DxxxxR0 draft 1: Stackable, thread local, signal guards

Document #:	DxxxxR0
Date:	2019-05-30
Project:	Programming Language C++ SG12 Undefined Behaviour study group SG18 LEWG Incubator study group WG14-WG21 liaison group

Reply-to:	Niall Douglas <s_sourceforge@nedprod.com>

A proposal for standard library support for executing a routine, guarding against compiler-unanticipated failure and interruption (i.e. 'signals'), with the possibility of recovering from the failure and continuing execution. This is a high level design, abstracting away for the majority of use cases any need to care about POSIX signal handlers or Win32 structured exception handling.

A reference implementation of the proposed library facility with API documentation can be found at `https://github.com/ned14/quickcpplib/blob/master/include/signal_guard.hpp`. It has been in production use for over a year at the time of writing, and has proven to be quite popular with some in the C++ ecosystem i.e. it has been lifted and borrowed by quite a few people, because it solves well an ever growing problem (see Introduction). It works well on Android, FreeBSD, MacOS, Linux and Microsoft Windows on ARM, AArch64, x64 and x86.

Contents

1 Introduction **2**

2 Example of use **3**

3 Impact on the Standard **5**

4 Proposed Design **7**

5 Design decisions, guidelines and rationale **10**
- 5.1 Separate handler install step . . . 10
- 5.2 `early_global_signals` . . . 11
- 5.3 Open design questions . . . 12

6 Acknowledgements **12**

7 References **12**

1 Introduction

Why propose this facility?

1. Recent versions of the major desktop and server operating systems proactively seek out and silently unmap from memory pages previously written to by the C++ program[1]. When the C++ program next refers to that memory page, a page fault occurs, faulting the memory page back into existence. If there are insufficient system resources to restore that memory page, a system exception is sent to the C++ program, which if unhandled, causes the C++ program to immediately terminate.

2. Even though memory mapped i/o is not currently supported by standard C++, if WG21 chooses to adopt [P1631] *Object detachment and attachment*, and possibly [P1031] *Low level file i/o*, then reading and writing mapped memory may also generate system exceptions e.g. 'disk full'.

3. Additionally, if WG21 chooses to adopt the 'default terminate' OOM model as proposed by [P0709] *Zero-overhead deterministic exceptions*, this would cause code which works with STL containers configured with the default allocator to become not stack unwindable when OOM by the container's default allocator occurs.

For the purposes of brevity, let us call all these three cases above a problem of *unanticipated interruptions*, because the C++ compiler does not, by default, generate code which is tolerant to these kinds of interruption (nor, incidentally, are many functions in the C and C++ runtime support libraries e.g. `malloc()`).

Proprietary mechanisms exist for C++ programs to be notified of unanticipated interruptions, however these are tricky to implement both correctly and with good performance, even for those very experienced in writing this kind of code. If the developer takes very special care when writing the C++ which works with such 'ghostly' memory[2] (e.g. never write code which could call a non-trivial destructor, only call async-signal-safe POSIX functions, etc), it is possible to write 'hardened' code which can safely recover from unanticipated interruptions, including system exceptions and Out Of Memory.

It is currently proposed that 'unanticipated interruption safe code' be left implementation defined outside conforming to the same requirements as for `longjmp()` as is currently in the standard, though see 'Impact on the Standard' for more discussion on this.

This paper proposes a standard library facility for calling a guarded routine in which unanticipated interruption may occur:

- One may specify which unanticipated interruptions ought to be guarded:
  - Process abort.
  - Undefined memory access.

[1] Linux may merge identical pages, and eliminate pages containing all bits zero e.g. those precleared by `std::vector` or `operator new[]`. Windows 10 memory compression removes and compresses pages not recently accessed, reexpanding on first access, and also eliminates pages containing all bits zero.

[2] Memory prone to disapperance and reappearance.

  - Illegal instruction.
  - Process interruption.
  - Broken pipe.
  - Segmentation fault.
  - Floating point error.
  - C++ out of memory (instead of throwing `std::bad_alloc`).
  - C++ termination (somebody has called `std::terminate()`).

  This is a subset of what could be available, and WG21 may wish to standardise all of what POSIX provides. However, the semantics of the less common options vary somewhat more in non-POSIX implementations. The list above was chosen precisely because of the common semantics between the major hosted implementations.

- One can configure a callable to be called if the guarded routine was aborted, and execution was resumed outside the guarded routine. This callable would typically clean up any interrupted state.
- One can furthermore configure a callable to be called at the exact moment when the interruption occurs, *in-situ*. This callable may be able to recover the problem, and resume execution by returning `true`. Alternatively, by returning `false`, it will cause execution to jump back to just before the guarded routine was entered, and to call the previously described cleanup handler.
- Finally, it is possible to nest guarded sections within other guarded sections.

It is recognised that this proposal involves a level of low level implementation detail which WG21 has not been keen on mentioning in the standard until now. However, I still feel it is worth asking, and see what WG21 thinks.

2 Example of use

The following is taken from the [P1031] *Low level file i/o* reference implementation of proposed `map_handle::write()`, which is working code in production use right now. I have decluttered and reformatted it a little, and added explanatory comments, otherwise it is identical.

```
/* This function implements synchronous gather write for map_handle,
which is an i/o handle working upon memory mapped storage.
Implementation is easy, simply memcpy() each buffer in the gather
buffer list into the mapped memory. However, if the disk runs out
of free space, a SIGBUS or equivalent shall be raised. We want to
trap that, and return it as an errc::no_space_on_device instead.
*/
map_handle::io_result<map_handle::const_buffers_type>
map_handle::write(io_request<const_buffers_type> reqs, deadline /*d*/) noexcept
{
  // const_buffers_type is a span<const_buffer_type>
```

```
// const_buffer_type is a span<const byte>
// io_request<T> supplies a const_buffers_type list of buffers to
// gather write, and an offset within the file at which to write them

// Where in memory we shall be writing to (addr is base of the map)
byte *addr = _addr + reqs.offset;

// Clamp the gather write to the end of the map (length is length of the map)
size_type togo = reqs.offset < _length ? static_cast<size_type>(_length - reqs.offset) : 0;

/* This signal_guard() function overload takes a bitfield of what
to guard against, a callable to be guarded, and a callable to be
called if the guarded callable is aborted. It returns whatever
the guarded callable, or the cleanup callable, returns, which in
this case is false for success, and true for failure.
*/
if(signal_guard(signalc::undefined_memory_access,
                [&]  // The guarded section of code
                {
                  for(size_t i = 0; i < reqs.buffers.size(); i++)
                  {
                    const_buffer_type &req = reqs.buffers[i];

                    // If this gather buffer's size exceeds that of
                    // the bytes before end of map, truncate the
                    // buffers returned to those actually written.
                    if(req.size() > togo)
                    {
                      memcpy(addr, req.data(), togo);
                      // We wrote togo bytes, not req.size() bytes
                      req = {addr, togo};
                      // Truncate gather list to buffers written
                      reqs.buffers = {reqs.buffers.data(), i + 1};
                      // Return success
                      return false;
                    }
                    memcpy(addr, req.data(), req.size());
                    // Return where the buffer was written to
                    req = {addr, req.size()};
                    addr += req.size();
                    togo -= req.size();
                  }
                  // Return success
                  return false;
                },
                [&](const raised_signal_info &info)  // the cleanup handler
                {
                  // Retrieve the memory location associated with the failure
                  auto *causingaddr = (byte *) info.address();

                  // This could be a undefined memory access not involving
                  // this map at all, if so, re-raise it.
                  if(causingaddr < _addr || causingaddr >= (_addr + _reservation))
                  {
                    // Not caused by this map, so re-raise it on this thread
                    info.reraise();
```

```
68
69                      // POSIX permit signal handlers to return, also the
70                      // handler may be set to SIG_IGN, so if undefined
71                      // memory access was not handled, abort.
72                      abort();
73                    }
74
75                    // The guarded routine failed due to undefined memory
76                    // access, so return true to cause no_space_on_device
77                    // to be returned by the write() function.
78                    return true;
79                }))
80    {
81      // If true was returned, we failed due to no space on device
82      return errc::no_space_on_device;
83    }
84    // Otherwise return buffers successfully written
85    return reqs.buffers;
86  }
```

3 Impact on the Standard

As mentioned earlier, it is currently proposed that the subset of C++ code which is 'unanticipated interruption safe' be left implementation defined, apart from the current C++ standard's requirements for code which uses `longjmp()`. `longjmp()` is permitted over automatic duration C++ objects if, and only if [csetjmp.syn]:

> A `setjmp`/`longjmp` call pair has undefined behavior if replacing the `setjmp` and `longjmp` by catch and throw would invoke any non-trivial destructors for any automatic objects.

Thus, one would borrow identical normative wording for guarded code sections, as recovery is implemented using `longjmp()`.

As unanticipated interruptions may occur at any time, one must only call async signal safe POSIX functions within guarded code, if one is on POSIX. POSIX.2017 requires the following functions to be async signal safe[3]:

- `_Exit()`
- `_exit()`
- `abort()`
- `accept()`
- `access()`
- `aio_error()`
- `aio_return()`
- `aio_suspend()`
- `alarm()`
- `bind()`
- `cfgetispeed()`
- `cfgetospeed()`
- `cfsetispeed()`
- `cfsetospeed()`
- `chdir()`
- `chmod()`
- `chown()`
- `clock_gettime()`
- `close()`
- `connect()`
- `creat()`
- `dup()`
- `dup2()`
- `execl()`
- `execle()`
- `execv()`
- `execve()`
- `faccessat()`
- `fchdir()`
- `fchmod()`
- `fchmodat()`
- `fchown()`
- `fchownat()`
- `fcntl()`
- `fdatasync()`
- `fexecve()`
- `ffs()`
- `fork()`
- `fstat()`
- `fstatat()`
- `fsync()`
- `ftruncate()`
- `futimens()`
- `getegid()`

[3] https://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html

- geteuid()
- getgid()
- getgroups()
- getpeername()
- getpgrp()
- getpid()
- getppid()
- getsockname()
- getsockopt()
- getuid()
- htonl()
- htons()
- kill()
- link()
- linkat()
- listen()
- longjmp()
- lseek()
- lstat()
- memccpy()
- memchr()
- memcmp()
- memcpy()
- memmove()
- memset()
- mkdir()
- mkdirat()
- mkfifo()
- mkfifoat()
- mknod()
- mknodat()
- ntohl()
- ntohs()
- open()
- openat()
- pause()
- pipe()
- poll()
- posix_trace_event()
- pselect()
- pthread_kill()
- pthread_self()
- pthread_sigmask()
- raise()
- read()
- readlink()
- readlinkat()
- recv()
- recvfrom()
- recvmsg()
- rename()
- renameat()
- rmdir()
- select()
- sem_post()
- send()
- sendmsg()
- sendto()
- setgid()
- setpgid()
- setsid()
- setsockopt()
- setuid()
- shutdown()
- sigaction()
- sigaddset()
- sigdelset()
- sigemptyset()
- sigfillset()
- sigismember()
- siglongjmp()
- signal()
- sigpause()
- sigpending()
- sigprocmask()
- sigqueue()
- sigset()
- sigsuspend()
- sleep()
- sockatmark()
- socket()
- socketpair()
- stat()
- stpcpy()
- stpncpy()
- strcat()
- strchr()
- strcmp()
- strcpy()
- strcspn()
- strlen()
- strncat()
- strncmp()
- strncpy()
- strnlen()
- strpbrk()
- strrchr()
- strspn()
- strstr()
- strtok_r()
- symlink()
- symlinkat()
- tcdrain()
- tcflow()
- tcflush()
- tcgetattr()
- tcgetpgrp()
- tcsendbreak()
- tcsetattr()
- tcsetpgrp()
- time()
- timer_getoverrun()
- timer_gettime()
- timer_settime()
- times()
- umask()
- uname()
- unlink()
- unlinkat()
- utime()
- utimensat()
- utimes()
- wait()
- waitpid()
- wcpcpy()
- wcpncpy()
- wcscat()
- wcschr()
- wcscmp()
- wcscpy()
- wcscspn()
- wcslen()
- wcsncat()
- wcsncmp()
- wcsncpy()
- wcsnlen()
- wcspbrk()
- wcsrchr()
- wcsspn()
- wcsstr()
- wcstok()
- wmemchr()
- wmemcmp()
- wmemcpy()
- wmemmove()
- wmemset()
- write()

If one is not on POSIX, then different lists of permitted versus non-permitted system calls exist, depending on the system in question. For example, on Microsoft Windows, almost none of the Win32 APIs which are implemented by `kernel32.dll` are permitted e.g. `WriteFile()`, whereas all the APIs implemented by `ntdll.dll` are safe e.g. `NtWriteFile()`.

All this said, the safest approach is to not call system functions at all in portable guarded code, which ought to be kept as short and as simple as possible in most use cases. Perhaps this is what

the C++ standard might wish to recommend in a non-normative comment, if WG21 smiles on this proposal.

4 Proposed Design

```
namespace signal_guard
{
  //! The signals which can be raised
  bitfield(signalc)
  {
    none = 0,

    abort_process = (1 << 0),            //!< The process is aborting
    undefined_memory_access = (1 << 1),  //!< Attempt to access a memory page which doesn't exist
    illegal_instruction = (1 << 2),      //!< Execution of illegal instruction
    interrupt = (1 << 3),                //!< The process is interrupted
    broken_pipe = (1 << 4),              //!< Reader on a pipe vanished
    segmentation_fault = (1 << 5),       //!< Attempt to access a memory page whose permissions
        disallow

    floating_point_error = (1 << 8),     //!< Floating point error
    // leave bits 9-15 for individual FP errors

    // C++ handlers
    out_of_memory = (1 << 16),  //!< A call to operator new failed, and a throw is about to occur
    termination = (1 << 17),    //!< A call to std::terminate() was made

    // Signal install flags
    early_global_signals = (1 << 28)  //!< CAUTION: Enable signals globally at install time, not at
        guard time. This is dangerous, see documentation.
  }
  bitfield(signalc)

  /*! On platforms where it is necessary (POSIX), installs, and potentially enables,
  the global signal handlers for the signals specified by `guarded`. Each signal installed
  is threadsafe reference counted, so this is safe to call from multiple threads or multiple times.

  If this call does anything at all, it is not fast, plus it serialises on a global mutex.

  ## POSIX only

  Changing the signal mask for a process involves a kernel transition, which costs perhaps
  500 CPU cycles. The default implementation enables the guarded signals for the local thread
  just before executing the guarded section of code, and restores the previous thread local
  signal mask on exiting the guarded section of code. This, inevitably, adds at least 1,000
  CPU cycles to each guarded code invocation, but it comes with the big advantage of predictability.

  One can set the `signalc::early_global_signals` flag during signal install which profoundly
  changes these semantics. When we install the handlers, we use `SA_NODEFER` to prevent the
  blocking of the raised signal during the execution of the signal handler. We enable the guarded
  signals immediately, and globally.

  These differences mean that `signal_guard` no longer needs to touch the signal mask during
  execution, and thus avoid all kernel transitions. Performance is enormously improved. The
```

```
  cost however is that signal handling becomes much less predictable. If the installed signal
  handlers cause the raising of a signal, an infinite loop occurs. Signal handlers are executed
  in all threads in the process, not just in the guarded code sections.

  Because of these profound differences, you cannot mix different types of signal install in
  the same process. An attempt to instantiate a second install with differing
  'signalc::early_global_signals' will throw an exception.
  */
  class signal_guard_install
  {
  public:
    explicit signal_guard_install(signalc guarded);
    ~signal_guard_install();
    signal_guard_install(const signal_guard_install &) = delete;
    signal_guard_install(signal_guard_install &&) = delete;
    signal_guard_install &operator=(const signal_guard_install &) = delete;
    signal_guard_install &operator=(signal_guard_install &&) = delete;
  };

  //! Thrown by the default signal handler to abort the current operation
  class signal_raised : public std::exception
  {
  public:
    //! Constructor
    signal_raised(signalc code);
    virtual const char *what() const noexcept override;
  };

  /*! \class raised_signal_info
  \brief Portable information about a raised signal.
  */
  class raised_signal_info
  {
  protected:
    raised_signal_info() = default;

  public:
    raised_signal_info(const raised_signal_info &) = delete;
    raised_signal_info(raised_signal_info &&) = delete;
    raised_signal_info &operator=(const raised_signal_info &) = delete;
    raised_signal_info &operator=(raised_signal_info &&) = delete;

    //! The signal raised
    virtual signalc signal() const noexcept = 0;

    /*! The faulting address for 'signalc::undefined_memory_access', 'signalc::segmentation_fault',
    'signalc::illegal_instruction' and 'signalc::floating_point_error'.
    */
    virtual void *address() const noexcept = 0;

    /*! The system specific error code for this signal, the 'si_errno' code (POSIX) or
    'NTSTATUS' code (Windows)
    */
#ifdef _WIN32
    virtual long error_code() const noexcept = 0;
#else
```

```
    virtual int error_code() const noexcept = 0;
#endif

    //! The OS specific ‘siginfo_t *‘ (POSIX) or ‘PEXCEPTION_RECORD‘ (Windows)
    virtual const void *raw_info() const noexcept = 0;

    //! The OS specific ‘ucontext_t‘ (POSIX) or ‘PCONTEXT‘ (Windows)
    virtual const void *raw_context() const noexcept = 0;

    //! Re-raise this signal on the calling thread, returning false if there is no handler available.
    virtual bool reraise() const = 0;
  };

  /*! Call a callable ‘f‘ with signals ‘guarded‘ protected for this thread only, returning whatever
  ‘f‘ returns.

  Note that on POSIX, if a ‘signal_guard_install‘ is not already instanced, one is temporarily
  installed, which is not quick. You are therefore very strongly recommended, when on POSIX, to
  call this function with a ‘signal_guard_install‘ already installed.

  Firstly, how to restore execution to this context is saved, if thread locally configured the
  guarded signals are enabled for the calling thread, and ‘f‘ is executed, returning whatever
  ‘f‘ returns, and restoring the signal enabled state to what it was on entry to this guard
  before returning. This is mostly inlined code, so it will be relatively fast. No memory
  allocation is performed if a ‘signal_guard_install‘ is already instanced. Approximate overhead
  on an Intel CPU:

  - Linux (thread local): 1450 CPU cycles (mostly the two syscalls to enable and disable the
  guarded signals)
  - Linux (early global): 52 CPU cycles
  - Windows: 85 CPU cycles

  If during the execution of ‘f‘, any one of the signals ‘guarded‘ is raised:

  1. ‘c‘, which must have the prototype ‘bool(const raised_signal_info &)‘, is called with the
  signal which was raised. You can fix the cause of the signal and return ‘true‘ to continue
  execution, or else return ‘false‘ to halt execution. Note that the variety of code you can
  call in ‘c‘ is extremely limited, the same restrictions as for signal handlers apply.

  2. If ‘c‘ returned ‘false‘, the execution of ‘f‘ is halted **immediately** without stack unwind,
  the thread is returned to the state just before the calling of ‘f‘, and the callable ‘g‘ is
  called with the specific signal which occurred. ‘g‘ must have the prototype
  ‘R(const raised_signal_info &)‘ where ‘R‘ is the return type of ‘f‘. ‘g‘ is called with this
  signal guard removed, though a signal guard higher in the call chain may instead be active.

  Obviously all state which ‘f‘ may have been in the process of doing will be thrown away. You
  should therefore make sure that ‘f‘ never causes side effects, including the interruption in
  the middle of some operation, which cannot be fixed by the calling of ‘h‘. The default ‘h‘
  simply throws a ‘signal_raised‘ C++ exception.
  */
  template<class F, class H, class C, class R = decltype(std::declval<F>()())>
  requires(std::is_constructible<R, decltype(std::declval<H>()(std::declval<const raised_signal_info
      >()))>::value
    && std::is_constructible<bool, decltype(std::declval<C>()(std::declval<const raised_signal_info>()
        ))>::value)
  inline R signal_guard(signalc guarded, F &&f, H &&h, C &&c);
```

```
158
159   /*! Convenience overload with preconfigured handlers:
160
161   - For the ‘h‘ parameter, a callable which throws an exception of type ‘signal_raised‘ i.e.
162   upon resumption of execution outside the guarded section, throw that exception.
163   - For the ‘c‘ parameter, a callable which always returns ‘false‘ i.e. halt execution immediately
164   without stack unwind, and call the callable which throws the aforementioned exception.
165   */
166   template <class F, class R = decltype(std::declval<F>()())>
167   inline R signal_guard(signalc guarded, F &&f);
168
169   /*! Convenience overload with preconfigured handlers:
170
171   - For the ‘c‘ parameter, a callable which always returns ‘false‘ i.e. halt execution immediately
172   without stack unwind, and call the callable specified by ‘h‘.
173   */
174   template<class F, class H, class R = decltype(std::declval<F>()())>
175   requires(std::is_constructible<R, decltype(std::declval<H>()(std::declval<const raised_signal_info
          >()))>::value)
176   inline auto signal_guard(signalc guarded, F &&f, H &&h);
177 }
```

5 Design decisions, guidelines and rationale

5.1 Separate handler install step

For those who have ever had the misfortune of working with them from library code, POSIX signals have many problems:

1. Their handlers are installed globally for a process, which creates problems for third party library code.
2. There is only the ‘current’ signal handler for a signal, which means that ‘filtering’ signal handlers need to check whether the signal’s cause applies to the specific cases they were installed for, and then call the previously installed signal handler.
3. If you install a handler, and then some other code then installs another handler, there is no way to remove your handler because it is now managed by whomever replaced your handler.
4. Installation and removal of signal handlers is not thread safe. However, each thread has a signal mask, which determines which signals can be delivered to it.
5. Some signals get delivered to any random thread for which its bit is enabled in that thread’s signal mask.

For those who have ever used structured exception handling on Microsoft Windows, you will instantly agree that their stackable per-thread approach is the correct way to implement signals. Not what POSIX does.

Implicit in the design presented above for standardisation is effectively stackable per-thread signal handling i.e. what Microsoft Windows does, and indeed on Microsoft Windows, one implements

this facility using a trivially simple structured exception handling implementation, as the system already implements everything for us.

On generic POSIX, however – and for the `std::set_terminate()` and `std::set_new_handler()` support on all platforms – one must emulate stackable per-thread signal handling using the global handlers. On generic POSIX, without using platform-specific extensions, this can be done by replacing the global signal handlers with ones which:

1. Check if a `signal_guard` instance for the specific unanticipated failure is present for the calling thread.
2. If so, invokes the guard.
3. If not, calls the previously installed global handler.

This implies that a thread local stack is kept of currently applicable `signal_guard` instances on POSIX, and for the terminate and new global handlers on Windows as well.

Because of this non-trivial setup overhead on POSIX, and the problem of race conditions if you modify the signal handlers outside of program bootstrap, we separate out global handler installation into the `signal_guard_install` class. It would be expected that C++ programs would instance that class somewhere in their static init or their `main()`, however third party libraries can also instance that class in their static init, as it is the combined set of `signalc` from all the `signal_guard_install` class instances which is actually used.

In other words, it is safe in the proposed design to instance as many `signal_guard_install` objects as you want, and to destruct them in any order. **However** be aware that on POSIX the final `signal_guard_install` instance destruction must abort the process if third party code has replaced the handler we installed with another one, as it is not possible to safely deinstall our handler.

(Aside: One would hope that if this proposal is standardised, POSIX implementations would internally implement a less broken solution to signal handling, and have this C++ support use that internal implementation instead of the POSIX standard semantics)

5.2 `early_global_signals`

The default signal mask for a thread is to block delivery of all signals. This implies that during the execution of guarded code, we must temporarily unblock the delivery of the guarded signals for the duration of the guarded code, taking care to remember that signal guards can be legally nested inside one another.

Unfortunately, most POSIX implementations store a thread's signal mask in the kernel, so modifying it involves a syscall, which means two syscalls per guarded code exection. This is approximately 1,000 CPU cycles on Linux, which is a lot if you are reading a single byte, for example.

By setting `early_global_signals`, we can avoid this overhead by enabling the guarded signals globally, for the entire process, until the last `signal_guard_install` instance is destructed. This reduces the signal guard overhead to a mere 50 CPU cycles, which is probably as low as is possible.

However, with standard POSIX signals, doing this is not risk free. Enabling signal delivery for all threads means that the global signal handlers are called from all threads. Obviously, our global signal handler implementation passes on the signal if it cannot find a signal guard instance for the calling thread, however because we are installing a global, filtering signal handler which is active for all threads, we must specify `SA_NODEFER` for the global handler i.e. don't disable the signal during signal handling. This is necessary to avoid deadlock, however the corollory is that if the handler itself causes a signal, it'll loop into itself forever, without termination.

Again, if this proposal were standardised, I would like to hope that POSIX implementations would take the opportunity to substantially refactor how signals are implemented by their C runtime support. I would strongly suggest replicating how Windows implements this, where there are both globally installable AND stackable, per-thread, handlers, with the ability to deinstall a globally installed handler without being the last piece of code to install a handler. The POSIX signal API would then be a subset API for the true, internal, implementation. For more information, see https://docs.microsoft.com/en-gb/windows/desktop/Debug/vectored-exception-handling.

5.3 Open design questions

- In the current design, `raised_signal_info` is not copyable nor moveable. It is passed by const lvalue ref to the cleanup handler of the signal guard.

  `raised_signal_info::raw_info()` and `raised_signal_info::raw_context()` are const member functions returning pointers to const data. This is because the data they return are to OS allocated internal structures, which should not be modified from a strict C++ standpoint, as we don't own them.

  However, the const-ness of these returned pointers makes them useless for reuse with OS-specific functions, which DO own them. So should we make these functions return non-const pointers, or simply insist that the end user casts off the const-ness as an explicit declaration of 'I know what I am doing'?

6 Acknowledgements

Todo

7 References

[P0709] Herb Sutter,
Zero-overhead deterministic exceptions
https://wg21.link/P0709

[P1031] Douglas, Niall
Low level file i/o library
https://wg21.link/P1031

[P1095] Douglas, Niall
Zero overhead deterministic failure – A unified mechanism for C and C++
https://wg21.link/P1095

[P1631] Douglas, Niall
Object detachment and attachment
https://wg21.link/P1631