Programmatic access to the call stack in C++

eli.thegreenplace.net/2015/programmatic-access-to-the-call-stack-in-c/

Sometimes when working on a large project, I find it useful to figure out all the places from which some function or method is called. Moreover, more often than not I don't just want the immediate caller, but the whole call stack. This is most useful in two scenarios - when debugging and when trying to figure out how some code works.

One possible solution is to use a debugger - run the program within a debugger, place a breakpoint in the interesting place, examine call stack when stopped. While this works and can sometimes be very useful, I personally prefer a more programmatic approach. I want to change the code in a way that will print out the call stack in every place I find interesting. Then I can use grepping and more sophisticated tools to analyze the call logs and thus gain a better understanding of the workings of some piece of code.

In this post, I want to present a relatively simple method to do this. It's aimed mainly at Linux, but should work with little modification on other Unixes (including OS X).

Obtaining the backtrace - libunwind

I'm aware of three reasonably well-known methods of accessing the call stack programmatically:

1. The gcc builtin macro `__builtin_return_address`: very crude, low-level approach. This obtains the return address of the function on each frame on the stack. Note: just the address, not the function name. So extra processing is required to obtain the function name.
2. glibc's `backtrace` and `backtrace_symbols`: can obtain the actual symbol names for the functions on the call stack.
3. `libunwind`

Between the three, I strongly prefer `libunwind`, as it's the most modern, widespread and portable solution. It's also more flexible than `backtrace`, being able to provide extra information such as values of CPU registers at each stack frame.

Moreover, in the zoo of system programming, `libunwind` is the closest to the "official word" you can get these days. For example, gcc can use `libunwind` for implementing zero-cost C++ exceptions (which requires stack unwinding when an exception is actually thrown) . LLVM also has a re-implementation of the `libunwind` interface in libc++, which is used for unwinding in LLVM toolchains based on this library.

Code sample

Here's a complete code sample for using `libunwind` to obtain the backtrace from an arbitrary point in the execution of a program. Refer to the libunwind documentation for more details about the API functions invoked here:

```
#define UNW_LOCAL_ONLY
#include <libunwind.h>
#include <stdio.h>

// Call this function to get a backtrace.
void backtrace() {
  unw_cursor_t cursor;
  unw_context_t context;

  // Initialize cursor to current frame for local unwinding.
  unw_getcontext(&context);
  unw_init_local(&cursor, &context);

  // Unwind frames one by one, going up the frame stack.
  while (unw_step(&cursor) > 0) {
    unw_word_t offset, pc;
    unw_get_reg(&cursor, UNW_REG_IP, &pc);
    if (pc == 0) {
      break;
    }
    printf("0x%lx:", pc);

    char sym[256];
    if (unw_get_proc_name(&cursor, sym, sizeof(sym), &offset) == 0) {
      printf(" (%s+0x%lx)\n", sym, offset);
    } else {
      printf(" -- error: unable to obtain symbol name for this
frame\n");
    }
  }
}

void foo() {
  backtrace(); // <-------- backtrace here!
}

void bar() {
  foo();
}

int main(int argc, char **argv) {
  bar();

  return 0;
}
```

`libunwind` is easy to install from source or as a package. I just built it from source with the usual `configure`, `make` and `make install` sequence and placed it into `/usr/local/lib`.

Once you have `libunwind` installed in a place the compiler can find , compile the code snippet with:

```
gcc -o libunwind_backtrace -Wall -g libunwind_backtrace.c -
lunwind
```

Finally, run:

```
$ LD_LIBRARY_PATH=/usr/local/lib
./libunwind_backtrace
0x400958: (foo+0xe)
0x400968: (bar+0xe)
0x400983: (main+0x19)
0x7f6046b99ec5: (__libc_start_main+0xf5)
0x400779: (_start+0x29)
```

So we get the complete call stack at the point where `backtrace` is called. We can obtain the function symbol names and the address of the instruction where the call was made (more precisely, the return address which is the next instruction).

Sometimes, however, we want not only the caller's name, but also the call location (source file name + line number). This is useful when one function calls another from multiple locations and we want to pinpoint which one is actually part of a given call stack. `libunwind` gives us the call address, but nothing beyond. Fortunately, it's all in the DWARF information of the binary, and given the address we can extract the exact call location in a number of ways. The simplest is probably to call `addr2line`:

```
$ addr2line 0x400968 -e
libunwind_backtrace
libunwind_backtrace.c:37
```

We pass the PC address to the left of the `bar` frame to `addr2line` and get the file name and line number.

Alternatively, we can use the dwarf_decode_address example from pyelftools to obtain the same information:

```
$ python <path>/dwarf_decode_address.py 0x400968
libunwind_backtrace
Processing file: libunwind_backtrace
Function: bar
File: libunwind_backtrace.c
Line: 37
```

If printing out the exact locations is important for you during the backtrace call, you can also go fully programmatic by using `libdwarf` to open the executable and read this information from it, in the `backtrace` call. There's a section and a code sample about a very similar task in my blog post on debuggers.

C++ and mangled function names

The code sample above works well, but these days one is most likely writing C++ code and not C, so there's a slight problem. In C++, names of functions and methods are mangled. This is essential to make C++ features like function overloading, namespaces and templates work. Let's say the actual call sequence is:

```
namespace ns {

template <typename T, typename U>
void foo(T t, U u) {
  backtrace(); // <-------- backtrace
here!
}

}  // namespace ns

template <typename T>
struct Klass {
  T t;
  void bar() {
    ns::foo(t, true);
  }
};

int main(int argc, char** argv) {
  Klass<double> k;
  k.bar();

  return 0;
}
```

The backtrace printed will then be:

```
0x400b3d: (_ZN2ns3fooIdbEEvT_T0_+0x17)
0x400b24: (_ZN5KlassIdE3barEv+0x26)
0x400af6: (main+0x1b)
0x7fc02c0c4ec5:
(__libc_start_main+0xf5)
0x4008b9: (_start+0x29)
```

Oops, that's not nice. While some seasoned C++ veterans can usually make sense of simple mangled names (kinda like system programmers who can read text from hex ASCII), when the code is heavily templated this can get ugly very quickly.

One solution is to use a command-line tool - `c++filt`:

```
$ c++filt _ZN2ns3fooIdbEEvT_T0_
void ns::foo<double, bool>(double,
bool)
```

However, it would be nicer if our backtrace dumper would print the demangled name directly. Luckily, this is pretty easy to do, using the `cxxabi.h` API that's part of libstdc++ (more precisely, libsupc++). libc++ also provides it in the low-level libc++abi. All we need to do is call `abi::__cxa_demangle`. Here's a complete example:

```
#define UNW_LOCAL_ONLY
#include <cxxabi.h>
#include <libunwind.h>
```

```
#include <cstdio>
#include <cstdlib>

void backtrace() {
  unw_cursor_t cursor;
  unw_context_t context;

  // Initialize cursor to current frame for local unwinding.
  unw_getcontext(&context);
  unw_init_local(&cursor, &context);

  // Unwind frames one by one, going up the frame stack.
  while (unw_step(&cursor) > 0) {
    unw_word_t offset, pc;
    unw_get_reg(&cursor, UNW_REG_IP, &pc);
    if (pc == 0) {
      break;
    }
    std::printf("0x%lx:", pc);

    char sym[256];
    if (unw_get_proc_name(&cursor, sym, sizeof(sym), &offset) == 0) {
      char* nameptr = sym;
      int status;
      char* demangled = abi::__cxa_demangle(sym, nullptr, nullptr, &status);
      if (status == 0) {
        nameptr = demangled;
      }
      std::printf(" (%s+0x%lx)\n", nameptr, offset);
      std::free(demangled);
    } else {
      std::printf(" -- error: unable to obtain symbol name for this
frame\n");
    }
  }
}

namespace ns {

template <typename T, typename U>
void foo(T t, U u) {
  backtrace(); // <-------- backtrace here!
}

}  // namespace ns

template <typename T>
struct Klass {
  T t;
  void bar() {
    ns::foo(t, true);
  }
};

int main(int argc, char** argv) {
  Klass<double> k;
  k.bar();

  return 0;
}
```

This time, the backtrace is printed with all names nicely demangled:

```
$ LD_LIBRARY_PATH=/usr/local/lib
./libunwind_backtrace_demangle
0x400b59: (void ns::foo<double, bool>(double, bool)+0x17)
0x400b40: (Klass<double>::bar()+0x26)
0x400b12: (main+0x1b)
0x7f6337475ec5: (__libc_start_main+0xf5)
0x4008b9: (_start+0x29)
```
