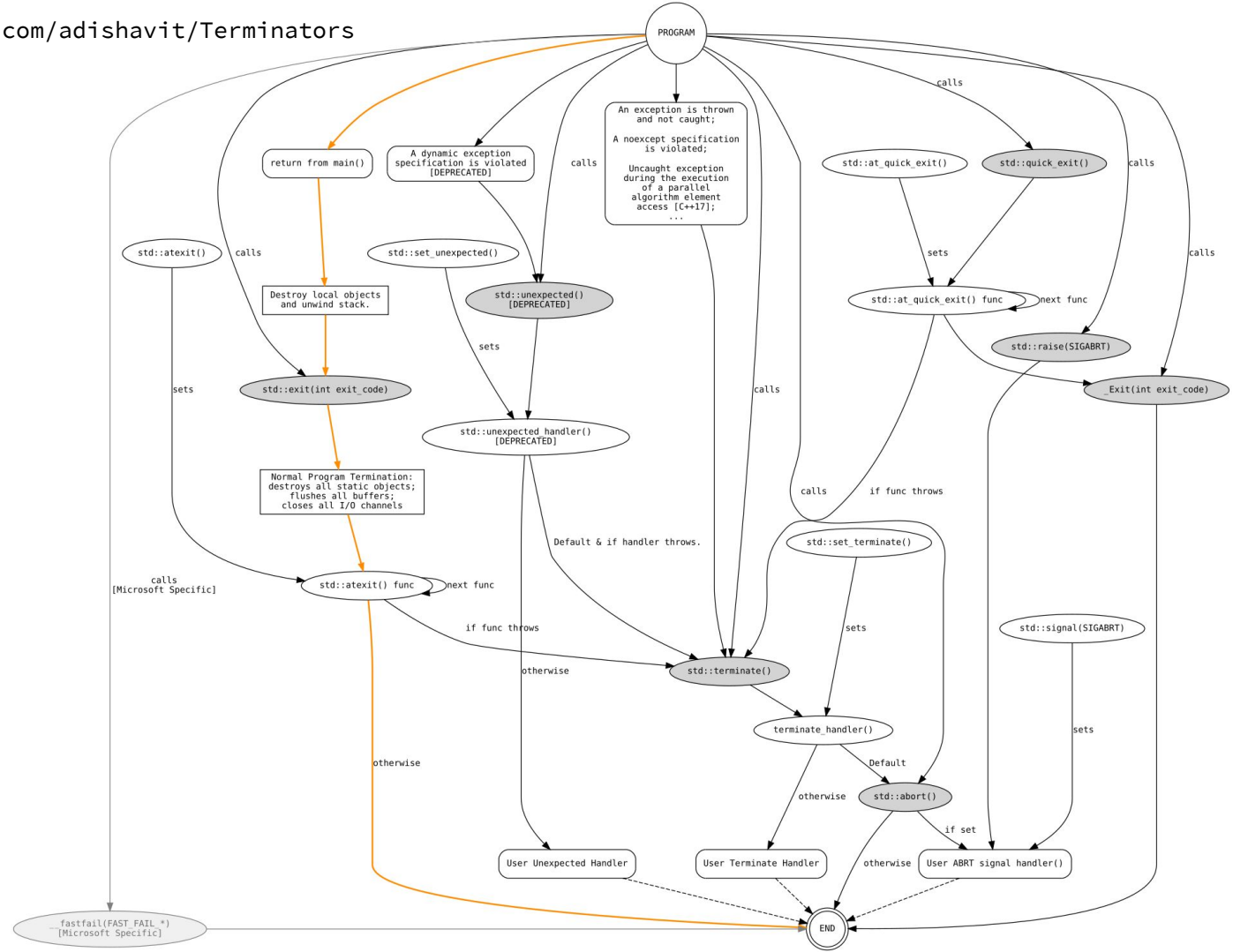
Catch yourself!

(or: what to do in “ooopsi” situations)

C++ User Group Aachen
Daniel Evers, 2018-11-08

What can possibly go wrong?

https://github.com/adishavit/Terminators



What can go wrong - C++ style

- uncaught exception
  - calls std::terminate()
- dynamic exception specification violation (pre C++17)
  - calls std::unexpected()
- “manual” calls to std::terminate() / std::abort()

What can go wrong - C style

- `abort()`
- `unhandled (OS) signal/event`
  - `stack overflow, seg-fault, div-by-zero, illegal instruction, …`
  - `highly OS-dependent...`

What can I do?

Out

Of

Options

Print

(Call)**S**tack

Information

OOOPSI !

https://github.com/dermojo/ooopsi

std::terminate

Defined in header <exception>

```
void terminate();                          (until C++11)
[[noreturn]] void terminate() noexcept;    (since C++11)
```

`std::terminate()` is called by the C++ runtime when exception handling fails for any of the following reasons:

1) an exception is thrown and not caught (it is implementation-defined whether any stack unwinding is done in this case)
2) an exception is thrown during exception handling (e.g. from a destructor of some local object, or from a function that had to be called during exception handling)
3) the constructor or the destructor of a static or thread-local object throws an exception
4) a function registered with `std::atexit` or `std::at_quick_exit` throws an exception
5) a noexcept specification is violated (it is implementation-defined whether any stack unwinding is done in this case)
6) a dynamic exception specification is violated and the default handler for `std::unexpected` is executed
7) a non-default handler for `std::unexpected` throws an exception that violates the previously violated dynamic exception specification, if the specification does not include `std::bad_exception`

(until C++17)

8) `std::nested_exception::rethrow_nested` is called for an object that isn't holding a captured exception
9) an exception is thrown from the initial function of `std::thread`
10) a joinable `std::thread` is destroyed or assigned to
11) a function invoked by a parallel algorithm exits via an uncaught exception and the execution policy specifies termination.

(since C++17)

`std::terminate()` may also be called directly from the program.

In any case, `std::terminate` calls the currently installed `std::terminate_handler`. The default `std::terminate_handler` calls `std::abort`.

If a destructor reset the terminate handler during stack unwinding and the unwinding later led to `terminate` being called, the handler that was installed at the end of the throw expression is the one that will be called. (note: it was ambiguous whether re-throwing applied the new handlers) (until C++11)

If a destructor reset the terminate handler during stack unwinding, it is unspecified which handler is called if the unwinding later led to `terminate` being called. (since C++11)

std::terminate_handler

Defined in header <exception>

```
typedef void (*terminate_handler)();
```

std::terminate_handler is the function pointer type (pointer to function that takes no arguments and returns void), which is installed and queried by the functions std::set_terminate and std::get_terminate and called by std::terminate.

The C++ implementation provides a default std::terminate_handler function, which calls std::abort(). If the null pointer value is installed (by means of std::set_terminate), the implementation may restore the default handler instead.

See also

terminate	function called when exception handling fails (function)
set_terminate	changes the function to be called by std::terminate (function)
get_terminate (C++11)	obtains the current terminate_handler (function)

std::unexpected

Defined in header <exception>

`void unexpected();`	(until C++11)
`[[noreturn]] void unexpected();`	(since C++11) (deprecated) (removed in C++17)

`std::unexpected()` is called by the C++ runtime when a dynamic exception specification is violated: an exception is thrown from a function whose exception specification forbids exceptions of this type.

`std::unexpected()` may also be called directly from the program.

In either case, `std::unexpected` calls the currently installed `std::unexpected_handler`. The default `std::unexpected_handler` calls `std::terminate`.

If a destructor reset the unexpected handler during stack unwinding and the unwinding later led to `unexpected` being called, the handler that was installed at the end of the throw expression is the one that will be called. (note: it was ambiguous whether re-throwing applied the new handlers)	(until C++11)
If a destructor reset the unexpected handler during stack unwinding, it is unspecified which handler is called if the unwinding later led to `unexpected` being called.	(since C++11)

std::abort

Defined in header <cstdlib>

```
          void abort();                    (until C++11)
[[noreturn]] void abort() noexcept;        (since C++11)
```

Causes abnormal program termination unless SIGABRT is being caught by a signal handler passed to std::signal and the handler does not return.

Destructors of variables with automatic, thread local (since C++11) and static storage durations are not called. Functions registered with std::atexit() and std::at_quick_exit (since C++11) are also not called. Whether open resources such as files are closed is implementation defined. An implementation defined status is returned to the host environment that indicates unsuccessful execution.

OS-specific signal/event-handling

SIGACTION(2) **Linux Programmer's Manual** **SIGACTION(2)**

NAME top

sigaction, rt_sigaction - examine and change a signal action

SYNOPSIS top

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

```
void
handler(int sig, siginfo_t *info, void *ucontext)
{
    ...
}
```

These three arguments are as follows

sig The number of the signal that caused invocation of the handler.

info A pointer to a *siginfo_t*, which is a structure containing further information about the signal, as described below.

ucontext
This is a pointer to a *ucontext_t* structure, cast to *void **. The structure pointed to by this field contains signal context information that was saved on the user-space stack by the kernel; for details, see sigreturn(2). Further information about the *ucontext_t* structure can be found in getcontext(3). Commonly, the handler function doesn't make any use of the third argument.

UnhandledExceptionFilter function

An application-defined function that passes unhandled exceptions to the debugger, if the process is being debugged. Otherwise, it optionally displays an **Application Error** message box and causes the exception handler to be executed. This function can be called only from within the filter expression of an exception handler.

Syntax

C++

```
LONG WINAPI UnhandledExceptionFilter(
  _In_ struct _EXCEPTION_POINTERS *ExceptionInfo
);
```

Parameters

ExceptionInfo [in]
A pointer to an EXCEPT
pointer is the return valu

This

_EXCEPTION_RECORD structure

02.10.2018 • 4 Minuten Lesedauer

Describes an exception.

Syntax

Kopieren

```
typedef struct _EXCEPTION_RECORD {
  DWORD                    ExceptionCode;
  DWORD                    ExceptionFlags;
  struct _EXCEPTION_RECORD *ExceptionRecord;
  PVOID                    ExceptionAddress;
  DWORD                    NumberParameters;
  ULONG_PTR                ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;
```

Code, anyone?