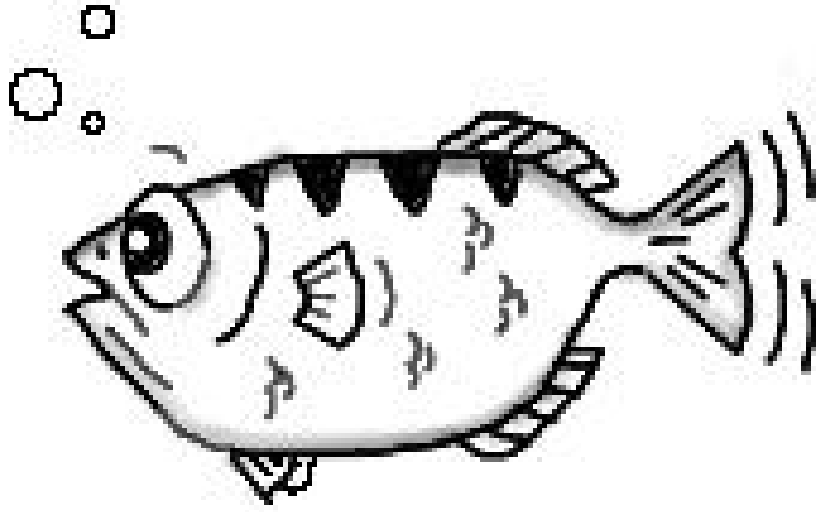
gdb: C++ conversion & dogfood

GDB: C++ convertion & dogfood

Pedro Alves <palves@redhat.com>

GNU Cauldron 2017



Outline

- GDB's C++ conversion
  - Status
  - What's next
- C++ debugging improvements
  - Breakpoints and namespaces
  - Breakpoints and ABI tags
  - TAB completion improvements

GDB's C++ conversion

C++ conversion, why?

- GDB was/is already written in poor-man's C++-in-C. Examples:
  - OO / subclasses: struct breakpoint. struct target_ops. Several more.
  - Virtual functions:
    - target_ops, breakpoint_ops, languages, values, more…
  - Overloaded functions.
  - Templates: observer, VEC, QUEUE, more…
  - Exceptions: used ubiquitously, along with …
  - RAII: Cleanups (since GDB 2.0, 1987 at least!)
    - but they are dynamic and more error-prone.

C++ conversion

What	When
Several attempts (public, private)	$time_immemorial
Global Maintainers reach consensus	2015/02
C++ by default (C still supported)	2016/04
GDB 7.12 branches	2016/08
GNU Cauldron 2016	2016/09
Master goes C++-only (drop C support)	2016/09
gdb::unique_ptr (C++03 replacement / C++11 catalyst)	2016/10
Require C++11!	2016/10
GDB 8.0 is released	2017/06

C++ conversion, requirements

- Baseline is gcc 4.8
- GDB's policy for migrating to C++NN
  - Must wait until the oldest compiler that supports C++NN is at least 3 years old.

C++ conversion, the immediate plan last September

- C++-ify C-style inheritance and tables of function pointers
  - struct target_ops; struct breakpoint_ops
  - struct serial_ops; struct extension_language_script_ops
  - struct value (lval_computed for starters)
  - several others...
- Use standard containers and algorithms.
- Wrap Python objects with C++ classes that manage refcounts?
- Cleanups, cleanups, cleanups (I mean, "struct cleanup")

C++ conversion, the immediate plan last September (II)

- All `make_cleanup` calls needs to be replaced (yes, all ~1900 uses...)
  - => objects with destructors / RAII
  - => smart pointers
- Until then, must keep using the TRY/CATCH macros.

Cleanups, a glimpse


```
extern int function_that_may_throw (const char *);
int foo (int num)
{
  struct cleanup *old_chain;
  int res;
  char *s;

  s = xstrprintf ("hello %d", num);
  old_chain = make_cleanup (xfree, s);

  res = function_that_may_throw (s);
  do_cleanups (old_chain);
  return res;
}
```

Cleanups, gone -> std::string


```
extern int function_that_may_throw (const char *);
int foo (int num)
{


  std::string s = string_printf ("hello %d", num);


  return function_that_may_throw (s.c_str ());


}
```

TRY_CATCH -> TRY/CATCH, ideally

Ideal/Goal:

```
try
  {
    something_that_reads_memory ();
  }
// ... but it's invalid here
catch (const memory_access_error &ex)
  {


  }
```

TRY_CATCH -> TRY/CATCH, after

Currently (since 7.10):

```
TRY
  { // save cleanups chain
    something_that_reads_memory ();
  } // restore cleanups chain
// no code can go here
CATCH (ex, RETURN_MASK_ERROR)
  {
     if (ex.error != MEMORY_ERROR)
       throw_exception (ex); // not what we wanted, rethrow
  }
END_CATCH // run cleanups && rethrow
```

C++ conversion, what's been done already

- many types class-ified (ui_file/ui_out, XML, DWARF, regcache, etc., etc.)
- VEC -> std::vector processing
- dynamic C strings -> std::string / std::unique_ptr processing
- gettimeofday -> std::chrono
- scoped_restore (save/restore globals)
- enum_flags [type safe bit flags]
- gdb::optional (simplified C++17 std::optional)
- gdb::function_view (for callbacks + lambdas)
- gdb_pyref -> gdb/python/ is free from cleanups!
- lots more...

C++ conversion, what's next (II)

- `make_cleanup` elimination
  - grep for "make_cleanup" shows a bit over 800 hits
    - was ~1900 last Sep

C++ conversion, what's next (III)

- Fix the gnulib macro problem.
  - Gnulib uses #defines to provide replacements:
    - #define printf rpl_printf
    - #define open rpl_open
    - etc.
  - Mostly fine for C, not so great in C++:

```
class target_ops {
  static void open (const char *args);
              ^^^^ -> rpl_open...
};
```

gnulib macros, fix attempt #1

- Make GDB use gnulib C++ namespace support
  - #define GNULIB_NAMESPACE `<some namespace name>`
  - #define GNULIB_NAMESPACE `gnulib`
- Works, but it's quite ugly...
  - Can't do `using gnulib::printf;` at top level
  - Adjust `foo` calls => `gnulib::foo` calls
    - ``printf `=>` gnulib::printf```
    - ``fwrite `=>` gnulib::fwrite```
    - (...)
  - ALL OF THEM!!!!11

gnulib macros, fix attempt #2

- Better would be to put *all* of gdb in a namespace

```
namespace gdb {
  using gnulib::printf;
}
```

- That works nicely, actually!

gdb:: everywhere

- However, gdb becomes harder to debug... :-(
  - Must prefix breakpoint locations with "gdb::"

```
(gdb) b gdb::internal_error
(gdb) b gdb::foo_bar
```

Ignore namespace breakpoints

- How about making GDB ignore missing specifiers (class/namespace):

```
void function () {}
namespace A { void function () {} }
namespace B::C { void function () {} }
```

```
(gdb) b function<TAB>
A::function()    B::C::function()    function()
```

```
(gdb) b function<enter>
Breakpoint 1: B::function. (3 locations)
```

namespace breakpoints, not so strange

Setting breakpoints by file ignores missing leading directories:

```
(gdb) b file.c:14
(gdb) info breakpoints
Num  Type         Disp Enb Address            What
1    breakpoint   keep y   <MULTIPLE>
1.1                      y     0x400635 in func() at .../one/file.c:14
1.2                      y     0x400691 in func() at .../two/file.c:14

(gdb) b one/file.c:14
(gdb) info breakpoints
2    breakpoint   keep y   0x400635 in func() at .../one/file.c:14
```

namespace breakpoints, not so strange, II

- Ada has always worked like that!

```
(gdb) b foo_proc
(gdb) info breakpoints
Num   Type         Disp Enb Address            What
1     breakpoint   keep y   <MULTIPLE>
1.1                     y     0x40137f in pck.foo_proc at pck.adb:19
1.2                     y     0x4013d5 in foo.foo_proc at foo.adb:26
(gdb) b pck.foo_proc
(gdb) info breakpoints
2     breakpoint   keep y   0x40137f in pck.foo_proc at pck.adb:19
```

(anonymous namespace)

- Helps with anonymous namespaces, even:

```
(gdb) b B::function<TAB>
(anonymous namespace)::B::function()
(anonymous namespace)::B::function() const
B::function()
B::function() const
Bn::(anonymous namespace)::B::function()
Bn::B::function()

(gdb) b B::function<enter>
Breakpoint 1: B::function. (6 locations)
```

What if I do want a fully-qualified name?

- `b -qualified` to the rescue:

```
(gdb) b -qualified B::function<TAB>
B::function()
B::function() const
B::function_const() const
```

C++ breakpoints, tab completion and overloads

Currently:

```
(gdb) b funct<TAB>       # gets you ...
(gdb) b function(        # ... this, and then you TAB again:
(gdb) b function(<TAB>   # ... this shows a bunch of useless completions
(gdb) b function(int <TAB>   # likewise
(gdb) b 'function(int <TAB>  # work around it by quoting
        ^
```

C++ breakpoints, no quoting necessary

- Get rid of the need for quoting linespecs.

```
(gdb) b function(in<TAB>
(gdb) b function(int)     # just works
```

C++ breakpoints, keyword completion

- Smart keyword completion

```
(gdb) b function(int) <TAB>
if      task    thread
```

C++ breakpoints, functions only

- No longer offers data symbols as possible completions

```
(gdb) b global_<TAB>
(gdb) b global_func()     # completes function only
```

While before:

```
(gdb) b global_<TAB>
global_func() global_var  # offered variable
(gdb) b global_var        # which is silly...
Function "global_var" not defined.
```

C++ breakpoints, expression awareness

- But context aware, expression/variable completion when needed:

```
(gdb) b function(int) if global_var1 + global_var<TAB>
global_var1 global_var2
```

C++ breakpoints, option awareness

- Explicit location option awareness + C++ operator awareness

```
(gdb) b -qualified test_op_SL::operato<TAB>
(gdb) b -qualified test_op_SL::operator<<(E, E) <TAB><TAB>
-function   -label      -line       -qualified  -source
if          task        thread
```

C++ breakpoints/ABI tags don't work


```
(top-gdb) b string_printf
Function "string_printf" not defined.
Make breakpoint pending on future shared library load? (y or [n])
```

```
/* Returns a std::string built from a printf-style format string.  */
std::string string_printf (const char* fmt, ...);
```

C++ breakpoints/ABI tags, fixed


```
(top-gdb) b string_printf
Breakpoint 6 at 0x6071c0: file gdb/common/common-utils.c, line 157.
(top-gdb) info breakpoints
Num Type           Enb Address  What
1   breakpoint y   0x6071c0 in string_printf[abi:cxx11](char const*, ...)
                                 at gdb/common/common-utils.c:157
(top-gdb)
```

C++ breakpoints/ABI tags, tab completion


```
(top-gdb) b string_printf<TAB>
(top-gdb) b string_printf(char const*, ...)

(top-gdb) b string_printf[ab<TAB>
(top-gdb) b string_printf[abi:cxx11](char const*, ...)
```

C++ breakpoints & TAB, under the hood

- Per language symbol name comparison functions (strcmp_iw)
- Symbol name match types : {FULL, WILD, EXPRESSION}
- Per language symbol name hashing functions
  - minsyms, psyms, symbols normalized
- Always hash symbol basename only
- Linespec completer reuses regular linespec parser
- strcmp_iw no longer ignores ALL whitespace
- Comprehensive testsuite & unit tests
- GDB <-> readline interaction completely reworked

C++ breakpoints & TAB, optimizations

- Ada ada_decode
- cp_find_first_component
- dwarf index & file seen cache
- psymtab & dwarf index use binary search (both completion and normal lookup)
- avoid recomputing symbol hash over and over

C++ breakpoints & TAB, status

- Posted upstream a few weeks ago.
- Started as 40 patches:

```
$ git diff --stat:
98 files changed, 10397 insertions(+), 2379 deletions(-)
```

- ~1/2 merged to master | 17 patches to go:

```
$ git diff --stat:
69 files changed, 7294 insertions(+), 1139 deletions(-)
```

The end, time for questions

- Try it at:
  - sourceware.org users/palves/cxx-breakpoint-improvements
- GDB's C++ conversion
  - Status
  - What's next
- C++ debugging improvements
  - Breakpoints and namespaces
  - Breakpoints and ABI tags
  - TAB completion improvements

More?

Other related usability improvements

- rvalue references support

```
void rval (std::string &&str) {}
```

```
(gdb) p str
$1 = <unknown type in /tmp/rval, CU 0x0, DIE 0xdf81>
     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
(gdb) p str
$1 = (std::__cxx11::string &&) @0x7fffffffd910: ""
```

Other related usability improvements, II

- `"list"` improvements around multiple locations

```
(gdb) list bar
file: "overload.cc", line number: 97, symbol: "bar(A)"
96
97      int bar (A) { return 11; }
file: "overload.cc", line number: 98, symbol: "bar(B)"
97      int bar (A) { return 11; }
98      int bar (B) { return 22; }
```

Other related usability improvements, III

- I'd like "list" to show current line:

```
(gdb) list
   1        #include <stdio.h>
   2
   3        int main (int argc, char **argv)
   4        {
=> 5          return 0;
   6        }
```

Other useful usability improvements, III

- GDB used to assume no-debug-info functions have type `int()`:

```
(gdb) p getenv ("PATH")
$1 = -6089                  # ????????? massive confusion
```

- No longer the case:

```
(gdb) p getenv ("PATH")
'getenv' has unknown return type;
cast the call to its declared return type
(gdb) p (char *) getenv ("PATH")
$1 = 0x7fffffffe83e "/usr/bin:/"...
```

The end, really!