parameters

`in`	*type*	the type of the memory to allocate
`out`	*error*	see errors.h

Returns

a pointer to dynamically allocated memory

7.5.2.4 #define r_mem_malloc_sized(*type, size, error*)

Allocate typed memory with size.

This macro allocates memory of size *size* and with type *type*.

This memory must be freed via a call to r_mem_unref().

Parameters

`in`	*type*	the type of the memory to allocate
`in`	*size*	the size of the memory to allocate This must be at least as large as the type represented by *type*
`out`	*error*	see errors.h

Returns

a pointer to dynamically allocated memory

7.5.3 Function Documentation

7.5.3.1 void∗ r_mem_calloc_fn (RType ∗ *type,* size_t *nmemb,* GError ∗∗ *error*)

Allocate zero'ed typed memory.

This is the function version of r_mem_calloc()

See Also

r_mem_calloc

Returns

a pointer to dynamically allocated memory

Parameters

in	*type*	an RType representing the type of the allocated memory
in	*nmemb*	the number of *type* sized elements to allocate memory for
out	*error*	see errors.h

7.5.3.2 void∗ r_mem_calloc_sized_fn (RType ∗ *type*, size_t *size*, size_t *nmemb*, GError ∗∗ *error*)

Allocate zero'ed typed memory, with size.

This is the function version of r_mem_calloc_sized()

See Also

r_mem_calloc_sized

Returns

a pointer to dynamically allocated memory

Parameters

in	*type*	an RType representing the type of the allocated memory
in	*size*	size the size of the memory to allocate. This must be at least as large as the type represented by *type*
in	*nmemb*	the number of *type* sized elements to allocate memory for
out	*error*	see errors.h

7.5.3.3 void∗ r_mem_malloc_fn (RType ∗ *type*, GError ∗∗ *error*)

Allocate typed memory.

This is the function version of r_mem_malloc()

See Also

r_mem_malloc

Returns

a pointer to dynamically allocated memory

Parameters

in	*type*	an RType representing the type of the allocated memory
out	*error*	see errors.h

7.5.3.4 void∗ r_mem_malloc_sized_fn (RType ∗ *type*, size_t *size*, GError ∗∗ *error*)

Allocate typed memory with size.

This is the function version of r_mem_malloc_sized().

See Also

r_mem_malloc_sized

Returns

a pointer to dynamically allocated memory

Parameters

in	*type*	an RType representing the type of the allocated memory
in	*size*	size the size of the memory to allocate. This must be at least as large as the type represented by *type*
out	*error*	see errors.h

7.5.3.5 void∗ r_mem_ref (void ∗ *mem*)

Increment the reference count of typed memory.

Returns

mem

Parameters

in	*mem*	the memory to increase the reference count of

7.5.3.6 size_t r_mem_size (void ∗ *mem*)

Retrieve the allocated size of typed memory.

Returns

the allocated size of *mem*

Parameters

in	*mem*	the typed memory to retrieve the allocated size of

7.5.3.7 RType∗ r_mem_type (void ∗ *mem*)

Retrieve the type of typed memory.

Returns

an RType representing the type of *mem*.

Parameters

in	*mem*	the memory to retrieve the type of

7.5.3.8 void r_mem_unref (void ∗ *mem*)

Decrease the reference count of typed memory.

This function frees *mem* if the reference count reaches zero.

Parameters

in	*mem*	the memory to decrease the reference count of

7.6 ruminate/ruminate.h File Reference

Top-level and utility functions.

Macros

- #define ruminate_get_type(expr, error)

    Get the type of an expression.

Functions

- bool ruminate_destroy (GError ∗∗error)

    De-initialize the ruminate framework.
- bool ruminate_init (const char ∗program_name, GError ∗∗error)

    Initialize the ruminate framework.
- RFrameList ∗ ruminate_backtrace (GError ∗∗error)

    Generate a backtrace.
- GPtrArray ∗ ruminate_get_types_by_name (const char ∗type_name, GError ∗∗error)

    Retrieve RTypes by name.
- RType ∗ ruminate_get_type_by_variable_name (const char ∗, GError ∗∗)
- RString ∗ ruminate_get_function_name (void ∗addr, GError ∗∗error)

    Get the name of a function by address.

7.6.1 Detailed Description

Top-level and utility functions.

7.6.2 Macro Definition Documentation

7.6.2.1 #define ruminate_get_type(*expr*, *error*)

Get the type of an expression.

Gets an instance of an RType representing the type of the provided expression.

Note that you must first have initialized the ruminate library via a call to ruminate_init().

Parameters

`in`	*expr*	The expression to determine the type of.
`out`	*error*	see errors.h

Returns

A pointer to an RType or `NULL` if an error occurred. This RType must be freed using r_type_unref().

7.6.3 Function Documentation

7.6.3.1 RFrameList$*$ ruminate_backtrace (GError $**$ *error*)

Generate a backtrace.

This function generates a backtrace of the caller's call stack.

Returns

A pointer to an RFrameList representing the frames found in the caller's call stack. This RFrameList must be freed using r_frame_list_unref().

Todo This method should return a GPtrArray rather than a custom list implementation.

Parameters

`out`	*error*	see errors.h

7.6.3.2 bool ruminate_destroy (GError $**$ *error*)

De-initialize the ruminate framework.

This function frees all internal resources of the ruminate framework. Undefined behavior results if any ruminate framework functions are called after this function returns `true`.

Returns

Whether or not an error occurred.

Parameters

`out`	*error*	see errors.h

7.6.3.3 RString$*$ ruminate_get_function_name (void $*$ *addr*, GError $**$ *error*)

Get the name of a function by address.

Returns

A RString containing the name of the function.

Parameters

in	*addr*	the address of the function to get the name of
out	*error*	see errors.h

7.6.3.4 RType∗ ruminate_get_type_by_variable_name (const char ∗ , GError ∗∗)

Todo document

7.6.3.5 GPtrArray∗ ruminate_get_types_by_name (const char ∗ *type_name*, GError ∗∗ *error*)

Retrieve RTypes by name.

This function retrieves all the types which are named *type_name*.

Returns

A GPtrArray of the types which are named *type_name*

Parameters

in	*type_name*	the name of the types to find
out	*error*	see errors.h

7.6.3.6 bool ruminate_init (const char ∗ *program_name*, GError ∗∗ *error*)

Initialize the ruminate framework.

This must be called before any other ruminate functions.

The argument *program_name* must either be the name of this program or `NULL`. If null, `g_get_prgname()` will be called to get the name of this program. If the program name has not been previously set via a call to `g_set_prgname()`, an error will occur.

Returns

Whether or not an error occurred.

See Also

`g_set_prgname`

Parameters

in	*program_name*	the name of this program, e.g. `argv[0]` or `NULL`
out	*error*	see errors.h

7.7 ruminate.h File Reference

The only file you should need to include.

7.7.1 Detailed Description

The only file you should need to include.

7.8 ruminate/type.h File Reference

The top level of the ruminate type hierarchy.

Enumerations

- enum RTypeId {
  R_TYPE_BUILTIN,
  R_TYPE_AGGREGATE,
  R_TYPE_TYPEDEF,
  R_TYPE_POINTER,
  R_TYPE_ARRAY,
  R_TYPE_UNKNOWN }

  An identifier denoting the child type of this RType.

Functions

- size_t r_type_size (RType ∗, GError ∗∗error)

7.8.1 Detailed Description

The top level of the ruminate type hierarchy.

See Also

RType

7.8.2 Enumeration Type Documentation

7.8.2.1 enum RTypeId

An identifier denoting the child type of this RType.

This identifier can be retrieved using r_type_id().

Enumerator

R_TYPE_BUILTIN a builtin type

See Also

RBuiltinType

R_TYPE_AGGREGATE an aggregate type

See Also

RAggregateType

R_TYPE_TYPEDEF a typedef

See Also

RTypedefType

R_TYPE_POINTER a pointer

See Also

RPointerType

R_TYPE_ARRAY an array

See Also

RArrayType

R_TYPE_UNKNOWN an unknown type

7.8.3 Function Documentation

7.8.3.1 size_t r_type_size (RType ∗ , GError ∗∗ *error*)

Todo Document this

7.9 ruminate/type_member.h File Reference

Type members.

Enumerations

- enum RTypeMemberId {
  R_TYPE_MEMBER_AGGREGATE,
  R_TYPE_MEMBER_ARRAY }

  An identifier denoting the child type of this RTypeMember.

7.9.1 Detailed Description

Type members. A RTypeMember represents a member of an array (RArrayType) or aggregate (RAggregateType) type.

See Also

RTypeMember

7.9.2 Enumeration Type Documentation

7.9.2.1 enum RTypeMemberId

An identifier denoting the child type of this RTypeMember.

This identifier can be retrieved using r_type_member_id().

Enumerator

R_TYPE_MEMBER_AGGREGATE a RAggregateMember

See Also

RAggregateMember

R_TYPE_MEMBER_ARRAY an array member

Index